



Teradata Database

---

# Database Design

Release 15.0  
B035-1094-015K  
June 2014



The product or products described in this book are licensed products of Teradata Corporation or its affiliates.

Teradata, Active Data Warehousing, Active Enterprise Intelligence, Applications-Within, Aprimo Marketing Studio, Aster, BYNET, Claraview, DecisionCast, Gridscale, MyCommerce, SQL-MapReduce, Teradata Decision Experts, "Teradata Labs" logo, Teradata ServiceConnect, Teradata Source Experts, WebAnalyst, and Xkoto are trademarks or registered trademarks of Teradata Corporation or its affiliates in the United States and other countries.

Adaptec and SCSISelect are trademarks or registered trademarks of Adaptec, Inc.

AMD Opteron and Opteron are trademarks of Advanced Micro Devices, Inc.

Apache, Apache Hadoop, Hadoop, and the yellow elephant logo are either registered trademarks or trademarks of the Apache Software Foundation in the United States and/or other countries.

Apple, Mac, and OS X all are registered trademarks of Apple Inc.

Axeda is a registered trademark of Axeda Corporation. Axeda Agents, Axeda Applications, Axeda Policy Manager, Axeda Enterprise, Axeda Access, Axeda Software Management, Axeda Service, Axeda ServiceLink, and Firewall-Friendly are trademarks and Maximum Results and Maximum Support are servicemarks of Axeda Corporation.

Data Domain, EMC, PowerPath, SRDF, and Symmetrix are registered trademarks of EMC Corporation.

GoldenGate is a trademark of Oracle.

Hewlett-Packard and HP are registered trademarks of Hewlett-Packard Company.

Hortonworks, the Hortonworks logo and other Hortonworks trademarks are trademarks of Hortonworks Inc. in the United States and other countries.

Intel, Pentium, and XEON are registered trademarks of Intel Corporation.

IBM, CICS, RACF, Tivoli, and z/OS are registered trademarks of International Business Machines Corporation.

Linux is a registered trademark of Linus Torvalds.

LSI is a registered trademark of LSI Corporation.

Microsoft, Active Directory, Windows, Windows NT, and Windows Server are registered trademarks of Microsoft Corporation in the United States and other countries.

NetVault is a trademark or registered trademark of Dell Inc. in the United States and/or other countries.

Novell and SUSE are registered trademarks of Novell, Inc., in the United States and other countries.

Oracle, Java, and Solaris are registered trademarks of Oracle and/or its affiliates.

QLogic and SANbox are trademarks or registered trademarks of QLogic Corporation.

Quantum and the Quantum logo are trademarks of Quantum Corporation, registered in the U.S.A. and other countries.

Red Hat is a trademark of Red Hat, Inc., registered in the U.S. and other countries. Used under license.

SAS and SAS/C are trademarks or registered trademarks of SAS Institute Inc.

SPARC is a registered trademark of SPARC International, Inc.

Symantec, NetBackup, and VERITAS are trademarks or registered trademarks of Symantec Corporation or its affiliates in the United States and other countries.

Unicode is a registered trademark of Unicode, Inc. in the United States and other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other product and company names mentioned herein may be the trademarks of their respective owners.

**THE INFORMATION CONTAINED IN THIS DOCUMENT IS PROVIDED ON AN "AS-IS" BASIS, WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. IN NO EVENT WILL TERADATA CORPORATION BE LIABLE FOR ANY INDIRECT, DIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS OR LOST SAVINGS, EVEN IF EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.**

The information contained in this document may contain references or cross-references to features, functions, products, or services that are not announced or available in your country. Such references do not imply that Teradata Corporation intends to announce such features, functions, products, or services in your country. Please consult your local Teradata Corporation representative for those features, functions, products, or services available in your country.

Information contained in this document may contain technical inaccuracies or typographical errors. Information may be changed or updated without notice. Teradata Corporation may also make improvements or changes in the products or services described in this information at any time without notice.

To maintain the quality of our products and services, we would like your comments on the accuracy, clarity, organization, and value of this document. Please email: [teradata-books@lists.teradata.com](mailto:teradata-books@lists.teradata.com).

Any comments or materials (collectively referred to as "Feedback") sent to Teradata Corporation will be deemed non-confidential. Teradata Corporation will have no obligation of any kind with respect to Feedback and will be free to use, reproduce, disclose, exhibit, display, transform, create derivative works of, and distribute the Feedback and derivative works thereof without limitation on a royalty-free basis. Further, Teradata Corporation will be free to use any ideas, concepts, know-how, or techniques contained in such Feedback for any purpose whatsoever, including developing, manufacturing, or marketing products or services incorporating Feedback.

**Copyright © 2000-2014 by Teradata. All Rights Reserved.**

# Preface

## Purpose

This book describes database design in the Teradata environment including the context of design within data warehousing, logical design, and physical design.

## Audience

This book is intended for database administrators and other technical personnel responsible for designing Teradata relational databases.

## Supported Software Releases and Operating Systems

This book supports Teradata® Database 15.0.

Teradata Database 15.0 is supported on:

- SUSE Linux Enterprise Server 10 SP3
- SUSE Linux Enterprise Server 11 SP1

Teradata Database client applications support other operating systems.

## Prerequisites

You should be familiar with the basic terminology and concepts of data modeling and with the Teradata software and hardware architectures.

It might be helpful to review the following books:

- *Introduction to Teradata*
- *Database Administration*
- *Data Dictionary*
- SQL book set
- *Utilities*

# Changes to This Book

Release	Description
Teradata Database 15.0 June 2014	<ul style="list-style-type: none"><li>Added JSON to the data types that cannot be used in these types of constraints: semantic database, column-level CHECK, UNIQUE, PRIMARY KEY, REFERENCES, and foreign key.</li><li>Corrected terminology for derived period columns. Derived period is not a data type.</li></ul>
Teradata Database 15.0 March 2014	<ul style="list-style-type: none"><li>Added the JSON and MBB 3-D geospatial data types to the chapter on database-level capacity planning considerations.</li><li>Updated the database limits section of the Teradata System Limits appendix to show 1 MB rows.</li><li>Added ANSI temporal tables support.</li><li>Removed references to Teradata Replication Services and Teradata Statistics Wizard, which are not supported in this release.</li></ul>

# Additional Information

URL	Description
<a href="http://www.info.teradata.com/">www.info.teradata.com/</a>	Use the Teradata Information Products Publishing Library site to: <ul style="list-style-type: none"><li>View or download a manual:<ol style="list-style-type: none"><li>Under <b>Online Publications</b>, select <b>General Search</b>.</li><li>Enter your search criteria and click <b>Search</b>.</li></ol></li><li>Download a documentation CD-ROM:<ol style="list-style-type: none"><li>Under <b>Online Publications</b>, select <b>General Search</b>.</li><li>In the <b>Title or Keyword</b> field, enter <i>CD-ROM</i>, and click <b>Search</b>.</li></ol></li></ul>
<a href="http://www.teradata.com">www.teradata.com</a>	The Teradata home page provides links to numerous sources of information about Teradata. Links include: <ul style="list-style-type: none"><li>Executive reports, white papers, case studies of customer experiences with Teradata, and thought leadership</li><li>Technical information, solutions, and expert advice</li><li>Press releases, mentions and media resources</li></ul>
<a href="http://www.teradata.com/t/TEN/">www.teradata.com/t/TEN/</a>	Teradata Customer Education delivers training that builds skills and capabilities for our customers, enabling them to maximize their Teradata investment.

URL	Description
<a href="https://tays.teradata.com/">https://tays.teradata.com/</a>	Use Teradata @ Your Service to access Orange Books, technical alerts, and knowledge repositories, view and join forums, and download software patches.
<a href="http://developer.teradata.com/">developer.teradata.com/</a>	Teradata Developer Exchange provides articles on using Teradata products, technical discussion forums, and code downloads.

To maintain the quality of our products and services, we would like your comments on the accuracy, clarity, organization, and value of this document. Please email [teradata-books@lists.teradata.com](mailto:teradata-books@lists.teradata.com).

## Product Safety Information

This document may contain information addressing product safety practices related to data or property damage, identified by the word Notice. A notice indicates a situation which, if not avoided, could result in damage to property, such as equipment or data, but not related to personal injury.

**Example:**

**Notice:** Improper use of the Reconfiguration utility can result in data loss.

## Teradata Database Optional Features

This book may include descriptions of the following optional Teradata Database features and products:

- Teradata Columnar
- Teradata Row Level Security
- Teradata QueryGrid: Teradata Database-to-Hadoop
- Teradata Temporal
- Teradata Virtual Storage (VS)
- Teradata QueryGrid: Teradata Database-to-Oracle Database

You may not use these features without the appropriate licenses. The fact that these features may be included in product media or downloads, or described in documentation that you receive, does not authorize you to use them without the appropriate licenses.

Contact your Teradata sales representative to purchase and enable optional features.



# Table of Contents

---

<b>Preface.....</b>	3
Purpose .....	3
Audience .....	3
Supported Software Releases and Operating Systems .....	3
Prerequisites .....	3
Changes to This Book.....	4
Additional Information .....	4
Product Safety Information .....	5
Teradata Database Optional Features .....	5
<b>Chapter 1: Designing for the Data Warehouse .....</b>	19
The Heart of the Data Warehouse.....	19
Data Marts and Data Warehouses.....	19
Data Marts.....	19
Common Problems With Data Marts.....	23
The Teradata Concept of the Data Warehouse .....	24
Special Design Considerations for the Teradata Data Warehouse Environment.....	25
Born To Be Parallel.....	25
Data Placement to Support Parallel Processing.....	26
Intelligent Internodal Communication .....	29
Optimizing Queries for a Parallel Environment .....	30
Request Parallelism.....	32
Synchronization of Parallel Operations .....	34
Design Considerations .....	36
Usage Considerations: OLTP and Data Warehousing.....	36
Usage Considerations: Summary Data and Detail Data .....	39
Usage Considerations: Simple and Complex Queries.....	41
Usage Considerations: Ad Hoc Queries .....	44

---

<b>Chapter 2: Logical Data Modeling .....</b>	.47
Databases and Data Modeling.....	.47
Database Design Life Cycle .....	.47
Designing for OLTP and Designing for Data Warehousing Support .....	.48
ANSI/X3/SPARC Three Schema Architecture .....	.49
Requirements Analysis .....	.53
Logical Database Design.....	.53
Activity Transaction Modeling .....	.54
Physical Database Design .....	.55
<b>Chapter 3: Requirements Analysis .....</b>	.57
Getting the System Right the First Time.....	.57
Supporting User Access to the System .....	.57
Producing a Reasonable Estimate of Costs.....	.57
Developing an Enterprise Data Model .....	.58
<b>Chapter 4: Semantic Data Modeling .....</b>	.61
The Entity-Relationship Model.....	.61
Entities, Relationships, and Attributes .....	.63
Translating Entities and Relationships Into Tables .....	.65
Relationship Theory .....	.67
One-to-One Relationships .....	.68
One-to-Many Relationships.....	.69
Many-to-Many Relationships .....	.70
Moving From an Entity-Relationship Analysis to Normalization .....	.71
<b>Chapter 5: The Normalization Process .....</b>	.73
Why Teradata Encourages a Fully Normalized Schema .....	.73
The Key, the Whole Key, and Nothing But the Key .....	.74
Properties of Relations and Their Logical Manipulation .....	.75
Functional, Transitive, and Multivalued Dependencies .....	.80

---

The Normal Forms .....	83
Third and Boyce-Codd Normal Forms.....	86
Decomposing Relations .....	89
Identifying Candidate Primary Keys.....	90
Foreign Keys .....	94
The Referential Integrity Rule .....	95
Domains and Referential Integrity .....	100
Normalization and Database Design Problems .....	103
General Procedure for Achieving a Normalized Set of Relations.....	107
Advantages of Normalization for Physical Database Implementation .....	109
Denormalized Physical Schemas and Ambiguity.....	113

---

## Chapter 6: The Activity Transaction Modeling Process ..... 123

Purpose .....	123
Goals.....	123
Terminology .....	124
Domains.....	128
Column Names and Constraints.....	133
Guidelines for Naming Columns.....	134
Key Values and Relationships Among Tables .....	137
Domains Form .....	138
Constraints Form .....	142
System Form .....	144
Application Form .....	145
Report/Query Analysis Form.....	146
Table Form .....	152
Filling Out the Table Form .....	154
Table Form Example.....	154
Table Form: Basic Information .....	155
Table Form: Column-Level Information .....	156
Table Form: Miscellaneous Column-Level Information.....	157
Table Form: Access Information .....	158
Table Form: Data Demographics for Single-Column Database Objects.....	158
Maximum and Typical Column Value Frequencies .....	160
Table Form: Data Demographics for Multicolumn Database Objects .....	165
Row Size Calculation Form .....	170

---

<b>Chapter 7: Denormalizing the Physical Schema .....</b>	175
Denormalization, Data Marts, and Data Warehouses.....	175
Denormalization Issues .....	176
Commonly Performed Denormalizations .....	178
Alternatives to Denormalization .....	178
Denormalizing With Repeating Groups.....	178
Denormalizing Through Prejoins .....	179
Denormalizing Through Join Indexes .....	181
Derived Data Attributes .....	181
Denormalizing Through Global Temporary and Volatile Tables .....	183
Denormalizing Through Views .....	185
Dimensional Modeling, Star, and Snowflake Schemas .....	187
<b>Chapter 8: Teradata Database Indexes and Partitioning .....</b>	191
Types of Indexes .....	191
Advantages of Indexes .....	195
Disadvantages of Indexes .....	195
Indexes and Partitioning .....	196
Using Indexes to Enhance Performance.....	204
Keys and Indexes.....	206
SQL and Indexes .....	209
Primary Indexed Tables, NoPI Tables, and Column-Partitioned Tables .....	211
Secondary Indexes.....	213
Join Indexes.....	215
Hash Indexes .....	217
Reference Indexes .....	219
Indexing and Hashing .....	220
Tradeoffs Between Hashing and Indexing .....	222
Teradata Database Hashing Algorithm.....	225
Hash Collisions .....	228
Hash Maps.....	229
Hash-Based Table Partitioning to AMPs .....	234
Hash-Related Functions.....	243
Hashed Row Access.....	246
Hashing and Data Types .....	250

---

Teradata Database Index Comparisons .....	252
Table Access Summary.....	255
Index Selection Summary.....	259

---

## **Chapter 9: Primary Indexes and NoPI Objects.....** 261

Primary Indexes .....	261
Primary Index Defaults.....	263
Unique and Nonunique Primary Indexes.....	264
Row-partitioned and Nonpartitioned Primary Indexes .....	266
NoPI Tables, Column-Partitioned Tables, and Column-Partitioned Join Indexes .....	280
Column-Partitioned Tables and Join Indexes .....	285
Column Partitioning Performance .....	297
Storage and Other Overhead Considerations for Partitioning.....	332
Memory Limitations and Partitioning .....	335
Advantages and Disadvantages of Partitioned Primary Indexes .....	341
Effects of Partition Cardinality On Query Performance.....	342
Selecting the Partitioning Granularity .....	342
Usage Recommendations For Partitioning.....	346
Single-Level Partitioning .....	362
Single-Level Partitioning Case Studies .....	370
Multilevel Partitioning .....	372
Detailed Multilevel Partitioning Example .....	393
Row Partition Elimination.....	400
Row-Partitioned and Nonpartitioned Primary Index Access for Typical Operations .....	403
Summary of Primary Index Selection Criteria.....	406
Principal Criteria for Selecting a Primary Index .....	407
Secondary Considerations for Selecting a Primary Index.....	416
Distribution Demographics .....	416
Access Demographics .....	417
Volatility of Indexed and Partitioning Column Values.....	420
Selecting a Primary Index for a Queue Table.....	421
Column Distribution Demographics and Primary Index Selection.....	422
Locating a Row Using Its Primary Index .....	436
Performance Considerations for Primary Indexes.....	441
Normalization, Denormalization, and Primary Index Usage.....	443
Duplicate Row Checks for NUPIs .....	444

Minimizing Duplicate NUPI Row Checks .....	448
Primary Index Uniqueness and Row Distribution .....	450

---

**Chapter 10: Secondary Indexes .....455**

Purpose .....	456
Restrictions .....	456
Space Considerations .....	457
Unique Secondary Indexes.....	457
Nonunique Secondary Indexes .....	467
Multiple NUSI Access.....	476
NUSI Bit Mapping .....	479
NUSIs and Query Covering.....	482
Value-Ordered NUSIs and Range Conditions .....	484
Selecting a Secondary Index.....	488
Secondary Index Access Summarized by Example .....	490
Secondary Index Build and Access Operations Summarized .....	494
Secondary Index Usage Summary.....	497

---

**Chapter 11: Join and Hash Indexes.....499**

Join Indexes.....	499
Using Join Indexes .....	509
Join Index Design Tips.....	517
Join Index Benefits and Costs .....	520
Cost/Benefit Analysis for Join Indexes .....	534
Join Index Types .....	539
Simple Join Indexes .....	541
Defining a Simple Join Index on a Binary Join Result.....	542
Defining and Using a Simple Join Index With an <i>n</i> -way Join Result.....	545
Single-Table Join Indexes.....	546
Single-Table Join Index .....	550
Aggregate Join Indexes .....	552
Sparse Join Indexes.....	556
Using Outer Joins in Join Index Definitions .....	558
Using Outer Joins to Define Join Indexes.....	559

Creating Join Indexes Using Outer Joins .....	561
Join Indexes and Tactical Queries .....	564
Join Index Definition Restrictions.....	572
Improving Join Index Performance .....	594
Join Index Storage.....	598
Value-Ordered Storage of Join Index Rows .....	602
Hash Indexes .....	602
Collecting Statistics on Hash Index Columns .....	608
Hash Index Definition Restrictions.....	611
Hash and Join Index Interactions With Other Teradata Database Systems and Features ..	613
Tradeoffs for Join or Hash Indexes .....	615

---

**Chapter 12: Designing for Database Integrity .....** 617

Sources of Data Quality Problems.....	618
Logical Integrity Constraints .....	625
How Relational Databases Are Built From Logical Propositions.....	631
Inclusion Dependencies .....	632
Semantic Integrity Constraint Types.....	633
Semantic Constraint Specifications.....	636
Semantic Constraint Enforcement .....	657
Updatable Cursors and Semantic Database Integrity .....	658
Semantic Integrity Constraints for Updatable Views .....	659
Summary of Fundamental Database Principles.....	663
Physical Database Integrity .....	664
Disk I/O Integrity Checking.....	666
Detecting Data Corruption Using Disk I/O Integrity Checksums.....	667
Integrity Checking Using a Checksum .....	668
Disk I/O Integrity Level Settings Based on Table Type .....	669
About Reading or Repairing Data from Fallback.....	672

---

**Chapter 13: Designing for Missing Information.....** 673

Semantics of SQL Nulls .....	673
Inconsistencies in How SQL Treats Nulls.....	674
Bivalent and Higher-Valued Logics.....	676

Alternatives To Nulls for Representing Missing Information .....	679
Systematic Use of Default Values .....	679
Redesigning the Database to Eliminate the Need for Nulls .....	680
Manipulating Nulls With SQL.....	682
Logical and Arithmetic Operations on Nulls .....	682
NULL Literals .....	686
Hashing on Nulls .....	687
Null Sorts as the Lowest Value in a Collation .....	687
Searching for Nulls Using a SELECT Request .....	687
Searching for Nulls and Nonnulls In the Same Search Condition.....	688
Excluding Nulls From Query Results .....	688
Nulls and the Outer Join .....	688

---

## Chapter 14: Database-Level Capacity Planning Considerations .....

691

Capacity Planning.....	691
Compression Types Supported by Teradata Database .....	695
Teradata Database Mechanisms for Multi-Value Compression.....	711
Storing Data Efficiently .....	713
Tradeoffs Between Multi-Value Compression and Storage Requirements for Compressed Values .....	717
Computing and Interpreting the Break-Even Point for Multi-Value Compression .....	718
Determining How Much Multi-Value Compression Can Be Realized .....	721
Calculating the Efficiency of Multi-Value Compression.....	725
Base Table Row Format .....	740
Hash and Join Index Row Structures .....	774
Secondary Index Subtable Row Structures.....	780
Presence Bits .....	780
Table Headers .....	787
Column Sizing Guidelines .....	791
Sizing Structured UDT Columns .....	792
System-Derived and System-Generated Columns.....	800
Data Type Considerations .....	822
Numeric Data Types.....	823
Integer Data Types .....	825
Non-INTEGER Numeric Data Types.....	826
Byte Data Types.....	829

DateTime Data Types.....	830
Interval Data Types.....	831
Period Data Types.....	838
Character Data Types .....	840
XML/XMLTYPE Data Type.....	842
JSON Data Type .....	843
User-Defined Data Types.....	843
Array Data Types.....	844
Geospatial Data Types .....	845
Row Size Calculation .....	846
Sizing Databases, Users, and Profiles .....	852
Sizing Base Tables, LOB Subtables, XML Subtables, and Index Subtables .....	858
Sizing Base Tables, Hash Indexes, and Join Indexes .....	859
Sizing a LOB or XML Subtable .....	861
Sizing a Unique Secondary Index Subtable .....	862
Sizing a Non-Unique Secondary Index Subtable.....	864
Sizing User-Defined Routines .....	866
Sizing a Reference Index Subtable.....	866
Sizing Spool Space.....	868
Sizing a Query Capture Database .....	870
Sizing Table Space Empirically .....	872

---

**Chapter 15: System-Level Capacity Planning Considerations .....** 875

Database Size Considerations .....	875
System Disk Contents.....	875
Data Disk Contents.....	876
Data Disk Space.....	877
Permanent Space Allocations.....	878
Estimating Database Size Requirements.....	880
Determining Available User Table Data Space.....	882
Designing for Backups .....	890

---

**Chapter 16: Design Issues for Tactical Queries.....** 893

Tactical Queries Defined .....	893
--------------------------------	-----

Scalability Considerations for Tactical Queries .....	894
Localizing the Work .....	896
Database Design Techniques to Support Localized Work .....	903
Single-AMP Queries and Partitioned Tables .....	905
Recommendations for Tactical Queries and Partitioned Tables .....	906
Sparse Join Indexes and Tactical Queries .....	907
All-AMP Queries .....	908
All-AMP Tactical Queries and Partitioned Tables .....	909
Application Opportunities for Tactical Queries .....	910
Other Tools Useful for Monitoring and Managing Tactical Queries.....	914
Monitoring Active Work .....	914
<hr/>	
<b>Appendix A: Notation Conventions.....</b>	<b>917</b>
Table Column Definition and Constraint Abbreviations .....	917
Character Symbols .....	918
Predicate Calculus and Set Theory Notation Used in This Manual.....	919
Dependency Theory Notation Used in This Manual.....	920
<hr/>	
<b>Appendix B: Teradata System Limits .....</b>	<b>921</b>
System Limits .....	921
Database Limits.....	927
Session Limits .....	938
<hr/>	
<b>Appendix C: Designing With Task-Oriented Profiles .....</b>	<b>941</b>
Concepts, Policies, User Profiles, and Rules.....	941
<hr/>	
<b>Appendix D: Summary Physical Design Scenario.....</b>	<b>945</b>
Prerequisites for the Process Review.....	945
Process Review .....	945

---

<b>Appendix E: Sample Worksheet Forms.....</b>	947
<b>Appendix F: Designing Tables for Optimal Performance ...</b>	959
Minimizing Table Size .....	959
Reducing the Number of Table Columns.....	959
Adjusting the DATABLOCKSIZE and MERGEBLOCKRATIO Table Parameters .....	959
Adjusting FREESPACE.....	960
Using Identity Columns, Compression, and Referential Integrity for Optimal Performance Design.....	961
Using Indexes to Enhance Performance.....	961
Understanding the Effects of Altering Tables.....	961
<b>Appendix G: References .....</b>	963
<b>Glossary .....</b>	965
<b>Index.....</b>	999

## Table of Contents

# CHAPTER 1 Designing for the Data Warehouse

---

This chapter introduces the topic of designing a relational database management system for the data warehouse environment.

Data marts are defined, their relationship with data warehouses are described, and their shortcomings are listed.

General guidelines concerning the basic requirements for a proactive-query-enabled data warehouse are described with limited industry-specific focus.

No detailed information about specific applications is introduced.

## The Heart of the Data Warehouse

Without a properly designed and configured database, no data warehouse can provide the sort of ready access to the enterprise-wide data store required to answer the ad hoc tactical and decision support queries and data mining requests that the modern business enterprise generates in its quest to remain ahead of its competition.

Different users require different sorts of data, ranging from canned batch reports to complex ad hoc queries to detailed exploratory analyses examining the finest-grained atomic data of the enterprise. The database must be capable of supporting all these users and all the possible requests they might make of the enterprise data store.

## Data Marts and Data Warehouses

You have no doubt read about data marts and data warehouses. The principal purpose of this topic is to differentiate between the two and to provide commentary on when each is or is not appropriate using Teradata technology.

The topic also introduces the concept of the active data warehouse and contrasts it with the historical definition of data warehousing.

## Data Marts

A data mart is generally a relatively small application- or function-specific subset of the data warehouse database created to optimize application performance for a narrowly defined user population.

Data marts are often categorized into three different types:

- Independent data marts

Independent data marts are isolated entities, entirely separate from the enterprise data warehouse. Their data derives from independent sources and they should be viewed as data pirates in the context of the enterprise data warehouse because their independent inputs, which are entirely separate from the enterprise data warehouse, have a high likelihood of producing data that does not match that of the warehouse.

These independent data marts are sometimes referred to as *data basements*, and Teradata strongly discourages their use (see “[Independent Data Marts](#)” on page 20).

- Dependent data marts

Dependent data marts are derived from the enterprise data warehouse. Depending on how a dependent data mart is configured, it might or might not be useful.

The recommended process uses only data that is derived from the enterprise data warehouse data store and also permits its users to have full access to the enterprise data store when the need to investigate more enterprise-wide issues arises.

The less useful forms of dependent data mart are sometimes referred to as data junkyards (see “[Dependent Data Marts](#)” on page 22).

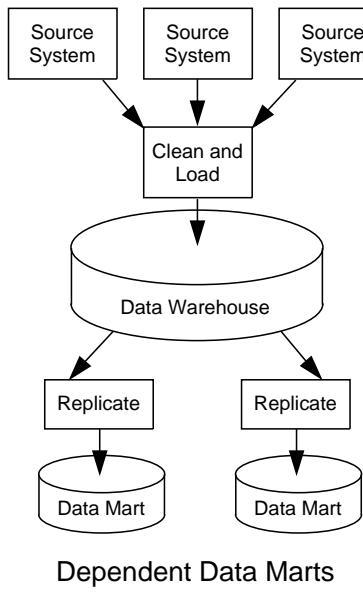
- Logical data marts

The logical mart is a form of dependent data mart that is constructed virtually from the physical data warehouse. Data is presented to users of the mart using a series of SQL views that make it appear that a physical data mart underlies the data available for analysis (see “[Logical Data Marts](#)” on page 23).

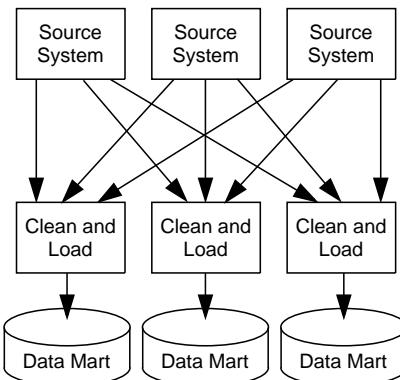
## Independent Data Marts

An independent data mart has neither a relationship with the enterprise data warehouse nor with any other data mart. Its data is input separately and its analyses are conducted autonomously. Because the data is not derived from the central warehouse, the likelihood that it does not match the enterprise data is high. Which version of reality is correct? How can a user know?

Teradata often discourages the use of independent data marts, sometimes referred to disparagingly as “data basements.” Implementation of independent data marts is antithetical to the motivation for building a data warehouse in the first place: to have a consistent, centralized store of enterprise data that can be analyzed in a multiplicity of ways by multiple users with different interests seeking widely varying information.



Dependent Data Marts



Independent Data Marts

1094A091

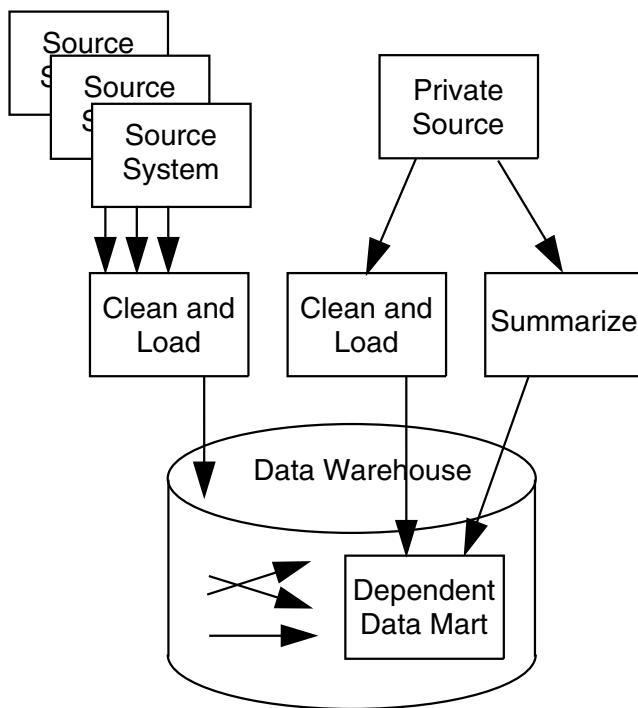
A data basement is a collection of independent data marts. Suppose you have parts that you decide to store in your basement. There is no particular rhyme or reason to what part is stored or where it is stored other than convenience. Continuing the analogy, what is stored in the basement depends on what any family member decides needs to be stored there. If you need to locate a part that you think might have been stored in the basement, you ask everybody in the family if they have seen it recently and then you make your search based on their recollections. If you need to visit more than one basement to find your parts, it is unlikely they will be compatible even if you are able to find them.

This method of storing data is essentially the same as the mix of paper databases and mixed hierarchical and relational online databases spread among multiple departments that supports many businesses today. It is the sort of situation that businesses generally want to escape, not automate.

## Dependent Data Marts

If you need to develop one or more physical data marts in the Teradata environment, you should strongly consider configuring them as dependent data marts. Dependent data marts can be built in one of two ways: either where a user can access both the data mart and the complete data warehouse, depending on need, or where access is limited exclusively to the data mart. The latter approach is deprecated and the type of data mart it produces is sometimes referred to as a data junkyard.

### Dependent Data Mart



FF07D364

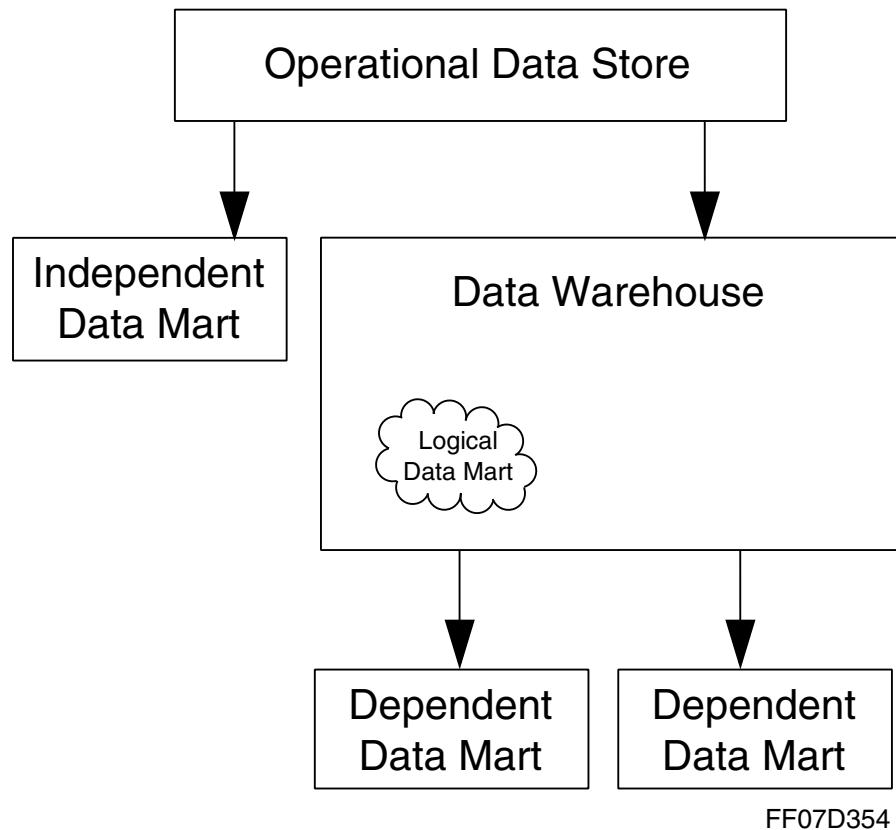
In the data junkyard, all data begins with a common source (in this analogy, “cars”), but they are scrapped, rearranged, and generally junked to get some common parts that the yard operator believes are useful to his customers. The parts collection in the junkyard relates more to what has been useful in the past: previous supply and demand determines what the user can access.

Continuing the analogy, you, as a user, visit the junkyard and search through the various wrecks you encounter in hopes of finding the part you need. To find your part (to answer your question), you will probably need to scavenge through several different junkyards.

The approach results in a decision support environment molded, and compromised, from a specific, well known set of questions and responses rather than around your ever-changing business needs.

## Logical Data Marts

Perhaps the ideal approach to incorporating the data mart concept into your data warehouse is to construct one or more logical, or virtual, data marts. By using a system of carefully constructed views on the detail data of the warehouse, you can design multiple user- or department-specific virtual data marts that provide the same sort of highly tailored information a physical data mart would without the need for massive data loads, cleansing, and other necessary transformations.



## Common Problems With Data Marts

Prospective builders of data warehouses are frequently advised to “start small” with a data mart and use that kernel to expand gradually into a full blown data warehouse. This approach to warehousing generally leads to failed projects for several reasons.

Sometimes the new data mart is so successful that the configuration is overrun by user demands. The databases grow too large too fast, response times become unacceptably long, and user frustration leads to searching for other ways to get the answers.

The more common reason for failure is that the data mart is immediately unsuccessful because it is designed in such a way that users are unable to retrieve the sort of information they want and need to extract from the data. Databases are highly denormalized to respond to a small set of canned queries; summaries, rather than detail data, comprise the database so

that fine-grained exploratory data analysis is not possible; and support for ad hoc queries is either absent or so poor as to discourage users from bothering with them.

The very factors that frequently defeat data mart projects are also the most commonly recommended approaches to designing data marts and data warehouses in the popular data warehousing literature:

- Denormalization (dimensional modeling)
- Storing aggregates at the expense of detail data
- Skewing performance toward a small, preselected set of queries at the expense of all other exploratory analyses

Teradata refers to this approach to building a data warehouse as data mart-centric. Instead of designing a data warehouse from the ground up, the data mart-centric approach begins with one or several highly customized data marts and then attempts to expand the kernel into a full blown data warehouse at some point.

## Avoiding the Data Mart-Centric Approach

Data marts can have a place in a well-designed data warehouse, but not when they are designed as dead ends for user information searches or they cannot meet the needs of your users. Both the data junkyard and the data basement are examples of data mart-centric strategies, and both are doomed to failure.

You are probably asking yourself “If this approach to warehousing is so counterproductive, why do so many vendors and data warehousing guides recommend it?” The answer to this question is fairly simple: vendors recommend these approaches because the products they are selling are incapable of performing at the levels required to do true data warehousing. Authors recommend these approaches because they are only experienced using systems whose deficiencies require data mart-centric design in order to perform at all.

An effective data warehouse is not only *not* data mart-centric, it is a foundation for any information analyses that can be made within the enterprise.

## The Teradata Concept of the Data Warehouse

The data warehouse as conceived by Teradata is a dynamic process. The Teradata data warehouse is an *active* data warehouse that supports the ongoing, day-to-day business of the living enterprise, including many aspects of the traditional operational data store. Enterprise business rules are isomorphic with the rules of the data warehouse. Its most characteristic attribute is its volatility. The Teradata data warehouse is a *dynamic* entity.

In summary, the heart of the data warehouse is your data modeled in such a way that it matches your business. Just as the warehouse is used as a tool to change the business, so it must change over time if it is to continue to be a critical tool for supporting the enterprise. A true data warehouse sows the seeds that force its own change. The only static warehouse is the warehouse with little or no value to the business.

# Special Design Considerations for the Teradata Data Warehouse Environment

An earlier topic in this chapter ([“Avoiding the Data Mart-Centric Approach” on page 24](#)) observed that the most frequently seen advice for how to design the database that supports a data warehouse is antithetical to the overriding reason for having a warehouse: unrestrained access to enterprise-wide business data for any possible question that could possibly be conceived.

Are there any special database design considerations for building a Teradata data warehouse? The short answer is no, there are no *special* concerns, but certain aspects of commonly recommended data warehousing strategies should be evaluated for their veracity before you begin designing the database that supports your data warehouse. For these evaluations, see:

- [“Usage Considerations: OLTP and Data Warehousing” on page 36](#)
- [“Usage Considerations: Summary Data and Detail Data” on page 39](#)
- [“Usage Considerations: Simple and Complex Queries” on page 41](#)
- [“Usage Considerations: Ad Hoc Queries” on page 44](#)
- [Chapter 16: “Design Issues for Tactical Queries.”](#)

The Teradata parallel technology is optimized to perform tasks in a normalized environment that other relational DBMSs cannot match with a denormalized schema. Teradata is also optimized to perform tasks in a denormalized environment. Among the special performance advantages built into the Teradata system are the following: star join and other join optimizations, full table scan optimization, specially designed index types, a full complement of SQL aggregate and ordered analytical functions, and the most sophisticated parallel-aware SQL query optimizer available.

## Born To Be Parallel

### Goals of a Teradata Data Warehouse

- A large capacity, parallel processing database machine with thousands of MIPS capable of storing terabytes of total user data and billions of rows in a single table.
- Fault tolerance, with no single point of failure, to ensure data integrity.
- Redundant network connectivity to ensure system throughput.
- Manageable, scalable growth.
- A fully relational database management system using a standard, non-proprietary access language.
- Faster response times than any competing relational database management systems.
- A centralized shared information architecture in which a single version of the truth is presented to users.

There is a difference between a single *version* (or source) of the truth and a single *view* of the truth. It is quite possible, and often very necessary, to have multiple views of the truth, but these multiple views should all be based on a single version of the truth if they are to be relied upon for decision making.

Among the fundamental aspects of Teradata parallelism are:

- “[Data Placement to Support Parallel Processing](#)”
- “[Intelligent Internodal Communication](#)”
- “[Request Parallelism](#)”
- “[Synchronization of Parallel Operations](#)”

## Designing for Parallel Processing in the Data Warehouse Environment

Because the Teradata architecture was designed from the outset to support a relational database management system, its component subsystems were all designed to optimally support the established norms for such system, including full normalization of the database schema. Significant for the handling of normalized data was the incorporation of parallel processing into the system.

For example, because it was designed to perform parallel processing from the outset, the Teradata architecture does not suffer from the allocation of shared resources that other system that have been adapted for parallelism experience. This is because the system is designed to maximize throughput while multiple dimensions of parallel processing are available for each individual system user. In other words, the so-called “pathologies of big data” (Jacobs, 2009) do not affect Teradata Database.

Repeating for emphasis, *the Teradata architecture is parallel from the ground up and has always been so*. Its file system, message subsystem, lock manager, and query optimizer all fit together snugly, all working in parallel.

# Data Placement to Support Parallel Processing

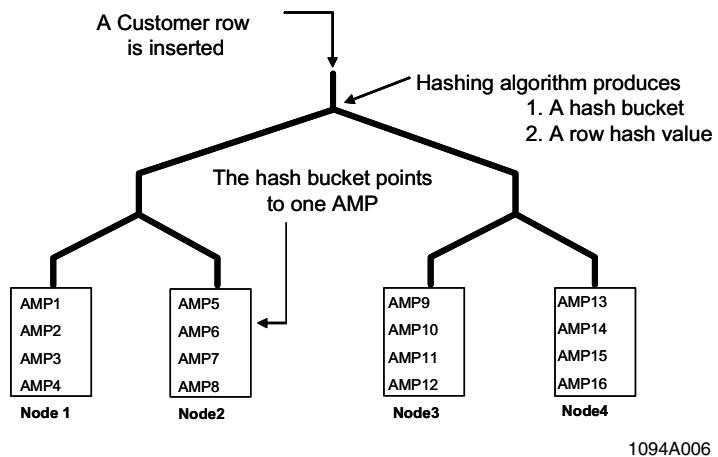
## Balanced Workloads

The key to parallel processing is balancing the workload of the database management system. Teradata Database balances its workload by distributing table rows evenly across its AMPs and by giving the AMPs the sole responsibility for the data they own. See [Chapter 8: “Teradata Database Indexes and Partitioning”](#) for more information.

## Row Distribution

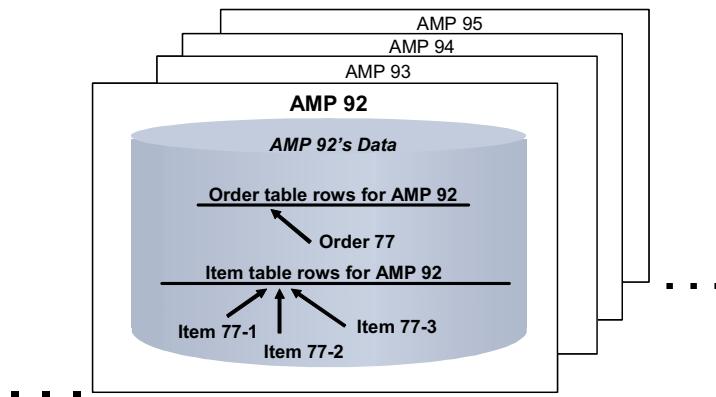
Teradata rows are hashed across the AMPs of a system using the row hash value of their primary index as the hash key. This does not apply for the rows of NoPI tables. See [“Row Allocation for Teradata Parallel Data Pump”](#) on page 237 and [“Row Allocation for FastLoad Operations Into Nonpartitioned NoPI Tables”](#) on page 238 for details.

Teradata Database also uses the row hash of its primary index to retrieve a row. The following graphic shows how a row is hashed and assigned to an AMP:



By carefully choosing the primary index for each table, you can ensure that rows that are frequently joined hash to the same AMP, eliminating the need to redistribute the rows across the BYNET in order to join them.

The following graphic shows how you can set up rows from commonly joined tables on the same AMP to ensure that a join operation avoids being redistributed across the BYNET:



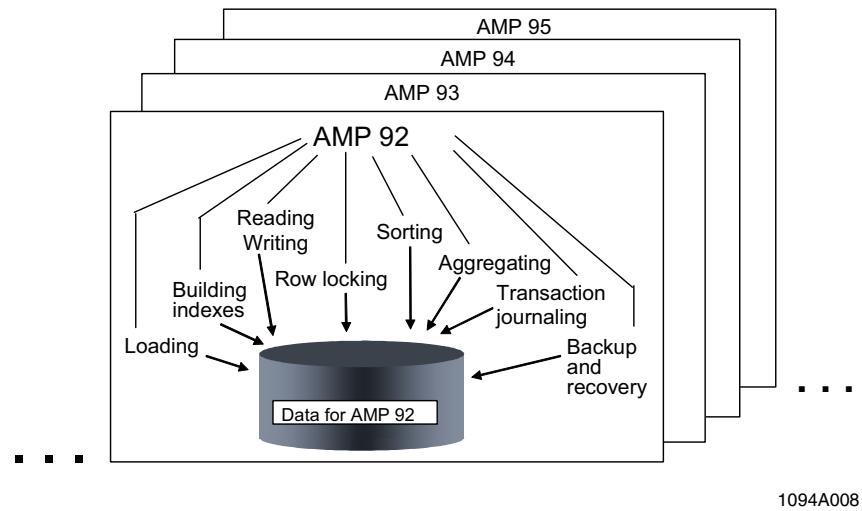
## Shared-Nothing Architecture

The term *shared-nothing architecture* was coined by Michael Stonebraker (1986) to describe a multiprocessor database management system in which neither memory nor disk storage is shared among the processors. The PE and AMP vprocs in the Teradata architecture share neither memory nor disk across CPU units; therefore, Teradata is a shared-nothing database architecture. It is this architecture that affords Teradata systems their scalability.

## AMP Ownership of Data

Because of its shared-nothing architecture, each AMP in a Teradata system exclusively controls its own virtual disk space. As a result, each row is owned by exactly one AMP. That AMP is the

only one in the system that can create, read, update, or lock its data. The AMP-local control of logging and locking not only enhances system parallelism, but also reduces BYNET traffic significantly. The following graphic shows how local autonomy provides each AMP (AMP 92 in this particular example) with total accountability for its own data.



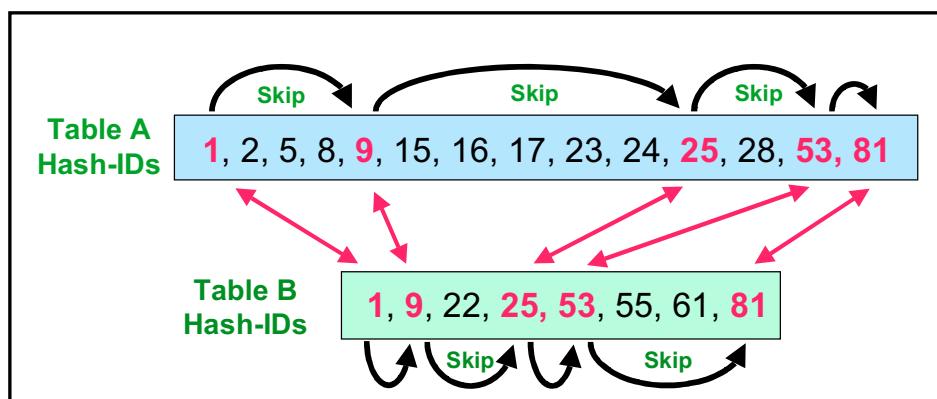
1094A008

## Other Applications of Hashing for Parallel Processing

Teradata Database employs row hashing for tasks beyond simple row distribution and retrieval.

Several types of join processing hash on join column values to determine which AMP is to handle the join of the particular rows. This enables Teradata Database to balance the join load across all AMPs, with each doing an even subset of the total work.

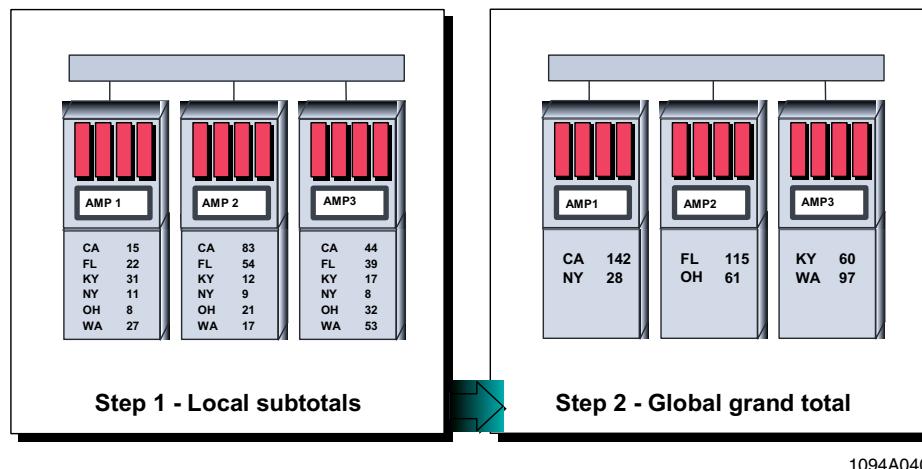
Similarly, the row hash match scan method sorts rows to be joined on their row hash values, then scans the joined tables in parallel to permit the scan of one of the tables to skip ahead in its hash scan to the row hash where the second table is already positioned, as indicated by the following graphic:



1094A039

In this example, *Table A* reads and joins row 1 to row 1 of *Table B*. *Table A* then obtains the row hash value of the next row in *Table B*, row 9, and joins its row 9 to row 9 of *Table B*, which has a matching row hash value. Unmatched rows are skipped without being read by hashing to the next highest candidate value for which a join might be possible. The saltatory processing of candidate joined rows can shorten the time required to perform this join significantly. See *SQL Request and Transaction Processing* for more information about this join method.

Aggregate processing also takes advantage of hashing to build its totals. For example, the following graphic indicates how each AMP aggregates its rows locally as the parallel first step in the global parallel aggregation process:



Following that, the fields in the GROUP BY key are hashed, and the resulting hash bucket for each distinct value points to the AMP responsible for building the global aggregate for that piece of the aggregate.

The illustration shows only 1 AMP per node for simplicity of presentation. A real Teradata system typically has multiple AMPs per node, and each would be involved in performing its own role in producing the global aggregate totals.

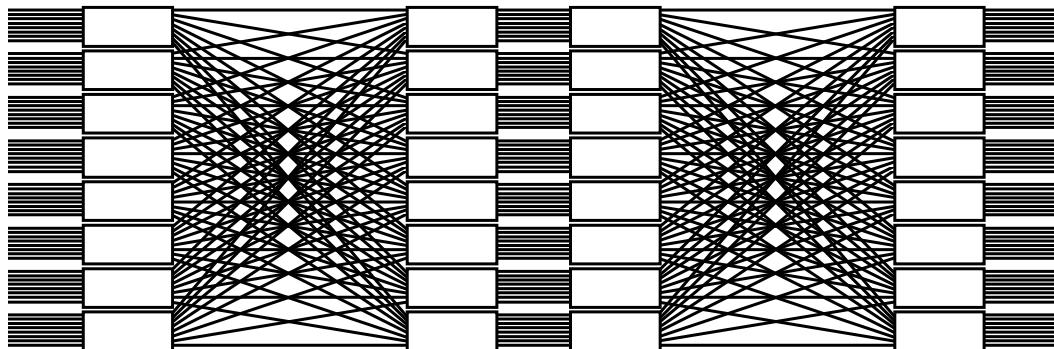
## Intelligent Internodal Communication

### Internodal Communication Over the BYNET

Suboptimal use of any resource quickly affects the other components of an MPP system because of the number of tasks parallel operations can spawn. This is equally true of the internodal communication network that supports the operations of Teradata Database, the BYNET.

All aspects of Teradata Database architecture were designed with parallelism in mind, and the hash-based row distribution and AMP-local processing described in “[Data Placement to Support Parallel Processing](#)” on page 26 work hand-in-hand with the BYNET to optimize parallel processing. By co-locating prospective joined rows on the same AMP and then processing tasks involving them directly on and by the AMP, BYNET traffic is minimized.

The following graphic diagrams the BYNET for a 64 node Teradata system configuration. Each system node has multiple BYNET paths to every other node in the system.



1094A009

## Optimal Network Communication Strategies

To minimize the cost of internodal communication, Teradata Database uses a non-collision, point-to-point monicast communication architecture whenever possible. As a result, a single sender is connected with a single receiver analogous to how calls between individuals are routed over a telephone network. This is less costly from the perspective of system resource usage than broadcasting the same information to all AMPs in the system.

Of course, whenever all AMPs require the same information, such as duplicating the rows of a table on another AMP to make a join or sending a dispatcher message for an all-AMPs operation such as a table-level lock, Teradata Database broadcasts the message across the BYNET.

## Special Purpose Communications Transport

Teradata Database optimizes BYNET communication by implementing a proprietary data communications protocol specifically designed for parallel query processing rather than a general purpose packet switching protocol like TCP/IP.

## Conservation of Network Bandwidth

The BYNET conserves bandwidth by optimizing tasks such as the assurance that messages are committed. For example, rather than performing multiple separate tasks in serial to commit message transmittal, the BYNET performs a single commit protocol analogous to a two-phase commit on each message it transmits.

# Optimizing Queries for a Parallel Environment

The Teradata Database query optimizer was designed from the outset to optimize queries executed in a parallel environment. Unlike the optimizers for many other competing relational database management systems, the Teradata Database optimizer, because it is designed only to optimize queries in a parallel architecture, does not first optimize for a

nonparallel environment and then perform further optimizations to take advantage of a parallel architecture.

When a query optimizer works out the cost of performing a 6 table join in various ways, it must know, within limits, how long each operation required to perform the query takes to perform so it can determine the optimal ordering of the joins. To gain insight into how important this evaluation can be, consider the following table:

Number of Tables Joined	Number of Possible Join Orders
4	24
6	720
8	40,320
10	3,628,000

An optimizer that estimates operational costs based on a single-threaded model will make inappropriate decisions for a parallel processing system. Using the previous table as a guide, you can easily see that any such inappropriate decisions are magnified exponentially as the number of tables to be joined increases.

See *SQL Request and Transaction Processing* for more information about how Teradata Database performs query optimization.

## Parallel Awareness

Among the demographics and statistics available for the Optimizer to use are information about the number of AMPs per node in the system, table cardinalities, and how many of those rows are likely to be touched for a particular operation. From this, the Optimizer knows the number of rows each AMP needs to manage, which it can use to manage file system I/Os most effectively.

The Optimizer determines an AMP-to-CPU ratio that permits it to compare the number of AMPs per node as a function of available CPU power to build the most efficient query plan possible. For example, operations that are more CPU-intensive, like a product join, can be avoided and less CPU-intensive operations substituted for them when possible. This means that the Optimizer will select a product join less frequently in a configuration that has less powerful or fewer CPUs than for a system that has a larger AMP-to-CPU ratio or more powerful CPUs.

Another example of the parallel awareness of the Teradata Database query optimizer is deciding whether to redistribute rows or not for a join operation. The Optimizer selects row redistribution less often in a configuration with many AMPs than it does for a system with few AMPs because row redistribution is an activity that impinges heavily on BYNET traffic as well as being a CPU-intensive operation.

## Request Parallelism

Request parallelism is multidimensional in Teradata Database. Among the more important of these dimensions are:

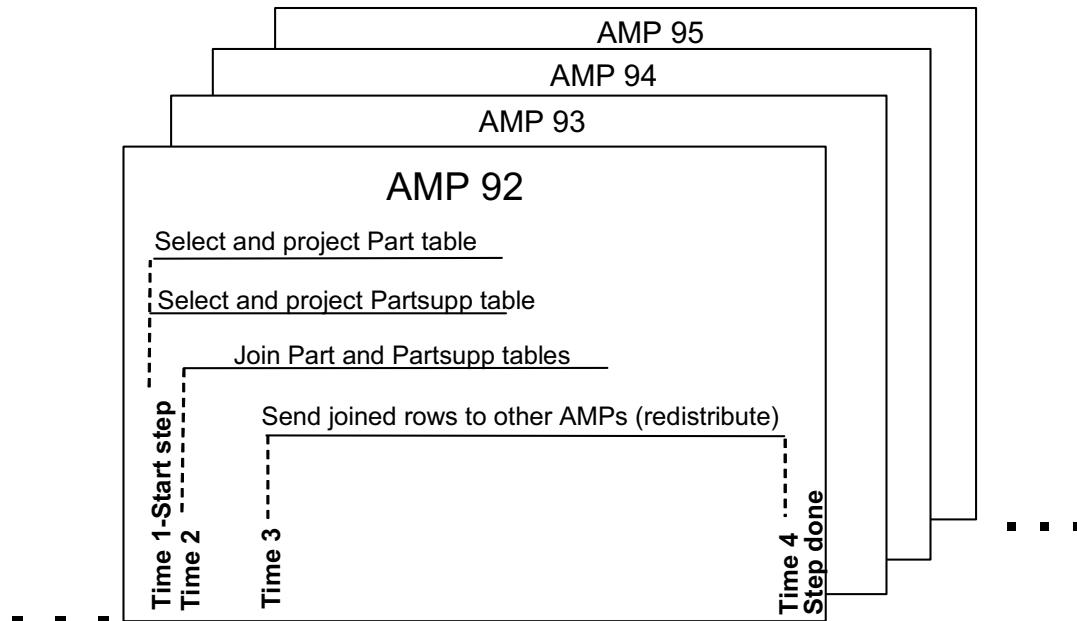
- Query parallelism
- Within-step parallelism
- Multistep parallelism
- Multistatement request parallelism

Teradata Database request parallelism is enabled by the hash distribution of rows across all AMPs in the system (see [“Data Placement to Support Parallel Processing” on page 26](#)). All relational operations perform in parallel, simultaneously, and unconditionally across the AMPs, and each operation on an AMP is performed independently of the data on other AMPs in the system.

### Within-Step Parallelism

When the Optimizer generates a request plan, it parcels out the components of a request into a number of suboperations referred to as concrete steps, which are then dispatched to the appropriate AMPs for execution. An individual step can perform a substantial quantity of work. Within an individual step, multiple relational operations are pipelined for parallel processing. For example, while a table scan is occurring, rows that have already been selected can be pipelined into an ongoing join process. The Optimizer chooses the mix of relational operations within a step very carefully to avoid the possibility of stalls within the pipeline.

The following graphic illustrates how the operations performed by a single step can be pipelined within an AMP:



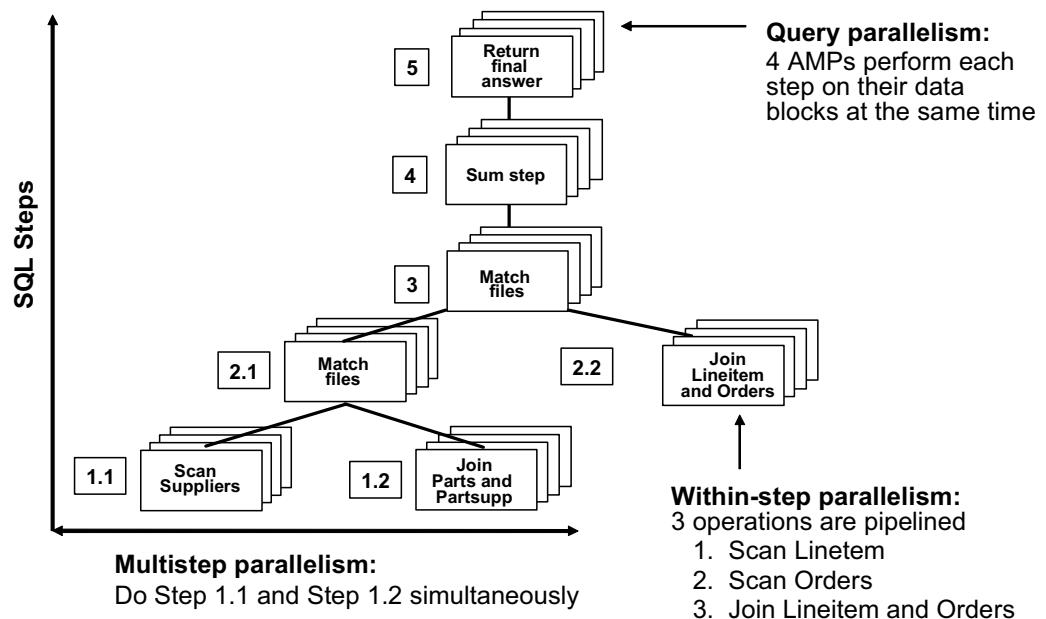
1094A004

## Multistep Parallelism

Multistep parallelism is the simultaneous performance of multiple steps across all units of parallelism in the system.

Teradata Database is the only commercially available relational database management system to implement multistep parallelism. Whenever possible, the Teradata Database invokes one or more processes for each step in a request to perform an operation. Multiple steps for the same request execute simultaneously as long as they are not dependent of the results of previous steps.

You can recognize multistep parallelism in EXPLAIN reports by the dot notation used for the parallel steps; multistep parallelism is illustrated in the following graphic:



1094A005

This diagram shows the steps the Optimizer might generate for a 4-AMP system configuration. Note the multistep parallelism with steps 1 and 2, where steps 1.1 and 1.2 run in parallel with one another as do steps 2.1 and 2.2, respectively.

## Multistatement Request Parallelism

Multistatement requests are a Teradata SQL extension that bundle any number of distinct SQL statements together in such a way that the Optimizer treats them as a single unit. Teradata Database always attempts to perform the SQL statements in a multistatement request in parallel. An example of multistatement parallelism is common subexpression elimination, an operation in which the Optimizer collapses any subexpressions that are common to the SQL statements in the request and performs the extracted operation one time only.

For example, suppose you bundle six SELECT statements together in one multistatement request, and each contained a common subquery. That common subquery is executed only once and the result substituted back into the respective individual select operations.

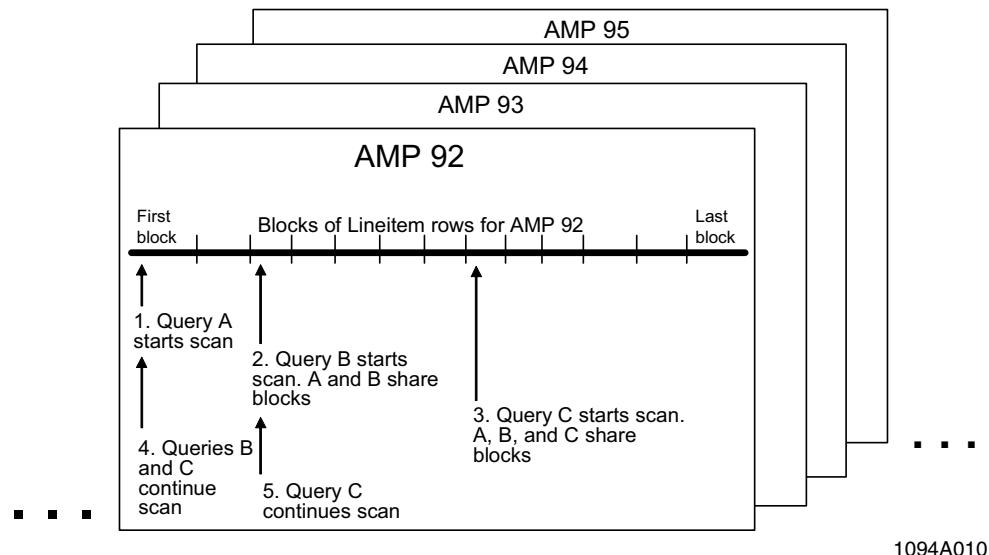
Even though the individual SQL statements within a multistatement request are performed in an interdependent, overlapping set of operations, each returns its own, distinct answer set.

## Synchronization of Parallel Operations

### Synchronized Table Scans

Teradata Database system architecture is not only inherently parallel, but it is also highly optimized to synchronize its parallel operations to effect further performance optimizations.

One of the more recent additions to the Teradata Database parallel processing story is synchronized scanning of cached large tables. Synchronized scanning permits new full table scans to begin at the current point of an ongoing scan of the same table, thus avoiding any I/O for the subsequent scans. Because synchronized scans might not progress at the same rate, any task can initiate the next read operation.



### Spool File Reuse

Teradata Database reuses the intermediate answer sets referred to as spools within a request if they are needed at a later point to process the same query. Two common examples of spool reuse are table self-joins and correlated subqueries.

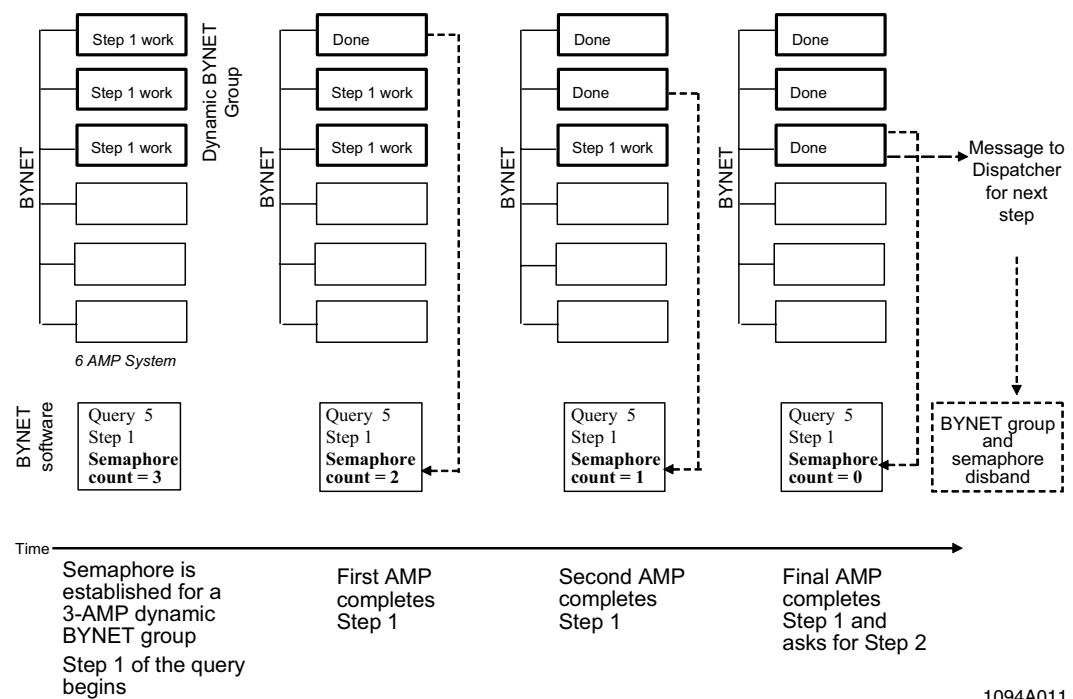
### Synchronized BYNET Operations

Several Teradata Database features act to minimize message flow within the database software. Primary among them are dynamic BYNET groups and global semaphores.

A dynamic BYNET group is an ad hoc grouping of the AMPs working on a single step. Any number of dynamic BYNET groups can coexist simultaneously. Dynamic BYNET groups are established when the first step of an optimized request is about to be dispatched to the AMPs.

Before the first step is dispatched, a message is sent via the BYNET to only those AMPs that will be involved in processing the step. As a result, a dynamic BYNET group might be composed of a single AMP, a group of AMPs, or all AMPs in the system. When an AMP acknowledges reception of the message, it is enrolled in the dynamic BYNET group, sparing the database software from having to initiate a separate communication.

Success and failure global semaphores are associated with dynamic BYNET groups. These objects exist in the BYNET software and signal the completion (or failure) of a step by the first and last AMPs in the dynamic BYNET group. Each success semaphore maintains a count of the number of AMPs in the group still processing the step. As each AMP reports its success, the semaphore count is reduced by 1, so when the last AMP in the group reports, the value of the semaphore is 0. Each AMP reads the value of the success semaphore when it finishes processing the step, and when its value reaches 0, that AMP knows it is the last in the group to complete the task. At that point, the AMP sends a message back to the Dispatcher to notify it that all AMPs in the dynamic BYNET group have finished processing the step, so it can then send the next step to the group. This is the only message sent to the Dispatcher for each step irrespective of the number of AMPs in the dynamic BYNET group.



Failure semaphores are the mechanism Teradata Database uses to support global abort processing. A failure semaphore signals to the other AMPs within a dynamic BYNET group that one of its members has experienced a serious error condition so they can abort the step and free system resources immediately for other uses.

With success and failure semaphores, no external coordinator is required. Global synchronization is built into Teradata Database architecture, and because there is no single point of control, performance can be scaled up to easily handle an increasing volume of system users.

## Design Considerations

- Whether third party vendors from whom you are considering buying tools are Teradata partners. Query tools, for example, should take advantage of the Teradata Database optimizations (for example, they should incorporate the standard Teradata Database aggregate and ordered analytical functions rather than performing in-line calculations of OLAP statistics).
- If you intend to perform a mix of ad hoc tactical and decision support queries or if you plan to undertake any serious data mining projects, then you should limit the physical denormalization of the database to a minimum.  
For example, if you perform frequent ROLAP analyses of your data and performance is not what you expect to see, consider off loading the data to a dependent data mart designed around the multidimensional model that many OLAP proponents advocate. You should not denormalize the entire database just to support a few OLAP applications.
- If you plan to use data mining technology, then a normalized database is the key to your success. Data mining techniques do not perform well in a denormalized environment.

## Usage Considerations: OLTP and Data Warehousing

### Principal Uses of Relational Database Management Systems

The two principal commercial uses for relational systems are online transaction processing (OLTP) and data warehousing (DW). The access patterns of these two approaches are very different and they make very different demands on the underlying database engine.

The next few topics examine the different access patterns of OLTP and DW processing. Once the two styles of processing have been compared, you should readily recognize just how dramatically different they are.

#### OLTP

Consider the following simple example: A customer, Mr. Brown, orders a calendar. The graphic diagrams the actions taken against the database.

**Item**

Item #	Quantity	Description
PK		
3509	10	Calendar
3360	935	8.5 x 14 paper
3474	0	Stapler
3345	875	8.5 x 11 paper
3421	0	#2 pencil

**Customer**

Customer #	Name	Phone
PK		
2	James	555-4444
3	Brown	444-3333
12	Adams	111-9999
9	Black	444-5555
13	Rice	888-9999

**Order**

Order #	Customer #	Order Date	Status
FK	FK		
7324	2	Mar 13	0
7325	3	Mar 13	0
<b>7326</b>	<b>3</b>	<b>Apr 5</b>	<b>0</b>

**Order Item Shipped**

Order #	Item #	Quantity	Ship Date
PK			
FK	FK		
7324	3360	100	Mar 14
7325	3509	30	Mar 14
7324	3474	30	Mar 14
7324	3421	0	Mar 14
7325	3474	10	Mar 14
<b>7326</b>	<b>3509</b>	<b>1</b>	<b>Apr 5</b>

**Order Item Backordered**

Order #	Item #	Quantity
PK		
FK	FK	
7324	3421	144
7325	3474	10

FF07D357

The transaction flow is outlined in the following process stages:

- 1 A new order, 7326, is opened on the *Order* table.
- 2 The remaining columns for the order are filled out, including the *Customer* number for Mr. Brown, 3, and the date the order was placed, April 5.
- 3 A new shipping order, with order number 7326 and item number 3509, is opened on the *Order Item Shipped* table.
- 4 The remaining columns for the shipper are filled out, including the quantity shipped, 1, and the date the order was shipped, April 5.
- 5 The transaction is complete.

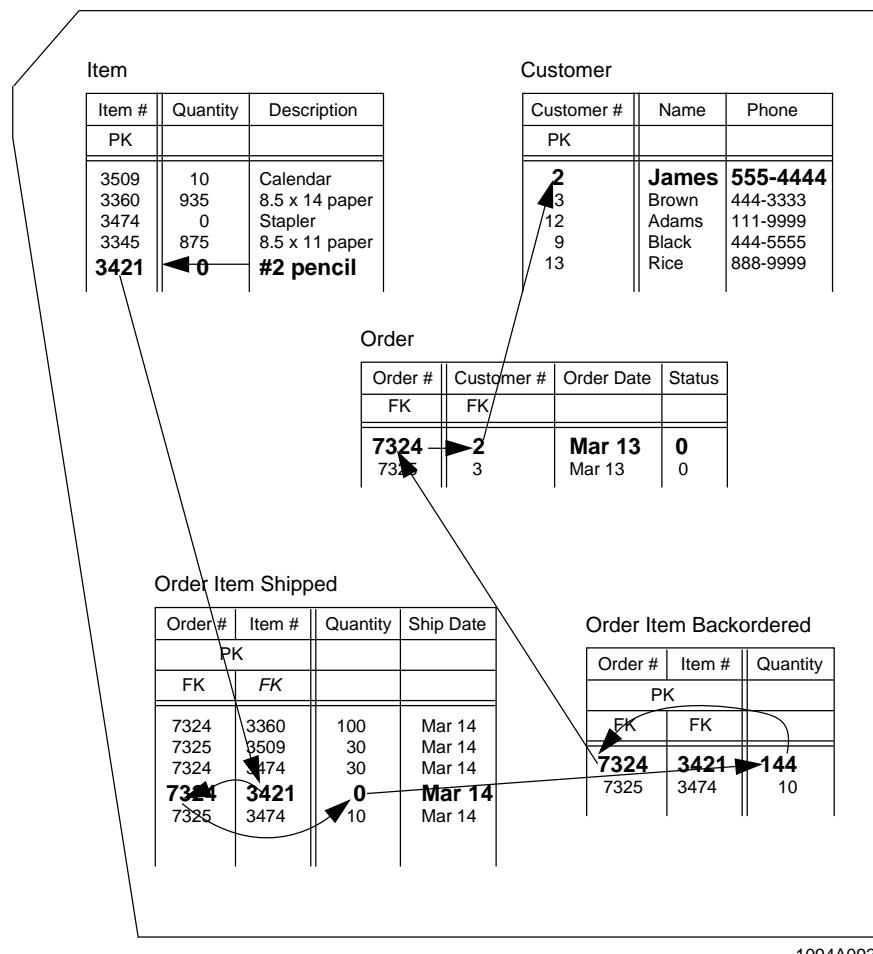
Three key aspects of this transaction deserve your attention:

- Only a few of many possible tables were accessed.
- None of the accessed tables, some of which might have billions of rows, were scanned.
- Very little I/O processing was required to complete the transaction.

These three attributes are characteristic of all OLTP environments. The requirements are simple and involve only a quick-in, quick-out approach to the database. Most modern relational database engines are designed specifically for maximum efficiency with these kinds of workloads.

## Data Warehousing

Consider the following simple decision support example using the same database to emphasize the contrast in processing requirements: A business analyst needs to find the answer to a specific question about pencil customers so he can target them for a promotion. The natural language query the analyst performs is this: which customers placed the majority of their orders for pencils in the first quarter of this year? The graphic diagrams the actions taken against the database.



1094A092

The processing flow is outlined in the following process stages:

- 1 The database engine interrogates the *Item* table for the item number for pencils, which is 3421.
- 2 The database engine next scans the *Order Item Shipped* table looking for all orders associated with item number 3421.

- 3 The particular row examined in this example shipped 0 pencils, indicating that the order was back ordered.
- 4 The engine accesses the *Order Item Backordered* table row having an order and item number matching that examined in stage 3 and finds that 144 pencils were back ordered.
- 5 The database engine must now locate the customer who placed this order because the whole point of the query was to identify customers who had ordered pencils in the first quarter of the year.

To do this, the engine must access the *Order* table, where it finds that customer number 2 placed the order in question.

- 6 The database engine accesses the *Customer* table to identify the name of the customer associated with customer number 2, which is James.

There are also three key aspects of this query that deserve your attention:

- The large number of tables that had to be accessed in order to answer it.  
This complex access path through many tables is a hallmark of decision support analysis.  
And remember: this is an extremely simple example!
- The tables were not just touched lightly in response to the query, but required massive searches, and sometimes multiple scans are required.
- Once the data required to answer the query has been gathered, it must be processed further using aggregation, joins, sorts, conditional requirements, and so on.

All this processing is heavily resource-intensive.

The following table presents a summary contrast of DW and OLTP processing.

OLTP Transaction	Attribute	DW Query
No	Multiple tables scanned	Yes
No	Large volumes of data examined	Yes
No	Processing-intensive	Yes
No	Response time is a function of database size	Yes

## Usage Considerations: Summary Data and Detail Data

This topic examines the nature of the data you keep in your data warehouse and attempts to indicate why storing detail data is a better idea, particularly for ad hoc tactical and decision support queries and data mining explorations.

## Observing the Effects of Summarization

Suppose we have an application that gathers check stand scanner data and stores it in relational tables. The raw detail data captured by the scanner includes a system-generated transaction number, codes for the individual items purchased as part of the transaction, and the number of each item purchased. The table that contains this detail data is named Scanner Data in the following graphic:

Scanner Data			Store Item Daily Sales				Store Item Weekly Sales			
Checkout Transaction No.	Item No.	Quantity Sold	Store No.	Item No.	Date	Quantity Sold	Store No.	Item No.	Week Ending	Quantity Sold
PK			PK			NN, DD	PK			NN, DD
FK	FK	NN	...	...	...	...	...	...	...	...
1234001	1563	12	1	2	Jun 01	110	1	2	Jun 07	1363
1234001	807	1	1	2	Jun 02	126	1	2	Jun 07	456
1234001	2	1	1	2	Jun 03	127	2	2	Jun 07	...
1234001	149	4	1	2	Jun 04	144	2	2	Jun 07	...
1234001	...	...	1	2	Jun 05	102	2	2	Jun 07	...
1234005	...	...	1	2	Jun 06	344	2	2	Jun 07	...
1234005	402	3	1	2	Jun 07	410	2	2	Jun 07	...
1234005	2	2	1	2	...	...	2	2	Jun 07	...
1234027	...	...	1	2	...	...	2	2	Jun 07	...
1234027	2	3	1	2	Jun 01	50	2	2	Jun 07	...
1234027	807	3	1	2	Jun 02	47	2	2	Jun 07	...
1234027	...	...	1	2	Jun 03	32	2	2	Jun 07	...
1234046	...	...	2	2	Jun 04	20	2	2	Jun 07	...
1234046	177	6	2	2	Jun 05	37	2	2	Jun 07	...
1234046	807	1	2	2	Jun 06	144	2	2	Jun 07	...
1234046	2	3	2	2	Jun 07	126	2	2	Jun 07	...
1234639	...	...	2	2	...	...	2	2	Jun 07	...
1234639	2	1	2	2	...	...	2	2	Jun 07	...
1234639	177	1	2	2	...	...	2	2	Jun 07	...
1234639	...	...	2	2	...	...	2	2	Jun 07	...

FF07D359

The middle table, *store\_item\_daily\_sales*, illustrates a first level summary of the data in *scanner\_data*. Notice that where we knew which items sold at which store at which time of day in *scanner\_data*, now we only know the quantity of an item sold for an entire business day. The clarity of the detail data has been replaced by a more fuzzy snapshot. Potential insights have been lost.

The right most table, *store\_item\_weekly\_sales*, illustrates a further abstraction of the detail data. Now all we have is the quantity of each item sold by each store for an entire week. Still fewer insights can be garnered from the data.

Of course, the data could be further abstracted. Summarization can occur at many levels. The important lesson to be learned from this study is that summaries hide valuable information. Worse still, it is not possible to reverse summarize the data in order to regain the original detail. The sharper granularity is lost forever.

Consider this simple, and highly logical, query that an analyst might ask of the sales data: How effective was the mid-week promotion we ran on sales for an item on Tuesday and Wednesday? If the only data available for analysis is a unit volume by week entity, then it is not possible to answer the question. The answer to the question is to be found in the detail, and the analyst has no way to determine the effectiveness of the promotion.

Other basic questions that cannot be answered by summary data include the following:

- What is the daily sales pattern for item 2 at any given store?
- When a customer purchases item 2, what other items are most frequently purchased with it?
- What is the profile of a customer who purchases item 2?

## Information Value of Summary Data

The information value of summary data is extremely limited. As we have seen, summary data cannot answer questions about daily sales patterns by store, nor can it reveal what additional purchases were made in the same market basket, nor can it tell you anything at all about the individual customer who made the purchase.

What summary data can provide is summary answers and nothing more. This puts you in the position of always being reactive rather than proactive. Two classic retail dilemmas posed by this summary-only situation indicate that both extremes of a given problem can be caused by only having access to summary data:

- An out-of-stock situation has only been discovered after it is too late to remedy the problem.
- There is too much stock on hand, forcing an unplanned price reduction promotion to eliminate the unwanted inventories.

## Proactive Use of Detail Data

Suppose the retailer in this case example used detail data to analyze which products tend to cluster in the same market basket. Once the product clusters have been determined, it is possible to rearrange the layout of shelf displays to encourage yet more of this buying behavior.

As an another example, a retailer could use detail data to determine what types of customer tend to buy a particular product or product family. With this information in hand, the retailer could then target a specialized promotion to cross-sell those customers on other products.

# Usage Considerations: Simple and Complex Queries

Users query their databases not because they need an answer as an end result; rather, they want guidance for performing a business action that will have optimal decisive effects. The actions taken cause your bottom line to change, not the answers that informed those actions.

A query can be simple or it can be complex. The answers provided by simple queries tend to be more expected than not, while the answers provided by complex queries tend to produce far less certain answers that are that much richer for resolving the questions they pose.

Consider the following case example from the financial industry. A bank offers its customers a financial instrument that a competitor has decided to offer “without charge.” This challenge

must be responded to quickly or the bank will lose many of its customers who are currently paying what they see as an unnecessary fee for the financial instrument under discussion.

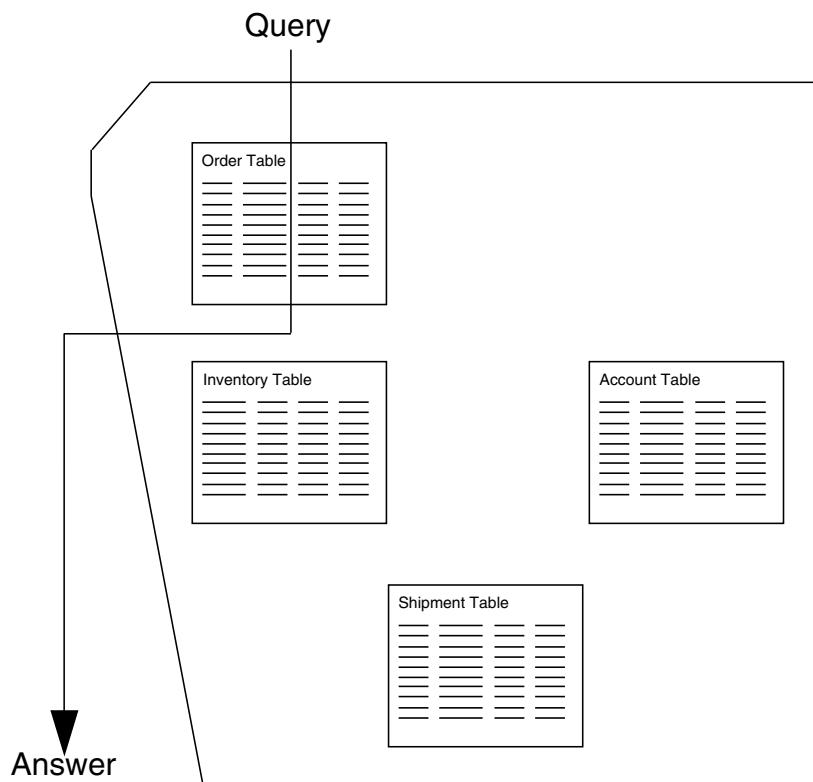
Suppose the bank responds to this situation by asking a simple question such as “which current customers use this product”? The most likely response to this information, which is merely a list of current customers, would be to eliminate the fee for the product. While this action is likely to forestall the erosion of the customer base, it also has the negative effect of reducing profits for the bank.

Suppose, instead, that the bank asks a more sophisticated question such as “which current customers using this product would remain profitable clients even if the fee were eliminated”? The bank now knows not only which consumers of its financial instrument are profitable for reasons other than their consumption of that product, but also knows which consumers do *not* otherwise contribute to the bottom line. The latter customer set can be released to the competition, which, by the way, almost certainly does not know that the new customers it is luring away from our bank are not profitable.

The impact of this more complex query is profit maximization, and its example illustrates very clearly the value of complex over simple queries.

## Relationship Between Query Complexity and the Value of Its Answer

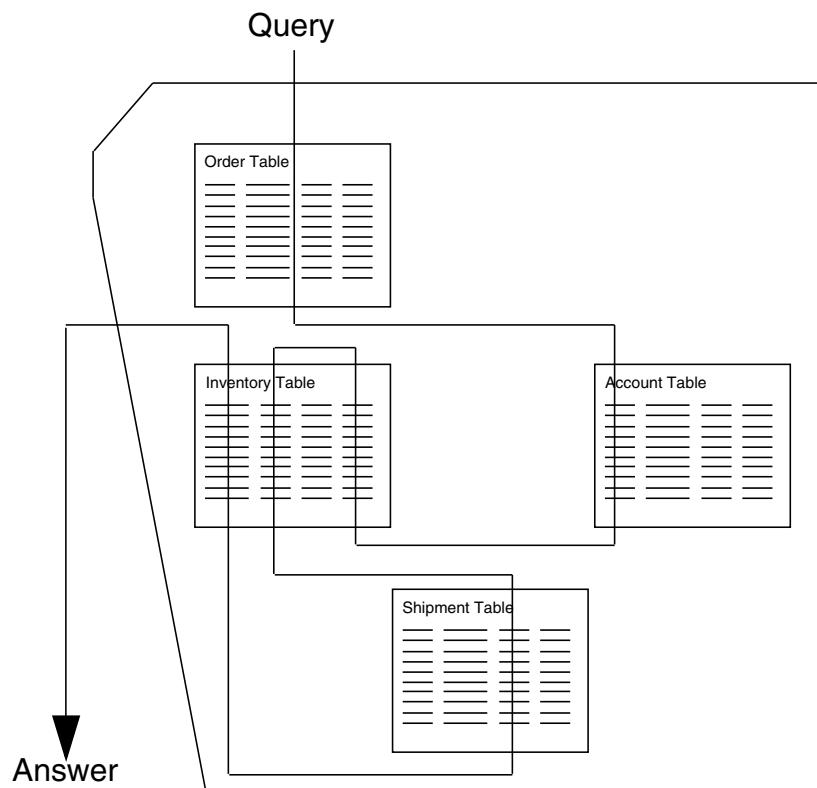
A simple query typically accesses only a few tables, as illustrated by the following graphic.



FF07D372

The simplicity of such a query maps directly into the simplicity of the answer it returns. In other words, simple queries tend to deliver low-value answers which, in turn, enable low-value actions.

More complex queries, on the other hand, investigate the multivariate relationships among many tables in search of the high-value answers that come from mining the many interrelationships among the tables accessed. The following graphic illustrates the concept of a moderately complex query. The example winds its way through four individual tables, accessing one of them several times. A more realistic complex query could easily access hundreds of tables, including as many as 128 of them in a single join!



FF07D360

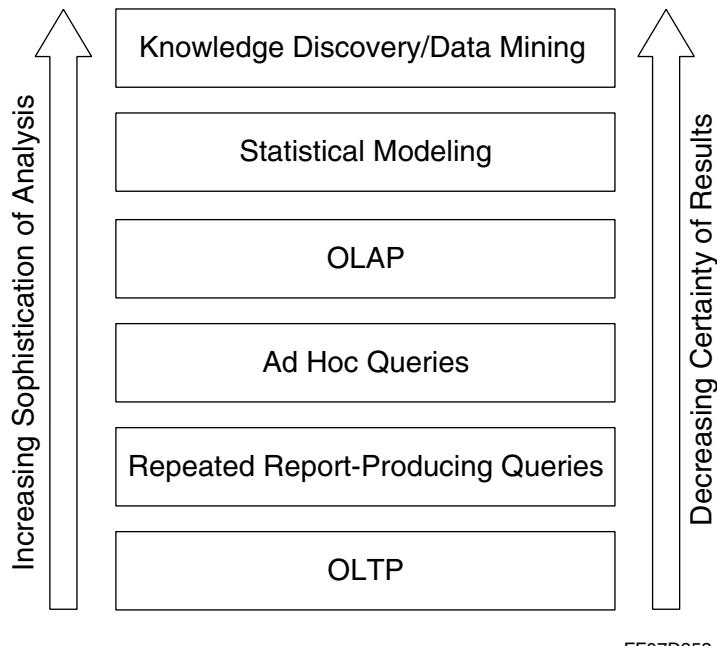
Query complexity exerts a processing I/O burden that many commercial relational database management systems are not capable of handling, which is the principal reason that most data warehouse vendors advocate the use of summary data.

The relationship between query complexity and the extent of detail in the database is direct and profound. For example, Foote and Krishnamurthi (2001), in a study of the data warehousing experiences of WalMart, reported that the company estimates that its power users earn more than \$12,000 for the enterprise per complex query, and that those users perform as many as 1,000 queries per day. While summary data is often good enough to answer simple queries, it cannot deliver the sorts of answers that more complex queries seek. This is an extremely important concept to understand before designing your databases, because you need to provide the level of detail in the data that can deliver answers to the types of questions you will be asking.

## Valuable Information and Time

The value of information is often inversely proportional to the length of time required to derive it.

As the following graphic illustrates, the more sophisticated the analysis, the less expected the answers obtained. More explicitly, this principle can be stated as the more complex the query, the more likely that heretofore unknown information hidden in the data is revealed.



FF07D353

Sophisticated explorations of the data universe might take longer to produce results, but when those results are finally produced, they are invaluable to the business. A quick response is a minimal requirement for simple queries, but such queries rarely provide a business-critical response, and designing a database to ensure nothing *but* quick responses is the quickest path to failure for your data warehouse project. Teradata Database parallel architecture ensures that *all* queries are answered in an optimal time frame.

Your data warehouse can be a source of unimaginable information richness, but only if it is designed with the thought that any possible question, no matter how involved or abstract, should be possible to answer as readily as a simple query.

## Usage Considerations: Ad Hoc Queries

In the data warehousing world, the phrase *ad hoc query* generally refers to a query composed at the keyboard for immediate performance (as opposed to a permanent query stored in a macro, stored procedure, or SQL application that is performed over and over again without alteration). Such a query frequently undergoes careful, extended planning before being performed for the first time and then is often revised and refined interactively until the desired

response is returned. The goal of this type of ad hoc query is not the reporting of data, but the discovery of information.

## Ad Hoc Queries and Enterprise Tactical Support

Tactical queries are focused on operational decision making rather than enterprise discovery or bookkeeping activities. They typically have the following characteristics:

- Relatively simple syntax
- Direct access
- Highly tuned
- Have expected response latencies on the order of 20 seconds or less, and are often expected to return a response in less than one second.

The Teradata data warehouse provides facilities such as the Priority Scheduler (see *Utilities: Volume 2 (L-Z)*) to fully integrate workload mixes of both tactical and strategic queries across the enterprise, enabling executives and frontline decision makers alike to access the same single version of the truth concurrently.

See [Chapter 16: “Design Issues for Tactical Queries”](#) for many of the special considerations you must take into account when designing to support tactical queries.

## Ad Hoc Queries and Enterprise Discovery

“[Usage Considerations: Simple and Complex Queries](#)” on page 41 established that discovery is the critical focus of data warehousing, not simple reporting. The trends fueling the data warehousing revolution are based on the need for information that can be acted upon proactively. Report-oriented systems rely on predefined questions that produce reports of what has already happened. This type of information can only produce evidence that a situation that has already occurred must be reacted to.

A data warehouse must have the capability of performing high performance, complex, ad hoc queries if it is to capitalize on the advantages of warehousing its business data. Furthermore, that data must be available for analysis in detail, as well as summary, formats. The value of data warehousing comes from being able to ask unplanned questions on detail data.

## Business Situations That Drive the Need for Ad Hoc Queries

Two general situations typically drive ad hoc queries: exploratory analysis to discover business opportunities and detailed analysis of why some complicated event that had a negative impact on the enterprise occurred. You must be able to ask questions as they present themselves in either situation and the data warehouse you use for your analysis must be capable of accommodating that need.

Consider the search for new business opportunities. The following table lists some likely business opportunity searches by industry:

Industry	Typical Query Focus
Financial	Cross-selling opportunities
Retail	Market basket analysis
Communications	Calling patterns indicating high risk of losing a customer to the competition.

Now consider the “what went wrong” type of analysis. The following table lists some likely scenarios for this type of ad hoc query by industry:

Industry	Typical Query Focus
Transportation	<ul style="list-style-type: none"><li>• Over capacity</li><li>• Under capacity</li></ul>
Communications	Sudden customer attrition

Businesses cannot know what questions they will need answered in the future, but they must be able to ask questions that permit them to influence the future positively. Neither issue can be dealt with by means of an analysis of the data warehouse if the system does not provide support for complex ad hoc queries.

## CHAPTER 2 Logical Data Modeling

---

This chapter introduces the concept of database modeling, both logical and physical, providing a preview for the remaining chapters in this manual.

### Databases and Data Modeling

As was noted in chapter 1 (see “[The Heart of the Data Warehouse](#)” on page 19), the heart of any data warehouse is its database. A data warehouse is a set of processes about a population of data. That population of data is its underlying database.

The usefulness of a data warehouse is directly proportional to the quality of the database that supports it.

The purpose of this manual is to provide guidelines to help you design a Teradata Database that provides any user of the warehouse with all of the following characteristics:

- A single, unified, unambiguous view of the data
- Optimal access for any possible valid query
- Freedom from navigation

These attributes can only be achieved by careful collection and analysis of requirements and through careful planning of how to implement those requirements. The name used to describe the planning of the structure and relationships of a database is data modeling. Note that the expression *data model* has two very different meanings. In the original definition, a data model is taken to be a set of abstract constructs that can be applied to many different specific applications. Examples would be the relational and CODASYL models for database management. In the case described in this section, the term refers to logical database modeling, which is a specific application of the properties provided by the relational data model. An analogy that is sometimes made is that a data model in the first sense is akin to a programming language while a data model in the second is akin to a particular application program written in a particular programming language.

Data modeling comes in two modes: logical and physical.

### Database Design Life Cycle

Database design is a living process: it never ends. Not only do new features and optimizations come with each new release of Teradata Database software, but new business needs and supporting information requirements continually present themselves and must be integrated with the production database.

In some circumstances, the integration process can be as involved as re-engineering entire components of the database. Fortunately, if you have adhered to an implementation that is fully normalized, the re-engineering process is a relatively harmless and predictable activity.

## Designing for OLTP and Designing for Data Warehousing Support

### Differences Between OLTP and Data Warehouse Processing

The following table examines some of the principal differences in OLTP and Decision Support processing that are critical to database design:

Attribute	Processing Type	
	OLTP Transaction	Decision Support Query
Multiple tables scanned?	No	Yes
Large data volumes?	No	Yes
Processing intensive?	No	Yes
Processing time a function of database size?	No	Yes

As you can see from this side-by-side comparison, OLTP transactions and decision support queries are dramatically different entities and, by inference, might be expected to require physically different database support.

### Data Warehousing Support

Relational databases that support data warehousing are optimized to support decision support applications in all the forms listed below and more.

- Ad hoc SQL or natural language-generated SQL queries
- Repeatedly stated queries in the form of macros, embedded SQL applications, or stored procedures
- Data mining and OLAP explorations

The data in the warehouse is stored in the form of multiple normalized tables that model your enterprise. The decision support applications that explore these data often perform full file scans of multiple large tables, making them processing intensive. Because processing time is directly related to data volume, the responsiveness of the system is heavily affected by the size of the underlying database.

## OLTP

In contrast, OLTP transaction processing tends to operate orthogonally. OLTP systems do not scan multiple tables simultaneously, nor do they access large volumes of data. This makes their consumption of processing and I/O resources minuscule in comparison with decision support query processing. Because the database size determinant is factored from the equation, size is not a factor in the responsiveness of the system.

## ANSI / X3 / SPARC Three Schema Architecture

In 1975, the ANSI/SPARC (American National Standards Institute/Standards Planning And Requirements Committee) Study Group on Data Base Management Systems made its initial interim report on an architecture for database management systems that the committee felt should become an industry standard for the design of commercial database management systems (Bachman et al., 1975).

The ANSI/SPARC architecture was *not* specifically designed for relational databases, which were not commercially available at the time the interim report was published. Explicitly referring to the hierarchical, network, and relational models, the report notes that “Much of the work [on the architecture] has been driven by the desire to accommodate the various requirements statements and differing viewpoints.”

Perhaps the most interesting aspect of the model for relational databases is its pioneering work highlighting the need for data independence. Data independence is a default property of relational databases that hides both the organization of the data and its access from the requesting user or user application.

The report introduces the concept of data independence in the following, rather long-winded and strangely worded section, which is the entirety of its Section 1.7: *Binding and Mapping of Objects to Each Other*:

“Entities and properties of entities can be represented by: objects, the descriptors of which are declared in a source program (e.g., a data division); objects, the descriptors of which are defined in a relational external schema, defined in a Cobol external schema, defined in an accountant’s external schema, and/or defined in a canonical external schema. The descriptors of objects declared in a source program, either directly or from an external schema, are bound to descriptors of objects defined in the conceptual schema. The objects, the descriptors of which are declared in these schemas are all abstract: the internal data is actually stored in objects, the descriptors of which are declared in the internal schema. The objects, the descriptions of which are defined in the conceptual schema, are mapped to objects, the description of which are defined in the internal schema. This does not imply any specific one-to-one mapping among these objects.

“An object that is local to an application’s external model, may be not represented by internal data, and may be not under the control of the enterprise administrator. In this case, there is no object defined in the conceptual schema to which the object defined in the external schema can be bound.

“The conceptual schema may contain descriptors of objects that may be not represented by internal data. This may be during the development of augmentations to the conceptual schema, before defining and collecting internal data, or to increase the understanding of the interrelationships between data stored in this data base and data stored in another data base or data that is not machine processable. In this case, it is not envisioned that any object defined in an external schema would be bound to such an object defined in the conceptual schema, except for testing.” (Bachman et al., 1975, pages II-5 - II-6).

This property makes it possible to extract information from a database without specifying, or even knowing, how to access the underlying data in the database. In a relational database management system, the query optimizer plays the role of a surrogate user and specifies all the particulars of accessing the data.

The implementation of data independence also makes it possible to change the storage structure of the database without changing, in any way, the code used to access the data in the database.

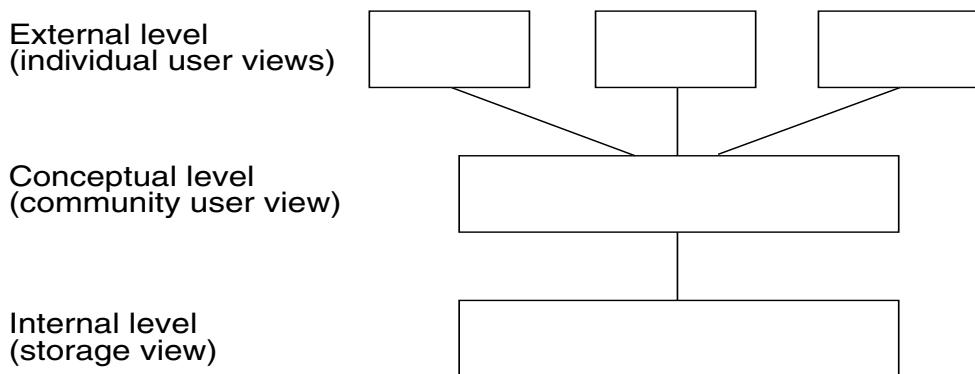
## High Level Architecture

The following graphic presents a high level (and highly simplified) view of the ANSI/SPARC architecture.

ANSI/SPARC specifically names three levels of the architecture.

- External
- Conceptual
- Internal

The External level is composed of all the different views (not specifically in the sense of views in relational systems) of the underlying physical database.



FF07D338

The Conceptual level is concerned with transparently mapping External level views to the physical storage of the actual data in the database. In terms of a relational system, this level is composed of the database management system and the file system.

The Internal level describes the physical storage of the data on the storage media.

## Detailed Architecture

The following diagram portrays the ANSI/X3/SPARC architecture in greater detail and explicitly relates the various levels to their manifestations in a relational database management system. Note the following points:

- Depending on the situation and object privileges, end users communicate with the database in one of two possible ways:
  - Indirectly through an external view
  - Directly
- Communications through views must be converted, or mapped, in both directions.
- Communications between the database and the disk subsystem are made through the file system.

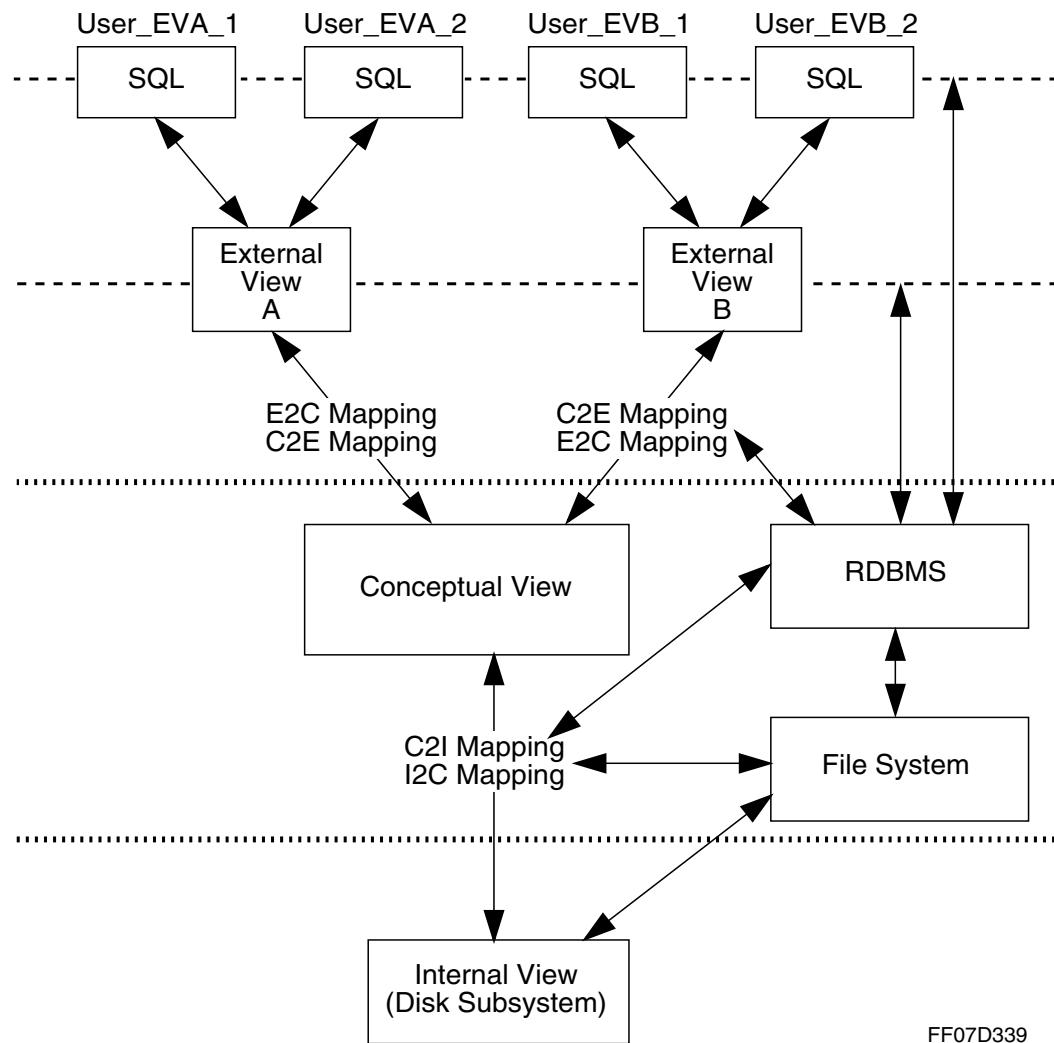
The higher level within the External level is represented by the SQL language for relational systems. The SQL language elements can be embedded within a client application, within a stored procedure, or presented to the database in the form of an ad hoc, interactive query made through a query manager like BTEQ or SQL Assistant or through a natural language SQL code generator application.

The lower level with the External level is represented by a relational view. Views not only mask the underlying database storage, but also the conceptual structure it supports, acting as a virtual tabular interface on the physical base tables.

Note that not all users communicate with the database through views: some, particularly administrative users, communicate with the database directly.

The Conceptual level of the architecture is represented by the relational database management system and the file system. The role of the file system is more nebulous than the diagram portrays and is actually intermediate between the Conceptual and Internal levels, but for purposes of this description of the architecture, it is regarded as a component of the Conceptual level.

The Internal level of the architecture is represented by the disk subsystem. Depending on the configuration, the Internal level could also include storage media such as tape and optical disk. From the perspective of the architecture, only the file system has direct access to the disk subsystem, and it must map data requests and responses in a manner appropriate to the level or component it communicates with.



## Key

The following table explains the abbreviations used in this detailed architecture flow diagram:

Abbreviation	Definition
EVA	External View A
EVB	External View B
E2C	External-to-Conceptual
C2E	Conceptual-to-External
C2I	Conceptual-to-Internal
I2C	Internal-to-Conceptual

## Requirements Analysis

Any design process must begin with the knowledge of what is to be designed. This includes not only the proposed morphology of the end product but also the systems, policies, and procedures - the processes - of the designed product.

This fundamental knowledge is derived through a process of accumulating facts about what the eventual users of the product require to do their work in support of the enterprise. The process includes, at minimum, the following tasks:

- Interviewing notable employees, both management and support staff, for information such as the following:
  - What information do they need?
  - What is the source of that information?
  - What are the tasks involved with creating and reporting the information?
  - How is the information used?
- Gathering all input screens and reports generated by the legacy system and interviewing management and support staff about what is right and wrong about these components as well as determining what sorts of new or different input and report items should be added to the new system.
- Compiling and circulating the cumulative research information you have gathered to obtain affirmation of its accuracy from all involved parties.
- Writing a requirements specification from the approved research information and making it available to the designer of the logical database.

## Logical Database Design

The requirements analysis phase of the design process reveals the real world objects and their attributes that the database must represent as well as the relationships among them.

The logical database design phase formalizes the objects, or entities, and their relationships. Another primary task of logical design is to ensure that the modeled entities are modified by attributes that uniquely pertain to them. No attribute should appear in an entity unless it describes the uniqueness identifier for the entity, its primary key.

See [Chapter 4: “Semantic Data Modeling”](#) for more information about logical database design.

## Normalization

Fortunately, the entire structure of relational database management is built on a solid theoretical underpinning that makes the operations used to associate the proper attributes with the proper entities predictable, repeatable, and formally correct.

This process is referred to as normalization. Normalization is not an art form; it is a science based on provable formal concepts such as dependency theory. The normal forms and their

derivations operate strictly within a series of inference rules and formal logical operations drawn from formal logic and set theory. This should not be taken to mean that only one correct logical database design can be developed for a set of relations. In fact, it is fairly easy to create numerous formally correct logical designs for the same set of relations. The point is that it is possible to develop both correct and incorrect logical designs, and by following the rules of normalization and dependency theory, you can always develop *a* correct logical design.

Unfortunately, the determination of entities and their relationships is subjective, unpredictable, unrepeatable, and more of an art form than a science. In spite of the subjectivity of entity-relationship-attribute analysis, the normalization process almost always rescues any blunders made in the determination of entities and relationships, so it all works out in the end.

See [Chapter 5: “The Normalization Process”](#) and the references cited there for more information about normalization and dependency theory.

## Activity Transaction Modeling

Once a fully normalized logical model has been realized, the next step is to enter the transition phase. In this phase, which Teradata refers to as Activity Transaction Modeling, you translate the entities, attributes, and relationships into a set of worksheets that feed into the physical database design process.

# Activity Transaction Modeling

The Activity Transaction Modeling (ATM) process extends the logical data model by beginning to attach physical attributes to it. In doing so, the ATM process undertakes the following activities:

- Identifies the business rules of the enterprise that apply to the information to be stored in the data warehouse
- Initiates the process of identifying and defining attribute domains and constraints for physical columns
- Identifies and models database applications
- Identifies and models application transactions
- Summarizes table and join accesses by column across transactions
- Compiles a preliminary set of data demographics by computing table cardinalities, value distributions, and attaching change ratings to columns

This information is compiled and used as input to the physical design process.

The ATM process is described in detail in [Chapter 6: “The Activity Transaction Modeling Process.”](#)

## Physical Database Design

Physical database design is the commitment of all the previous design stages to a physical reality.

Entities, attributes, and relationships are identified and normalized in previous phases of the design process. Attribute sets are assigned to domains.

The physical design phase identifies and creates the actual databases, base tables, constraints, indexes, partitioning, views, macros, triggers, and other objects that define the physical database that drives your data warehouse.

See [Chapter 8: “Teradata Database Indexes and Partitioning”](#) through [Chapter 13: “Designing for Missing Information”](#) for more information about physical database design.



# CHAPTER 3 Requirements Analysis

---

The principal reasons for performing a requirements analysis include:

- Getting the system right the first time
- Supporting easy user access to the system
- Producing reasonable and reliable estimates of costs

## Getting the System Right the First Time

A formally agreed upon set of requirements is necessary to get the system right the first time.

Industry analysts have reported IT project failure rates as high as 70%, largely because no requirements were derived for the project apart from IT guesses as to what their users needed.

## Supporting User Access to the System

As more and more of the task of updating and maintaining the database is shifted to the user community, their input into the design and implementation of the system becomes critical.

## Producing a Reasonable Estimate of Costs

Without requirements, it is not possible to estimate the developmental and operational costs of a new database management system. Projects are often terminated by enterprise management because they see costs spiraling out of control. This is as much a problem of psychology as it is an issue of failure to plan.

When a project estimate of X dollars is approved by management, they expect to see IT spend X dollars on the project and no more. When they see costs exceeding initial estimates by an order of magnitude or more, they perceive a project to be a failure even when the system is progressing at a normal rate simply because costs so greatly exceed their expectations. When all parties are aware of what a project is going to cost, the psychological dissonance of budgets is greatly reduced if not eliminated altogether.

Without a requirements analysis, you cannot have any grasp of how much an IT project will cost.

# Developing an Enterprise Data Model

An enterprise data model is a blueprint: a means of ensuring that standards for creating the IT function exist and that they are appropriately integrated with the overall function of the business enterprise the function is designed to support.

Note that an enterprise data model is a data model in the second sense as defined by Date (see “[Definition of a Data Model](#)” on page 61).

## Things To Avoid When Developing an Enterprise Data Model

IT organizations often commit several common mistakes in the course of developing the architecture for their enterprise data model. Instead of providing the usual checklist of recommended practices for developing a successful enterprise data architecture, Rehkopf and Wybott (2003) identify ten of the most frequently made errors in the process of developing an enterprise architecture.

Their list of common errors is as follows, listed in no particular order:

- Treating the architecture as if it were a finished product rather than a process.

Like the enterprise data warehouse, the enterprise data architecture is an ongoing process, not a fixed, concrete thing. As such, it has a continuous life cycle that supports the following areas:

- Identification of particular areas that particular decisions and technology selections affect.
- Provision of guidelines for investment decisions.
- Identification of the major processes, skills, and components needed to support the IT projects that support the enterprise.
- Assuming that technical personnel make the best architects.

Of course, an enterprise data architect *must* be highly technical. The point Rehkopf and Wybott are making is that architects must possess a wide ranging skill set, and that sheer technical competence in the absence of other equally important skills such as the following is a formula for failure:

- Understanding and matching business objectives and needs to technological systems
- Auditing architectural compliance to ensure its overall integrity
- Educating and consulting with users about various architectural issues and so on
- Rejecting input because it does not fit the existing (or preconceived) architecture or, as Rehkopf and Wybott put it simply, "saying 'no'."

Ideally, such decisions should be made by a Chief Information or Technology Officer, not by the architecture staff. The role of the architects is to keep the enterprise architecture life cycle process working, not to assume (or reject) the risks of technology decisions.

- Failing to communicate early and often.

As stated previously, the enterprise data architecture is an ongoing process that must not only be updated continuously, but also sold and re-sold to its constituents. Rehkopf and Wybott suggest several effective methods for achieving this:

- Report all important modifications to the architecture to all affected members of the user community.
- Summarize and report case studies, both of works in progress and of completed projects, of how the architecture has been put into effect by users, including both successful and unsuccessful applications.
- Conduct periodic refresher presentations and seminars on the architecture at department or staff meetings.
- Produce architecture documentation in the form of checklists, templates, and contact lists.
- Partition the enterprise architecture want list documentation into broad sets such as *must have*, *should have*, and *nice to have*.
- Create an architecture intranet site or portal so users can easily access all enterprise project documentation.
- Failing to explain concepts in simple, non-technical language.

It is almost always a mistake to present an architecture project in purely technical terms. Instead, describe projects in terms of the business problems they are intended to reduce or solve.

For example, whether a user-defined function or stored procedure must be written to solve the problem is almost never relevant, nor is the question of whether the procedure should be written in SQL or C. Only the business problem to be solved is important along with a non-technical explanation of how the problem is solved by the new application.

- Mistaking standards for architecture.

Without being embedded in an enterprise data architecture, standards are nothing more than a list of things that are (or are not) permitted. For example, your standard might state that a tool for a particular category of tasks must support x, y, and z. When embedded within an enterprise architecture, those standards would also be augmented by a set of usage guidelines that map specific categories of tools to specific types of problems that the architecture is designed to solve.

- Forgetting to assess people and process impacts.

Organizations frequently evaluate technology in a void, failing to consider the impact of new technologies on people and processes. Not only are ergonomic factors often overlooked, but even such basic business components as the staffing and administration of new technologies are often not evaluated.

With respect to processes, factors such as capacity planning, security management, user administration, and operations management are also frequently forgotten.

- Aligning with strategies rather than business goals and cultural values.

The problem with this approach is that business strategies not only change frequently, but often fail. If the enterprise needs to revise its strategy, it should not also have to revise its data architecture.

Instead, consider aligning your IT architecture with the business goals and cultural values of the corporation. For example, a commonly stated business goal is to provide mid-market products at the best price with the least possible inconvenience.

An example of aligning the corporate information architecture with this goal might be optimizing supply chain performance with low cost and maximally efficient transaction processing.

- Compelling unwilling constituents to participate in architectural decisions.

Rehkopf and Wybolt describe this behavior as acting like an uninvited party crasher.

Enterprise architects should seek out individuals within the organization who understand the value of a sound enterprise data architecture and are willing to participate in developing that architecture.

Avoid the temptation to coerce the participation of unwilling individuals, no matter how valuable you perceive their input to be.

- Introducing technology before its time.

Avoid introducing new technologies just because the industry perceives them to be the next big thing. Instead, focus on finding proven technologies that can address existing business problems or that can enable the delivery of new business value.

To facilitate this, Rehkopf and Wybolt identify 5 best practices for assessing the readiness of candidate technologies to support existing business problems and make it possible to bring new value into the enterprise:

- Create a technology assessment program.
- Establish pilot projects to evaluate and assess the readiness of new technologies.
- Devise training programs to enhance the skills of the staff members who will be using the new technologies.
- Design phased migration plans that parallel the evolution of updated staff member skills.
- Consider engaging in partner relationships with the technology vendors to help introduce new technologies.

## CHAPTER 4 Semantic Data Modeling

---

This chapter describes the basics of semantic, or logical, data modeling, focusing on entity-relationship analysis.

Teradata created several different industry-specific logical data model (LDM) frameworks, including LDMs for the following industries:

- Communications
- Financial Services
- Healthcare
- Insurance
- Manufacturing
- Media
- Retail
- Transportation and Logistics
- Travel and Hospitality Industry
- Utilities

Refer to <http://www.teradata.com/t/roadmaps-and-models> or contact your Teradata representative for details about the availability of these and other LDMs.

## The Entity-Relationship Model

### Definition of a Data Model

The term data model is used in two distinctly different ways in database management. Date (2004, p. 16) distinguishes between them as follows (emphasis in original):

- “A **data model** is an abstract, self-contained, logical definition of the objects, operators, and so forth, that together constitute the *abstract machine* with which users interact. The objects allow us to model the *structure* of data. The operators allow us to model its *behavior* ... In a nutshell: The model is what users have to know about; the implementation is what users do not have to know about ... A data model in the first sense is like a *programming language*—albeit one that is somewhat abstract—whose constructs can be used to solve a wide variety of specific problems, but in and of themselves have no direct connection with any such specific problem.”
- “A data model in the second sense is like a *specific program* written in that language. In other words, a data model in the second sense takes the facilities provided by some model in the first sense and applies them to some specific problem. It can be regarded as a *specific application* of some model in the first sense.”

When this manual uses the term *data model*, it is in the sense of the first definition provided by Date unless explicitly stated otherwise.

## The Relational Model for Database Management

After E.F. Codd introduced the relational model for database management in 1969, a number of alternative models immediately began to appear in the literature. The usual justification given for introducing these new models was a perceived failure of the relational model to capture sufficient data semantics. The failure of the original relational model to capture a satisfactory level of data semantics was acknowledged by Codd himself, who has extended the model several times in an attempt to better capture data semantics, most famously in his RM/T paper (Codd, 1979). This effort has even led to a second version of the relational model, which is presented in his book *The Relational Model for Database Management, Version 2* (Codd, 1990).

## The Entity-Relationship Model for Database Management

Peter Chen (Chen, 1976) introduced the Entity-Relationship (E-R) model as a data model that was intended to replace the relational model. The E-R model was a serious attempt to extend the semantics of the relational model to capture more meaning from the data stored in a database. There is a naive appeal to the E-R model because it is based on a far less abstract approach to the world than that used by the relational model. In the E-R approach, a real world object is captured as an *entity*. Each entity has certain unique characteristics that are captured as *attributes*. Just as real world objects interact with one another, so do entities in the E-R model, and they do this by way of *relationships*. Of all the alternatives to the relational model proposed since its introduction, the Entity-Relationship model has perhaps garnered the most support, albeit not in the manner its author had intended.

In large part, the failure of the E-R model to catch on as a serious data model derives from its informality: the very attribute that makes it so appealing in the first place. Because the E-R model is not derived from formal principles, it is art masquerading as science, greatly lacking in precision and predictability, both of which are the most appealing attributes of the relational model.

Of all the models proposed to replace the relational model in his lifetime, Codd himself found the E-R model to be the most objectionable. He writes, “The major problem in the entity-relationship approach is that one person’s entity is another person’s relationship. There is no general and precisely defined distinction between these two concepts, even when discussion is limited to a particular part of a business that is to be modeled by means of a database. If there are 10 people in a room and each is asked for definitions of the terms “entity” and “relationship,” 20 different definitions are likely to be supplied for each term” (Codd, 1990, p. 477).

The lack of precision in the definition of relationships in the Chen model is also noted by Kent (2000) and underscored by the ongoing research effort to generalize the concept and to provide a formal definition almost 30 years after the E-R model was originally described (see, for example, Dey, Storey, and Barron, 1999 and Wand, Storey, and Weber, 1999).

## Applications of the E-R Model

Because of its intuitive appeal and naïve usefulness, the E-R model has become the most widespread technique used for deriving an initial logical model for relational databases. Virtually every commercially available CASE tool in existence uses some form of the E-R approach as the basis for its design tools.

It is this use of the E-R model, as a data model in the second sense of Date (2004), that is presented here.

# Entities, Relationships, and Attributes

## Definition of an Entity

An entity is a database object that represents a thing in the real world. Entities are expressed as nouns.

Entities can be concrete, like buildings and employees or they can be more abstract things like departments and accounts.

Loosely speaking, an entity corresponds to a relation in relational theory. When a relation is made physical, it is normally referred to as a table, though the term is also used to describe physical as well as conceptual tables.

## Types of Entities

There are several different schemes for categorizing entities based on qualitatively different criteria. For purposes of this manual, the following two schemes are defined:

- Major and minor entities
- Supertype and subtype entities

The following table provides definitions for these types:

Entity Type	Definition	Example
Major	An entity with relatively large cardinality and degree that is updated frequently.	Order table
Minor	An entity with small cardinality and degree that is rarely updated. Minor entities are typically used in a single, 1:M association, and their primary key is often nonnumeric.	Nation Code table
Supertype	A generic entity that is a superclass of one or more subtype entities. Supertype and subtype entities model the same real world entity at a high level. Supertypes must, by definition, have one or more reciprocal subtypes.	Publications table

Entity Type	Definition	Example
Subtype	<p>A specific entity that is a disjoint subclass of one and only one supertype entity.</p> <p>Subtype and supertype entities model the same real world entity at a high level.</p> <p>Subtype entities typically have a higher degree than their supertypes, with the additional attributes describing detailed characteristics of the subtype that distinguish it from the other subtype entities of a mutual supertype.</p>	<ul style="list-style-type: none"> <li>• Book table</li> <li>• Magazine table</li> <li>• Professional journal table</li> <li>• Conference proceedings (all as subtypes of the supertype Publications)</li> </ul>

## Definition of a Relationship

A relationship is an association among two or more entities or other relationships.

Relationships are expressed as verbs.

Relationships among entities are described by one of three ratios:

Relationship	Shorthand Notation
One-to-one	1:1
One-to-many	1:M
Many-to-many	M:M

Relationships as defined in the E-R model have no direct counterpart in relational theory. The closest property of relational theory to expressing what a relationship is in E-R theory is the primary key-foreign key relationship.

[“Relationship Theory” on page 67](#), [“One-to-One Relationships” on page 68](#), [“One-to-Many Relationships” on page 69](#), and [“Many-to-Many Relationships” on page 70](#) describe the properties of relationships in greater detail.

## Definition of Attribute

An attribute is a characteristic of an entity. Every entity has at least one attribute: its primary key (More accurately, a *candidate* key). Attributes are expressed as nouns qualified by adjectives that clarify their role.

An attribute plays one of three possible roles in any table:

- Primary key attributes identify the entity or relationship modeled by a table (note the fuzzy boundaries among entities, relationships, and tables, which led Codd to his pointed criticism of the E-R approach).
 

Primary key attributes are said to be *identifier* attributes because they uniquely identify an instance of an entity.
- Foreign key attributes define relationships between and among entities or among entities and relationships.

A foreign key attribute can be an identifier attribute if it is part of a composite primary key; otherwise, foreign key attributes are descriptor attributes.

- Nonkey attributes further describe the entity or relationship modeled by a table.  
Nonkey attributes are said to be *descriptor* attributes because they specify a nonunique characteristic of an instance of an entity.

In the relational model, attributes have the same properties as they do in E-R theory.

## Definition of a Derivative Attribute

So-called derivative attributes violate the rules of normalization in relational theory because they are not atomic. A derivative attribute is any attribute that can be derived by calculation from other data in the model.

The issue of derivative attributes should not concern you during the logical design phase other than knowing that they should not be modeled. Derivative attributes are an important consideration for physical database design, where they are often modeled as a means for enhancing system performance.

Note that Teradata Database offers several features like hash and join indexes, aggregate join indexes, and global temporary tables that lessen the temptation to denormalize the physical design of your base tables by using derivative attributes.

# Translating Entities and Relationships Into Tables

## Definition of a Prime Table

A prime table is a table that has a single column primary key.

Prime tables always model entities and all entities are modeled by prime tables.

Ensure that all prime tables have been defined prior to defining any associative tables (see “[Definition of an Associative Table](#)” on page 66).

The following table, *Table\_A*, is prime because it has a simple primary key (see the definition for “Simple key” under “[Definitions](#)” on page 75):

Table_A
A_Key
PK
A1
A2
A3

## Definition of a Non-Prime Table

A non-prime table is a table that has a composite primary key.

The following table, *Table\_B*, is non-prime because it has a composite primary key (see the definition for “Composite key” under “[Definitions](#)” on page 75):

Table_B	
B_Key_1	B_Key_2
PK	
B1_1	B2_3
B1_2	B2_2
B1_3	B2_1

## Definition of an Associative Table

An associative table is a non-prime table whose primary key columns are all foreign keys.

Because associative tables model pure relationships rather than entities, the rows of an associative table do not represent entities. Instead, they describe the relationships among the entities the table represents.

Always define all your prime tables before defining any associative tables.

The following associative table, *table\_a-b*, associates prime table entity *table\_a* with prime table entity *table\_b*:

table_a-b	
a_Key	b_Key
PK	
FK	FK
A1	B1
A2	B5
A3	B2

## Guideline for Naming Associative Tables

Use the following general form to name associative tables.

`prime_table_name_A-prime_table_name_B`

This convention helps to keep the alphabetic and logical sequences of tables synchronized, making it easier to locate the prime tables for the foreign keys that make up its composite primary key.

# Relationship Theory

## Definition of Degree

The degree of a relationship is the number of entities associated in the relationship. Typical degree descriptions are provided in the following table:

For a relationship among this many entities ...	The following term is commonly used to describe the relationship ...
1	Unary
2	Binary
3	Tertiary
$n$	$n$ -ary

## Definition of Connectivity

Connectivity refers to the mapping of entity occurrences in a relationship. The possible values for the connectivity of a relationship are three:

- 0
- 1
- Many

## Definition of Cardinality

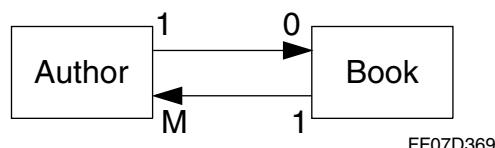
With respect to E-R theory, the term cardinality refers to the integer number represented by the symbol  $M$  in a 1:M or M:M relationship. This number describes the constraint on the number of entity instances that are related through the relationship.

In this context, cardinality does *not* refer to the number of tuples in a relation.

## Definition of Existence Dependency

Whenever the existence of an entity depends on the existence of another entity, that relationship is described as an existence dependency (ED).

For example, suppose there are two entities named Author and Book. A writer might have been provided with an advance to write a book for a publisher, but if this is the first book written by the writer for this publisher, there will be no entry in the Book entity until the contracted book has been written and published. In this case, the relationship between Author and Book is 1:0, indicating that the existence of an instance of Book is optional.



FF07D369

# One-to-One Relationships

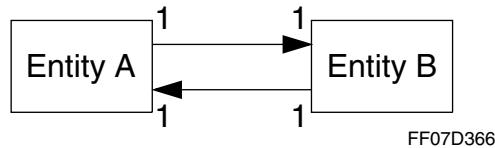
Assume two entities,  $A$  and  $B$ .

A  $1:1$  relationship exists between  $A$  and  $B$  when each occurrence of entity  $A$  is related to at most one occurrence of entity  $B$ , while each occurrence of entity  $B$  is related to at most one occurrence of entity  $A$ .

Somewhat more formally, Date defines  $1:1$  relationships as follows: “Let  $A$  and  $B$  be sets, not necessarily distinct. Then a one-to-one correspondence from  $A$  to  $B$  is a rule that pairs each element of  $A$  with exactly one element of  $B$  and each element of  $B$  with exactly one element of  $A$ . Equivalently, we might just say the one-to-one correspondence is that pairing itself” (2006, page 14).

$1:1$  relationships are not commonly seen in real world situations.

$1:1$  relationships are graphed as follows:



## Modeling 1:1 Relationships

$1:1$  relationships are modeled by placing the primary key of entity  $A$  as a foreign key component of entity  $B$  with no duplicates allowed. Because the relationship is symmetrical, you could just as well place the primary key of entity  $B$  as a foreign key component of entity  $A$ .

## Guideline for Placing the Foreign Key

Place the foreign key in whichever entity minimizes or eliminates the possibility of nulls.

## Example

Because of the symmetry of  $1:1$  relationships, the following entity pairs both model the same relationship:

A	B		A	B	B
A_Key	B_Key	A_Key	A_Key	B_Key	B_Key
PK	PK	FK, ND	PK	FK, ND	PK
A1	B1	A1	A1	B1	B1
A2	B2	A3	A2	B5	B2
A3	B3	A5	A3	B2	B3

# One-to-Many Relationships

Assume two entities, *A* and *B*.

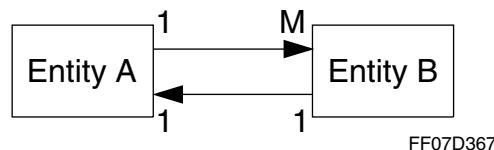
A  $1:M$  relationship exists between entity *A* and entity *B* when each occurrence of entity *A* is related to 0, 1, or more occurrences of entity *B*, while each occurrence of entity *B* is related to at most one occurrence of entity *A*.

Somewhat more formally, Date defines  $1:M$  relationships as follows: “Let *A* and *B* be sets, not necessarily distinct. Then a one-to-many correspondence from *A* to *B* is a rule that pairs each element of *A* with *at least* one element of *B* and each element of *B* with *exactly* one element of *A*. Equivalently, we might just say the one-to-many correspondence is that pairing itself.” (2006, page 16, emphases added).

Conversely, Date defines a  $M:1$  relationships as follows: “Let *A* and *B* be sets, not necessarily distinct. Then a many-to-one correspondence from *A* to *B* is a rule that pairs each element of *A* with *exactly* one element of *B* and each element of *B* with *at least* one element of *A*. Equivalently, we might just say the many-to-one correspondence is that pairing itself” (2006, page 15, emphases added).

$1:M$  relationships are commonly seen in real world situations. For example, each department has many employees, but each employee has only one department.

$1:M$  relationships are graphed as follows:



## Modeling 1:M Relationships

$1:M$  relationships are modeled by placing the primary key of entity *A* as a foreign key component of entity *B*.

### Example

Because  $1:M$  relationships are asymmetric, the entity *A* key must be placed in entity *B*. The reciprocal relationship does not model the same E-R relationship.

A	B
A_Key	B_Key
PK	FK
A1	B1
A2	B2
A3	B3

A	B
A_Key	B_Key
PK	FK
	A1
	A3

## Many-to-Many Relationships

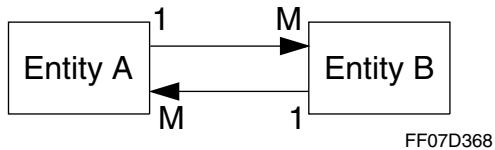
Assume two entities, *A* and *B*.

A  $M:M$  relationship exists between entity *A* and entity *B* when each occurrence of entity *A* is related 0, 1, or many occurrences of entity *B* and each occurrence of entity *B* is related to 0, 1, or many occurrences of entity *A*.

Somewhat more formally, Date defines  $M:M$  relationships as follows: “Let *A* and *B* be sets, not necessarily distinct. Then a many-to-many correspondence from *A* to *B* is a rule that pairs each element of *A* with at least one element of *B* and each element of *B* with at least one element of *A*. Equivalently, we might just say the many-to-many correspondence is that pairing itself.” (2006, page 17).

$M:M$  relationships are commonly seen in real world situations.

$M:M$  relationships are graphed as follows:



### Modeling M:M Relationships

$M:M$  relationships are modeled by placing the primary key of entity *A* and the primary key of entity *B* as foreign keys comprising the entire primary key of a separate entity referred to as an associative table. This is another example of the subjectivity of E-R theory, because a relationship is expressed as a physical table in the database.

### Guideline for Drawing M:M Tables for Users

Draw entities that represent  $M:M$  relationships in a way that data is presented in a form most familiar to users.

## Example

The following set of entities models the  $M:M$  relationship between  $A$  and  $B$ :

A	B	A-B	
A_Key	B_Key	A_Key	B_Key
PK	PK	PK	
A1	B1	FK	FK
A2	B2	A1	B1
A3	B3	A2	B5
		A3	B2

## Moving From an Entity-Relationship Analysis to Normalization

The goal of an E-R analysis is to generate a reasonable set of entities and to map the relationships among them.

The next step in the process of achieving a logical data model for your database is to translate those entities into a set of fully normalized tables. The principal task of this translation is to cluster attributes with entities, or relations, in such a way that they uniquely describe the primary keys of those relations. Attributes that do not describe the primary key of a relation must be migrated to a relation whose primary key they *do* describe uniquely.

The topics of normal forms and the normalization process are described in [Chapter 5: “The Normalization Process.”](#)



# CHAPTER 5 The Normalization Process

---

This chapter reviews some concepts of the normalization process. The assumption is made that any designer will use a CASE tool that has its own set of notational structures and conventions, so the presentation is as generic as possible.

Note that many of the dependencies described in this chapter can also be used to optimize SQL requests (see *SQL Request and Transaction Processing*).

## Why Teradata Encourages a Fully Normalized Schema

The full normalization approach Teradata advocates in its philosophy of database design originates in the massively parallel architecture of Teradata Database (see “[Born To Be Parallel](#)” on page 25, “[Data Placement to Support Parallel Processing](#)” on page 26, “[Intelligent Internodal Communication](#)” on page 29, “[Request Parallelism](#)” on page 32, and “[Synchronization of Parallel Operations](#)” on page 34 for background information on the massively parallel architecture of Teradata Database systems).

### Built-In Support for Fully-Normalized Databases

Teradata Database parallel technology is optimized to perform relational database management tasks in a normalized *physical* environment that other commercially available relational DBMSs cannot match operating with a *denormalized* physical schema. Among the special performance advantages built into Teradata Database are the following:

- Star join and other join optimizations (see *SQL Request and Transaction Processing*)
- Full-table scan optimization (see *SQL Request and Transaction Processing*)
- Specially designed index types to optimize parallel query operations (see [Chapter 8: “Teradata Database Indexes and Partitioning,”](#) [Chapter 9: “Primary Indexes and NoPI Objects,”](#) [Chapter 10: “Secondary Indexes,”](#) and [Chapter 11: “Join and Hash Indexes”](#))
- Row and column partitioning
- A full complement of SQL aggregate and ordered analytical functions (see *SQL Functions, Operators, Expressions, and Predicates*)
- A sophisticated parallel-aware SQL query optimizer (see *SQL Request and Transaction Processing*)

Teradata was founded at the time relational technology had initially emerged as the dominant force in database management, and the company has, from the outset, focused solely on delivering an enterprise-wide decision support system with high performance, direct-access, application-neutral, and extensible facilities that present a single version of the truth.

## Original Teradata Design Goals Strongly Coupled With Normalization

Two of the principal design goals for the original Teradata parallel architecture were:

- To overcome the performance barriers that existing decision support systems faced
- To eliminate redundant, inconsistent data banks and the costs, both tangible and intangible, of those redundancies and inconsistencies

Teradata has always promoted full database normalization. The reasons for this unwavering support for fully normalized databases include the following factors:

- Normalization is a provable, logical, and consistent method for designing provably correct database schemas. In their research that won the ACM PODS best paper award, Arenas and Libkin (2003) demonstrate the rigor of normalization by showing that its correctness can be proven not only with formal logic, but also by employing entropy measures derived from information theory.
- The Teradata parallel architecture was designed from the outset to support normalized databases.

Because of these factors, this chapter focuses on the principles of designing fully normalized databases. Significantly, other commercially available relational database engines cannot simultaneously achieve acceptable performance levels and support fully normalized physical database schemas.

Teradata Database architecture is optimized and parallelized so thoroughly that its performance is neutral with respect to the particulars of schema design (see Hurwitz Group, 1999), and the Optimizer attempts to generate an optimal query path irrespective of the physical schema of the database. For example, see *SQL Request and Transaction Processing* for a detailed explanation of the various star join optimizations available to the Optimizer. Of course, these optimizations cannot compensate for the update anomalies that exist in any database schema that is not fully normalized.

Furthermore, the appropriate application of views can effectively mask the underlying physical database schema from end users by presenting the look and feel of one or many very different virtual schemas (see “[Dimensional Views](#)” on page 186).

## The Key, the Whole Key, and Nothing But the Key

The normalization process helps you to structure your thinking about the entities you have identified and the relationships they share among one another. It heightens your awareness of the problems that can occur when all the attributes of a schema are not sorted out and stored in one and only one place.

### Normalization as a Logical Process

It is important to understand that database normalization is a logical process; therefore, the term denormalization is a misnomer when it is applied to the physical design of the database

rather than its logical derivation. The application of large scale denormalization to the physical database model is not necessary to support an enterprise data warehouse based on Teradata Database. Because most advocates of data warehousing begin with the assumption that a denormalized physical database design is a necessity, you might find this statement to be confusing. The reason for this assumption is that most commercially available database engines are not capable of delivering the level of performance required to support massive data warehouses based on normalized physical design schemas. Because of its innate parallel architecture, Teradata easily supports multiterabyte databases within a physical design that is directly derived from a fully normalized logical design (see “[Born To Be Parallel](#)” on page 25).

Because database management systems map logical relations directly to physical tables, it can sometimes appear difficult to separate the logical model from its physical realization. Nevertheless, you should always design a fully normalized logical model, then, only if necessary to achieve performance levels that are otherwise not possible to realize, denormalize the physical design.

Because of the widespread usage of the term, this manual often refers to denormalization in the context of physical database design. This is done with the understanding that the logical and physical models of a database are independent things, that the physical model applies *only* to implementation, and that the term denormalization *strictly* applies only to logical modeling. In spite of this, industry usage of the term nearly always applies to the physical design.

## The Cost of Normalization

There can be a cost to normalization when the logical model is implemented as a 1-to-1 physical mapping. In a very real sense, normalization optimizes update performance at the expense of retrieval performance. Because of this, it sometimes becomes necessary to “denormalize” physical tables to some extent in order to achieve reasonable overall system performance. This so-called denormalization is actually an implementation issue, not a logical design issue (see [Chapter 7: “Denormalizing the Physical Schema,”](#) for a description of “denormalization.”)

Before examining the details of the normalization process, a number of properties of relations and their manipulation need to be spelled out.

# Properties of Relations and Their Logical Manipulation

## Definitions

Term	Definition
Alternate key	Any candidate key not selected to be the primary key for a relation.

Term	Definition
Attribute	<p>A property of a relation that describes its primary key. Each attribute has a unique name, is drawn from a domain, and can be constrained in various ways.</p> <p>Because its values are all drawn from the same domain, the data for any attribute is homogeneous by definition.</p> <p>Attribute is the term used in set theory and logical design. Column is the equivalent term used in physical design and database management.</p>
Body	<p>The body of a relation is the composite value set assigned to its tuple variables.</p> <p>Each SQL relation <i>must</i> have a body (see “<a href="#">Types of Missing Values</a>” on <a href="#">page 673</a> about Table_Dee and Table_Dum for why this might not be an optimal situation).</p>
Candidate key	<p>An attribute set that uniquely identifies a tuple.</p> <p>A candidate key has the following minimal properties:</p> <ul style="list-style-type: none"> <li>• The value of the key uniquely identifies the tuple in which it appears.</li> <li>• Attributes defining the candidate key cannot be redundant. If an attribute is removed from the candidate key, then its uniqueness must be destroyed else it is not a properly defined candidate key. Compare with “Superkey.”</li> </ul> <p>In the completed physical design of a relational database, candidate keys are easy to identify because they are always constrained as UNIQUE NOT NULL.</p>
Composite key	<p>A key defined on more than one attribute. Compare with “Simple key.”</p>
Domain	<p>The set of all possible values that can be specified for a given attribute.</p> <p>The physical representation of a domain is a data type, and the ideal representation of a domain is a distinct user-defined data type (see <i>SQL Data Definition Language</i> and <i>SQL External Routine Programming</i> for more information about UDTs).</p>
Field	<p>The intersection of a tuple and an attribute.</p> <p>Columns in a table are often referred to as fields, but strictly speaking, that is incorrect.</p>
Foreign key	<p>An attribute set based on an identical attribute set that acts as a candidate key (typically the primary key) for a different relation.</p> <p>Foreign keys reflect relationships between tables and are often used as join columns.</p>
Heading	<p>Each attribute of a relation must have a heading. Each such attribute has two required parts: a name and a domain, or data type. It is common practice not to call out the domain of an attribute unless it is germane to the problem at hand, but that does not render the typing of each attribute in a relation any less necessary.</p>
Instance	<p>A tuple drawn from the complete set of tuples for a relation. The term is sometimes used to describe any selected set of tuples from a relation.</p>

Term	Definition
Intelligent key	<p>An overloaded simple key that encodes more than one fact (a combination of identification as well as characterization facts.), which is a violation of 1NF. The principal implementation problem with intelligent keys is that if any of the components of the key change, then all applications that access the key are affected. This is why you should always select unchanging attributes for your keys.</p> <p>The classic example of an intelligent key is the International Standard Book Number (ISBN) used by publishers to identify individual books. Each ISBN is composed of a group identifier, a publisher identifier, a title identifier, and a check digit.</p>
Key	<p>An attribute set that uniquely identifies each tuple in a relation. Implicitly synonymous with primary key, though it applies equally well to any candidate or foreign key. See “Primary key.”</p>
Natural key	<p>The representation of a real world tuple identifier in a relational database. For example, a common identifier of employees in a corporation is a unique employee number. An employee is assigned an employee number whether that information is stored within the database or not.</p> <p>Natural keys are sometimes confused with intelligent keys (see “Intelligent key”), but they are very different concepts.</p>
Order independence	<p>The <i>logical</i> ordering of tuples and attributes has no meaning. In other words:</p> <ul style="list-style-type: none"> <li>• Tuples have no essential up-down order.</li> <li>• Attributes have no essential right-left order.</li> </ul> <p>This is an area where the relational model is at odds with axiomatic set theory, because set theoretic relations are, by definition, ordered left-to-right. For example, Stoll (1961, p. 23) writes, “A (binary) relation is used in connection with pairs of objects considered in a definite order,” while Tarski (1995, p. 88) is more subtle, writing “Any thing having the relation <math>R</math> to some thing <math>y</math> we call a PREDECESSOR WITH RESPECT TO THE RELATION <math>R</math>; any thing <math>y</math> for which there is a thing <math>x</math> such that <math>x R y</math> is called a SUCCESSOR WITH RESPECT TO THE RELATION <math>R</math>.”</p> <p>When attributes are manifested physically as columns, their order <i>does</i> have significance for SQL.</p>

Term	Definition
Primary key	<p>The primary key for a relation is an attribute set that uniquely identifies each tuple in the relation. By the entity integrity rule, primary keys cannot contain nulls. See “<a href="#">Rules for Primary Keys</a>” on page 92 for an explanation of the entity integrity rule.</p> <p>Every relation must have one and only one primary key. More current thinking argues that every relation must have a <i>candidate</i> key, but that it need not necessarily be declared as the <i>primary</i> key for the relation.</p> <p>Note that you cannot use XML, BLOB or CLOB columns to define a physical key or other database constraint (see <a href="#">Chapter 12: “Designing for Database Integrity”</a>), nor can you use XML, BLOB, or CLOB columns to define the physical primary key for a global temporary trace table. See “CREATE GLOBAL TEMPORARY TRACE TABLE” in <i>SQL Data Definition Language Detailed Topics</i>.</p> <p>The primary key is often used as the primary index for a table when a relation is manifested physically (see <a href="#">Chapter 8: “Teradata Database Indexes and Partitioning”</a> and <a href="#">Chapter 9: “Primary Indexes and NoPI Objects”</a>).</p>
Relation	<p>A two-dimensional <i>representation</i> of data in tabular form. Note that database relations are <i>n</i>-dimensional, <i>not</i> 2-dimensional as is commonly asserted.</p> <p>Relation is the term used in set theory and logical design. Table is the analogous term used in physical design and database management.</p>
Relational schema	A set of relations in a logical relational model. In the physical model, a relational schema is manifested as a database.
Repeating group	A collection of logically related attributes that occur more than once in a tuple.
Simple key	A key defined on a single attribute. Compare with “Composite key.”
Superkey	Any set of attributes that uniquely identifies a tuple, whether redundantly or not. The allowance of redundant attributes within a super key distinguishes it from a simple candidate key (see <i>Candidate key</i> ).
Surrogate key	<p>An artificial simple key used to identify individual entities when there is no natural key or when the situation demands a noncomposite key, but no natural noncomposite key exists.</p> <p>Surrogate keys do not identify individual entities in a meaningful way: they are simply an arbitrary method to distinguish among them. You should only resort to surrogate keys if there is no other way to uniquely identify the rows of a table.</p> <p>Surrogate keys are typically arbitrary system-generated sequential integers. See “<a href="#">Identity Columns</a>” on page 818 and “CREATE TABLE” in <i>SQL Data Definition Language Detailed Topics</i>, for information about how to generate surrogate keys in Teradata.</p>
Tuple	<p>A unique instance of a relation consisting of, at minimum, a primary key and zero or more attributes that describe the primary key.</p> <p>Tuple is the term used in set theory and logical design. Row is the equivalent term used in physical design and database management. Nonrelational systems use the term record in the same way relational systems use row.</p>

Term	Definition
Uniqueness	<p>Duplicate rows are not permitted.</p> <p>When relations are manifested physically as tables in SQL databases, duplicate rows are permitted for multiset tables only. In fact, the ANSI/ISO SQL standard is bag (multiset)-oriented rather than set-oriented by definition. This is at odds with one of the fundamental properties of the relational model.</p> <p>The term <i>duplicate row</i> is used here as a row whose columnar values match every columnar value of one or more other rows in a relation.</p> <p>This is not a recommended practice: you should always either define your tables as set tables to avoid the many problems that multiset tables present or define them as multiset tables, but specify at least one of the columns to be UNIQUE NOT NULL.</p>

## Logical Operations on Relations

Relations are formally decomposed and constructed using logical relational operators and the relational algebra. The relational algebra is a set of procedural constructs for manipulating relations, while the relational calculus is a set of *nonprocedural* constructs for performing the same operations (SQL is a somewhat uneasy combination of both the algebra and the calculus). Codd, with his so-called reduction formula, sometimes referred to as Codd's theorem, proved the equivalence of the relational algebra and the relational calculus. Codd was apparently not aware of the distantly related work on relation algebras and relation calculi that Chin and Tarski had developed 20 years earlier, nor that Schröder had developed the first relation algebra nearly 80 years earlier! This does not detract from the later development by Codd of the relational algebra and calculus, because his work is a very distant relative of the various relation algebras and calculi developed in various fields of pure mathematics.

The first four operators in the following table derive directly from set theory, while the last four were developed by Codd specifically to suit the needs of relational databases.

Although algebras of relations have existed in one form or another since the time that set theory became a discipline of mathematics, it was only with the work of the great logician Alfred Tarski that relation algebra began to be studied seriously. Codd was also apparently not aware that several workers had begun similar work earlier in the same decade. Subsequent to the relational algebra developed by Codd, the relational algebra has continued to develop, particularly through the work of C.J. Date and his colleagues. These workers have extended the relational algebra to allow it to perform computations, among other things, a capability the operators introduced by Codd did not have.

### Set Theory Operators

Logical Operator	Description
DIFFERENCE	$X - Y$ The set of all attributes contained in relation X but not in relation Y.

Logical Operator	Description
INTERSECTION	$X \cap Y$ The set of all attributes contained in both relation <b>X</b> and relation <b>Y</b> .
PRODUCT	$X \cdot Y$ The set of all multiples of all attributes contained in relations <b>X</b> and <b>Y</b> .
UNION	$X \cup Y$ The set of all attributes contained in either relation <b>X</b> or relation <b>Y</b> or both.

### Special Relational Database Operators

Logical Operator	Description
DIVIDE	The division of relation <b>R</b> of degree $m + n$ by relation <b>S</b> of degree $n$ produces a quotient relation of degree $m$ .
JOIN	The join of relation <b>R</b> on attribute <b>X</b> with relation <b>S</b> on attribute <b>Y</b> is the set of all tuples $t$ such that the concatenation of a tuple $r$ , belonging to <b>R</b> , and a tuple $s$ , belonging to <b>S</b> , and the predicate $r.R \text{ equality\_operator } s.S$ evaluates to TRUE. In SQL, this is expressed in an ON clause in a DML join statement.
PROJECT	The projection of relation <b>R</b> on attributes <b>X</b> , <b>Y</b> , ..., <b>n</b> is the set of all tuples $(x, y, \dots, n)$ such that a tuple $n$ appears in <b>R</b> with <b>X</b> value $x$ , <b>Y</b> value $y$ , and so on. Less formally, a projection on relation <b>R</b> is any subset of the attributes of <b>R</b> . In SQL, this is expressed as a column list in a DML statement.
RESTRICT/ SELECT	The restriction of relation <b>R</b> is the set of all tuples $t$ such that the comparison $t.X \text{ operator } t.Y$ evaluates to TRUE. Less formally, a restriction (or selection) on relation <b>R</b> is any subset of the tuples of <b>R</b> satisfying the condition <b>X</b> <i>equality_operator</i> <b>Y</b> . In SQL, this is expressed in a WHERE clause in a DML statement.

## Functional, Transitive, and Multivalued Dependencies

### Definitions

Term	Definition
Determinant	For any relation <b>R</b> , the set of attributes <b>X</b> in the functional dependency $X \rightarrow Y$ is the determinant group for the dependency.
Full functional dependence	An attribute set <b>X</b> is fully functionally dependent on another attribute set <b>Y</b> if <b>X</b> is functionally dependent on the whole set of <b>Y</b> , but not on any subset of <b>Y</b> .

Term	Definition
Functional dependence	For any relation $R$ , attribute $X$ is functionally dependent on attribute $Y$ if for every valid instance, the value of $Y$ determines the value of $X$ . The symbolic notation for this is $Y \rightarrow X$ .
Multivalued dependence	For any relation $R$ with attribute set $(X,Y,Z)$ , the set $X$ multivalue determines the set $Y$ if, for every pair of tuples containing duplicates in $X$ , the instance also contains the pair of tuples obtained by interchanging the $Y$ tuples in the original pair. The symbolic notation for this is $X \Rightarrow Y$ .
Transitive dependence	Transitive dependencies are dependencies that exist among three or more attributes in a relation in such a way that one attribute determines a third by way of an intermediate attribute. For example, consider the relation $R(X,Y,Z)$ , where $X$ is the primary key. Attribute $Z$ is transitively dependent on attribute $X$ if attribute $Y$ satisfies the following dependencies, where the symbol $\overline{\rightarrow}$ indicates <i>does not determine</i> : <ul style="list-style-type: none"> <li>• <math>X \overline{\rightarrow} Y</math></li> <li>• <math>Y \overline{\rightarrow} Z</math></li> <li>• <math>Y \overline{\rightarrow} X</math></li> </ul>

## Inference Axioms for Functional Dependencies

The concept of functional dependencies for relations was introduced by Armstrong. Beeri, Fagin, and Howard produced a complete set of inference axioms for functional and multivalued dependencies for relations. Their axioms, which are sometimes referred to as the Armstrong axioms, are described in the following table:

Axiom	Formal Expression					
Reflexive rule	$X \rightarrow X$					
Augmentation rule	IF $X \rightarrow Y$ THEN $XZ \rightarrow Y$					
Union rule	IF $X \rightarrow Y$ AND $X \rightarrow Z$ THEN $X \rightarrow YZ$					
Decomposition rule	IF $X \rightarrow Y$ THEN $X \rightarrow Z$ WHERE $Z$ is a subset of $Y$					
Transitivity rule	IF $X \rightarrow Y$ AND $Y \rightarrow Z$ THEN $X \rightarrow Z$					
Pseudotransitivity rule	IF $X \rightarrow Y$ AND $YZ \rightarrow W$ THEN $XZ \rightarrow W$					

For the sake of some examples, assume the attribute set  $R, S, T, U, V$  with the following set of dependencies:

- $R \rightarrow S$       •  $V \rightarrow T$
- $TU \rightarrow R$       •  $SU \rightarrow T$
- $T \rightarrow V$

The following table lists many of the dependencies among these attributes implied by the inference axioms:

Dependency	Supporting Axiom
$R \rightarrow R$	Reflexive rule
$RT \rightarrow S$	Augmentation rule
$TU \rightarrow RV$	<ul style="list-style-type: none"> <li>Augmentation rule</li> <li>Union rule</li> </ul>
$RU \rightarrow T$	Pseudotransitivity rule
$UV \rightarrow R$	Pseudotransitivity rule
$TU \rightarrow S$	Transitivity rule
$SU \rightarrow V$	Transitivity rule
$VU \rightarrow R$	Pseudotransitivity rule
$RU \rightarrow V$	<ul style="list-style-type: none"> <li>Transitivity rule</li> <li>Pseudotransitivity rule</li> </ul>
$UV \rightarrow S$	<ul style="list-style-type: none"> <li>Transitivity rule</li> <li>Pseudotransitivity rule</li> </ul>

Bernstein developed an algorithm for synthesizing a minimum set of tables in 3NF using the Armstrong axioms, and Teorey provide a nice example of determining a minimum set of 3NF relations using the Bernstein algorithm.

## Inference Axioms for Multivalued Dependencies

The following table lists the complete set of multivalued dependencies derived by Beeri, Fagin, and Howard:

Axiom	Formal Expression
Reflexive rule	$X \Rightarrow X$
Augmentation rule	IF $X \Rightarrow Y$ THEN $XZ \Rightarrow Y$
Union rule	IF $X \Rightarrow Y$ AND $X \Rightarrow Z$ THEN $X \Rightarrow YZ$
Decomposition rule	IF $X \Rightarrow Y$ AND $X \Rightarrow Z$ THEN $X \Rightarrow Y \cap Z$ AND $X \Rightarrow (Z - Y)$
Transitivity rule	IF $X \Rightarrow Y$ AND $Y \Rightarrow Z$ THEN $X \Rightarrow (Z - Y)$
Pseudotransitivity rule	IF $X \Rightarrow Y$ AND $YW \Rightarrow Z$ THEN $XW \Rightarrow (Z - YW)$
Complement rule	IF $X \Rightarrow Y$ AND $Z \Rightarrow R - XY$ THEN $X \Rightarrow Z$

Axiom	Formal Expression				
Mixed inference rules	IF $X \rightarrow Y$ THEN $X \Rightarrow Y$ IF $X \Rightarrow Y$ AND $Z \Rightarrow W$ WHERE $W$ is contained in $Y$ AND $Y \cap Z$ is not empty				
	THEN $X \ W$				

## The Normal Forms

Normalization theory is constructed around the concept of a universe of normal forms that define a system of constraints. If the form of a relation meets the constraints of a particular normal form, it is said to be *in* that form.

Use the third normal form to model data for Teradata Database.

### The Objective of Normalization

The intent of normalizing a relational database can be reduced to one simple aphorism: One Fact In One Place. By decomposing your relations into fully normalized forms, you can eliminate the majority of update anomalies that can occur when data is stored in unnormalized tables. Decomposition is attained through a series of projections of a relational schema into a set of normalized relations that optimize the concision of attributes.

A slightly more detailed statement of this principle would be the definition of a relation (or table) in a normalized relational database: A relation consists of a primary key (more accurately, a *candidate key*.), which uniquely identifies any tuple, and zero or more additional attributes, each of which represents a single-valued (atomic) property of the entity type identified by the primary key (once again, it is more accurate to say *candidate key* in place of primary key).

**Note:** You cannot use XML, BLOB, or CLOB columns to define a key or other database constraint (see [Chapter 12: “Designing for Database Integrity”](#)).

### Types of Decomposition

Relations can be decomposed in one of two ways: horizontally or vertically.

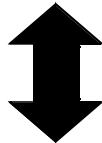
A horizontal decomposition is one in which a relation is partitioned along its cardinality dimension. In other words, entire tuples are divided into two or more tuple sets. Conceptually, this is pure relational restriction without projection, as indicated by the following graphic:

**Supplier\_Parts**

SuppNum	SuppName	PartNum	Quantity
1	Gozzo	P-001	251
1	Gozzo	P-002	719
17	Nagata	P-942	105
23	Albertine	P-063	10
84	Lê	P-346	622

**Supplier\_Parts\_1**

SuppNum	SuppName	PartNum	Quantity
1	Gozzo	P-001	251
1	Gozzo	P-002	719
17	Nagata	P-942	105



**Supplier\_Parts\_2**

SuppNum	SuppName	PartNum	Quantity
23	Albertine	P-063	10
84	Lê	P-346	622

1094A051

A vertical decomposition is one in which a relation is partitioned along its arity dimension. In other words, the attributes of a relation are decomposed into two or more sets of projections.

Conceptually, this is pure relational projection without restriction, as indicated by the following graphic:

### Supplier\_Parts

SuppNum	SuppName	PartNum	Quantity
1	Gozzo	P-001	251
1	Gozzo	P-002	719
17	Nagata	P-942	105
23	Albertine	P-063	10
84	Lê	P-346	622

### Supplier

SuppNum	SuppName
1	Gozzo
1	Gozzo
17	Nagata
23	Albertine
84	Lê

### Parts

SuppNum	PartNum	Quantity
1	P-001	251
1	P-002	719
17	P-942	105
23	P-063	10
84	P-346	622

1094A050

Notice that the vertical decomposition in this example is not a simple splitting of attributes between relvars (see “[Relations, Relation Values, and Relation Variables](#)” on page 627), but an actual normalization step (to BCNF), so the *SuppNum* attribute is duplicated, once as a PK in Supplier and again as an FK in Parts.

A more pure example of vertical composition is the projection of a subset of the columns of a large table into a single-table join index (see “[Single-Table Join Indexes](#)” on page 546).

For more information about decomposing relations, see “[Decomposing Relations](#)” on page 89.

## Third and Boyce-Codd Normal Forms

Third Normal Form (3NF) deals with the elimination of nonkey attributes that do not describe the primary key (more accurately, 3NF deals with the elimination of nonkey attributes that do not describe any *candidate* key for the relation).

### Definition of Third Normal Form

A relation variable  $R$  is said to be in Third Normal Form when it is in 2NF and one of the following statements is also true for every nontrivial functional dependency (assume the functional dependency  $X \rightarrow A$ ):

- The attribute set  $x$  is a superkey.
- Attribute  $A$  is part of some candidate key.

Another way of stating the rule is this: The relationship between any two nonkey attributes or attribute sets (excluding attributes having *no duplicates allowed* constraints) must not be one-to-one in either direction.

Using the phrase coined by Kent (1983), the relation is in 3NF when all its attributes depend on the key, the whole key, and nothing but the key. This condition is met when both of the following criteria have been met:

- Every nonkey attribute depends on all attributes of the primary key: the *entire* primary key.

Again, this is more accurately stated as the entire *candidate* key.

**Note:** You cannot use XML, BLOB, or CLOB columns to define a key or other database constraint (see [Chapter 12: “Designing for Database Integrity”](#)).

- No non-key attribute is functionally dependent on another non-key attribute of the relation.

The formal definition for Third Normal Form is as follows: For a relation to be in 3NF, the relationship between any two nonprimary key attributes (more accurately, between any two non-*candidate* key attributes) or groups of attributes in a relation must not be one-to-one in either direction. In other words, the nonkey attributes are nontransitively dependent upon each other and the key. Having no transitive dependencies in a relation implies no mutual dependencies.

Attributes are said to be mutually independent if none of them is functionally dependent on any combination of the others. This mutual independence ensures that individual attributes can be updated without any danger of affecting any other attribute in a tuple.

This is an incomplete definition for 3NF that fails to account for the case where functional or transitive dependencies occur. To account for this special case, Codd and Raymond Boyce completed 3NF with the definition described in [“Definition of Boyce-Codd Normal Form” on page 87](#). BCNF was first worked out and reported by Ian Heath, though he is rarely given credit for the discovery.

## Definition of Boyce-Codd Normal Form

When the relational model of database management was originally proposed, it only addressed what are now known to be the first three normal forms. Later theoretical work with the model showed that 3NF required further refinement to eliminate certain update anomalies.

The classic definition for third normal form does not handle situations in which a relation  $R$  has multiple composite candidate keys with overlapping attributes such as the following, where  $CK_1$ ,  $CK_2$ , and  $CK_3$  represent overlapping candidate keys on the overlapping attribute composites  $A_1-A_2$ ,  $A_2-A_3$ , and  $A_3-A_4$ , respectively:

R					
$A_1$	$A_2$	$A_3$	$A_4$	$A_5$	$A_6$
$CK_1$		$CK_3$			
	$CK_2$				

While this situation does not occur frequently, it does present itself from time to time, and once the problem was recognized, a solution had to be found.

To eliminate the deficiencies of the classic 3NF definition, Heath, Codd, and Boyce developed what has traditionally been called Boyce-Codd Normal Form (BCNF), which reduces to 3NF whenever the special situation that defines this problem does not apply.

A relation is in BCNF if and only if every determinant is a candidate key. This means that only determinants can be candidate keys.

Somewhat more formally, a relation is said to be in Boyce-Codd Normal Form when it is in 2NF and the following is true: if whenever  $X \rightarrow A$  and  $A$  does not belong to  $X$ , then  $X$  is a superkey.

## Relationship Between 3NF and BCNF

Zaniolo demonstrates that BCNF is strictly stronger than 3NF with an elegant proof. First, consider his definition of 3NF.

Let  $R$  be a relation variable, let  $X$  be any subset of the attributes of  $R$ , and let  $A$  be any single attribute of  $R$ .

$R$  is in 3NF iff for every functional dependency  $X \rightarrow A$  in  $R$ , at least one of the following assertions is true:

- $X$  contains  $A$ , making the functional dependency trivial.
- $X$  is a superkey.
- $A$  is contained in a candidate key of  $R$ .

If the third assertion is eliminated from consideration, the definition for BCNF follows, clearly indicating that BCNF is a stronger form than 3NF. Note that the third assertion constitutes the inadequacy of the original formulation of 3NF proposed by Codd.

## Meaning of Third Normal Form in This Manual

For purposes of this manual, use of the term 3NF generally covers BCNF as well. Relations in BCNF are often awkward to deal with, however, and are usually more tractable if decomposed into other relations in 4NF.

## Example: 3NF

For example, consider the following relations:

Customer	
CustNum	CustName
PK	
1	Wright
2	Adams

Order		
OrderNum	OrderDate	CustNum
PK		FK
1	2000/03/15	1
2	2000/03/17	2
3	2000/04/15	1

If these relations had been designed so that the *CustName* attribute was in the *Order* relation, 3NF would have been violated because there would be a one-to-one relationship between two nonkey attributes, *CustNum* and *CustName*.

## Example: Violating 3NF

Suppose the *Order* relation had been structured like this:

Order			
OrderNum	OrderDate	CustomerNum	CustomerName
PK		FK	
1	2000/01/15	1	Wright
2	2000/02/17	2	Adams
3	2000/02/01	1	Wright

The potential update anomalies associated with this violation of 3NF are the following:

- Changed customer name.  
To change the name of a customer, you must find every occurrence of its customer name.
- Inconsistent customer names.

Adams might be misspelled Addams in an occurrence, which would cause the tuple containing the misspelled customer name to be missed in a query having the WHERE predicate customername = Adams.

Worse still, customer number 2 might be labeled Adams in one tuple and Zoller in another.

- Inability to add new customer names unless they have an order placed.

## Example: BCNF

Now consider the following entity, which is in 3NF, but still has some problems:

Schedule

Campus	Class	Section	Date	Building/Room
PK				
Rancho Bernardo	Physical Database Design	1	2000/05/15	A/225
El Segundo	Relational Database Modeling Workshop	2	2000/08/15	ES/16-201
Dayton	Teradata Basics	3	2000/06/30	CTEC/120
Rancho Bernardo	Advanced SQL for Teradata Database	1	2000/06/30	A/225

Note that the buildings determine campus because no two buildings on any of the Teradata campuses have the same abbreviation. As a result, Building/Room → Campus. The relation is not in Boyce-Codd normal form.

Normalize the relation by decomposing it into the two following BCNF relations:

Campus-Class

Campus	Class	Section	Date
PK			
Rancho Bernardo	Physical Database Design	1	2000/05/15
El Segundo	Relational Database Modeling Workshop	2	2000/08/15
Dayton	Teradata Basics	3	2000/06/30
Rancho Bernardo	Advanced SQL for Teradata Database	1	2000/11/23

Building-Campus

Building/Room	Campus
PK	
A/225	Rancho Bernardo
ES/16-201	El Segundo
CTEC/120	Dayton

## Decomposing Relations

The principal goal of normalization is to eliminate update anomalies. By decomposing your relations into normalized forms, you can eliminate the vast majority of update anomalies.

Advocates of dimensional modeling argue against the method of decomposition because normalization tends to produce many relations. This is an implementation issue, not a logical

design issue, and it originates in the inability of most vendor database products to make reasonably high-performing joins.

When you design enterprise database schemas using the dimensional model (see “[Dimensional Modeling, Star, and Snowflake Schemas](#)” on page 187), you create only a few fact relations with a composite primary key that is not free of transitive dependencies (see “[Functional, Transitive, and Multivalued Dependencies](#)” on page 80), and smaller satellite relations called dimensions that contain detailed data about the fact relation.

An enterprise schema design using the dimensional model (see “[Dimensional Modeling, Star, and Snowflake Schemas](#)” on page 187) creates only a few fact relations with a composite primary key that is not free of transitive dependencies (see “[Functional, Transitive, and Multivalued Dependencies](#)” on page 80), and smaller satellite relations called dimensions that contain detailed data about the fact relation.

## Decomposing Relations for an Inherently Parallel Database Environment

Because the Teradata architecture is inherently parallel and optimized for join performance, it suffers no performance deficits when dealing with a physical schema that has been mapped directly from a fully normalized logical design.

See the following topics for details:

- [“Born To Be Parallel” on page 25](#)
- [“Data Placement to Support Parallel Processing” on page 26](#)
- [“Intelligent Internodal Communication” on page 29](#)
- [“Request Parallelism” on page 32](#)
- [“Synchronization of Parallel Operations” on page 34](#)

## Identifying Candidate Primary Keys

The primary key for a relation variable is an attribute set that uniquely identifies each tuple in that relation variable. This property is true for any alternate key, not just the candidate key that is selected to be the primary key for a relation.

**Note:** You cannot use XML, BLOB, CLOB, ARRAY, VARRAY, or Geospatial columns to define a key or other database constraint (see [Chapter 12: “Designing for Database Integrity”](#)).

Primary keys do *not* denote either of the following properties.

- Order  
Tuples within a relation are not ordered in any way.
- Access path

Keep in mind that normalization is a logical process, not a physical implementation of the database.

Primary keys are not used to access rows on disk, though they are a frequent choice as the primary index for tables (see [Chapter 8: “Teradata Database Indexes and Partitioning”](#) and

[Chapter 9: “Primary Indexes and NoPI Objects”](#)). The primary index does define a storage path and at least one access path for table rows.

## Procedure for Identifying Primary Keys

The selection of a primary key for a relation is the final step in a (possible) series of identifications of unique attribute sets as follows:

- 1 Identify any superkeys that exist in the attribute set.

A superkey is any set of (possibly redundant) attributes that uniquely identifies the tuples of a relation.

Every relation has at least one superkey and might have only one.

- 2 Eliminate any redundant, or otherwise unnecessary, attributes in the identified superkeys and produce a candidate key set.

A candidate key is a nonredundant attribute set that uniquely identifies the tuples of a relation. Note that it is possible for composite candidate keys to overlap.

By definition, every relation has at least one candidate key and often has *only* one candidate key.

- 3 Select the primary key from the set of identified candidate keys.

Selection of the primary key from a set of candidate keys is ultimately an arbitrary decision, but when there are multiple candidate keys to choose from, a good rule of thumb is to select the one having the fewest attributes, particularly if you plan to use the primary key as the primary index for the table.

See “[Performance Considerations for Primary Indexes](#)” on page 441 for specific information on selecting primary indexes to optimize retrieval and hashing performance.

The unselected candidate keys, if any, are referred to as alternate keys. Assign a UNIQUE constraint to any alternate key not selected to be the primary key for a relation to ensure its integrity with respect to the *referential integrity rule* (see “[The Referential Integrity Rule](#)” on page 95).

Every relation has one and only one primary key.

## Surrogate Keys

Situations occur where the identification and choice of a simple primary key is difficult, if not impossible. There might be no single column that uniquely identifies the tuples of a relation variable or, looking ahead to physical design, there might be performance or query condition considerations that argue against using a composite key. In these situations, and only in these situations, surrogate keys are an ideal solution.

As previously defined (see “Surrogate key” under “[Definitions](#)” on page 75), a surrogate key is an artificial, simple key used to identify individual entities when there is no natural key or when the situation demands a simple key, but no natural simple key exists.

Surrogate keys do not identify individual entities in a meaningful way: they are simply an arbitrary method to distinguish among those entities (see Hall, Owlett, and Todd, 1976). As a

result, it is often difficult to maintain referential integrity relationships among tables with surrogate keys.

Surrogate keys are typically arbitrary system-generated sequential integers. See “[Identity Columns](#)” on page 818 and “[CREATE TABLE \(Column Definition Clause\)](#)” in *SQL Data Definition Language Detailed Topics*, for information about how to generate surrogate keys in Teradata Database.

## Rules for Primary Keys

There are several rules that define the bounds of primary keys. Note that these rules actually apply to all *candidate* keys for a relation, not just to its primary key.

The first and second rules in the following list are absolute, and the third is strongly advised:

- Primary key attributes cannot be null.

This principle is known as the *entity integrity rule*, and it is one of the fundamental principles of relational database theory. See “[The Referential Integrity Rule](#)” on page 95 for a definition of the other fundamental integrity rule.

By definition, nulls are not unique because they represent missing values that cannot be distinguished from one another.

Although the database relational model does not explicitly state that alternate keys cannot be null, the constraint is implicit because a column set cannot be a potential primary key (a candidate key) if it is null or contains nulls.

- Primary and alternate key attributes cannot contain duplicate values.

By definition, a primary key is a unique identifier. If it contains duplicate values, it cannot be unique (this means that multiset tables cannot have true primary keys).

- Primary and alternate key values should never be modified.

This rule is neither part of the relational model, nor is it absolute, because there are occasions when primary key updates *must* be made.

Whenever a primary key is updated, it is possible, and even probable, that a series of coordinated foreign key updates must also be made to maintain the consistency of the database. However, it is equally likely that those cascaded updates will never be performed, leaving the database in a state that does not reflect reality *even if all the system integrity constraints have been satisfied*, which is the principal reason primary key updates are discouraged so strongly.

- Primary and alternate keys cannot be defined on columns defined with the BLOB or CLOB data types.

## Guidelines for Selecting Primary Keys

This topic presents several recommendations for selecting the attributes that make up the primary key for a relation variable. These guidelines apply, the necessary changes being made, to all candidate keys:

- Select numeric attributes as primary keys.

Numeric keys are both easier to produce uniquely and easier to manage than character data.

Always assign numeric attributes as the primary key if the key is to be system-generated.

- Whenever possible, use system-assigned keys (see “Surrogate key” under [“Definitions” on page 75](#)) to simplify the maintenance of uniqueness on the primary key attribute set. This is another factor that argues in favor of numeric keys.
- Select primary key attributes that can remain unique for the life of a relation variable.
- Construct primary keys on attributes that rarely, if ever, change during the life of a relation variable.
- Never use intelligent keys.

An intelligent key carries semantics about the tuple it identifies.

The expression is also used to describe a key constructed around one or more misprojected attributes that would not be in the relation variable were the database properly normalized.

Note that intelligent keys and natural keys are *not* the same thing. See “Intelligent key” and “Natural key” under [“Definitions” on page 75](#) for details.

- Whenever relation variables are paired in a supertype-subtype relationship, always assign the same primary key to both.
- Use a consistent convention when naming primary key attributes (see [“Guidelines for Naming Columns” on page 134](#)).

Because it is important to track the originating entity for attributes, always identify foreign keys with a unique code.

For example, if you use the column naming convention described in [“Guidelines for Naming Columns” on page 134](#), you might extend the convention as indicated in the following example:

Suppose the relation *Lineitem* has the following primary key composed of the three attributes indicated by preliminary attribute names:

line_item		
line_item_number	order_number	order_year_date
PK		
	FK	FK

You might consider naming the primary key attributes as follows.

- *line\_item\_number*
- *line\_item\_order\_FK\_number*
- *line\_item\_order\_FK\_year\_date*

The FK embedded in *order\_number* and *order\_year\_date* indicate that the attributes are foreign keys that reference the *order* relation variable.

When attribute naming has been completed, the primary key attribute names should look like the following relation definition fragment:

line_item		
line_item_number	order_FK_number	order_FK_year_date
PK		
	FK	FK

## Foreign Keys

A foreign key is an attribute set in one relation based on an identical attribute set that acts as a primary or alternate key in a different relation.

Foreign keys are a special form of [Inclusion Dependencies](#).

Foreign keys provide a mechanism for performing primary index joins, sometimes referred to as prime key joins.

Foreign keys are also used to maintain referential integrity among related tables in a database (see [“The Referential Integrity Rule” on page 95](#)).

### Rules

- Foreign key values are restricted to three types.
  - A mirror image, drawn from the same domain, of a primary or alternate key in an associated relation.
  - Wholly null
  - Partially null

The partially null case applies to compound foreign keys only, where one or more columns of the key might be null while others might contain references to primary keys in other tables, which are, by definition, non-null.

Because of the myriad problems nulls present in database management (see [Chapter 13: “Designing for Missing Information”](#)), you should avoid creating foreign keys that are either wholly or partially null.

Admitting nulls into the relational model is arguably the only serious error Codd ever made in his development of the theory. In fact, Codd did not introduce nulls into the theory until a full 10 years after he had initially proposed its original version.

Chris Date and his colleagues Hugh Darwen, David McGoveran, and Fabian Pascal have argued forcefully for the elimination of nulls from the relational model based not only on the violation of the principles of the first order predicate logic and set theory they present, but also on the myriad practical difficulties that nulls present to practitioners and to the

integrity of any database that permits them. See [Chapter 13: “Designing for Missing Information”](#) for a brief summary of their objections to nulls.

- You cannot use BLOB or CLOB columns to define a physical foreign key or other database constraint (see [Chapter 12: “Designing for Database Integrity”](#)).
- You cannot define a foreign key for a global temporary trace table. See “CREATE GLOBAL TEMPORARY TRACE TABLE” in *SQL Data Definition Language Detailed Topics*.

## The Referential Integrity Rule

If a base relation **R** has a foreign key FK that matches the primary key PK of base relation **S**, then every value of FK in **R** must have one of the following properties:

- Equal to the value PK of some tuple in **S**.
- Wholly null (each value in the attribute set that defines FK is null).

This definition presents an obvious contradiction and is not supported by relational theory. By definition, a candidate key (and the FK in a referential relationship must be a candidate key for its relation) must be a *unique* identifier for its relation, yet a key that is wholly null contains *no* values, only markers for values that are missing; therefore, it *cannot* be unique.

If your enterprise business rules prohibit nulls in foreign key columns, you can enforce a non-null constraint on a column by defining the column with a NOT NULL attribute using either CREATE TABLE or ALTER TABLE.

In other words, a row cannot exist in a table with a (non-null) value for a referencing column if no equal value exists in its referenced column.

This relationship is referred to as referential integrity and the act of ensuring that this rule is enforced is referred to as maintaining referential integrity.

This principle is known as the *referential integrity rule*, and it is one of the fundamental principles of relational database theory. Besides ensuring database integrity, referential integrity constraints have the added benefit that they are frequently used to optimize join plans when implemented in your physical database design.

The referential integrity rule applies equally, the necessary changes being made, to all candidate keys, not just the key selected to be the primary key for a relation.

### Purpose

The intent of the referential integrity rule is to ensure that if some tuple *r* in relation variable **R** references some tuple *s* in relation variable **S**, then tuple *s* must exist.

This prevents you from corrupting the integrity of the database by deleting data that is still being used by another relation.

## Entity Integrity and Foreign Key Nulls

Permitting a foreign key to be wholly null seems to contradict the entity integrity rule (see “[Rules for Primary Keys](#)” on page 92). This property is, to be sure, somewhat ad hoc, but it can have some practical usefulness. For example, consider the following case: an employee is working for some enterprise but has not yet been assigned to a department. Assuming that *Department Number* is a FK in the *Employee* relation, you can readily see that even when all the remaining information required to create an *Employee* relation tuple for the employee in question is available, you cannot create such a tuple in the *Employee* relation unless you also allow the FK to be null.

The tuple in question should be updated with the appropriate *Department Number* data just as soon as it is available, but allowing the null FK permits the enterprise to pay this employee even though she is not yet assigned to a department.

Alternatively, the problem can be avoided entirely with a minor change in the logical design of the database. See “[Redesigning the Database to Eliminate the Need for Nulls](#)” on page 680 for an example.

As another alternative, you can assign default values to the FK field set and then update with the appropriate *Department Number* as soon as you know what it is. SQL facilities for assigning default values to columns are described in *SQL Data Types and Literals*.

## Enforcing the Referential Integrity Rule

The relational model states the Referential Integrity rule without providing a mechanism to enforce it. In practice, the rule is enforced through the PRIMARY KEY and FOREIGN KEY clauses in the CREATE TABLE statement. The keyword REFERENCES describes the table and column set in which the primary key image of the foreign key resides.

The REFERENCES WITH NO CHECK OPTION specification explicitly instructs Teradata Database *not* to enforce referential integrity on the specified relationship. See “[Foreign Key Constraints](#)” on page 644 for further information.

When referential integrity is enforced, you cannot delete a row from the referenced (parent) table as long as there is a row in the referencing table whose foreign key matches it.

## Enforcing Referential Integrity in Teradata

To implement referential integrity (RI) in Teradata, you have three choices, ranked in their order of preference:

- 1 Use the declarative referential integrity constraint checks supplied and enforced by the database software.
- 2 Write your own, site-specific macros, triggers, or stored procedures to enforce RI.
- 3 Enforce constraints through application code.

See “CREATE TABLE” in *SQL Data Definition Language Detailed Topics* for a description of the problems inherent in attempting to enforce database integrity using nondeclarative constraints.

## Definition of a Referencing (Child) Table

The referencing table is referred to as the Child table, and the specified Child table columns are called referencing columns.

Referencing columns should be of the same number and have the same data type as the referenced table key.

## Definition of a Referenced (Parent) Table

A child must have a parent, and the referenced table in a PK-FK relationship is referred to as the Parent table. The key columns in the Parent table that see a Child table are called referenced columns.

Since the referenced columns are defined as unique constraints, the referenced column set must be one of the following:

- A unique primary index (UPI), not null
- A unique secondary index (USI), not null

## Definition of a Primary Key

With respect to RI, a primary key (more accurately, a *candidate* key) is a parent table column set that is referred to by a foreign key column set in a child table.

## Definition of a Foreign Key

With respect to RI, a foreign key is a child table column set that refers to a primary key column set in a parent table.

## Why Referential Integrity Is Important

Referential integrity is a mechanism to keep you from corrupting your database. Suppose you have a table like the following:

order\_part

order_num	part_num	quantity
PK		NN
FK	FK	
1	1	110
1	2	275
2	1	152

Part number and order number, each a foreign key in this relation, also form the composite primary key.

Suppose you were to go to the *part\_number* table and delete the row defined by the primary key value 1. The keys for the first and third rows in the *order\_part* table are now parentless because there is no row in the *part\_number* table with a primary key of 1 to support them. Such a situation exhibits a loss of referential integrity.

Now, suppose you had a mechanism to prevent this from happening? If you attempt to delete the row with a primary key value of 1 from the *part\_number* table, the database management system does not allow you to do so. This is the way Teradata Database maintains referential integrity.

Besides data integrity and data consistency, referential integrity has the following benefits:

Benefit	Description
Increases development productivity	It is not necessary to code SQL statements to enforce referential integrity constraints because Teradata automatically enforces Referential Integrity by means of declarative RI constraints.
Requires fewer programs to be written	All update activities are programmed to ensure that established declarative referential integrity constraints are not violated because Teradata enforces Referential Integrity in all environments: no additional programs are required.

## Referential Integrity Constraints

The combination of the foreign key, the parent key (more accurately, a *candidate* key), and the relationship between them defined by the referential integrity rule is called the referential integrity constraint.

This is a constraint defined on a table column set (using the CREATE TABLE or ALTER TABLE SQL statements) that represents a referential integrity link between two tables.

A referential integrity constraint is always defined for a foreign key column set in the child table in a relationship.

Note that you cannot use UDT, Period, Geospatial, BLOB, CLOB, or XML columns to define a referential integrity relationship or other database constraint (see [Chapter 12: “Designing for Database Integrity”](#)).

See [Chapter 12: “Designing for Database Integrity”](#) for additional information about how referential integrity is essential to maintaining the integrity of databases.

Teradata Database provides two other features related to referential integrity constraints: batch referential integrity constraints and referential constraints. The basic differences among the different referential constraint types are summarized in the following table.

Referential Constraint Type	CREATE TABLE Syntax	Does It Enforce Referential Integrity?	Level of Referential Integrity Enforcement
<ul style="list-style-type: none"> <li>• Referential Constraint</li> <li>• Temporal Relationship Constraint</li> </ul> <p>For more information, see <i>ANSI Temporal Table Support</i> and <i>Temporal Table Support</i>.</p>	REFERENCES WITH NO CHECK OPTION	No	None
Batch referential integrity constraint	REFERENCES WITH CHECK OPTION	Yes	All child table rows must match a parent table row
Referential integrity constraint	REFERENCES	Yes	Row

Referential constraints and temporal relationship constraints do not enforce the referential integrity of the database. Instead, they signal the Optimizer that certain referential relationships are in effect between tables, thus providing a means for producing better query plans without incurring the overhead of system enforcement of the specified RI constraints.

You should specify referential constraints and temporal relationship constraints only when you enforce the integrity of the referential relationship in some other way, or if the possibility of query errors and the potential for data corruption is not critical to your application. See “[Foreign Key Constraints](#)” on page 644 and “[CREATE TABLE Column Definition Clause](#)” in *SQL Data Definition Language* for more information.

Batch referential integrity constraints are externally identical to regular referential integrity constraints. Because they enforce referential integrity in an all-or-none manner for an entire implicit transaction, they can be more high-performing than standard referential integrity constraints. In most circumstances, there is no semantic difference between the two, only an implementation difference. Batch referential integrity becomes important when a single statement inserts many rows into a table, like the massive inserts that can occur with `INSERT ... SELECT` requests.

## Example: Referential Integrity Constraints

The following CREATE TABLE statement defines the following referential integrity constraints on *table\_A*.

- A column-level constraint on the foreign key column, *column\_A1*, and the parent table (*table\_B*) key column, *column\_B1*.
- A table-level constraint on the composite foreign key column set (*column\_A1*, *column\_A2*) and the parent table *table\_C*.

```
CREATE TABLE table_A (
    column_A1 CHARACTER(10) REFERENCES table_B (column_B1),
    column_A2 INTEGER
    PRIMARY KEY (column_A1),
    FOREIGN KEY (column_A1, column_A2) REFERENCES table_C);
```

According to this definition, the single-column primary key *column\_A1* must reference the primary key column *column\_B1* of the parent table *table\_B*. *Table\_B* must have a primary key, of which one column is *column\_B1*, having the following definition.

CHARACTER(10) NOT NULL

The composite foreign key (*column\_A1*, *column\_A2*) must reference the primary key of the parent table *table\_C*. *Table\_C* must have a two-column primary key, the first column of which has the following definition.

CHARACTER(10) NOT NULL

The second column of the *Table\_C* primary key must have the following definition.

INTEGER NOT NULL

## Rules for Referential Integrity Constraints

Referential integrity constraints must obey the following set of rules:

- The parent key must exist when the referential integrity constraint is defined.
- The parent key columns must be either a unique primary index (UPI) or a unique secondary index (USI).
- The foreign and parent keys must have the same number of columns and their data types must match.
- The foreign and parent keys cannot exceed 64 columns.
- Duplicate referential integrity constraints are not allowed.
- The columns on which a referential integrity relationship is defined cannot be defined with the XML, BLOB, CLOB, ARRAY, VARRAY, or Geospatial data types.
- You cannot drop or alter foreign key or parent key columns using an ALTER TABLE statement after a referential integrity constraint has been defined on them.

To drop a foreign or parent key column after a referential integrity constraint has been defined on it, you must first drop the referential constraint and then alter the table to drop the foreign or parent key columns.

- A foreign key value must be equal to its parent key value or it must be null.
- Self-reference (a condition in which the Parent and Child tables are the same table) is allowed, but the foreign and parent keys cannot consist of identical columns.
- You can define no more than 64 referential integrity constraints per table.
- You cannot define a referential integrity relationship on a global temporary trace table. See “CREATE GLOBAL TEMPORARY TRACE TABLE” in *SQL Data Definition Language Detailed Topics*.

## Domains and Referential Integrity

This topic describes various domain rules for primary (more accurately *candidate*) and foreign keys.

## Definition of a Domain

A domain is a data type defined with explicit constraints. A data type is a well-defined set of values. For example, the data type INTEGER represents all the possible integer numbers, while an INTEGER column defined with an explicit CHECK constraint condition such as “count BETWEEN 25 AND 50” defines a domain on the set of numbers having the INTEGER data type. More unambiguous data types are usually application-specific and you can define them using the various facilities available to you for defining user-defined types and their associated constructs such as methods, orderings, and so on. See *SQL Data Definition Language* and *SQL External Routine Programming* for more details about user-defined data types.

## Table Definition for Examples

An Employee table is used for the examples described in this topic. The definition of the Employee table is given by the following table:

employee		
employee_number	last_name	supervisor_employee_number
PK, SA		FK
1	Smith	null
2	Brown	1
3	White	1
4	Gibson	2
5	Black	2
6	Jones	3
7	Mason	3

The following domain rule applies to the *employee\_number* column: Its data type is INTEGER and any value inserted into the column must have a value greater than zero.

## Domain Rules for Primary Keys

The following rules apply to primary key domains and referential integrity. The domain rules for primary keys apply equally, the necessary changes being made, to all candidate keys:

- New primary key values must be drawn from the domain set of all valid values for the column on which the primary key value is defined.
- You cannot define a primary key using any column defined with an XML, BLOB, or CLOB data type.

- You cannot delete a primary key that references foreign keys because that violates the referential integrity rule.

The rules for deleting a primary key that has foreign key references are as follows:

Rule Name	Rule Description
Prevent	<ul style="list-style-type: none"><li>• Do not delete a PK if it references an existing FK.</li><li>• Do not change the value of PK if it references an existing FK.</li></ul>
Reassign	<ul style="list-style-type: none"><li>• Change the value of FK to a different PK value before deleting the old PK row.</li><li>• Change the value of FK to a different PK value before changing the old PK value.</li></ul>
Nullify	<ul style="list-style-type: none"><li>• Change the FK value to NULL before deleting the old PK row.</li><li>• Change the FK value to NULL before changing the old PK value.</li></ul>
Cascade	<ul style="list-style-type: none"><li>• Delete the FK row before deleting the old PK row.</li><li>• Delete the FK row before changing the old PK value.</li></ul>

Each of these rules preserves the referential integrity of a PK-FK relationship.

## Questions and Answers for Primary Key Domains

The following questions and answers about primary keys indicate some examples of how domain integrity is enforced. The questions refer to the table defined in “[Table Definition for Examples](#)” on page 101.

Question	Answer	Explanation
Is it valid to add Employee 12345?	Yes	12345 is a positive integer.
Is it valid to delete the row for Employee 5?	Yes	Employee 5 is not referenced by any other row.
Is it valid to add Employee -23.67?	No	-23.67 is a negative decimal number.
Is it valid to delete the row for Employee 2?	Yes	Cascade rule.

## Domain Rule for Foreign Keys

Foreign key values can be added only when they are drawn from the set of existing primary key values.

## Questions and Answers for Foreign Key Domains

The following questions and answers about foreign keys indicate some examples of how domain integrity is enforced. The questions refer to the table defined in “[Table Definition for Examples](#)” on page 101.

Question	Answer	Explanation
Is it valid to have Employee 6 report to Employee 4?	Yes	Employee 4 is drawn from the set of existing employee numbers.
Is it valid to delete the row for Employee 7?	No	Employee 7 has an assigned foreign key value.
Is it valid to have Employee 6 report to Employee 8?	No	Employee 8 is not drawn from the set of existing employee numbers.

## Normalization and Database Design Problems

Although normalization is the only component of database design based in provably correct mathematics, it also retains some of the more subjective elements common to other aspects of database design. For example, it is usually possible to normalize a database schema in multiple ways. Normalization theory does not guarantee, or even suggest, that it produces a canonical set of relations for any given database. Instead, it guarantees that a fully normalized database schema is free of a number of problems that otherwise cannot be excluded, the most widely known of these problems being update anomalies.

This topic, which is slightly modified from the presentation given by Date (1998), describes some of the problems commonly encountered in database design that cannot be solved merely by fully normalizing the database schema.

### Preserving Functional Dependencies

It is often true that a relation can be nonloss-decomposed in more than one way; however, some of those ways are better than others. For example, consider the following relvar:

employee		
emp_num	dept_num	budget
PK		

This relvar has the following FDs:

- $\text{emp\_no} \rightarrow \text{dept\_no}$
- $\text{dept\_no} \rightarrow \text{budget}$
- $\text{emp\_no} \gg \text{budget}$   
where  $\gg$  indicates the dependency is transitive.

A relvar like *employee* suffers from certain well known update anomalies that can be avoided by nonloss-decomposition into various projections. Relvar *employee* can be nonloss-decomposed into two valid 5NF projections.

For example, consider the following set of 5NF projections:

emp_dept	
emp_num	dept_num
PK	FK

dept_budget	
dept_num	budget
PK	

Call this decomposition *A*. The second valid 5NF decomposition of *employee* is as follows:

emp_dept	
emp_num	dept_num
PK	

emp_budget	
emp_num	budget
PK	

Call this decomposition *B*.

Note that the projection *emp\_dept* is the same for both decompositions *A* and *B*.

Even though both decompositions are valid, decomposition *A* is better than decomposition *B*. For example, it is not possible to represent the fact that a given department has a budget unless that department also has at least one employee in decomposition *B*.

You can make the determination that decomposition *A* is better than decomposition *B* solely on an analysis of the FDs in the original *employee* relvar. Note that the projections in decomposition *A* correspond to the nontransitive FDs in *employee* (indicated by  $\rightarrow$ ). As a result of this, you can update either projection without regard for the other as long as the referential constraint from *emp\_dept* to *dept\_budget* is satisfied. Provided only that the update in question is legal within the context of the given projection, which means only that it must not violate the primary key constraint for that projection, the join of the two projections after the update is still be a valid value for the *employee* relvar. In other words, the join cannot possibly violate the FD constraints on *employee*.

In decomposition *B*, by contrast, one of the two projections corresponds to the transitive dependency in *employee* (indicated by  $\gg$ ). As a result, updates to either projection must at least be monitored to ensure that the FD  $dept\_no \rightarrow budget$  is not violated (in other words, if two employees have the same department, they must also have the same budget - consider, for example, the machinations required in decomposition *B* to move an employee from department D1 to department D2.)

The problem with decomposition *B* is that the FD  $dept\_no \rightarrow budget$  spans two relvars. On the other hand, while it is true that the transitive FD  $dept\_no \gg budget$  in Decomposition *A*

spans relvars, it is also true that the FD is enforced by default as long as the FDs  $emp\_no \rightarrow dept\_no$  and  $dept\_no \rightarrow budget$ , which do not span relvars, are enforced, which only requires enforcement of the corresponding primary key uniqueness constraints.

The point being made is that you should decompose relvars in such a way that functional dependencies are preserved.

## Full Normalization and Dependency Preservation

Sometimes the goals of decomposing to full normalization and decomposing them in a way that preserves functional dependencies can conflict. Consider the following example relvar:

*student\_subject\_teacher*

student	subject	teacher
CK		
	CK	
Smith	Logic	Alonzo Church
Smith	Physics	Robert Millikan
Jones	Logic	Alfred Tarski
Jones	Physics	Max Planck

The predicate for the *student\_subject\_teacher* relvar is as follows: the student named *student* is taught the subject named *subject* by the teacher named *teacher*.

Assume the following restrictions for purposes of illustration:

- For each subject, each student of that subject is taught by only one teacher.
- Each teacher teaches only one subject, but each subject is taught by several teachers.

What are the FDs for the *student\_subject\_teacher* relvar?

- The first bullet implies the following FD:  
*student, subject — teacher*
- The second bullet implies the following FD:  
*teacher — subject*
- The second bullet also implies that the following FD is *not true*:  
*subject — teacher*

For the sake of illustration, this set of dependencies is portrayed symbolically as follows:  
*student — teacher — subject*.

As the representation of relvar *student\_subject\_teacher* indicates, it has two candidate keys, the composite column sets *student-subject* and *student-teacher*.

Both CKs have the property that all columns of *student\_subject\_teacher* are functionally dependent on them, and that property no longer holds if any column is discarded from either column set.

Note, too, that *student\_subject\_teacher* also satisfies the FD *teacher—subject*, which is not implied by either CK. This absence of implication is indicative that the *student\_subject\_teacher* relvar is not in BCNF. As a result, *student\_subject\_teacher* suffers from certain update anomalies. Suppose, for example, that you want to delete the information that Jones is studying physics. You cannot do this without also losing the information that Max Planck teaches physics.

As usual, you can avoid the update anomalies by performing a nonloss decomposition into projections. In this particular case, the projections, both of which are in 5NF, are as follows:

student_teacher		teacher_subject	
student	teacher	teacher	subject
PK		PK	
Smith	Alonzo Church	Alonzo Church	Logic
Smith	Robert Millikan	Alfred Tarski	Logic
Jones	Alfred Tarski	Robert Millikan	Physics
Jones	Max Planck	Max Planck	Physics

It is now possible to delete the information that Jones is studying physics by deleting the row for Jones and Max Planck from projection *student\_teacher* without losing the row for Max Planck and Physics from projection *teacher\_subject*. So the problem is solved ... or is it?

Yes, that particular problem is solved. A remaining problem, however, is that this decomposition violates the FD preservation objective described in “[Preserving Functional Dependencies](#)” on page 103.

Specifically, the FD *student, subject—teacher* now spans two relvars, *student\_teacher* and *teacher\_subject*, so it cannot be deduced from the FD *teacher—subject*, which is the only nontrivial FD represented in *student\_teacher* and *teacher\_subject*. As a result, *student\_teacher* and *teacher\_subject* cannot be updated independently of one another.

For example, an attempt to insert a row for Smith and Max Planck into *student\_teacher* must be rejected, because Max Planck teaches physics and Smith is already being taught physics by Robert Millikan, yet this fact cannot be detected without examining *teacher\_subject*.

The point of this analysis is to highlight the fact that the following equally desirable objectives of database design can sometimes clash:

- Decomposing relvars to the their ultimate normal form (or even just to BCNF)
- Decomposing relvars in such a way that preserve FDs

In other words, it is not always possible to achieve both objectives simultaneously.

Other points arise from this example:

- Neither design is superior to the other on its face. The principles of normalization suggest that one design is better, while the principle of FD preservation suggest that the other is.

As a result, the ultimate design choice must be based on other criteria.

- Assume that you settle on the two-relvar design.

In this case, the relvar-spanning FD must be declared, at least for documentation purposes, even if the design cannot enforce it.

A fully formal version of that declaration might look something like this:

```

FORALL s IN S, t1,t2 IN T, j1,j2 IN J
  IF      EXISTS { S:s, T:t1 } IN ST
    AND EXISTS { S:s, T:t2 } IN ST
    AND EXISTS { T:t1, J:j1 } IN TJ
    AND EXISTS { T:t2, J:j2 } IN TJ
  THEN j1 , j2

```

where the student, teacher, and subject domains are represented by S, T, and J, respectively.

If you think this example is overly contrived, and that involved dependency structures such as *student—teacher—subject* never occur in practice, consider the following, more familiar, example:

**address**

street	city	state	zip_code
--------	------	-------	----------

This relvar satisfies the following set of FDs:

- $\text{zip\_code} \rightarrow \text{city}, \text{state}$
- $\text{street}, \text{city}, \text{state} \rightarrow \text{zip\_code}$

In other words, it is identical to the *student\_subject\_teacher* example, where *student* maps to the composite *street-city, state* maps to *subject*, and *zip\_code* maps to *teacher*.

## General Procedure for Achieving a Normalized Set of Relations

Although a designer having relatively little experience with normalizing a relational database can often put relations into 2NF and even 3NF without giving the process much thought, the following procedure is provided so that even a novice can pursue a set of steps through to completion and achieve normalization of a fairly simple set of relations.

The following procedure provides a high-level method for achieving a normalized set of relations:

- 1 Identify the attributes of the database.
- 2 Group related attributes into relations.
- 3 Identify the candidate keys for each relation.
- 4 Select the most useful primary key from among the set of candidate keys.
- 5 Identify and remove all repeating groups.

The result is a relation in 1NF.

- 6 If any of the resulting relations have identical primary keys, then combine them into a single relation.
- 7 Identify all functional dependencies between the attributes of a relation and its primary key.
- 8 Decompose the relations to a form where each nonkey attribute is dependent on all the attributes of the key. Do this by taking projections of the 1NF relation that eliminate any non-full functional dependencies.

The result is a set of relations in 2NF.

- 9 If any of the resulting relations have identical primary keys, then combine them into a single relation.
- 10 Identify any transitive dependencies in the relations decomposed to this point.
  - a Examine relations for dependencies among nonkey attributes.
  - b Examine relations for dependencies among key within the primary key.
- 11 Remove transitive dependencies by decomposition. Do this by taking projections of the 2NF relations produced by steps 8 and 9 that eliminate any transitive functional dependencies.

The result is a set of relations in 3NF.

- 12 If any of the resulting relations have identical primary keys, then combine them into a single relation if and only if no transitive dependencies result from the combination.
- 13 Remove any remaining functional dependencies from the 3NF set of relations. Do this by taking projections of the 3NF relations produced by steps 8 and 9 that eliminate any remaining functional dependencies where the determinant is not a candidate key.

The result is a set of relations in BCNF.

- 14 Remove any multivalued dependencies that are not also functional dependencies. Do this by taking projections of the BCNF relations produced by step 13.

The result is a set of relations in 4NF.

Also see Date and Fagin (1992).

- 15 The likelihood that any join dependencies not implied by the candidate keys remain by this point is virtually nil. If you can identify any such relations, then take projections of them to eliminate the non-implied join dependencies.

The result is a set of relations in 5NF.

Also see Date and Fagin (1992).

- 16 Particularly for temporal data, but sometimes to eliminate nulls as well, take projections to 6NF.

# Advantages of Normalization for Physical Database Implementation

Teradata Database is the only commercially available database management system that is capable of physically implementing a fully-normalized logical model. Independent data warehousing consultant Neil Raden has written, “...3NF designs don’t support query and analysis [...] The only routine exception to this is Teradata implementations: Because of the unique characteristics of the massively parallel architecture and database (sic) optimizer, Teradata can process analytical SQL against a 3NF schema with acceptable performance”. Note that relational query optimizers optimize SQL *requests* against databases, not the databases themselves.

The following list summarizes the advantages of physically implementing a normalized logical model for Teradata Database:

- Greater number of relations
  - More primary index choices
  - Optimal distribution of data
  - Fewer full-table scans

For example, consider the multicolumn primary index of a Fact table, which has the primary key of each of its dimension tables as a component. The Optimizer cannot retrieve a row with a partial primary index, so many, if not all, accesses to the Fact table must use a full-table scan.

This assumes that you implement the natural primary key of the fact table as the primary index. If you instead define a surrogate key column to be the primary index, the full-table scan issue is moot (see the definition for “Surrogate key” under [“Definitions” on page 75](#) and [“Identity Columns” on page 818](#)). This practice is not generally advised.

- More joins possible
- Enhanced likelihood the Optimizer will use the high-performing merge or nested join methods
- Optimal data separation to eliminate redundancy from the database
- Optimal control of data by eliminating update anomalies
- Fewer columns per row
  - Optimal application separation
  - Optimal control of data
- Smaller rows
  - Optimal data blocking by the file system
  - Reduced transient and permanent journal space
  - Reduced physical I/O

## How Normalization Is Beneficial for Physical Databases

The fundamental objective for a relational database management system is to keep data independent of the applications or analysis that use it. There are two reasons for this:

- Different applications require different views of the same data.
- Data must be extendable, and a framework must exist to support the introduction of new applications and analyses without having to modify existing applications.

This topic develops some of themes described elsewhere in this manual by explaining their relevance to everyday exploratory data analysis problems in data warehousing. The subject matter for this topic is adapted from Russell and Armstrong.

## Data Dependence and Data Independence

Applications implemented in pre-relational database systems are data-dependent, meaning that both the physical representation of the data and the methods of accessing it are built directly into the application code. This makes even the slightest change to the physical design of a database an extraordinarily laborious effort.

The main objective of relational DBMSs is data independence. For years, the relational database management systems used to run businesses, often referred to as OLTP systems, made data independence obligatory. In an OLTP database, data is stored in nonredundant tables that demand that every column of the table be rigorously related to its primary key alone and to no other tables. This ensures that information is available to all applications and analyses that use it, and it provides a mechanism for maintaining consistency and reliability across applications: a single source of each particular data element, a single version of the truth.

Data independence works well for OLTP systems because the applications accessing the data generally access single tables or join only a few, small tables in relatively simple queries. With the introduction of the data warehouse, previously unheard of demands were placed on the relational database management systems underlying them. In the data warehouse environment, large tables must be scanned and large result sets are frequently returned. Many tables are joined together, complicated calculations are made, and detailed data is aggregated directly in the queries. In addition, large data volumes are extracted, transformed and loaded into the tables concurrently with users running queries against the data. It quickly became apparent that databases created and tuned for OLTP could not sustain the performance levels required to support the demands of business intelligence processing. The OLTP databases could not perform the queries within their allotted time window or, in some cases, at all.

This situation highlights the potential for contradiction between designing databases for optimum integrity and designing databases for optimum performance. The key to data independence is data normalization, and normalized data schemas are the most demanding of system performance.

To address the issue of poor performance, data independence has often been abandoned in many environments and denormalized schemas have been used to address a few particular, rather than all general, analytical needs of the enterprise.

**Note:** The term *denormalized* is used because of its familiarity in the industry, not because of its technical accuracy. As described elsewhere in this book (see, for example, “[Normalization as a Logical Process](#)” on page 74), normalization is a logical concept, not a physical, concept. Therefore, it is incorrect to speak of denormalization in the context of physical database design.

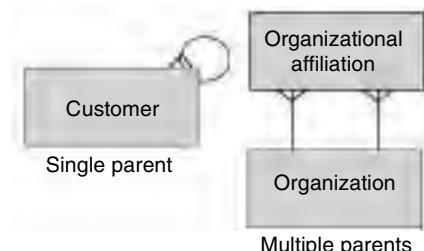
Although this arrangement addresses short-term decision support needs, it compromises the enterprise view of the data and its adaptability. Data independence, adaptability, and cross-enterprise functionality go hand in hand, and a normalized data schema is critical to reaching these objectives.

The following topics provide some detail about why this is true.

## Recursive Relationships

The star schema (see “[Dimensional Modeling, Star, and Snowflake Schemas](#)” on page 187), which is the most common form of denormalization used in contemporary data warehousing, cannot handle every kind of relationship that can exist comfortably in a fully-normalized environment. Recursive relationships are one such example. Recursion, as the term is generally used in computer science, is only a small subset of the recursive function theory of formal logic.

A recursive relationship exists when the parent of a member in a hierarchical relationship is also a member of the same entity. As demonstrated by the following figure, there are two ways that this can manifest itself: with only a single parent or with multiple parents:



1094-012A

The most commonly used example of a single-parent recursive relationship is an *employee* table, where both an employee and its manager have rows. From an E-R perspective, you would say a manager *has* employees. But managers, too, are employees. This also means that managers can have managers who are employees, and so on.

In the diagram, the single-parent recursive relationship is a *customer* table in which a customer can be a customer of yet another customer in the table. The classic multiple-parent recursive relationship is the bill of material. The diagram shows an example in which multiple organizations can have multiple organizational affiliations. Project work breakdown hierarchies are another common example of a a multiple parent recursive structure.

In a recursive structure, there can be an unlimited number of levels without knowing how many levels each member hierarchy currently has or potentially can have. One hierarchy have only two levels, while another might be 15 levels deep. Herein lies the limitation of the star

schema for handling recursive relationships: it requires a fixed number of levels, because each level is set up by a series of fixed columns in a dimension table. Because you do not know the number of levels in a recursive structure, you cannot predefine the columns.

The most critical entities in an enterprise data model frequently have recursive structures. Organizational hierarchies such as internal, customer, supplier, and competitor entities are usually recursive relationships.

## Arguments Against Denormalizing the Physical Database Schema to Increase Its Usability

Many data warehouse designers argue that denormalized physical database schemas are easier for end users to navigate than fully normalized schemas.

Denormalized physical schemas certainly seem more user-friendly than the complexity of a highly generalized, normalized data model. However, denormalized physical schemas are driven by known queries, so their ease of use is somewhat illusory. Formulating queries to address novel requirements, a task that is nearly definitive of the data warehouse process model, is made more difficult, if not impossible, in a denormalized environment. A fully normalized enterprise data model is flexible enough to support the undertaking of any new analyses of the data.

That said, the reality is that end users typically do not write queries anyway, and when they do, they are likely to use a third party natural language query generator, so the usability argument is often moot. Coding novel queries is often the responsibility of an application developer or a natural language query writing tool.

More importantly, you can create “denormalized” views to implement a semantic layer that makes the normalized data model easier to navigate (see [“Denormalizing Through Views” on page 185](#)). Few sites permit users to query base tables directly anyway, so creating views on base tables that look exactly like their denormalized table counterparts should not be an issue.

If there were no issues of performance for those database management systems that lack the parallel processing power of Teradata Database, then denormalization could be handled universally by implementing views (see [“Arguments Against Denormalizing for Performance” on page 113](#)). Star schemas, snowflakes, summary tables, derived data, and the like could be built as virtual clusters of tables that look exactly like their physical counterparts. By handling denormalization virtually, the relationships within, between, and among the underlying base tables of the schema remain intact, and referential integrity can be maintained by the system regardless of how many virtual denormalized relationships are created. This flexibility frees DBAs to create any number of denormalized views for users while simultaneously maintaining semantic data integrity and eliminating the data redundancies required by denormalized physical schemas.

DBAs can create virtual, subject-oriented schemas for specific applications as well as creating views for more general database access without affecting the underlying base table data. These same views can also be used to enforce security constraints for the different communities of business users who must access the database.

Consider another argument that favors the ease of use of a fully-normalized database schema over a denormalized schema. A physical star schema has physical dimensions that support a physical fact table. However, for some dimensions there can be mutually exclusive substitutes for the same data. For example, suppose an airline is interested in both the point-to-point travel of customers between segments in addition to their travel between their true origins and destinations. This discussion abbreviates this dimensional family as O&D.

The true O&D dimension is different from the segment O&D, although superficially, it looks the same. Moreover, their respective consolidation of facts is different as well, although the detailed base table data is the same. If the star schemas are physicalized, two very large schemas must be created, maintained, and coordinated to represent them, whereas with virtual star schemas, the data is maintained only in the base tables, producing a single, consistent version of the truth.

## Arguments Against Denormalizing for Performance

Data warehousing authorities often argue that physical denormalization of the logical data model offers better performance than a physical instantiation of the normalized logical data model. This question is mitigated fully when the database engine can handle a normalized physical design and scale linearly. Teradata Database does just that (see “[Born To Be Parallel](#)” on page 25).

If further need to improve performance remains aside from scaling the database, then implement a well-planned data propagation strategy that maintains and complements the underlying normalized base table substructure.

To begin, consider propagating denormalized data within the same database instance. Weigh propagating to another environment only if there are other considerations for the target data source, such as geographic needs or the need to support proprietary data structures. In any case, the propagation is from the data warehouse and not directly from source systems.

These reasons support this strategy.

- The fully-parallel capabilities of the data warehouse can be used to optimize the propagation of data.
- By keeping the data in the same instance of the database, you can perform hybrid queries that take advantage of both the denormalized and normalized data. For example, large volumes of nonvolatile data from transaction detail rows can be propagated into new physical fact tables, and smaller volume, highly volatile dimensional data can be built into virtual dimension tables.
- The resulting administration of the complete data warehousing environment is not only easier, but also less expensive.

## Denormalized Physical Schemas and Ambiguity

A denormalized physical schema creates ambiguity because within a denormalized table, it is not possible to determine which columns are related to the key, to parent tables, or to one

another (see “[Functional, Transitive, and Multivalued Dependencies](#)” on page 80, “[The Referential Integrity Rule](#)” on page 95, and “[Domains and Referential Integrity](#)” on page 100 for explanations of how fully normalized tables avoid this ambiguity). Normalized models maintain all relationships through the association of primary keys with their functionally dependent attributes and with foreign keys.

Consider the following non-normalized table, where all but one of the column names have been abbreviated to fit on the page.

com_id	com_name	grp_id	grp_name	fam_id	fam_name	sequence
PK						

The expanded column heading names are given in the following table.

Abbreviated Column Name	Actual Column Name
com_id	commodity_id
com_name	commodity_name
grp_id	group_id
grp_name	group_name
fam_id	family_id
fam_name	family_name

This table meets the criteria for 2NF, but not for 3NF, because not only does every non-key attribute *not* depend on the primary key, but several non-key attributes are functionally dependent on other non-key attributes (see “[Definition of Third Normal Form](#)” on page 86).

You cannot perceive the relationships among the *commodity\_id*, *group\_id*, and *family\_id* attributes by looking at this table. While it is fairly obvious that *commodity\_name* is a functional dependency of the primary key, it could also be true that the commodity attribute has separate relationships with groups and families, or it could be true that commodities are related to groups, which are, in turn, related to families. Moreover, the relationship of the Sequence attribute to the primary key, or to anything else, cannot be resolved. Is it used to order commodities within groups, commodities within families, or groups within families? It is surprising how often such dangling relationships are found in denormalized logical models.

In contrast, normalized models maintain all relationships through the association of primary keys with their attributes and with foreign keys. As a result, you can clearly see the relationships among the various tables in the database schema simply by looking at the data model.

## Referential Integrity and a Denormalized Schema

Referential integrity cannot be effectively maintained within a denormalized database schema.

By definition, denormalized structures compromise data quality. Relational database management systems enforce referential integrity to ensure that whenever foreign keys exist, the instances of the objects to which they refer also exist. Denormalized tables cannot support referential integrity through declarative constraints. For example, users have no guarantee that the children in a denormalized table have parents in that table unless they implement some other, more costly method of programmatically ensuring semantic data integrity. The problems with maintaining referential integrity by means of application code rather than through declarative constraints are myriad (see [Chapter 12: “Designing for Database Integrity”](#)).

Also, as new incremental data loads are inserted into the database, changed fields are not reflected in the old data. When an attribute is changed in the source system, only the newly loaded rows reflect those changes. For example, suppose the marital status, last name, or another field in the customer table changes. Any new loads for that customer reflect the different data in the changed fields, causing inconsistencies and incorrect results in certain types of analyses because both the changed and unchanged attributes are stored redundantly, violating the concept of a single version of truth that is so easily maintained with a fully normalized database schema.

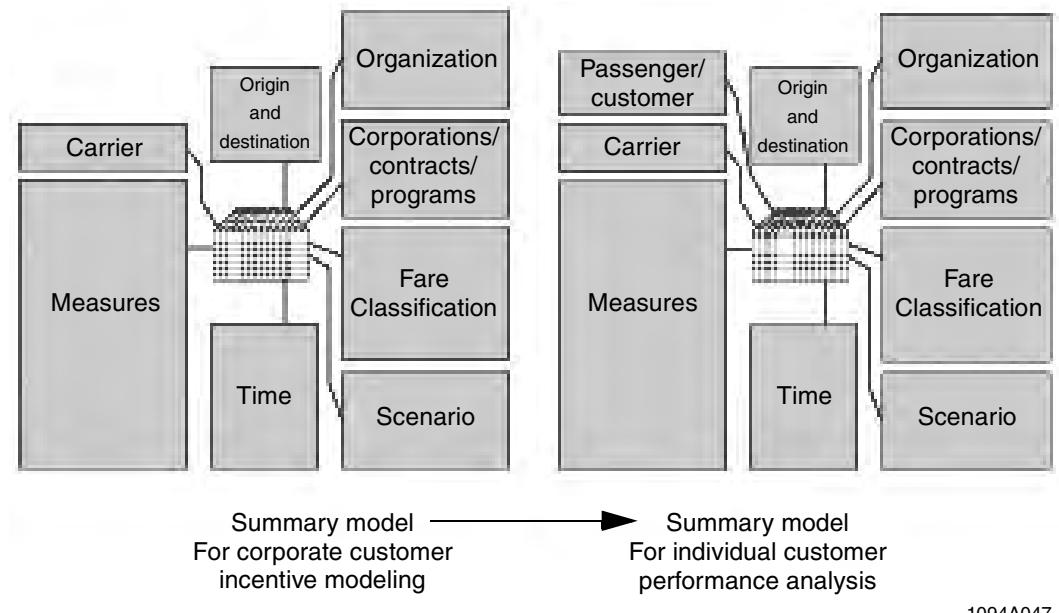
## Denormalized Views Versus Physical Denormalization of the Database Schema

A fully-normalized database schema accessed through denormalized views is far more flexible in creating and managing dimensional consolidation hierarchies for OLAP models than a star schema.

Note that Ross and Kimball do not address dimensional views as a solution to resolving the differences between normalized and dimensional database design. Their “hybrid approach” is conceptually similar, and leads to the identical outcome, but it is explicitly redundant, requiring two separate physical databases to implement. The dimensional view approach, by using one physical database, eliminates both the additional costs and the data quality problems of maintaining redundant data.

DBAs who deal with denormalized methods typically define a single set of mutually exclusive dimensions for a given OLAP model. However, it is possible to create many nonexclusive dimensional paths for an individual OLAP model that also are reusable across many different OLAP models.

The following figure shows two denormalized virtual schemas that share a number of common dimensional views:



The number of virtual schemas that can be created and aligned with particular users or user communities is unlimited. As DBAs become more sophisticated in their application of this concept, they can manage an extensive number of database views tailored toward the specific needs of individual users as well as the specific needs of various user groups. It is not possible to maintain such an extensive schema management program when the data must be physically propagated for performance reasons.

Consider another example: suppose there are eight potential dimensional consolidation paths that could arise from the following three entities:

- Households
- Customers
- Accounts

In OLAP terminology, a *consolidation path* is a set of hierarchically related data. Consolidation itself is the process of aggregating the hierarchical data to form subtotals. The highest level in the consolidation path is the dimension for the data.

The eight possible consolidation paths for this data are the following, where → indicates a downward path through a hierarchy and in E-R terminology is equivalent to the word *have*:

- *Households*
- *Customers*
- *Accounts*
- *Households* → *Customers* → *Accounts*
- *Households* → *Accounts* → *Customers*
- *Households* → *Customers*
- *Households* → *Accounts*
- *Customers* → *Accounts*

When either these consolidation paths or the previously mentioned schemas are created virtually, any problems with managing referential integrity are rendered moot because the only data that must be maintained is in the base tables. However, when these tables are instead instantiated as separate physical tables, managing referential integrity across the multiple alternate dimensional paths becomes very difficult to maintain. As more dimensional entities and potentially hundreds more dimensional consolidation paths are added to the schema, it becomes impossible to continue to propagate data physically. As a result, if a physical star schema is implemented, compromises to the maintenance of the semantic integrity of the data must be made to accommodate the resulting complexity.

## Dimensional Analysis

A star schema is designed specifically to support dimensional analysis, but not all analysis is dimensional. Many types of quantitative analysis, such as data mining, statistical analysis, and case-based reasoning, are actually inhibited by a physical star schema design.

Although not addressing the topic of normalization directly, Xiong et al. (2006) address related issues when they write the following passages concerning the problem of enhancing data mining by removing noise from the database, “...data objects that are irrelevant, or only weakly relevant can ... significantly hinder data analysis. Thus, if the goal is to enhance the data analysis as much as possible, these objects should ... be considered as noise, at least with respect to the underlying analysis” (p. 304, italics not in original).

Later in the same paper, they write, “...we will refer to the objects that are eliminated as noise since this use of the word falls within the general meaning of noise as meaningless or irrelevant data” (p. 306).

Recall the definition of 3NF given by Kent: the key, the whole key, and nothing but the key (see [“Definition of Third Normal Form” on page 86](#)). Given this definition, any attributes that do not modify the key, the whole key, and nothing but the key are “noise” in the context of normalization theory.

If you consider the household/customer/account example (see [“Denormalized Views Versus Physical Denormalization of the Database Schema” on page 115](#)) in a different light, you encounter an interesting problem with denormalization. An OLAP modeler, focused on creating dimensional views and drill-downs, is likely to assume that a household can have many customers but that a customer can belong to only one household. However, the underlying normalized data model might reveal a many-to-many relationship between households and customers. For example, a customer can belong to many different households over time.

A modeler focused on OLAP applications typically is not concerned with this relationship because it is unlikely that OLAP users would want to drill down into previous households. However, a data mining user might find tremendous value in exploring this relationship for patterns of behavior that revolve around the changing composition of households.

For example, consider the following possible household change behaviors:

- Children leave their parents to start their own households.
- Couples marry and form one household from two.

- Married couples divorce and form two households from one.

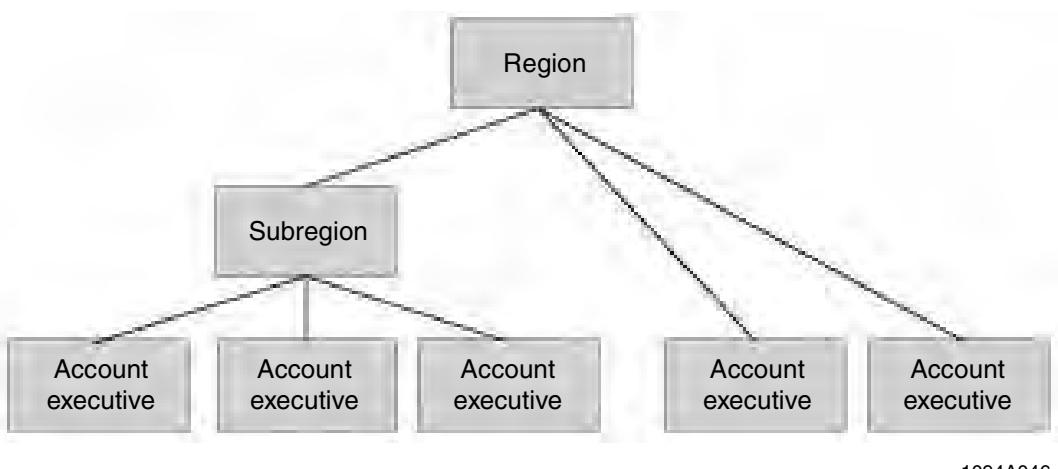
And so on.

The enterprise needs a logical model to address the precise needs of *any* analytical methodology, whether it be a complicated ad hoc SQL query, an OLAP analysis, exploratory work using data mining, or something entirely different.

## Unbalanced Hierarchies and Unnormalized Physical Schemas

You cannot make the universal assumption that all levels in a dimensional model are balanced. A star schema dimension table, which requires fixed columns for each level, has considerable difficulty in handling unbalanced hierarchies.

An unbalanced hierarchy, as shown in the following figure, is a hierarchy in which a particular level can have its parent at different levels above it. An account executive, for example, might report to a region or a subregion.



1094A046

Because a star schema creates specific columns to handle each level in a hierarchy, it requires that all hierarchies be balanced so that, using the current example, an account executive would always need to be at level three.

## Analyses Driven By Normalized Relationships

Many types of data warehouse analysis are driven by the normalized relationships in the database.

There is, for example, a considerable amount of analysis made within the data warehouse that is not quantitative in any way. For example, market basket studies determine clusters of products that are most frequently purchased together. A market basket study does no counting, aggregating, or any other quantitative measure or analysis.

Tracking the cycle times for various entities in a data warehouse requires a sophisticated method of following the relationships between statuses and events. Modeling strategic frameworks such as value chains, supply chain management, competency models, industry structure analysis, and total quality management requires sophisticated relational models for which denormalized approaches are inadequate.

## Costs of Denormalization

In the long term, the actual cost of maintaining a denormalized environment for an enterprise data warehouse exceeds the costs of a normalized environment, and the cost of decreased cross-functional information opportunities is much greater.

There are economic costs associated with denormalization that often are not considered. In a star schema, each row in a dimension table contains all of the attributes of every entity mapped within it. In the accounts, customers, and households example of “[Denormalized Views Versus Physical Denormalization of the Database Schema](#)” on page 115, you would carry all the redundant household data and all the redundant customer data for each account. Not only does the redundancy from the expansion of columns exist, but in the case of the many-to-many relationship between accounts and customers, the number of rows also increases because a separate row is required for each legitimate account-customer combination. When there are millions of accounts, thousands of which are jointly held, this horizontal and vertical redundancy can add significant storage overhead.

These costs generally are not significant in light of performance gains, but they are significant in terms of the additional DBA and application coding costs incurred to programmatically maintain referential integrity.

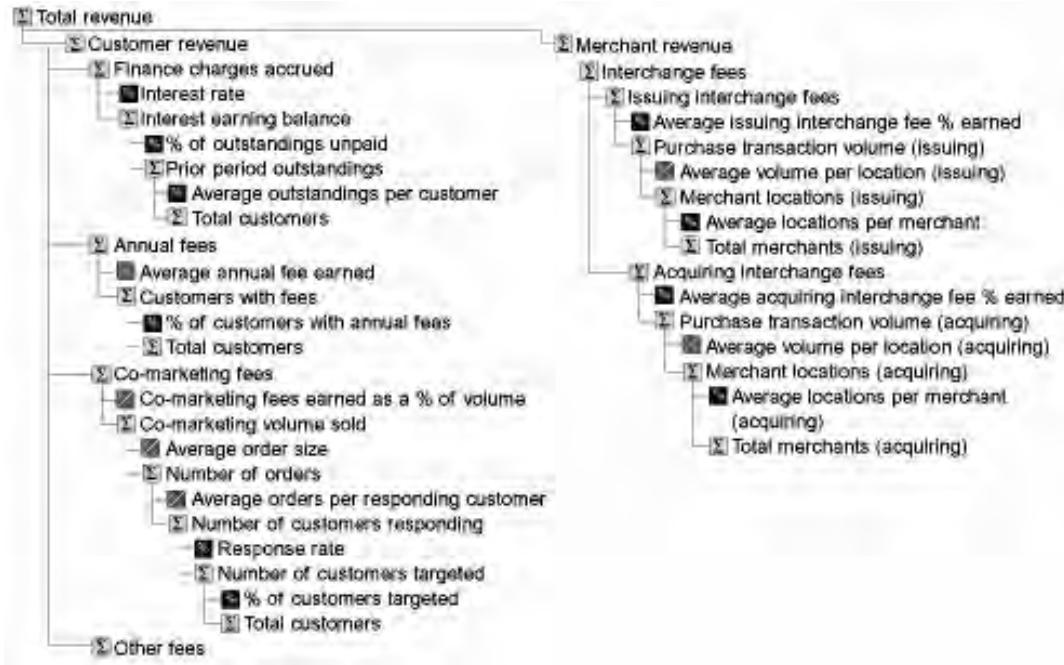
More important is the cost of business opportunities lost because of compromises that render entire categories of data analysis difficult, if not impossible, to perform. Total benefits are much greater for the normalized approach because of its adaptability and generalizability. As an enterprise begins to define its business analysis requirements, it initially identifies only a fraction of what it ultimately needs. Possibly as much as 95% of the real analytic needs of the enterprise go undefined in the infancy of the data warehouse.

The key to being able to capitalize on unsuspected opportunity is flexibility. By building an adaptable database schema from the beginning, an enterprise enables itself to address new business needs as they are identified without having to compromise or restructure the database. The enterprise is also able to address any new challenges in the shortest time frame because it does not need to involve its IT staff in designing, building, and propagating data for the new queries. The faster a company can respond to unexpected challenges and opportunities, the higher the business value that can be realized from its data.

## Cross-Functional Analysis and Denormalization

Cross-functional analysis becomes increasingly difficult as the data warehouse becomes more and more denormalized.

Data warehousing provides a means to build complex interrelated models for cross-subject area analyses in ways no other system can. These models can move beyond the traditional financial measures to begin interrelating internal process measures and customer-oriented measures as well. More importantly, these more sophisticated analytical models can begin to push from results-oriented or outcome-oriented measures toward measures directly linked to organizational activities. The following figure shows an example of interrelated measures from a credit-card model.



1094A048

The problem with denormalized data for this example is that measures cross many different denormalized schemas. Even though Teradata Database is optimized to handle them, star joins are cumbersome to process, and making joins across multiple models also makes them extremely complex. In this particular example, there could easily be four or more distinct star schema structures.

## No Crystal Ball

There is no crystal ball for predicting the future data analysis needs of the enterprise. The tendency to compromise and build the enterprise data model based solely on current needs is much greater for those building denormalized enterprise database schemas than for those building fully normalized schemas.

Even when aware of the compromises being made, individuals undertaking a denormalized implementation of the logical model tend to discount the possibilities of those compromises working against future growth of the system because they believe they have already anticipated all future analytical opportunities.

However, as business discovery evolves, it inevitably becomes clear that those initial projections do not fully support the requirements and vision of the business. For such an enterprise database design, it is inevitable.

## Design For Decision Support and Tactical Analysis

Not only is basic schema normalization an important design consideration, but normalization tailored specifically for the analytical needs of business intelligence is also critical. Even fully normalized data warehouses are often built as if they were designed to support OLTP applications.

Designers often fail to examine the following list of considerations:

- Integration of subject areas
- Versioning across time
- Generalization of key entities in anticipation of change
- Interrelation of life cycles across many cross-subject area entities and events
- Integration of measures, calculations, and dimensional context paths across the enterprise

These factors all support the immediate and long-term benefits provided by a fully normalized enterprise data model. The fully normalized schema provides a framework both for continued growth and for increasing user insight.

The moral of the story is this: it is often a mistake to compromise your logical data model when you select tools to extend the value of your data and to optimize the delivery of information to your users. When necessary to facilitate ease of use or to improve tool performance, implement views to help ensure the continued integrity of your logical data model. When you are evaluating the data model you need to support your data warehouse, ensure that model not only supports your business today, but that it is fully capable of supporting the enterprise well into the future.

## Design for Flexible Access Using Views

You should design your applications to always access the data warehouse through views rather than directly accessing base tables. Administrators sometimes avoid application designs that access the database through views because of a perceived performance penalty. While this can be true for delete, insert, and update operations, it is rarely true for access operations, and then only in exceptionally rare cases. In fact, view projections of their underlying base tables can minimize the use of spool space by joins by reducing the number of columns that must be redistributed to make the join. Furthermore, you can more readily control locking for access through careful view design, and so minimize the complexity of applications that access the database through those views.

As a general rule, the best practice is to design applications to access the database *only* through views rather than accessing base tables directly.

Besides providing you with the ability to provide pseudo-denormalized access for enhanced usability (see “[Denormalized Views Versus Physical Denormalization of the Database Schema](#)” on page 115), views also provide a degree of data independence that permits you to make changes either to your applications or to the physical database those applications access without having to worry about rewriting application software. Views can also provide a coarse level of schema versioning that is otherwise not available (see “[Design For Decision Support and Tactical Analysis](#)” on page 120).

The list of design advantages that views present for database access does *not* mean that it is generally advisable to perform delete, insert, or update operations through views. As a general rule, the best practice is to design applications that update the database to access base tables directly rather than through views.



# CHAPTER 6 The Activity Transaction Modeling Process

---

This chapter first describes and defines some of the concepts underlying the Activities and Transaction Modeling (ATM) process and then describes the ATM process and its forms, providing examples of how to fill them out and apply them to the physical database model.

## Purpose

You use the early ATM process forms as input to more complex forms later in the process. You use the final ATM process forms when you create the physical objects in your database, ensuring that the universe of domains, constraints, table accesses, and reports and queries are understood well in advance of the process of creating the physical database objects that derive from those constructs.

The ATM process is the first step in the transition from your logical data model to the implementation of your physical data.

The second and final step in that transition is to collect or prototype data demographics.

## Goals

- 1 Define all domains and constraints.
- 2 Identify all applications.
- 3 Model application processing activities including their transactions and run frequencies.
- 4 Model each transaction using the following information.
  - a Identify tables used.
  - b Identify columns required for value and join access.
  - c Estimate qualifying cardinalities.
- 5 Summarize value and join access information across all transactions.
- 6 Add data demographics to the Table Access Summary by Columns report.
  - Table cardinalities
  - Column value distributions (histograms)
  - Column change ratings

The following table indicates the activities of this process and the forms required to complete each activity.

Step	Action	Form Used
1	Define all domains in the system.	Domains
2	Define all constraints for the system.	Constraints
3	Identify all applications in the system.	System
4	Model each identified application.	Application
5	Model each identified transaction.	Report/Query Analysis
6	Summarize all value and join accesses.	
7	Transfer access information.	Table
8	Compile or estimate data demographic information.	
9	Identify column change ratings.	

Note that row sizing calculations are covered in [Chapter 14: “Database-Level Capacity Planning Considerations”](#) in the topic [“Row Size Calculation”](#) on page 846.

## Terminology

Term	Definition
Domain	<p>A well-defined, closed set of values from which column data can be drawn. Domain is really just another term for data type.</p> <p>Predefined data types generally do not provide sufficiently distinct domains, and you should consider using distinct user-defined types whenever domain integrity is important to your databases or applications. Note that you cannot specify any type of constraint for a UDT column. You can otherwise use various constraints, particularly check constraints, to restrict the range of values a column will accept, so if your application requires any kind of check, uniqueness, or referential constraints, you cannot define a domain for the column using a distinct type.</p> <p>See <i>SQL Data Definition Language</i> and <i>SQL External Routine Programming</i> for information about creating UDTs and their associated database objects.</p> <p>Note that Teradata Database does not support the <i>physical</i> concept of domains.</p>

Term	Definition
Constraint	<p>A well-defined physical restriction that can be defined for a column or table.</p> <p>Constraints include the following:</p> <ul style="list-style-type: none"> <li>• UNIQUE</li> <li>• PRIMARY KEY</li> <li>• FOREIGN KEY</li> <li>• CHECK <i>expression</i></li> <li>• REFERENCES</li> </ul> <p>The following rules and recommended practices apply to constraints.</p> <ul style="list-style-type: none"> <li>• Always name constraints.</li> <li>• Constraint names must be 30 or fewer characters.</li> <li>• A constraint name must be unique among all other constraint names defined for a table.</li> <li>• You can specify constraints both at the column and table levels.</li> <li>• The system does not assign names to constraints you do not name.</li> <li>• You cannot define any type of constraint on an XML, BLOB, or CLOB column.</li> </ul> <p>For detailed information about the various constraints, see “CREATE TABLE” in <i>SQL Data Definition Language Detailed Topics</i>.</p> <p>For design-related information about column and table constraints, see <a href="#">Chapter 12: “Designing for Database Integrity.”</a></p>
Table	<p>A multidimensional, abstract representation of an entity constructed from the following components:</p> <ul style="list-style-type: none"> <li>• Rows, representing tuples</li> <li>• Columns, representing attributes</li> </ul> <p>Tables are sometimes referred to as relations, though the correspondence between relations and tables is not always direct.</p> <p><a href="#">“Example 1: The Location Entity Before It Is Fully Attributed” on page 127</a> shows the structure of the Location entity before it is fully attributed.</p>
Row	<p>An instance of an object in a relational table. Rows in relational tables are not ordered. The expression is a synonym for record, a term not used for relational database systems.</p> <p>Rows are sometimes referred to as tuples, though the formal term for the corresponding concept is <i>n</i>-tuple.</p> <p>The number of rows in a table is referred to as its cardinality.</p> <p><a href="#">“Example 2: A Randomly Selected Row From the Location Entity” on page 127</a> indicates a randomly selected row from the <i>Location</i> entity.</p>

Term	Definition
Column	<p>A unique, atomic attribute of a relational entity. Columns in relational tables are not ordered logically, though they are ordered physically and can be referred to by their column number.</p> <p>Columns are sometimes referred to as attributes.</p> <p>The number of columns in a table is referred to as its degree or arity.</p> <p><a href="#">“Example 3: CustNum Column From the Location Entity” on page 127</a> indicates the <i>CustNum</i> column from the <i>Location</i> entity.</p>
Primary Key	<p>The primary key is a column set that uniquely identifies a tuple within a relation. Every relation must have one and only one primary key defined during logical design, but a primary key need not be formally defined for a table corresponding to a logically defined relation.</p> <p>A table can have multiple candidate primary keys, but only one defined primary key. A candidate primary key not selected as the primary key for a table is referred to as an alternate key.</p> <p>A primary key cannot be null and must be unique.</p> <p>No component of a primary key can have the XML, BLOB, or CLOB data types.</p> <p><a href="#">“Example 4: The Primary Key Column for the Location Entity” on page 128</a> indicates the primary key column for the <i>Location</i> entity.</p>
Identity Column	<p>A column for which the values are unique and system-generated. Values for identity columns can be generated by the system in all cases or only in those cases for which users do not provide a value.</p> <p>Identity columns are frequently used to generate surrogate keys.</p> <p>You cannot define an identity column having the XML, BLOB, or CLOB data types.</p> <p>See <a href="#">“Identity Columns” on page 818</a> and the detailed description of identity columns and their use documented by “CREATE TABLE” in <i>SQL Data Definition Language Detailed Topics</i> for more information about identity columns.</p>
Foreign Key	<p>The foreign key is a column set that identifies a relationship between the table for which it is a foreign key and one or more other tables in the database.</p> <p>Foreign keys are used both as join conditions and to maintain referential integrity between tables.</p> <p>A foreign key must be the primary key for the table it references and it can be null unless you define the foreign key column set for the table to exclude nulls.</p> <p>No component of a foreign key can have the XML, BLOB, or CLOB data types.</p> <p><a href="#">“Example 5: The Three Foreign Keys For the Location Entity” on page 128</a> indicates the three foreign key for the <i>Location</i> entity. Notice that the <i>CustNum</i> column, which is constrained to be not null, relates <i>Location</i> to <i>Customer</i>, the <i>State</i> column relates <i>Location</i> to <i>State</i>, and the <i>Country</i> column, which is also constrained to be not null, relates <i>Location</i> table to <i>Country</i>.</p>

Term	Definition
Normalization	A method for segregating the attributes of a database into individual tables in such a way that those attributes uniquely modify (or depend upon) the primary key for that table.  Somewhat more formally, a relation is said to be fully normalized if all its nonkey attributes are functionally dependent on its primary key.

### Example 1: The Location Entity Before It Is Fully Attributed

LocationNum	CustNum	State	Country
PK, SA	FK, NN	FK	FK, NN
2	10	CA	USA
7	0	CA	USA
1	2	NY	USA
4	7	PR	USA

### Example 2: A Randomly Selected Row From the Location Entity

LocationNum	CustNum	State	Country
PK, SA	FK, NN	FK	FK, NN
2	10	CA	USA
7	0	CA	USA
1	2	NY	USA
4	7	PR	USA

### Example 3: CustNum Column From the Location Entity

LocationNum	CustNum	State	Country
PK, SA	FK, NN	FK	FK, NN
2	10	CA	USA
7	0	CA	USA
1	2	NY	USA

LocationNum	CustNum	State	Country
PK, SA	FK, NN	FK	FK, NN
4	7	PR	USA

#### Example 4: The Primary Key Column for the Location Entity

LocationNum	CustNum	State	Country
PK, SA	FK, NN	FK	FK, NN
2	10	CA	USA
7	0	CA	USA
1	2	NY	USA
4	7	PR	USA

#### Example 5: The Three Foreign Keys For the Location Entity

LocationNum	CustNum	State	Country
PK, SA	FK, NN	FK	FK, NN
2	10	CA	USA
7	0	CA	USA
1	2	NY	USA
4	7	PR	USA

## Domains

The concept of domain as used in relational database theory derives directly from the more formal definition given in set theory and function theory. A variable is defined as a set of points having both a domain and a range, where the domain defines the specific type of data represented by the set and the range defines the bounds on that type.

A more concise way of expressing this is to say that domains are data types. It is frequently a good idea to extend this definition to include constraints defined for a column. Extended domains of this type are frequently referred to as business rules. A simple example would be to define *employee\_number* as an INTEGER type and extend the definition with a constraint to disallow negative integer numbers as valid employee numbers. Such an example conflates the

mathematical notions of range and domain. Strictly speaking, a domain is only a data type, and range constraints are not part of its definition.

Domains have been a fundamental concept supporting the theory of relational databases since the inception of that theory. The variables described by a domain in a relational database are the valid values an attribute, or column, can have.

Unfortunately, the ANSI/ISO SQL standard does not support a rigorous atomic definition of domain, so it is up to the database designer to define the domains for a database and their ranges rigorously by creating various constraints on table columns, by creating appropriate user-defined data types (the behaviors of distinct UDTs are based on the behaviors of the predefined data types from which they are derived. Distinct UDTs can optionally also have user-defined behaviors.

The behaviors of the other category of user-defined data types, structured UDTs, are exclusively user-defined. See *SQL Data Definition Language* for more information about creating distinct and structured UDTs and *SQL Data Definition Language* and *SQL External Routine Programming* for details about user-defined data types and their associated database objects), and by ensuring that applications do not make nonsensical cross-domain data manipulations such as subtracting an integer part number from an integer inventory count (see “[Column Comparisons](#)” on page 132).

**Note:** You cannot specify a referential integrity constraint for a UDT or CHECK constraint column, though UDT columns (with the exception of ARRAY, VARRAY, Period, XML, JSON, and Geospatial types) can be used in the definition of a UNIQUE or PRIMARY KEY constraint or a primary or secondary index. You can otherwise use various constraints, particularly CHECK constraints, to restrict the range of values a column will accept, so if your application requires any kind of check, uniqueness, or referential constraints, you cannot define a domain for the column using a distinct type.

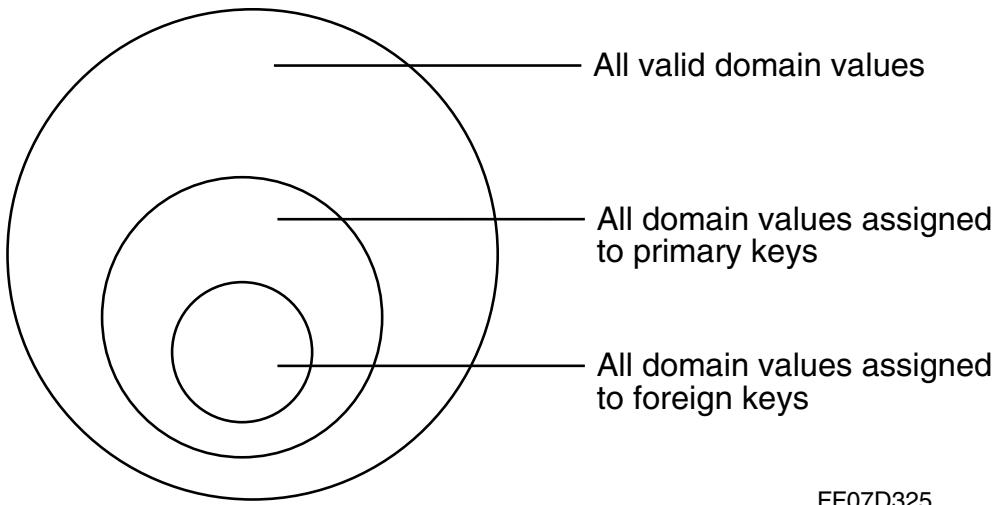
## Properties of Domains In Teradata Database

Domains in Teradata Database are defined to have the following properties:

- A domain defines the set of possible values that can be written to a table column.
- The values of a domain cannot be decomposed to component values; they are atomic.  
Note that this is not true for domains having INTERVAL, TIMESTAMP, or structured UDT data types, which are not atomic, but the concept generally holds true otherwise.
- A domain must have a name.
- A domain must have an assigned data type.

## Domains and Keys

The following diagram indicates the relationship among primary and foreign keys drawn from the same domain.



FF07D325

The following rules apply to these relationships.

- All primary key values for a table are drawn from the domain that defines all possible values for that column.  
That domain cannot be defined with an XML, BLOB, or CLOB data type.
- All foreign key values that reference a primary key value in another table are drawn from the same domain as the values for that primary key.  
That domain cannot be defined with an XML, BLOB, or CLOB data type.

## Teradata Database Data Types

When you applications require a strict domain type to ensure domain integrity, particularly for column comparisons and arithmetic operations, you should consider defining user-defined distinct data types for those domains.

See *SQL Data Definition Language* for information about creating user-defined data types and their associated database objects.

See *SQL External Routine Programming* for information about writing external code for the methods you define to work with user-defined types.

The following table lists the valid Teradata Database predefined data types you can use to define a domain or to create new user-defined data types:

Data Type	Definition
ARRAY VARRAY	Not valid. You cannot create a distinct UDT using a ARRAY/VARRAY data types, so you cannot create a domain using an ARRAY/VARRAY predefined type as its base.
BLOB BYTE VARBYTE	A binary integer used to store digital images.

Data Type	Definition
BINARY LARGE OBJECT (BLOB)	A large binary string used to store binary objects such as musical recordings, videos, and other multimedia.
CLOB CHARACTER VARCHAR LONG VARCHAR GRAPHIC	Any glyph from a supported language. For the English language, this type is often referred to as alphanumeric. The CLOB type is typically used to define large character objects whose length exceeds 64KB.
CHARACTER LARGE OBJECT (CLOB)	A large character string used to store documents, possibly encoded using tag languages such as XML.
XML	A large binary string used to store XML documents in XML format.
DATE	A valid, named 24-hour epoch from the Gregorian calendar.
DECIMAL	Any base-10 real number, including those with a fractional part.
NUMBER (Exact form)	Any base-10 real number, including those with a fractional part.
FLOAT REAL DOUBLE PRECISION	A rational number expressed in exponential format which, depending on the value, might be exact or might be an approximation to a real number.
NUMBER (Approximate form)	Any base-10 rational number expressed in exponential format.
BIGINT INTEGER BYTEINT SMALLINT	Any natural number.
INTERVAL	A time duration with optional fractional precision. Interval data is not implemented atomically, though it is generally treated logically as if it were atomic.
TIME	A valid time expressed using 24-hour notation with optional fractional precision. The TIME type can also be defined with a TIME ZONE.
TIMESTAMP	A valid date and time expressed using 24-hour notation with optional fractional precision. The TIMESTAMP type can also be defined with a TIME ZONE.
Period	Not valid. You cannot create a distinct UDT using the Period data type, so you cannot create a domain using the Period predefined type as its base.

See *SQL Data Types and Literals* for complete listings and descriptions of the data types available for Teradata Database table columns.

You can also use these predefined data types to create your own user-defined data types that may be more suitable for your particular application workloads. See *SQL Data Definition Language* and *SQL External Routine Programming* for details.

## Domains in the Logical Data Model

You should always define and assign domains to your table attributes during the logical design phase of your database design.

Domains can be bounded specifically, nonspecifically, or both simultaneously. For example, you can define the employee number domain to be both not null (a nonspecific boundary) and constrained within a restricted range such as greater than 100 and less than 1 000 000. Unfortunately, you cannot specify CHECK or any other types of constraints on UDT columns, which lessens their usefulness for defining domains.

## Column Comparisons

The necessity for naming your domains becomes clear when you examine the topic of column comparisons.

According to the relational model, columns can be compared if and only if their values are drawn from the same domain. You cannot compare XML, BLOB, or CLOB columns using the built-in SQL operator set. You *can* write UDFs to make such comparisons, however, and if you create UDTs based on BLOB or CLOB types, you can create methods for those UDTs that would permit you to make such comparisons (see *SQL External Routine Programming* and *SQL Data Definition Language* for further information). If an application is to compare data from different columns in any way, all of the following statements must be true for all of the columns being compared.

- They must share the same set of values.
- The same value represents the same real world object in all cases.
- The values can be compared, added, subtracted, and joined.

Note that you cannot operate on XML, BLOB, or CLOB data types

In terms of domains, these rules can be stated as follows.

- Columns from the same domain always have the same defined domain name, which can be based on a user-defined data type.
- Columns from the same domain always have the same defined constraints.
- Columns from the same domain always have the same data type.

Particularly in the case of user-defined data types, the domain and the data type are often isomorphic.

Commercial relational database management systems relax these comparison restrictions to a greater or lesser degree. For example, you can compare INTEGER and DECIMAL values in commercially available systems because, the reasoning goes, both are numeric types.

Database management systems usually provide internal data type conversion routines to ensure such comparisons can be made. Programming languages generally refer to this as weak typing. The more strict domain comparison rules of the relational model, which are more strongly typed, do not permit these types of comparison to be made. UDTs are strongly typed and do not permit careless comparisons unless you write casts specifically to permit them. See CREATE CAST in *SQL Data Definition Language* for more information about creating cast functionality for UDTs.

# Column Names and Constraints

The topic of domains leads naturally to the topic of naming columns and assigning constraints to them.

Always use a consistent method to define the names of the columns in your databases.

The following set of columns illustrates an unsophisticated example of why it is important to name your columns carefully.

## Case Study, Part 1

Suppose you have defined the following two entities:

PrimaryKeyColumn	ColumnA	Date	Date	Date
PK, SA				
		03 Apr 1960	17 Apr 1981	29 Nov 1987
		31 Jan 1937	03 Apr 1960	30 Jun 1981
		05 Mar 1930	21 Aug 1950	03 Apr 1960

PrimaryKeyColumn	Date	Date	Date
PK, SA			
	03 Apr 1960	10 Apr 1960	13 Apr 1960
	29 Mar 1960	03 Apr 1960	06 Apr 1960
	24 Mar 1960	31 Mar 1960	03 Apr 1960

Can you tell if the highlighted dates in these tables all refer to the same point in time? Can you tell if *any* of them do? If not, then you should not compare them. But how do you know whether the dates are drawn from the same domain or not?

## Case Study, Part 2

Now assume that you have identified and named all the domains in your database. The next step is to apply those domains to the identified entities in your logical model.

Consider the same two entities we examined before, only now their columns have been assigned names that represent the domains from which those column values are drawn.

employee				
emp_num	last_name	birth_date	hire_date	termination_date
PK, SA				
		03 Apr 1960	17 Apr 1981	29 Nov 1987
		31 Jan 1937	03 Apr 1960	30 Jun 1981
		05 Mar 1930	21 Aug 1950	03 Apr 1960

order				
order_number	order_date	shipping_date	billing_date	
PK, UA				
	03 Apr 1960	10 Apr 1960	13 Apr 1960	
	29 Mar 1960	03 Apr 1960	06 Apr 1960	
	24 Mar 1960	31 Mar 1960	03 Apr 1960	

## Guidelines for Naming Columns

As it has been defined so far, this column naming convention fails to define the Date domain with enough rigor to prevent an application developer from comparing a shipping date with a birth or hire date, but it provides a sufficient example for illustration as well as a first approximation to how you might define a domain rigorously.

Because ANSI/ISO SQL does not have facilities for defining domains with this level of rigor, your only course of action is to do so textually or to create appropriate distinct user-defined data types (see *SQL Data Definition Language* and *SQL External Routine Programming* for details about how to create UDTs and their associated database objects). Be aware that you cannot specify constraints on UDT columns, so their usefulness for defining domains is less than it might otherwise be.

You can record these distinctions in several ways.

- *DBC.TVFields.CommentString*

Create this comment string using the COMMENT SQL statement.

View the comment you create using the Columns system view.

- Column naming conventions

- Metadata repositories
- ATM Domains form

After you have rigorously defined your domains textually, your next step should be to name your columns by qualifying those domain names.

## Column Naming Convention

Teradata recommends the following syntax for naming columns:

*qualifier —— domain\_name*

FF07D324

where:

Syntax element ...	Specifies ...
<i>qualifier</i>	a unique name to differentiate one use of a domain from all the other uses of that domain.
<i>domain_name</i>	the name of the domain being qualified.

Using this convention, you could define date domains such as the following example names:

- HRDate
- HR\_Date
- OrderDate
- Order\_Date

The more finely defined domain names naturally lead to tables having columns with names like those in the following tables:

employee				
employee_number	last_name	birth_HR_date	hire_HR_date	term_HR_date
PK, SA				
		03 Apr 1960	17 Apr 1981	29 Nov 1987
		31 Jan 1937	03 Apr 1960	30 Jun 1981
		05 Mar 1930	21 Aug 1950	03 Apr 1960

order			
order_number	order_orrdate	shipping_orrdate	billin_gorddate
PK, UA			
	03 Apr 1960	10 Apr 1960	13 Apr 1960
	29 Mar 1960	03 Apr 1960	06 Apr 1960
	24 Mar 1960	31 Mar 1960	03 Apr 1960

With this convention, an application programmer would know never to compare an employee birth date with an order shipping date because they do not represent the same thing even though they share the same set of valid values.

A more rigorous approach might be to create distinct user-defined data types on *birth\_HR\_date*, *hire\_HR\_date*, and *term\_HR\_date*, and then create methods that only permit intra-domain comparisons on those domains. See *SQL Data Definition Language* and *SQL External Routine Programming* for information about how to create UDTs and their associated database objects.

Note, too, that superficially “identical” values are not necessarily drawn from the same domain. The following dollar amounts, for example, do not refer to the same real world objects:

- \$1.00 USD
- \$1.00 Canadian
- \$1.00 Australian

In the physical design phase, you can further limit column values with constraints imposed on them using the CREATE TABLE or ALTER TABLE statements.

For example, the HR\_Date domain would probably exclude all weekend and holiday dates, and if your company only ships goods on Fridays, then you would place a Fridays-only constraint on the *shipping\_orrdate* column.

## Naming Foreign Key Columns

You should also name foreign key columns to match their names in their primary tables. For example, the foreign key columns in the following relation are not only indicated by the name of their primary table (*order*) but are also demarcated with the characters FK to redundantly indicate their origin in another table.

lineitem		
lineitem_number	order_FK_number	order_FK_year_date
PK	FK	FK

## Metadata Definitions

All domain definitions should be stored in a metadata database and those definitions should be made available to all users of the database.

## ATM Domains Form

As part of the transition from the logical model of your database to its physical implementation, you should record all domains that you identify on a Domains form. See “[Domains Form](#)” on page 138.

# Key Values and Relationships Among Tables

One result of database normalization is the identification of relationships among the tables defined in the database. These relationships are defined on the basis of primary key values shared between tables.

## Example

Consider the following basic case. The *employee* table was defined as a major entity during the logical design phase. Its columns are defined as follows:

employee				
emp_num	emp_last_name	emp_first_name	emp_middle_initial	emp_phone
PK				
...	...	...	...	...

The *employee\_phone* table was defined as a minor entity, or entity subtype, related to the major entity, or entity supertype, *employee* table. Its columns are defined as follows:

employee_phone		
emp_num_FK_employee	emp_phone_number	emp_phone_comment
PK, FK	PK	
...	...	...

Note that the *employee\_phone* table has a foreign key. That foreign key, *emp\_num*, is the primary key of the *employee* table. The two tables relate to one another through this primary key relationship.

In this particular relationship, the minor entity, *employee\_phone*, is said to be the Child table because it references another table. That referenced table, *employee*, is said to be the Parent table in the relationship.

As a sidebar, note that because the referenced column is a primary key, it is by definition uniquely constrained, so it must be defined physically as either of the following two index types.

- Unique primary index, NOT NULL.
- Unique secondary index, NOT NULL.

Such a constraint only needs to be defined physically if the database management system enforces it. If the constraint is defined with a Referential Constraint, Teradata Database does not enforce the uniqueness on the parent table, so a UPI or USI is not required and the constraint is just assumed to be valid.

Certain database semantics derive from such relationships, and those semantics require a particular type of database constraint to maintain their integrity. You cannot create referential constraints between columns typed as Period, XML, BLOB, or CLOB.

This constraint is referred to as a referential constraint and it is said to maintain referential integrity.

You cannot define database constraints of any kind on columns having the XML, BLOB, or CLOB data types. See [Chapter 12: “Designing for Database Integrity”](#) for more information.

See [“The Referential Integrity Rule”](#) on page 95 and [“Domains and Referential Integrity”](#) on page 100 for more information about referential integrity.

## Domains Form

Lists and describes all the domains for a database.

This form supports the first step in the ATM process (see [“Goals”](#) on page 123).

### Information Recorded in the Domains Form

The Domains form records the following information about each domain:

Information Recorded	Description
ELDM Page	Page of the Extended Logical Data Model to which this domain pertains.
System	Name of the system on which this domain is defined.
Domain Name	Name of the domain. This is often the same name as the UDT corresponding to the domain.
Domain Description	Full text description of the domain.

Information Recorded	Description
Data Type	Encoded data type for the domain (see “ <a href="#">Data Type Codes</a> ” on page 139). You must create your own encodings for UDTs if you use them to define your domains.
Max Bytes	Maximum number of bytes a column value drawn from this domain can occupy in disk storage.  Note that the number of bytes per character does not necessarily map 1:1 to the number of characters defined for the column by the defined data type. Multibyte character sets are often represented by multiple bytes per single character, and you must account for this information when filling out the Domains form.
Print Format	Representation of the FORMAT clause (if any) used in the definition of columns drawn from this domain.
Constraint #	Constraint number (as recorded on the Constraints form) that applies to this domain, if any.

## Data Type Codes

You should also maintain a list of UDTs and the codes you assign to them, staying as close to the existing code naming convention as possible.

The following table lists the valid predefined data types and their DBC.TVFields codes.

Data Type	Code
ARRAY (one-dimensional)  VARRAY (one-dimensional)  You cannot use any of the predefined UDT data types as the base type for creating a distinct UDT, so you cannot create a domain using a distinct UDT based on a one-dimensional ARRAY/VARRAY type.	A1
ARRAY (multidimensional)  VARRAY (multidimensional)  You cannot use any of the predefined UDT data types as the base type for creating a distinct UDT, so you cannot create a domain using a distinct UDT based on a multidimensional ARRAY/VARRAY type.	AN
BINARY LARGE OBJECT (BLOB)	BL(n)
BIGINT	I8
BLOB	BO
BYTE	B(n)
BYTEINT	I1
CHARACTER	C(n)

Data Type	Code
CHARACTER VARYING	CV(n) CV(n) is the same as VC(n)
CLOB	CO
CHARACTER LARGE OBJECT (CLOB)	CL(n)
DATE	D
DECIMAL	DEC (n[,m])
DOUBLE PRECISION	DP
FLOAT	F(n)
GRAPHIC	G(n)
INTEGER	I
INTERVAL YEAR	IY
INTERVAL YEAR TO MONTH	IYM
INTERVAL MONTH	IMO
INTERVAL DAY	ID
INTERVAL DAY TO HOUR	IDH
INTERVAL DAY TO MINUTE	IDM
INTERVAL DAY TO SECOND	IDS
INTERVAL HOUR	IH
INTERVAL HOUR TO MINUTE	IHM
INTERVAL HOUR TO SECOND	IHS
INTERVAL MINUTE	IMI
INTERVAL MINUTE TO SECOND	IMS
INTERVAL SECOND	IS
LONG CHARACTER VARYING	LVC
LONG GRAPHIC VARYING	LVG
NUMBER (both exact and approximate forms)	N
NUMERIC	NUM (n[,m])
PERIOD(DATE)  You cannot use any of the predefined Period data types as the base type for creating a distinct UDT, so you cannot create a domain using a distinct UDT based on a PERIOD type.	PD
PERIOD(TIME)	PT

Data Type	Code
PERIOD(TIME WITH TIME ZONE)	PTTZ
PERIOD(TIMESTAMP)	PTS
PERIOD(TIMESTAMP <UNTIL_CHANGED>)  When the ending element value for PERIOD(TIMESTAMP) is UNTIL_CHANGED, the stored value for the ending element is only 1 byte; otherwise it is 10 bytes.	PTS_UC
PERIOD(TIMESTAMP WITH TIME ZONE)  When the ending element value for PERIOD(TIMESTAMP) WITH TIME ZONE is UNTIL_CHANGED, the stored value for the ending element is only 1 byte; otherwise, it is 12 bytes.	PTSTZ
PERIOD(TIMESTAMP WITH TIME ZONE <UNTIL_CHANGED>)	PTSTZ_UC
PERIOD (with derived period columns)	PP
REAL	R
SMALLINT	I2
TIME	T
TIMESTAMP	TS
TIME WITH TIME ZONE	TTZ
TIMESTAMP WITH TIME ZONE	TSTZ
VARBYTE	VB(n)
VARCHAR	VC(n) VC(n) is the same as CV(n)
VARCHAR(n) CHARACTER SET GRAPHIC	VG(n)
XML	XML

You should develop your own data type codes for the UDTs your site uses.

## Example

This example assumes you are using a single byte character set to define the *Addr*, *City*, and *Comment* domains.

Domains  
Page: \_\_\_\_ of: \_\_\_\_ ELDMD Page: \_\_\_\_  
System: \_\_\_\_

Domain Name	Domain Description	Data Type	Max Bytes	Print Format	Constraint Number
Addr	Address	CV(30)	30		
Amt	Dollar Amount	D(10,2)	8	\$\$\$\$\$\$9.99	

Domains Page: ____ of ____		ELDM Page: _____ System: _____			
City	City Name	CV(30)	30		
Comment	Comment	CV(100)	100		
...	...	...	...	...	...

## Constraints Form

Database constraints and validity checks are sometimes loosely referred to as business rules (see “[Semantic Data Integrity Constraints](#)” on page 623, and “[Semantic Constraint Specifications](#)” on page 636). They define conditions that must be met before a given value is permitted to be written to a column such as value ranges, equality or inequality conditions, and intercolumn dependencies. Teradata Database supports constraints at both the column and table levels. Constraints are defined using the CREATE TABLE and ALTER TABLE SQL statements.

**Note:** You cannot define any type of database constraint on columns defined with a UDT, XML, BLOB, or CLOB data type.

Database triggers, too, are often referred to as business rules. They define certain actions that are to be taken when a particular condition occurs during the update of the table on which they are defined.

The Constraints form:

- Lists all constraints, triggers, and validity checks defined for a database.
- Provides input to the Table form.

This form supports the second step in the ATM process (see “[Goals](#)” on page 123).

### Information Recorded on the Constraints Form

The Constraints form records the following information about each constraint, trigger, and validity check.

Information Recorded	Description
ELDM Page	Page of the Extended Logical Data Model to which this constraint pertains.
System	Name of the business system in which the applications using this constraint are defined.
Constraint Number	The number that applies to this constraint. This number is used by other forms as a “foreign key” value to refer to the constraints defined by this form.

Information Recorded	Description
Constraint Description	Full text description of the constraint.

## Constraint Codes

Code	Definition
FK	Foreign key
ID	Identity column
NC	No changes permitted
ND	No duplicates permitted
NN	Not null
PK	Primary key
SA	System-assigned
UA	User-assigned

## Example

Constraints  
Page: \_\_\_ of: \_\_\_      ELDM Page: \_\_\_  
System: \_\_\_

Constraint Number	Constraint Description
001	ND, NN, NC, PK
002	Must be greater than zero (whole positive integer)
003	Mutually exclusive: One required
004	Mutually exclusive: All required
005	No recursion and no loops
006	Excludes Saturdays, Sundays, and legal holidays
007	Prevent delete rule
008	Reassign delete rule
009	Nullify delete rule
010	Cascade delete rule
011	Copy rows to History table before deleting
012	Trigger update to OrderPart Category table on insert to Order or Part tables.
...	...

## System Form

A system is a set of application programs, or applications, grouped by business function. Applications are grouped into systems because each cluster of applications typically serves a different user community within the enterprise.

In terms of ATM, an application is a major business function. Each application is implemented by a set of transactions (see “[Application Form](#)” on page 145).

Ensuring that applications are clustered into the appropriate systems is the most important activity undertaken during this stage of the ATM process. The information described by the System form is generic and applies irrespective of the eventual physical implementation of the database.

The System form has two purposes:

- Identify all applications, grouping them by their business system.
- Provide input to the Application and Table forms.

This form supports the third step in the ATM process (see “[Goals](#)” on page 123).

### Information Recorded

Information Recorded	Description
ELDM Page	Page of the Extended Logical Data Model to which this system pertains.
System Name	Name of the business system in which the applications using this constraint are defined.
System Description	Brief description of the function of the system.
Application ID	Unique identifier for the application being documented.
Application Name	Full text name of the application being documented.
User Contacts	Names and other useful information pertaining to principal users of the application.

### Example

System                    ELDM Page: 1  
                            Page: \_\_\_ of \_\_\_                    System: PTS

System Name:            Part Tracking System (PTS)

System Description:    Tracks current status and history of each part over its lifetime.

System	ELDM Page: 1 Page: ___ of ___	System: PTS
Application ID	Application Name	User Contacts
Cust	Customer tracking	Joy Calder X829
PCat	Part categories	Bill Alpert
Rpt	Parts reporting	Hall Werts
...	...	...
...	...	...
...	...	...

## Application Form

The term *transaction* is used very loosely in the context of the Application form. Any of the following activities equates to a transaction.

- Report
- Single-row SELECT query
- Batch load job

The Application form has two purposes.

- Identify the transactions for each application in the system to estimate run frequencies and volumes.
- Provide input to the physical design of the database.

This form supports the fourth step in the ATM process (see “[Goals](#) on page 123”).

### Estimating Run Frequency and Duration

A typical business year has the following characteristics:

Attribute	Count
Number of business days per week	5
Number of hours per business day	8
Number of business weeks	52
Number of business days per year	260
Number of business hours per year	2,080

For each transaction, estimate the typical and peak run frequencies and identify when the action is performed and the duration of its performance.

The task is to estimate five parameters for each identified transaction. The parameters to be estimated are the following.

- Normal (nonpeak period) run frequency
- Peak period run frequency
- Point in the business cycle when the peak period occurs
- Duration of the peak period

## Example

The following example form models some of the transactions for the Order Tracking application:

Application		ELDM Page: 1		System: PTS			
Application ID: <u>Ord</u>		Application Name: Order Entry_____					
Application Description: <u>Generates and tracks part orders</u>		Run Frequencies					
Transaction ID	Transaction Name	Typical	Peak	When	Length	Total	
OrdCl	Close an order	50/day	75/day	EOQ	2 weeks	14 800	
			100/day	EOY	4 weeks		
OrdCn	Cancel an order	1/quarter	N/A	N/A	N/A	4	
OrdD	Delete closed orders	1/month	N/A	First day of month	N/A	12	
...	...	...	...	...	...	...	

## Report/Query Analysis Form

The Report/Query Analysis form:

- Captures a simple category of data demographics for each individual transaction within an application within a system:
  - Lists the tables accessed during a transaction.
  - Lists columns used for valued access (if any).
  - Lists columns used for join access (if any).
- Provides input to the Table form.

This form supports the fifth and sixth steps in the ATM process (see “[Goals](#)” on page 123).

## Terminology

Term	Definition
Access column	A column name that would be specified in the WHERE clause of a query.  In the following WHERE clause, column_name is the name of the access column:  <code>WHERE column_name = 'xyx'</code>
Join column	A column name that would be specified in the ON clause of a join query.  In the following ON clause, table_1.column_name and table_2.column_name are the names of join columns:  <code>ON table_1.column_name = table_2.column_name</code>

## Procedure

Perform the following procedure to complete the Report/Query Analysis form for each transaction:

- 1 Begin the analysis with standard reports, load requirements, and similar data.
- 2 Identify the tables involved and enter the names of each on a Report/Query Analysis form.
- 3 Identify which columns are printed and enter the names of each on the same Report/Query Analysis form that documents their tables.
- 4 Identify whether rows are accessed by specific column values.  
  
If so, enter the names of each column for which access values must be obtained and mark them with a V.  
  
Note that these must specify equijoin conditions matching on a single value.
- 5 Identify whether tables must be joined.  
  
If so, enter the names of the join columns and mark them with a J.  
  
Note that these must specify equijoin conditions.
- 6 Identify the number of rows that qualify for processing. This is not the number of rows in the answer set, but the number of rows that must be processed in order to obtain the answer set.  
  
Use these guidelines to determine what values to use.
  - a Identify the number of rows, if any, that qualify for value access.
  - b Identify the number of rows, if any, that qualify for join access.
  - c Select the smaller of 1 and 2 and use it as the value for the number of rows processed.
  - d If there are no value access rows and no join access rows, then use the number of rows in the table.

## Examples

There are over 1,000 patients in the hospital database used for these examples. Each patient is hospitalized long-term (where *long-term* means  $\geq$  one year).

### Example 1

The Patient Detail by Room Assignment report derives from the *patient* and *bed* tables. Values and selected demographics for those tables are modeled as follows.

Patient

Cardinality: 1,000

Patient_ID	Bed_Number	Nurse_ID	Doctor_ID
PK	FK, ND	FK	FK
701	A	201	501
702	B	201	502
703	C	202	502
...	...	...	...

Bed

Cardinality: 1,500

Bed_Number	Room_Number
PK, ND	FK
A	101
B	102
C	103
...	...

Patient Detail by Room Assignment

Room_Number	Patient_ID	Bed_Number	Nurse_ID
101	701	A	201
	716	F	233
	723	T	205
102	702	B	201
	725	Q	201
	748	L	222

Patient Detail by Room Assignment

...	...	...	...
-----	-----	-----	-----

Frequency or Importance Ranking: \_\_\_\_\_

Report/Query Description: Patient detail by room assignment

Table: Patient

Number of Output Rows: 1,000

Input Value Or Source	Bed Bed Number	
Access Column(s)	Bed Number	

Table: Bed

Number of Output Rows: 1,500

Input Value Or Source	Patient Bed Number	
Access Column(s)	Bed Number	

Table:

Number of Output Rows:

Input Value Or Source		
Access Column(s)		

Table:

Number of Output Rows:

Input Value Or Source		
Access Column(s)		

## Example 2

The Billing Report for Patient 705 is more complex than the report analyzed in the previous example. The cardinality of the *PatientServiceHistory* table is based on the assumption that

Chapter 6: The Activity Transaction Modeling Process  
 Report/Query Analysis Form

there is one year of service data kept in the history file, each patient receives an average of two services per day, and there are roughly 1,000 patients in the hospital at any time.

The calculation is done as follows.

$$1,000 \text{ patients} * 365 \text{ days} * 2 \text{ services} = 730,000 \text{ rows.}$$

Service		Cardinality: 125
Service_ID	Description	Cost
PK		
801	Appendectomy	\$ 500.00
802	Tonsillectomy	\$ 350.00
803	Anesthesia	\$1,000.00
...	...	...

Patient_Service_History		Cardinality: 730,000
Patient_ID	Service_ID	Timestamp
PK		
FK	FK	
701	801	2007-08-05 11:39
701	803	2007-08-05 23:59
705	801	2007-08-05 8:23
705	802	2007-08-05 13:43
705	803	2007-08-05 18:32
705	814	2007-08-05 23:56
...	...	...

Billing Report for Patient 705, August 2000

Date	Service_ID	Description	Cost
------	------------	-------------	------

**Billing Report for Patient 705, August 2000**

08/05/2007	801	Appendectomy	\$ 500.00
	802	Tonsillectomy	\$ 350.00
	803	Anesthesia	\$1,000.00
	Total for 08/05/2007		\$1,850.00
08/28/2007	814	Face lift	\$3,200.00
	Total for 08/28/2007		\$3,200.00

**Example 3**

This report is based on the same tables as that of the previous example. It reports summary information from those tables for all patients on a particular date. The value for the *date* column is obtained using the EXTRACT function on the Timestamp value for a performed service.

**Billing Report for August 5, 2007**

Patient_ID	Number_of_Services	Total_Cost
701	2	\$ 1,500.00
705	4	\$ 2,100.00
713	1	\$ 320.00
783	12	\$ 2,400.00
795	2	\$ 890.00

**Frequency or Importance Ranking:**

Report/Query Description: Billing report for a specific date

Table: Patient\_Service\_History

Number of Output Rows: 2,000

Input Value Or Source	Current_Date	Service Service_ID
Access Column(s)	Current_Date	Service_ID

Table: Service

Number of Output Rows: 2

Frequency or Importance Ranking:

Input Value Or Source	Patient_Service_History Service_ID	
Access Column(s)	Service_ID	

Table:

Number of Output Rows:

Input Value Or Source		
Access Column(s)		

Table:

Number of Output Rows:

Input Value Or Source		
Access Column(s)		

## Table Form

The Table form:

- Extends the fully attributed table model created through the logical database design by collecting table- and column-level data demographic information and adding it to the model.
- Provides input to the physical design of the database.

This form introduces data demographics to the ATM process. Supporting analysis requires you to determine various cumulants and then to add that data to information derived from the Column Names and Constraints and Report/Query Analysis forms.

The form supports the seventh, eighth, and ninth steps in the ATM process (see “[Goals](#)” on page 123).

### Approaches to Filling Out the Table Form

Because the determination of data demographics for columns that will be treated as multicolumn indexes is far more complex than the determination of similar data for single

column index columns, the sample analysis and the way you fill out the Table form differs slightly for the single column and multicolumn situations.

The following table points to the Table form topics that apply to the two situations:

Topic	Applies to Single Column Data?	Applies to Multicolumn Data?
<a href="#">“Table Form”</a>	Yes	Yes
<a href="#">“Table Form: Basic Information”</a>	Yes	Yes
<a href="#">“Table Form: Column-Level Information”</a>	Yes	Yes
<a href="#">“Table Form: Miscellaneous Column-Level Information”</a>	Yes	Yes
<a href="#">“Table Form: Access Information”</a>	Yes	Yes
<a href="#">“Table Form: Data Demographics for Single-Column Database Objects”</a>	Yes	No
<a href="#">“Maximum and Typical Column Value Frequencies”</a>	Yes	Yes
<a href="#">“Table Form: Data Demographics for Multicolumn Database Objects”</a>	No	Yes

## Information Collected for the Table Forms

Table form information falls into these broad categories:

- Miscellaneous
- Access
- Data demographics

## Miscellaneous Information

- Column name
- Identity column
- Primary key/foreign key
- Constraint number
- Primary index/secondary index

Note the following things about this point.

- This form is filled out during the logical design phase and indexes should not be defined this soon in the process.
- A table might be a nonpartitioned NoPI object, and a table or join index might be a column-partitioned object, in which case it would not have a primary index.
- Sample data

Transcribe this information from the completed Column Names and Constraints form to the Table form.

## Access Information

The following Table form information describes value and join accesses:

- Value access frequency
- Join access frequency
- Join access rows

Transcribe this information from the completed Report/Query Analysis form to the Table form.

## Data Demographics

The following Table form information describes the demographics of your data:

- Distinct values
- Maximum rows/value
- Rows/null
- Typical rows/value
- Change rating

This information applies only to those columns that have been identified by the Report/Query Analysis activity as taking part in value and join accesses.

The Table form introduces data demographics to the ATM process for the first time. The sources for this information are varied and you might have to use a variety of resources to obtain the data.

Details of this activity are provided in “[Table Form: Data Demographics for Single-Column Database Objects](#)” on page 158.

## Filling Out the Table Form

Because the Table form is both complicated and extensive, the next several topics provide a brief overview of its components as well as instructions for how to derive the information requested.

## Table Form Example

This example is the first page of the Table form for the *Patient\_Service\_History* table. A continuation page for this table, which collects data for multicolumn index candidates, is illustrated in “[Table Form: Data Demographics for Multicolumn Database Objects](#)” on page 165.

Table Page: of		ELDM Page: System: Patient Tracking	
Table Name: Patient_Service_History	Table Type: History	Cardinality: 730,000	Data Protection: RAID 5
Column Name	Patient_ID	Service_ID	Timestamp
PK/FK/ID	PK		
	FK	FK	
Constraint Number			
Value Access Frequency	1,000	52	365
Join Access Frequency		232	
Join Access Rows		1,400,000	
Distinct Values	1,000	125	365
Maximum Rows/Value	3,650	15,000	4,000
Rows/Null	0	0	0
Typical Rows/Value	720	5840	2,000
Change Rating	0	0	0
PI/SI			
Sample Data	701 702 703	801 801 802	19920101 19920105 19920105

## Table Form: Basic Information

This topic explains how to fill out the general information requests named at the top of the Table form.

### Sources for the Required Information

The following table points to the sources for the data requested at the top of the Table form.

Information	Source
System	System form
Table name	Entity name from your logical data model
Table type	Entity type from your logical data model
Row count	<p>Varies.</p> <ul style="list-style-type: none"> <li>If an existing electronic database contains the information, query that database for the cardinality of the table in one of two ways.           <ul style="list-style-type: none"> <li>Examine current statistics for the table.</li> <li>Execute a simple SELECT COUNT (*) request on the table.</li> </ul> </li> <li>If no existing database contains the information, consult with your users to make a best guess cardinality estimate based on estimates of known maxima, not averages.</li> </ul>
Data protection	<p>Varied, depending on how your company operates. The following items are all likely sources for this information.</p> <ul style="list-style-type: none"> <li>Users</li> <li>Enterprise data model</li> <li>IT policies and procedures</li> </ul> <p>The following methods for protecting data are available. Note that while you can specify fallback at the table and join index level, RAID protection applies to your entire hardware configuration.</p> <ul style="list-style-type: none"> <li>Fallback</li> <li>Disk I/O integrity checking</li> <li>RAID 1</li> <li>RAID 5</li> <li>RAID S</li> <li>Disk I/O integrity checks</li> </ul>

## Table Form: Column-Level Information

Column-level information derives from several different sources. This topic parses column-level information into three types. Additional topics examine each of those three types in greater detail.

### Column-Level Information Types

Column-level information separates into three categories, as indicated by the following table:

Category	Topic to See for More Detailed Information
Miscellaneous	<a href="#">“Table Form: Miscellaneous Column-Level Information” on page 157</a>
Value and join access	<a href="#">“Table Form: Access Information” on page 158</a>

Category	Topic to See for More Detailed Information
Data demographics	"Table Form: Data Demographics for Single-Column Database Objects" on page 158

## Table Form: Miscellaneous Column-Level Information

This topic explains how to fill out all the column-level information not categorized as either access or data demographics data.

### Sources for the Required Information

The following table points to the sources for the miscellaneous column-level data requested by the Table form.

Information	Source
Column name	Column name from your logical data model.
PK/FK/ID	Primary and foreign key designations from your logical data model. Identity column designation identified by this step.
Constraint number	Constraints form.
PI/SI	Primary and secondary index designations identified by this step. Note the following things about this information. <ul style="list-style-type: none"> <li>• This form is filled out during the logical design phase and you should not define indexes this soon in the process.</li> <li>• If you ignore the previous bullet, be aware that neither indexes nor database constraints can be defined on columns that have any of the following data types. <ul style="list-style-type: none"> <li>• ARRAY/VARRAY</li> <li>• BLOB</li> <li>• CLOB</li> <li>• XML</li> <li>• Geospatial</li> <li>• Period</li> <li>• JSON</li> </ul> </li> <li>• A table might be a non partitioned NoPI object, and a table or join index might be a column-partitioned object, in which case it would not have a primary index.</li> </ul>
Sample data	Sample data from your logical data model.

## Table Form: Access Information

This topic explains how to fill out the column access information requests named in the Table form.

### Sources for the Required Information

The following table points to the sources for the access data requested by the Table form:

Information	Source
Value access frequency	Report/Query Analysis form
Join access frequency	
Join access rows	

## Table Form: Data Demographics for Single-Column Database Objects

This topic explains how to derive column demographics and how to add that information to the Table form.

Capture demographic data only for access and join column sets and primary key column sets.

By collecting this information, you are emulating some of the fundamental data collection activities performed by the COLLECT STATISTICS statement. The data is used for similar purposes as well: to help you to model your physical database in a way that its performance across all systems and applications is optimal.

### Sources for the Required Information

The following table points to the sources for the data demographics requested by the Table form:

Information	Source
Distinct values	<p>Varies.</p> <p>This parameter determines how many unique values (including nulls) exist for the specified column.</p> <ul style="list-style-type: none"> <li>• If an existing electronic database contains the information, query that database for the number of distinct values in the column in one of two ways.           <ul style="list-style-type: none"> <li>• Examine current statistics for the table.</li> <li>• Execute a simple SELECT COUNT DISTINCT request over the required rows.</li> </ul> </li> <li>• If there is no existing electronic database that contains the information, consult with your users to make a best guess estimate.</li> </ul>
Maximum rows/value	<p>Varies.</p> <p>This parameter determines the maximum value for the specified column.</p> <ul style="list-style-type: none"> <li>• If there is an existing electronic database that contains the information, query that database for the value that occurs most frequently in the column in one of two ways.           <ul style="list-style-type: none"> <li>• Examine current statistics for the table.</li> <li>• Execute a simple SELECT MAX <i>column_name</i> request over the required rows.</li> </ul> </li> <li>• If no existing electronic database contains the information, consult with your users to make a best guess estimate basing your estimate on known maxima, not averages.</li> </ul>
Rows/null	<p>This parameter determines how many rows are null for the specified column.</p> <ul style="list-style-type: none"> <li>• If there is an existing electronic database that contains the information, query that database for the number of rows having a null in this in one of two ways.           <ul style="list-style-type: none"> <li>• Examine current statistics for the table.</li> <li>• Execute a simple SELECT COUNT <i>column_name</i> WHERE <i>column_name</i> IS NULL request over the required rows.</li> </ul> </li> <li>• If no existing electronic database contains the information, consult with your users to make a best guess estimate basing your estimate on known maxima, not averages.</li> </ul>
Typical rows/value	<p>This parameter determines a typical number of rows per value for the specified column.</p> <p>The typical number of rows per value for a column is not necessarily the average value, and the frequency can be so widely dispersed that a reasonably typical value cannot be determined.</p> <p>See “<a href="#">Maximum and Typical Column Value Frequencies</a>” on page 160 for examples of how to determine a value for this parameter.</p>

Information	Source
Change rating	<p>Varied, depending on how your company operates. The following items are all likely sources for this information.</p> <ul style="list-style-type: none"><li>• Users</li><li>• Enterprise data model</li><li>• IT policies and procedures</li></ul> <p>Note that the Teradata Index Wizard uses column change ratings as a parameter in determining the candidacy of columns for use as indexes.</p> <p>See the following topics for more information about how the Teradata Index Wizard uses this information:</p> <ul style="list-style-type: none"><li>• <i>SQL Request and Transaction Processing</i>.</li><li>• <i>SQL Data Definition Language</i>.</li></ul> <p>The change rating codes are as follows.</p> <ul style="list-style-type: none"><li>• 0 means the data for this column never changes. Examples include primary key columns and columns that contain historical information.</li><li>• 1 means the data for this column rarely changes.</li><li>• 2 - 8 are user-determined and cover anything not covered by codes 0, 1, and 9.</li><li>• 9 means the data for this column frequently changes.</li></ul>

## Maximum and Typical Column Value Frequencies

This topic compares the concepts of maximum value and typical value and indicates methods for estimating the typical value for a column.

Variables are often distributed in such a way that they have no “typical” value, in which case the average, which represents the maximum likelihood value, is the only reasonable choice.

Each of these case studies examines different demographics for the same column. In each case, the data examined is for a *City* column.

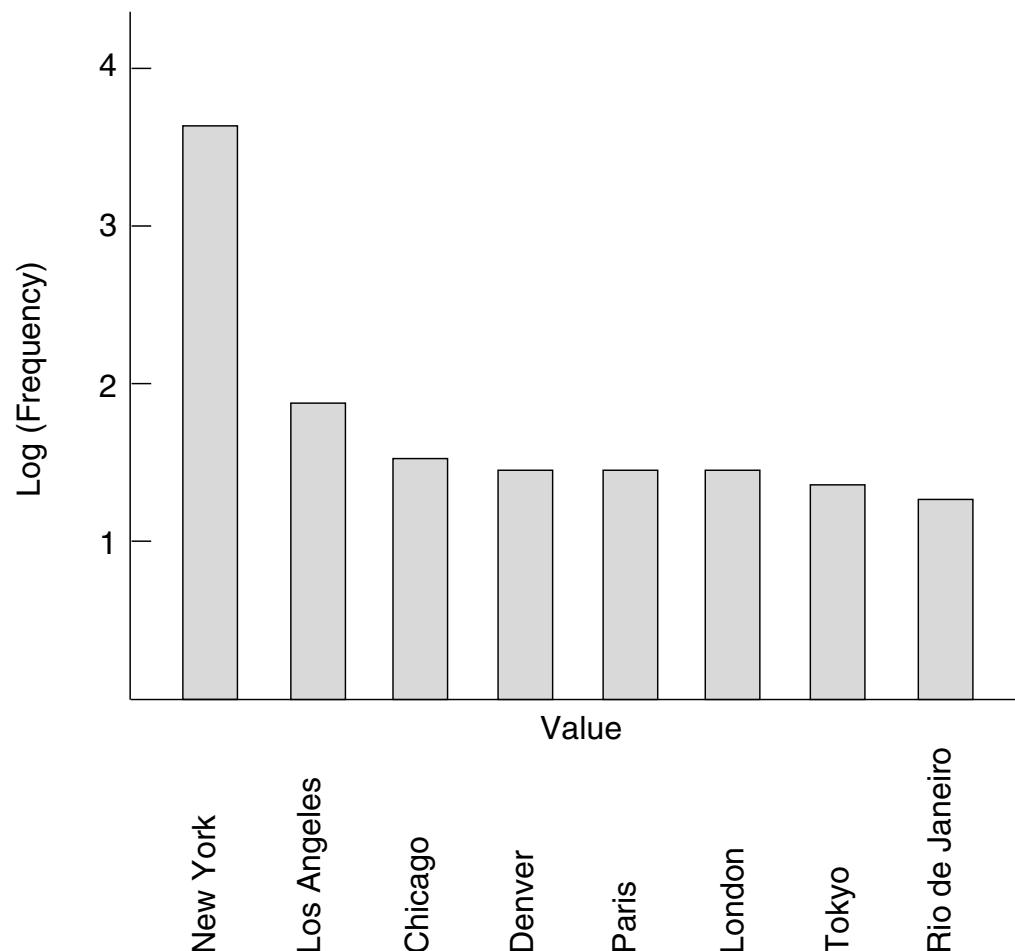
### Case Study 1

The following table indicates the number of distinct occurrences of *City* column values.

City Value	Frequency of Value	Log Frequency of Value
New York	4,000	3.602
Los Angeles	80	1.903
Chicago	35	1.544

City Value	Frequency of Value	Log Frequency of Value
Denver	30	1.477
Paris	30	1.477
London	30	1.477
Tokyo	25	1.398
Rio de Janeiro	20	1.301

The following histogram graphs the logarithm of the number of rows as a function of row values:



FF07D329

The maximum value for this set is 4,000, but what is the typical value?

It is easy to see by visual inspection of the table under “[Case Study 1](#)” on page 160 that the typical value for a variable is about 30. Note that the *average* value for this variable is 531.25, which is by no means typical of the values for the variable.

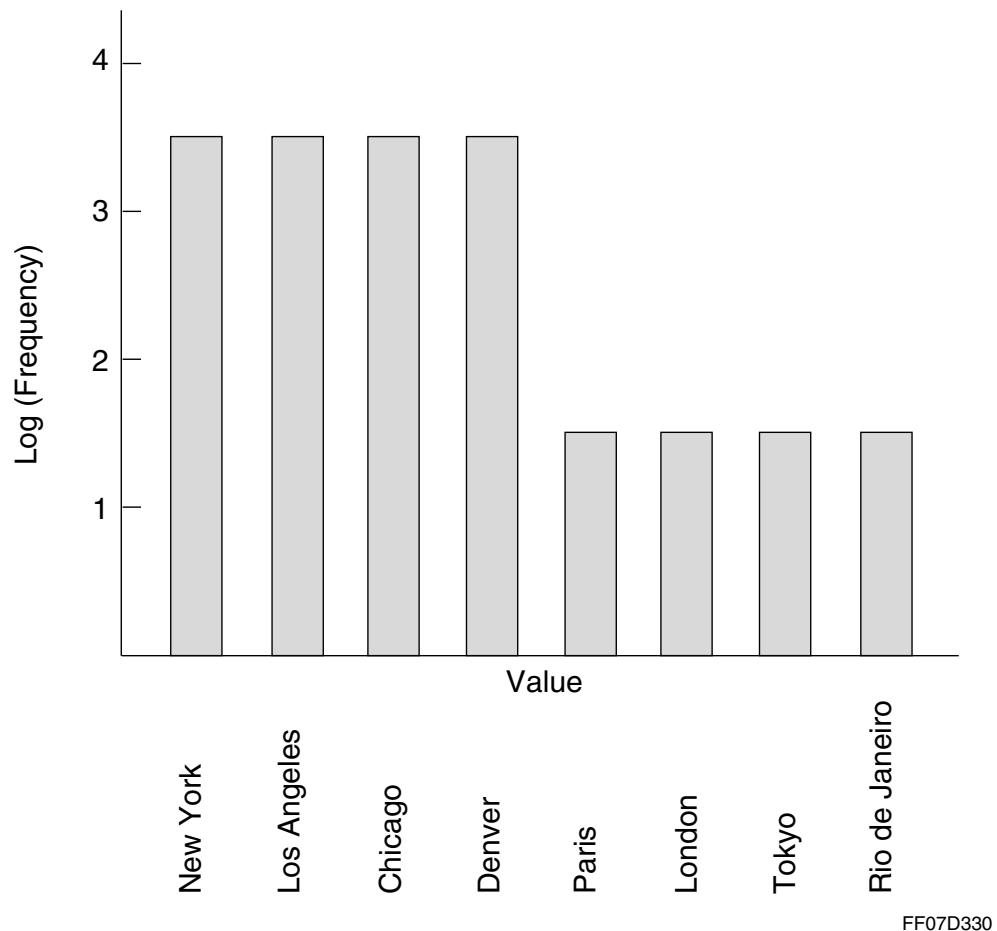
Maximum Value	Typical Value
4,000	30

## Case Study 2

The following table indicates the number of distinct occurrences of *city* column values.

City Value	Frequency of Value	Log Frequency of Value
New York	4,000	3.602
Los Angeles	4,000	3.602
Chicago	4,000	3.602
Denver	4,000	3.602
Paris	30	1.477
London	30	1.477
Tokyo	30	1.477
Rio de Janeiro	30	1.477

The following histogram graphs the logarithm of the number of rows as a function of row values:



The maximum value for this set is 4,000, but what is the typical value?

It is impossible to determine a typical value for the scenario provided by this case history. When you encounter a situation like this, the optimum solution is to use the worst case as your typical value. In this case, that value is 4,000. Note that the *average* value for this variable is 2,015, which is not only not a typical value for the distribution of the variable, it is *never* a value for the variable in this case.

Maximum Value	Typical Value
4,000	4,000

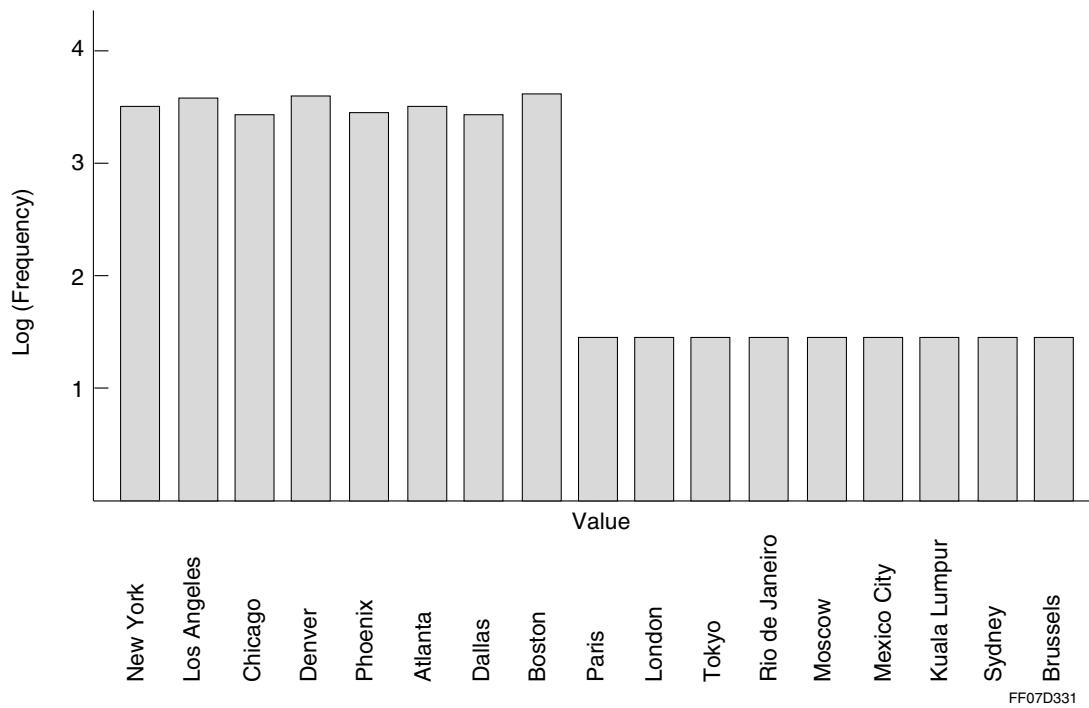
### Case Study 3

The following table indicates the number of distinct occurrences of City column values.

City Value	Frequency of Value	Log Frequency of Value
New York	4,000	3.602

City Value	Frequency of Value	Log Frequency of Value
Los Angeles	4,100	3.613
Chicago	3,800	3.580
Denver	4,200	3.623
Phoenix	3,900	3.591
Atlanta	4,000	3.602
Dallas	3,800	3.580
Boston	4,150	3.618
Paris	30	1.477
London	30	1.477
Tokyo	30	1.477
Rio de Janeiro	30	1.477
Moscow	30	1.477
Mexico City	30	1.477
Kuala Lumpur	30	1.477
Sydney	30	1.477
Brussels	30	1.477

The following histogram graphs the logarithm of the number of rows as a function of row values:



The maximum value for this set is 4,200, but what is the typical value?

It is impossible to determine an accurate “typical” value for the scenario provided by this case history. Like the scenario presented by “[Case Study 2](#)” on page 162, the distribution of values has two peaks at widely diverse points in the distribution. Unlike “[Case Study 2](#)” the value set clustered around the value 4,000 is not constant.

When you encounter a situation like this, the optimum solution is to use a value around which the largest values cluster as your typical value. In this case, that value is 4,000.

Maximum Value	Typical Value
4,200	4,000

## Table Form: Data Demographics for Multicolumn Database Objects

This topic describes how to determine data demographics for multicolumn objects such as composite indexes. Composite indexes are also referred to as multicolumn indexes.

No index of any kind can be based on columns typed as XML, BLOB, or CLOB.

## Sources for the Required Information

The following table points to the sources for the multicolumn data demographics requested by the Table form. With the exception of the information documented here, sources are identical to those documented by “[Sources for the Required Information](#)” on page 158.

Information	Source
Distinct values	See the following topics.
Maximum rows/value	<ul style="list-style-type: none"> <li>• <a href="#">“Calculating Multicolumn Demographics” on page 168</a></li> </ul>
Rows/null	<ul style="list-style-type: none"> <li>• <a href="#">“Example Cases for These Guidelines” on page 168</a></li> </ul>
Typical rows/value	

## Example

This example form, which continues the form illustrated in “[Table Form Example](#)” on page 154 indicates two multicolumn index candidates.

Information acquisition for the second of the two multicolumn specifications is made more simple because the specified columns constitute the primary key for the table.

Because the index candidates are both composite, you cannot use an identity column to define them.

This implies certain demographics by default, as described by the following list. The listed implications hold true irrespective of whether the primary key is uni- or multicolumnar.

- All demographics must be unique.
  - No demographics can be null.
- Therefore, the value for Rows Null is always 0.
- The value for Distinct Values always matches the Cardinality for the table and the value for Maximum Rows/Value is always 1.

Note that the value for Change Rating is also 0 by default because primary key values should never be changed.

Table, continued (Multicolumn index considerations) Page: of		ELDM Page: System: Patient Tracking	
Table Name: Patient_Service_History	Table Type: History	Cardinality: 730,000	Data Protection: RAID 5
Column Name	Patient_ID Service_ID	Patient_ID Service_ID Timestamp	
PK/FK/ID	FK	PK	

Table, continued (Multicolumn index considerations) Page: of		ELDM Page: System: Patient Tracking	
Constraint Number			
Value Access Frequency		120,000	
Join Access Frequency	2,200	300,000	
Join Access Rows	200,000	300,000	
Distinct Values	8,000	730,000	
Maximum Rows/Value	365	1	
Maximum Rows Null	0	0	
Typical Rows/Value	90	1	
Change Rating	0	0	
PI/SI			

## Demographic Trends as a Function of the Number of Columns

Certain characteristics of the demographics of a candidate index change in predictable ways as more columns are added to the index definition.

Selectivity, defined in more detail in [“Selectivity of Indexes and Partitioning” on page 197](#), is a measure of the ability of an index to return a highly discriminating subset of rows from a table. The higher the selectivity, the fewer rows retrieved.

- The number of distinct values increases because each additional index column further enhances the singularity of the row.
- The number of rows per value decreases because the number of values increases proportionately.
- The selectivity of the candidate index increases because any set of index column values points to fewer rows.

The following table summarizes these trends:

Index Columns	Number of Distinct Values	Number of Rows Per Value	Selectivity
State	Least	Most	Lowest

Index Columns	Number of Distinct Values	Number of Rows Per Value	Selectivity
State + Zip Code	•	•	•
State + Zip Code + Last Name	Most	Least	Highest

## Calculating Multicolumn Demographics

You must have an understanding of the underlying data before you can undertake a legitimate calculation of the demographics for multicolumn situations.

The following table provides some calculation guidelines for determining the number of distinct values in a binary (two column) situation. The guidelines scale as more columns are added to the candidate index.

IF distinct column_1 values ...	THEN the total number of distinct values in the two-column index equals ...	ELSE the number of distinct values is ...
can be paired with distinct values from column_2	the product of the number of distinct values in column_1 and the number of distinct values in column_2	approximately equal to the product of the number of distinct values in column_1 and the average number of values in column_2 that can be paired with a single column_1 value.

## Example Cases for These Guidelines

The following examples illustrate the guidelines stated in “[Calculating Multicolumn Demographics](#).”

- Consider the following table fragment. The candidate index for which multicolumn demographics are to be estimated is  $(ProductCode, ColorCode)$ .

	...	ProductCode	ColorCode	...
	...	...	...	...
	...	...	...	...

The product  $(ProductCode) * (ColorCode)$  is true iff every product is available in every color.

- Consider the following table fragment. The candidate index for which multicolumn demographics are to be estimated is  $(StateCode, ZipCode)$ .

	...	StateCode	ZipCode	...
	...	...	...	...
	...	...	...	...

- The product  $(StateCode) * (ZipCode)$  is not an accurate estimate of the number of distinct values in the multicolumn index because the resulting product has fewer distinct values than the product of distinct *StateCode* and distinct *ZipCode* values.
- If you cannot pair every value in the *StateCode* column with every value in the *ZipCode* column, then the number of distinct values in the multicolumn index is approximately equal to (Number of distinct *StateCode* values) \* (Average number of *ZipCode* values that can be paired with a single *StateCode* value).

For example, the number of distinct values is the product of the number of states in the US and the average number of zip codes per state, which we estimate to be 20.

Therefore, the number of distinct values for the *StateCode + ZipCode* index would be roughly  $50 * 20$ , or 1,000.

- The same considerations hold most of the other rows per value calculations.

For example, if every value of *column\_1* can be paired validly with every value for *column\_2*, then the result of the division is a fairly accurate estimate.

For each value estimated, the minimum is never less than the number of *column\_1* rows per value divided by the number of distinct *column\_2* values.

The exception is the count of Rows Null (labeled Rows/Null). See “[Treating the Rows Null Counts for Composite Indexes](#)” on page 169 for information on handling multicolumn indexes that are wholly or partially null.

## Treating the Rows Null Counts for Composite Indexes

An index can be partly or wholly null. This is almost never a good idea, however. See [Chapter 13: “Designing for Missing Information”](#) for information about SQL nulls and the many problems they cause with database integrity.

Consider the following example:

employee_number	department_number
PK	
FK	FK
103794	7012
?	7012
105378	?
?	?

- The first row has both *employee* and *department* numbers.
- The second row has a null *employee* number and a *department* number.
- The third row has an *employee* number and a null *department* number.
- The fourth row has nulls for both its *employee* number and its *department* number.

All four rows are valid primary index rows. Should they all contribute to the count for the Rows Null value or should they be treated differently?

**Note:** Only the first of these 4 rows is a valid primary key because by the entity integrity rule (see “[Rules for Primary Keys](#)” on page 92), a primary key cannot contain nulls by definition. This is only one of the many reasons that indexes should not be an issue during logical database design.

The following table provides recommended treatment guidelines for partially and wholly null composite indexes. Composite indexes are also referred to as multicolumn indexes.

IF a composite index value is ...	THEN you should ...
wholly null	add it to the Rows Null count.
partially null	<i>not</i> add it to the Rows Null count.

The Optimizer can detect the number of distinct values from the rows that are partially null and it tracks them separately from the rows that are wholly null. If a column has many nulls, but is not usually specified in predicates, you probably should not include that column in the multicolumn statistics you collect.

For example, suppose you are considering collecting multicolumn statistics on columns 1, 2, and 3 of a table, and their values are like the following, where a QUESTION MARK character represents a null.

<u>column 1</u>	<u>column 2</u>	<u>column 3</u>
1	2	3
?	2	3
1	?	3
1	2	?
?	?	?
1	2	?
1	2	?
1	2	?
1	2	?

You would probably not want to collect multicolumn statistics that include column 3 unless it is specified in a large number of predicates.

## Row Size Calculation Form

The Row Size Calculation form is used to estimate the size of your tables in bytes. See [Chapter 14: “Database-Level Capacity Planning Considerations”](#) for details about computing row sizes.

**Note:** This estimate is of the raw row size without considering all the aspects of physical database design that can affect table size both positively and negatively.

## Sources for the Required Information

The following table points to the sources for the miscellaneous column-level data requested by the Row Size Calculation form.

Information	Source
Column name	Column name from your logical data model.
Type	Table form reference to the Constraint form.
Max	See “ <a href="#">Data Type Considerations</a> ” on page 822.
Avg	For variable length types, must be computed; otherwise, use the value for <i>Max</i> .

## Things To Consider When Filling Out the Row Size Calculation Form

Keep the following things in mind as you fill out the Row Size Calculation form:

- The size of the row ID and row header depends on the table partitioning.

Table	Row ID Size (bytes)	Row Header Size (bytes)
not partitioned	8	12
up to 15 partition levels (65,535 combined partitions)  2-bytes used for partition level number in row ID portion of row header	10	14
up to 62 partition levels (more than 65,535 combined partitions)  8-bytes used for partition level number in row ID portion of row header	16	20

These values are true whether the table has a primary index or not.

- BLOB, CLOB, and XML column values are stored outside the row.  
  
The value you should enter for BLOB, CLOB, and XML columns in the Row Size Calculation form is their Object Identifier, which is always 39 bytes.  
  
The size of each XML, BLOB, or CLOB subtable must be calculated separately (see [Chapter 14: “Database-Level Capacity Planning Considerations”](#) for details).
- The following system-derived columns consume the following amount of space per row, respectively. Note that the first ROWID can only occur in a join index, and the second two only in result rows.

System-Derived Column Name	Data Type	Number of Bytes Stored for the Internal Information From Which They Are Derived
ROWID	BYTE	<ul style="list-style-type: none"> <li>• 10 for 2-byte partitioning or no partitioning</li> <li>• 16 for 8-byte partitioning</li> </ul>
PARTITION	INTEGER	4 for 2-byte partitioning
	BIGINT	8 for 8-byte partitioning
PARTITION#Ln	INTEGER	4 for 2-byte partitioning
	BIGINT	8 for 8-byte partitioning

- The following system-generated columns consume the following amount of space per row, respectively.

System-Generated Column Type	Data Type	Number of Bytes Stored
Object Identifier (OID)	VARBYTE	39
Identity	<p>Any of the following</p> <ul style="list-style-type: none"> <li>• BIGINT</li> <li>• BYTEINT</li> <li>• DECIMAL(<math>n,0</math>)</li> <li>• INTEGER</li> <li>• NUMERIC(<math>n,0</math>)</li> <li>• NUMBER(<math>n,0</math>)</li> </ul> <p>You can only use the fixed form of NUMBER for identity columns. The floating form is not valid.</p> <ul style="list-style-type: none"> <li>• SMALLINT</li> </ul>	<p>1 - 8, depending on the data type.</p> <p>This is true even when the DBS Control parameter MaxDecimal is set to 38, because the values generated by an identity column are restricted to a maximum of DECIMAL(18,0) NUMERIC(18,0) or NUMBER(18,0).</p>

- Compressed values and nulls have no effect on the length of their field in a column (they are “stored” as 0 bytes for the field), but can add to the row overhead because of their effect on the number of presence bits in the row header (for details, see [“Presence Bits” on page 780](#)).

Be aware that each compressed value consumes space in the table header (for details, see [“Determining How Much Table Header Space is Used for Compression” on page 711](#)).

- The ending timestamp values for any PERIOD(TIMESTAMP) or PERIOD(TIMESTAMP WITH TIME ZONE) values stored with an UNTIL\_CHANGED ending element value consume only 1 byte of storage for their ending element, while PERIOD(TIMESTAMP) and PERIOD(TIMESTAMP WITH TIME ZONE) values stored with a specified ending element value consume 20 and 24 bytes of storage, respectively, for their ending element.

## Procedure

Perform the following procedure to complete the Row Size Calculation form for each table.

- 1 Identify each column in the table and enter its name in the *Column Name* column.
- 2 Identify the data type for each column and enter it in the *Type* column next to the column name.

IF the type is ...	THEN ...
predefined	use the sizing factors on the right-hand side of the form to determine the size of values stored for each data type.
BLOB, CLOB, or XML	enter 39 bytes for the value of its OID. BLOB, CLOB, and XML values are stored outside of the row in their own subtables, which also must be accounted for in your capacity planning.
a distinct, structured, or ARRAY UDT	use the back of the form to tally its size. When you have identified and tallied all UDT columns in the table, copy their sum into the UDTs item in the sizing factors column on the right-hand side of the form.

- 3 Determine the maximum value of the specified data type for each column in the table and enter the value in the *Max* column.
- 4 Compute the average size of the specified data type for each column in the table and enter the value in the *Avg* column.
- 5 Enter the total physical row size, rounded up to an even number of bytes, as *Physical Size*.

Chapter 6: The Activity Transaction Modeling Process  
Row Size Calculation Form

# CHAPTER 7 Denormalizing the Physical Schema

---

This chapter describes some of the common ways to denormalize the physical implementation of a fully-normalized model. The chapter also briefly describing the popular technique of dimensional modeling and shows how the most useful attributes of a dimensional model can be emulated for a fully-normalized or partially-normalized physical database implementation through the careful use of dimensional views.

The term denormalization describes any number of physical implementation techniques that enhance performance by reducing or eliminating the isomorphic mapping of the logical database design on the physical implementation of that design. The result of these operations is usually a violation of the design goal of making databases application-neutral. In other words, a “denormalized” database favors one or a few applications at the expense of all other possible applications.

Strictly speaking, these operations are not denormalization at all. The concept of database schema normalization is logical, not physical. Logical denormalization should be avoided. Develop a fully-normalized design and then, if necessary, adjust the semantic layer of your physical implementation to provide the desired performance enhancement. Finally, use views to tailor the external schema to the usability needs of users and to limit their direct access to base tables (see “[Dimensional Views](#)” on page 186).

## Denormalization, Data Marts, and Data Warehouses

The following quotation is taken from the web site of Bill Inmon, who coined the term *data warehousing*. It supports the position argued here: the more general the analyses undertaken on the warehouse data store, the more important the requirement that the data be normalized. The audience size issue he raises is a reflection of the diversity of analysis anticipated and the need to support any and all potential explorations of the data.

“The generic data model represents a logical structuring of data. Depending on whether the modeler is building the model for a data mart or a data warehouse the data modeler will wish to engage in some degree of denormalization. Denormalization of the logical data model serves the purpose of making the data more efficient to access. In the case of a data mart, a high degree of denormalization can be practiced. In the case of a data warehouse a low degree of denormalization is in order.

“The degree of denormalization that is applicable is a function of how many people are being served. The smaller the audience being served, the greater the degree of denormalization. The larger the audience being served, the lower the degree of denormalization.”

## Denormalization Issues

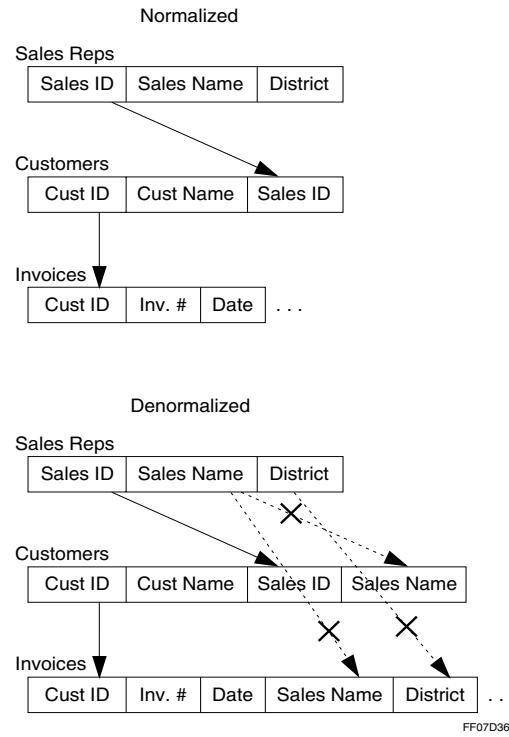
The effects of denormalization on database performance are unpredictable: as many applications can be affected negatively by denormalization as are optimized. If you decide to denormalize your database, make sure you always complete your normalized logical model first. Document the pure logical model and keep your documentation of the physical model current as well.

Denormalize the implementation of the logical model only after you have thoroughly analyzed the costs and benefits, and only after you have completed a normalized logical design.

Consider the following list of effects of denormalization before you decide to undertake design changes:

- A denormalized physical implementation can increase hardware costs.  
The rows of a denormalized table are always wider than the rows of a fully normalized table. A row cannot span data blocks; therefore, there is a high probability that you will be forced to use a larger data block size for a denormalized table. The greater the degree of a table, the larger the impact on storage space. This impact can be severe in many cases.  
Row width also affects the transfer rate for all I/O operations; not just for disk access, but also for transmission across the BYNET and to the requesting client.
- While denormalization benefits the applications it is specifically designed to enhance, it often decreases the performance of other applications, thus contravening the goal of maintaining application neutrality for the database.
- A corollary to this observation is the fact that a denormalized database makes it more difficult to implement new, high-performing applications unless the new applications rely on the same denormalized schema components as existing applications.
- Because of the previous two effects, denormalization often increases the cost and complexity of programming.
- Denormalization introduces update anomalies to the database. Remember that the original impetus behind normalization theory was to eliminate update anomalies.

The following graphic uses a simple database to illustrate some common problems encountered with denormalization:



Consider the denormalized schema. Notice that the name of the salesman has been duplicated in the Customers and Invoices tables in addition to being in the Sales Reps table, which is its sole location in the normalized form of the database.

This particular denormalization has all of the following impacts:

- When a sales person is reassigned to a different customer, then all accounts represented by that individual must be updated, either individually or by reloading the table with the new sales person added to the accounts in place of the former representative.  
Because the Customers, or account, table is relatively small (fewer than a million rows), either method of updating it is a minor cost in most cases.
- At the same time, because the ID and name for the sales person also appear in every Invoice transaction for the account, each transaction in the database must also be updated with the information for the new sales person. This update would probably touch many millions of rows in the Invoice table, and even a reload of the table could easily take several days to complete. This is a very costly operation from any perspective.
- Denormalized rows are always wider rows. The greater the degree of a table, the larger the impact on storage space. This impact can be severe in many cases.

Row width also affects the transfer rate for all I/O operations; not just for disk access, but also for transmission across the BYNET and to the requesting client.

Evaluate all these factors carefully before you decide to denormalize large tables. Smaller tables can be denormalized with fewer penalties in those cases where the denormalization significantly improves the performance of frequently performed queries.

## Commonly Performed Denormalizations

The following items are typical of the denormalizations that can sometimes be exploited to optimize performance:

- Repeating groups
- Prejoins
- Derived data (fields) and summary tables (column aggregations)

## Alternatives to Denormalization

Teradata continues to introduce functions and facilities that permit you to achieve the performance benefits of denormalization while running under a direct physical implementation of your fully normalized logical model.

Among the available alternatives are the following:

- Views
- Hash and join indexes
- Aggregate join indexes
- Global temporary tables

## Denormalizing With Repeating Groups

Repeating groups are attributes of a non-1NF relation that would be converted to individual tuples in a normalized relation.

### Example

For example, this relation has six attributes of sales amounts, one for each of the past six months:

Sales_History						
EmpNum	Sales Figures for Last 6 Months (US Dollars)					
	Sales	Sales	Sales	Sales	Sales	Sales
	PK					
	FK					
	UPI					
	2518	32,389	21,405	18,200	27,590	29,785
						35,710

When normalized, the Sales History relation has six tuples that correspond to the same six months of sales expressed by the denormalized relation:

**Sales\_History**

EmpNum	SalesPeriod	SalesAmount (US Dollars)
PK		
FK		
NUPI		
2518	20011031	32,389
2518	20011130	21,405
2518	20011231	18,200
2518	20010131	27,590
2518	20010228	29,785
2518	20010331	35,710

## Reasons to Denormalize With Repeating Groups

The following items are all possible reasons for denormalizing with repeating groups:

- Saves disk space
- Reduces query and load time
- Makes comparisons among values within the repeating group easier
- Many 3GLs and third party query tools work well with this structure

## Reasons Not to Denormalize With Repeating Groups

The following items all mitigate the use of repeating groups:

- Makes it difficult to detect which month an attribute corresponds to
- Makes it impossible to compare periods other than months
- Changing the number of columns requires both DDL and application modifications

## Denormalizing Through Prejoins

A prejoin moves frequently joined attributes to the same base relation in order to eliminate join processing. Some vendors refer to prejoins as materialized views.

## Example

The following example first indicates two normalized relations, Job and Employee, and then shows how the attributes of the minor relation Job can be carried to the parent relation Employee in order to enhance join processing:

Job		Employee		
JobCode	JobDesc	EmpNum	EmpName	JobCode
PK	NN, ND	PK, SA		FK
UPI		UPI		
1015	Programmer	22416	Jones	1023
1023	Analyst	30547	Smith	1015

This is the denormalized, prejoin form of the same data. This relation violates 2NF:

Employee			
EmpNum	EmpName	JobCode	JobDesc
PK, SA		FK	
UPI			
22416	Jones	1023	Analyst
30547	Smith	1015	Programmer

## Reasons to Denormalize Using Prejoins

The following items are all possible reasons for denormalizing with prejoins:

- Performance can be enhanced significantly.
- The method is a good way to handle situations where there are tables having fewer rows than there are AMPs in the configuration.
- The minor entity is retained in the prejoin so anomalies are avoided and data consistency is maintained.

## Reasons Not to Denormalize Using Prejoins

You can achieve the same results obtained with prejoins without denormalizing your database schema by using any of the following methods:

- Views with joins (see “[Denormalizing Through Views](#)” on page 185)
- Join indexes (see “[Denormalizing Through Join Indexes](#)” on page 181)
- Global temporary tables (see “[Denormalizing Through Global Temporary and Volatile Tables](#)” on page 183)

## Denormalizing Through Join Indexes

Join indexes provide the performance benefits of prejoin tables without incurring update anomalies and without denormalizing your logical or physical database schemas.

Although join indexes create and manage prejoins and, optionally, aggregates, they do not denormalize the physical implementation of your normalized logical model because they are not a component of the fully normalized physical model.

Remember: normalization is a logical concept, not a physical concept.

### Example

Consider the prejoin example in “[Denormalizing Through Prejoins](#)” on page 179. You can obtain the same performance benefits this denormalization offers without incurring any of its negative effects by creating a join index.

```
CREATE JOIN INDEX EmployeeJob
    AS SELECT (JobCode, JobDescription), (EmployeeNumber, EmployeeName)
    FROM Job JOIN Employee ON JobCode;
```

This join index not only eliminates the possibility for update anomalies, it also reduces storage by row compressing redundant *Job* table information.

### Reasons to Denormalize Using Join Indexes

The following items are all reasons to use join indexes to “denormalize” your database by optimizing join and aggregate processing:

- Update anomalies are eliminated because the system handles all updates to the join index for you, ensuring the integrity of your database.
- Aggregates are also supported for join indexes and can be used to replace base summary tables.

### Related Topic

For a full description of join indexes, see [Chapter 11: “Join and Hash Indexes”](#).

## Derived Data Attributes

Derived attributes are attributes that are not atomic. Their data can be derived from atomic attributes in the database. Because they are not atomic, they violate the rules of normalization.

Derived attributes fall into these basic types:

- Summary (aggregate) data
- Data that can be directly derived from other attributes

## Approaches to Handling Standalone Derived Data

There are occasions when you might want to denormalize standalone calculations for performance reasons. Base the decision to denormalize on the following demographic information, all of which is derived through the ATM process described in [Chapter 6: “The Activity Transaction Modeling Process”](#):

- Number of tables and rows involved
- Access frequency
- Data volatility
- Data change schedule

## Guidelines for Handling Standalone Derived Data

As a general rule, using an aggregate join index or a global temporary table is preferable to denormalizing the physical implementation of the fully normalized logical model.

The following table provides guidelines on handling standalone derived data attributes by denormalization. The decisions are all based on the demographics of the particular data.

When more than one recommended approach is given, and one is preferable to the other, the entries are ranked in order of preference.

Access Frequency	Change Rating	Update Frequency	Recommended Approach
High	High	Dynamic	<ol style="list-style-type: none"><li>1 Use an aggregate join index or global temporary table.</li><li>2 Denormalize the physical implementation of the model.</li></ol>
High	High	Scheduled	Use an aggregate join index or global temporary table.
High	Low	Dynamic	Use an aggregate join index or global temporary table.
High	Low	Scheduled	<ul style="list-style-type: none"><li>• Use an aggregate join index or global temporary table.</li><li>• Produce a batch report that calculates the aggregates whenever it is run.</li></ul>
Low	Unknown	Unknown	Calculate the information on demand rather than storing it in the database.

Any time the number of tables and rows involved is small, calculate the derived information on demand.

## Reasons Not to Denormalize Using Derived Data

The following items deal with the issues of derived data without denormalizing user base data tables:

- Aggregate join index (see “[Aggregate Join Indexes](#)” on page 552)
- Global temporary table with derived column definitions
- View with derived column definitions

## Denormalizing Through Global Temporary and Volatile Tables

Global temporary tables have a persistent stored definition just like any base table. The difference is that a global temporary table is materialized only when it is accessed by a DML request for the first time in a session and then remains materialized for the duration of the session unless explicitly dropped. At the close of the session, all rows in the table are dropped. Keep in mind that the containing database or user for a global temporary table must have a *minimum* of 512 bytes of PERM space per AMP in order to contain the table header. This means that the minimum amount of permanent space per global temporary table for the database is 512 bytes for each times the number of AMPS on your system.

Analogously, volatile tables can have a persistent stored definition if that definition is contained within a macro. When used in this manner, the properties of global temporary and volatile tables are largely identical in regard to persistence of the definition (see “[CREATE TABLE](#)” in *SQL Data Definition Language Detailed Topics* for other distinctions and differences).

Global temporary tables, like join and hash indexes, are not part of the logical model. Because of this, they can be denormalized to any degree desired, enhancing the performance of targeted applications without affecting the physically implemented normalization of the underlying database schema. The logical model is not affected, but all the benefits of physical schema denormalization are accrued.

It is important to remember that a materialized instance of a global temporary table and a volatile table are local to the session from which they are materialized or created, and only that session can access its materialized instance.

This also means that multiple sessions can simultaneously materialize instances of a global temporary table definition (or volatile tables) that are private to those sessions.

### Using Global Temporary Tables and Volatile Tables to Avoid Denormalization

You can use global temporary and volatile tables to avoid the following denormalizations you might otherwise consider:

- Prejoins
- Summary tables and other derived data

This final point is important as an alternative for applications that do not require persistent storage of summary results as offered, for example, by aggregate join indexes.

## Using Global Temporary and Volatile Tables to Enhance Performance

You can use global temporary tables to enhance performance in the following ways:

- Simplify application code
- Reduce spool usage
- Eliminate large numbers of joins

This final point is important as an alternative for applications that do not require persistent storage of prejoin results as offered, for example, by join indexes.

### Example: Simple Denormalization for Batch Processing

The following global temporary table serves 500 different transactions that create the output it defines. These transactions collectively run over one million times per year, but 95% of them run only on a monthly batch schedule.

With the following table definition stored in the dictionary, the table itself, which violates 2NF, is materialized only when one of those batch transactions accesses it for the first time in a session:

TemporaryBatchOutput

DeptNum	EmpNum	DeptName	LastName	FirstName
PK				
FK	FK			
...	...	...	...	...

### Example: Aggregate Summary Table

The following global temporary table definition, if used infrequently and is not shared, might be an alternative to using an aggregate join index to define the equivalent summary table:

DepartmentAggregations

DeptNum	Period	SumSalary	AvgSalary	EmpCount
PK				
FK				
NUPI				
...	...	...	...	...

## Example: Prejoin

Prejoins are a form of derived relationship among tables. The following table definition, if used infrequently, might be an alternative to using a join index to define the equivalent prejoin table.

This particular table saves the cost of having to join the *Order*, *Location*, and *Customer* tables:

OrderCustomer		OrdCost
OrdNum	CustNum	
PK	FK	
FK	NUPI	
...		...

## Denormalizing Through Views

You cannot denormalize a physical database using views, though views can be used to provide the *appearance* of denormalizing base relations without actually implementing the apparent denormalizations they simulate.

Denormalized views can be a particularly useful solution to the conflicting goals of dimensional and normalized models because it is possible to maintain a fully-normalized physical database while at the same time presenting a virtual multidimensional database to users through the use of a semantic layer based on dimensional views (see “[Dimensional Views](#)” on page 186 and “[Design for Flexible Access Using Views](#)” on page 121).

### Prejoin With Aggregation

The following example creates a prejoin view with aggregation. Note that you can create a functionally identical object as a join index.

```

REPLACE VIEW LargeTableSpaceTotal
  (DBname,Acctname,Tabname,CurrentPermSum,PeakPermSum,      NumVprocs)
AS SELECT DatabaseName,AccountName,TableName,
SUM (CurrentPerm)(FORMAT '---,---,---,--9'),
SUM (PeakPerm)(FORMAT '---,---,---,--9'),
COUNT(*) (FORMAT 'ZZ9')
FROM DBC.TablesizeV
GROUP BY 1, 2, 3
HAVING SUM (currentperm) > 10E9;

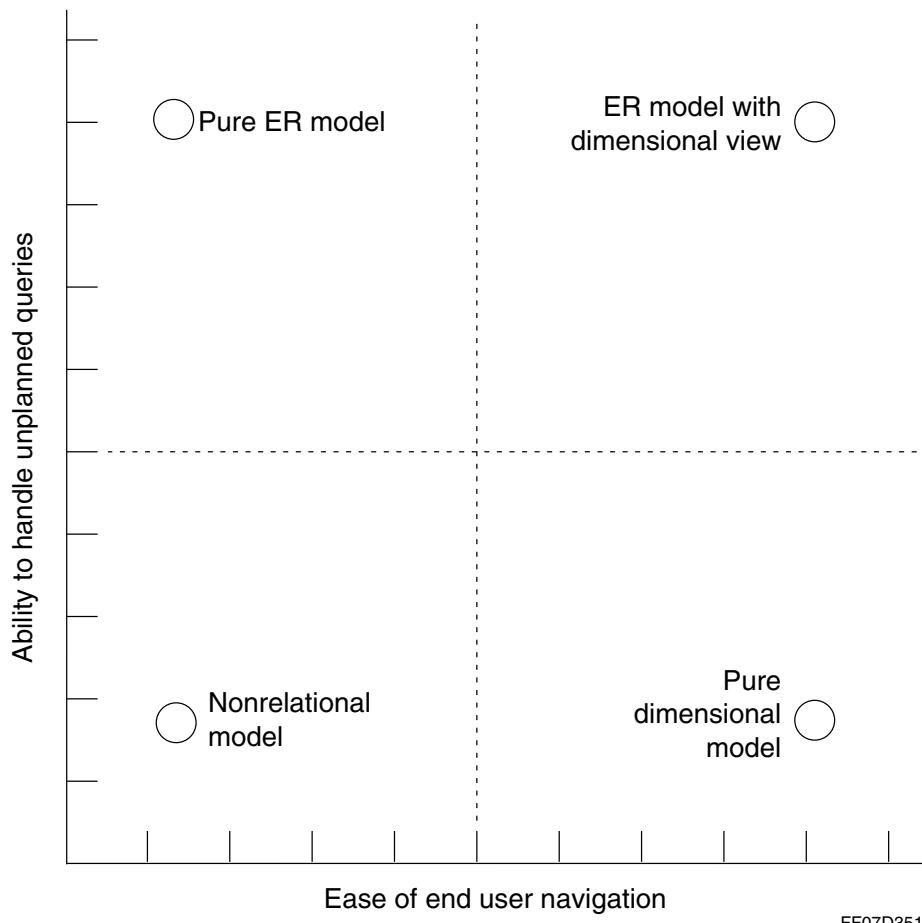
SELECT DatabaseName (CHAR(10), TITLE 'DbName'),
AccountName (CHAR(10),TITLE 'AcctName'),
TableName (CHAR(16),TITLE 'TableName'), Vproc,
CurrentPerm (FORMAT '---,---,---,--9'),
CurrentPerm * 100.0 / CurrentpermSum (AS PctDist, TITLE ' % // '
Distrib',FORMAT 'ZZ9.999'),PctDist * NumVprocs
(AS PctofAvg,TITLE '% of//AVG ', FORMAT 'ZZ9.9')
```

```
FROM LargeTableSpaceTotal, DBC.TablesizeV
WHERE DBname    = TablesizeV.DatabaseName
AND   AcctName  = TablesizeV.AccountName
AND   TabName   = TablesizeV.TableName
AND   PctofAvg > 125.0
ORDER BY 1, 2, 3, 4;
```

## Dimensional Views

A dimensional view is a virtual star or snowflake schema layered over detail data maintained in fully-normalized base tables. Not only does such a view provide high performance, but it does so without incurring the update anomalies caused by a physically denormalized database schema.

The following illustration, adapted from a report by the Hurwitz Group (1999), graphs the capability of various data modeling approaches to solve ad hoc and data mining queries as a function of ease-of-navigation of the database. As you can see, a dimensional view of a normalized database schema optimizes both the capability of the database to handle ad hoc queries and the navigational ease of use desired by many end users.



FF07D351

Many third party reporting and query tools are designed to access data that has been configured in a star schema (see “[Dimensional Modeling, Star, and Snowflake Schemas](#)” on

[page 187](#)). Dimensional views combine the strengths of the E-R and dimensional models by providing the interface for which these reporting and query tools are optimized.

Access to the data through standard applications, or by unsophisticated end users, can also be accomplished by means of dimensional views. More sophisticated applications, such as ad hoc tactical and strategic queries and data mining explorations can analyze the normalized data either directly or by means of views on the normalized database.

The following procedure outlines a hybrid methodology for developing dimensional views in the context of traditional database design techniques:

1 Develop parallel logical database models.

It makes no difference which model is developed first, nor does it make a difference if the two models are developed in parallel. The order of steps in the following procedure is arbitrary:

- a Develop an enterprise E-R model.
- b Develop an enterprise DM model.
- 2 Develop an enterprise physical model based on the E-R model developed in step 1.
- 3 Implement the physical model designed in step 2.
- 4 Implement dimensional views to emulate the enterprise DM model developed in step 1 as desired.

Several Teradata customers use this hybrid methodology to provide a high-performing, flexible design that benefits data manipulation while simultaneously being user- and third-party-tool friendly.

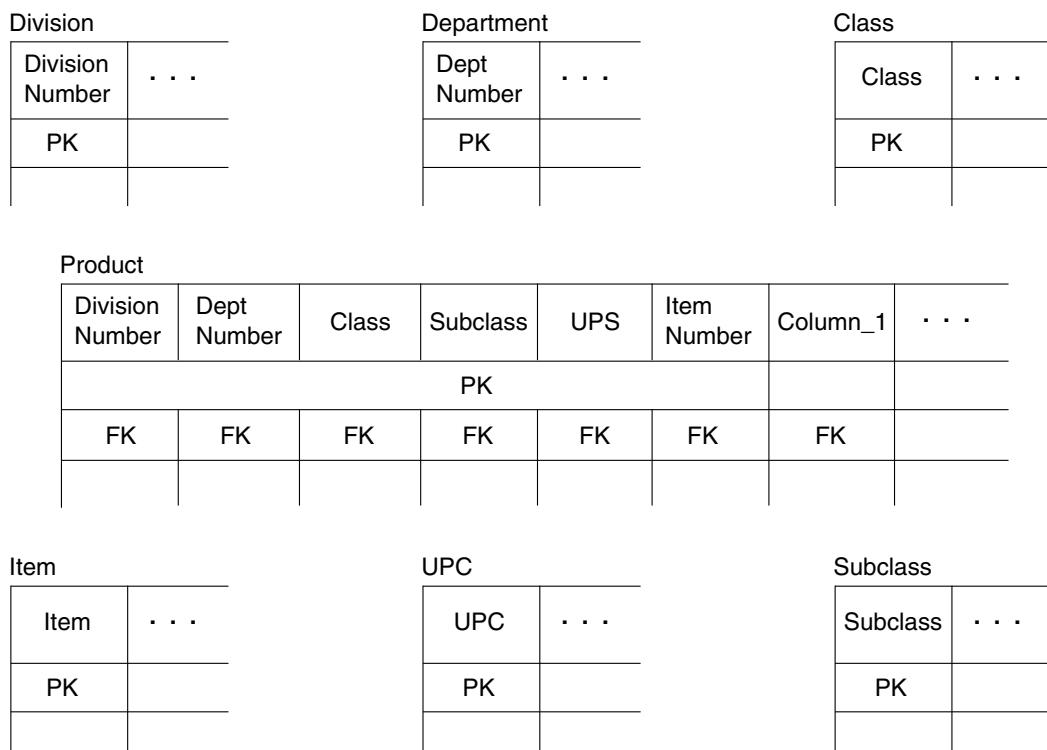
Martyn (2004) examines dimensional views from a research-oriented perspective and concludes that dimensional views are an optimal means for overcoming the objections to normalized databases *visa-a-vis* DM models.

## Dimensional Modeling, Star, and Snowflake Schemas

### Definition of Dimensional Modeling

According to Ralph Kimball, the creator of the dimensional modeling methodology, “DM is a logical design technique that seeks to present the data in a standard, intuitive framework that allows for high-performance access. It is inherently dimensional, and it adheres to a discipline that uses the relational model with some important restrictions. Every dimensional model is composed of one table with a multipart key, called the fact table, and a set of smaller tables called dimension tables. Each dimension table has a single-part primary key that corresponds exactly to one of the components of the multipart key in the fact table” (Kimball, 1997).

The graphic indicates a simplified example of a fact table (Product) and its associated dimension tables (Division, Department, Class, Item, UPC, and Subclass).



FF07D352

## Fact Tables and Dimension Tables

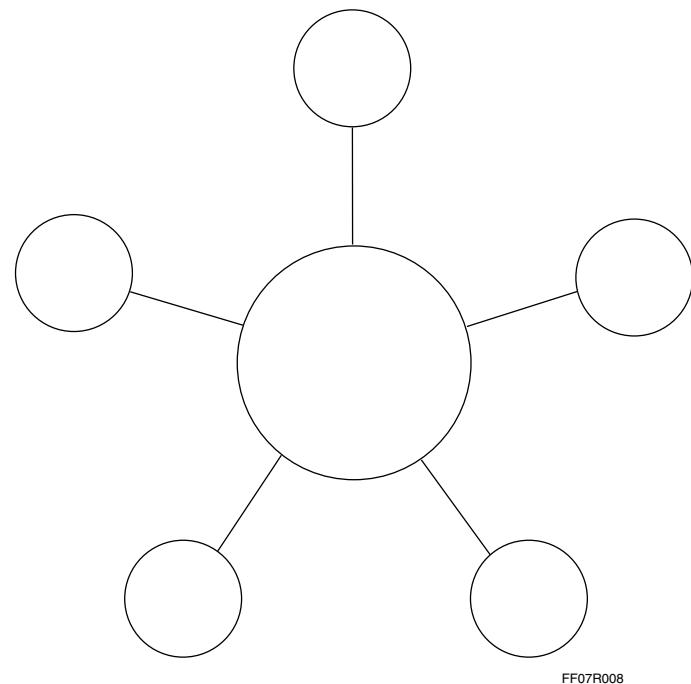
The structure of a dimension model somewhat resembles that of a crude drawing of a star or snowflake (see the graphics “[Star Schema](#)” on page 188 and “[Snowflake Schema](#)” on page 189).

In a dimensional model, fact tables always represent M:M relationships (see “[Many-to-Many Relationships](#)” on page 70). According to the model, a fact table should contain one or more numerical measures (the “facts” of the fact table) that occur for the combination of keys that define each tuple in the table.

Dimension tables are satellites of the central fact table. They typically contain textual information that describes the attributes of the fact table.

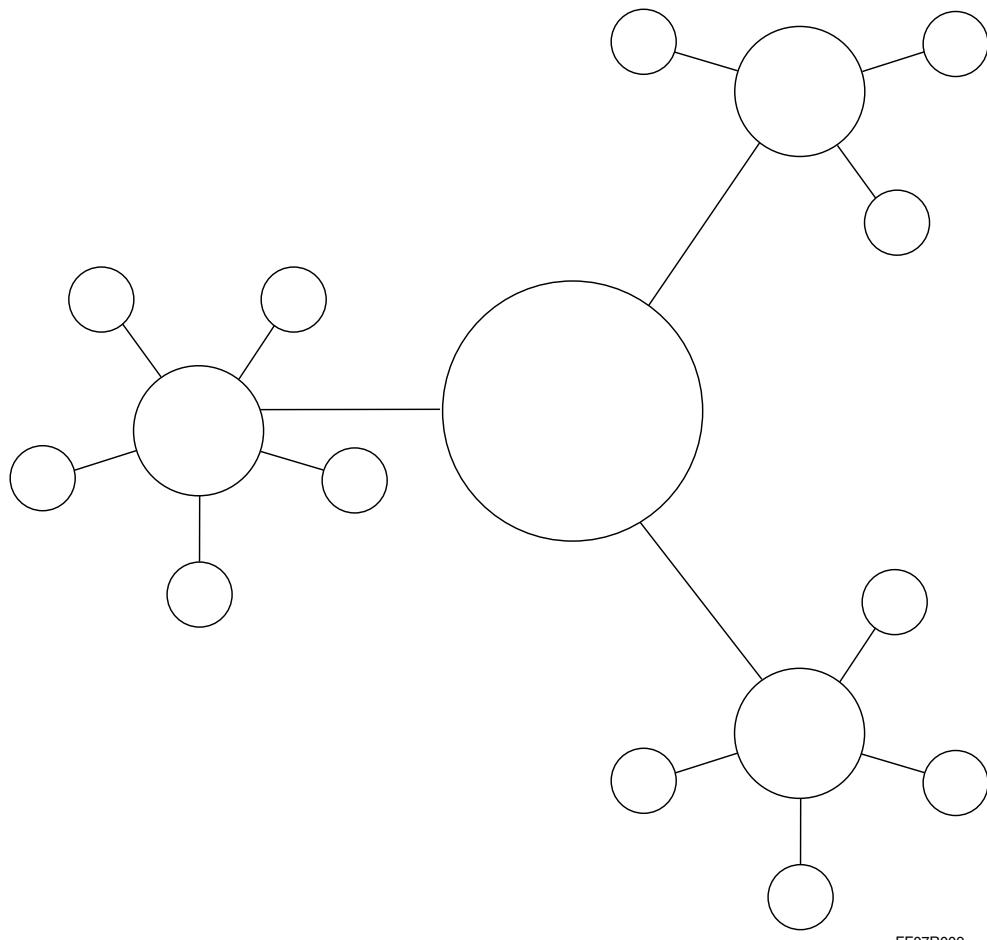
## Star Schema

The following graphic illustrates the classical star schema:



## Snowflake Schema

The following graphic illustrates the classical snowflake schema:



## The E-R Model Versus the DM Model

While a table in a normalized E-R-derived database represents an entity and its relevant atomic descriptors, tables in a DM-derived database are said to represent dimensions of the business rules of the enterprise. The meaning of business rule used here is somewhat different from that used by writers in the business rules community, where the term applies more directly to the declarative domain, range, uniqueness, referential, and other constraints you can specify in the database.

While advocates of implementing a normalized physical schema emphasize the flexibility of the model for answering previously undefined questions (see, for example, Inmon, 2000), DM advocates emphasize its usability because the tables in a DM database are configured in a structure more akin to their business use.

The E-R model for an enterprise is always more complex than a DM model for the same enterprise. While the E-R model might have hundreds of individual relations, the comparable DM model typically has dozens of star join schemas. The dimension tables of the typical DM-derived database are often shared to some extent among the various fact tables in the database.

# CHAPTER 8 Teradata Database Indexes and Partitioning

---

Teradata Database provides indexing and partitioning options for optimizing the performance of your relational databases:

- You can apply primary indexes to non-column-partitioned tables and join indexes.
- You can apply secondary, hash, and join indexes to all tables.
- You can apply secondary indexes to join indexes.

Hash and join indexes are not true indexes; instead, they are user-inaccessible tables that the Optimizer can use to resolve queries covered by the hash or join index without necessarily accessing the underlying base tables. The Optimizer can also use secondary, hash, and join indexes to partially cover queries, joining the partial covering result to the indexed base table to capture data that is not covered by an index.

Teradata Database primary and secondary indexes are not really indexes, but hash keys. Definitions of indexes and hash keys are provided in “[Indexing and Hashing](#)” on page 220 and “[Tradeoffs Between Hashing and Indexing](#)” on page 222.

This chapter describes Teradata Database indexing and introduces the types of indexes you can specify.

The chapter also briefly describes how the rows of tables and indexes are distributed to the AMPs using different methods of row allocation, and how rows and columns can be partitioned on each AMP.

## Types of Indexes

### Unique Indexes

A unique index column set, like a primary key column set constraint, enforces a unique value for that column set for each row in a table.

Teradata Database defines two different types of unique index:

- Unique primary index (UPI)

UPIs provide optimal data distribution and, when used, are typically assigned to the primary key for a table.

UPIs are described in detail in “[Unique Primary Indexes](#)” on page 264.

- Unique secondary index (USI)

USIs guarantee that each complete index value is unique, while ensuring that data access based on it is always at most a two-AMP operation. A USI is typically assigned to the

primary key for a table when the table does not have a UPI for the primary key and enforcement by the system of a primary key or unique constraint is a requirement. USIs are described in detail in “[Unique Secondary Indexes](#)” on page 457.

## Nonunique Indexes

A nonunique index does not require its values to be unique. There are occasions when a nonunique index is the best choice as the primary index for a table.

Teradata Database defines two different types of nonunique index.

- Nonunique primary index (NUPI)

NUPIs facilitate joins between major and minor entities by defining them with the same PI (as a UPI in the case of the major entity and as a NUPI in the case of the minor entity) to ensure that the rows to be joined hash to the same AMP.

How evenly a NUPI distributes the rows for a table across the AMPs depends on the relative singularity of the data comprising the primary index. As a rule of thumb, you should avoid assigning a NUPI to a column set for which many more than several hundred to several thousand rows in the table are likely to have the same NUPI value, depending on the cardinality.

NUPIs are described in detail in “[Unique Primary Indexes](#)” on page 264.

- Nonunique secondary index (NUSI)

NUSIs can be very useful for providing set selection in many decision support applications.

NUSIs are described in detail in “[Nonunique Secondary Indexes](#)” on page 467.

## Primary Index Types

Except for nonpartitioned NoPI tables (see “[NoPI Tables, Column-Partitioned Tables, and Column-Partitioned Join Indexes](#)” on page 280), column-partitioned tables and join indexes, and global temporary trace tables, each Teradata Database table requires a primary index to distribute table rows to the AMPs based on their primary index values.

Nonpartitioned NoPI table rows and column-partitioned table and join index rows are distributed using a different mechanism (see “[Row Allocation for Teradata Parallel Data Pump](#)” on page 237 for both nonpartitioned NoPI and column-partitioned tables and “[Row Allocation for FastLoad Operations Into Nonpartitioned NoPI Tables](#)” on page 238 for nonpartitioned NoPI table rows only).

Teradata Database provides the following primary index types:

- Unique primary index (UPI)
- Nonunique primary index (NUPI)

The NUPI for a join index can be locally value-ordered in some cases. See the description of the CREATE JOIN INDEX statement in *SQL Data Definition Language* for details.

- Nonpartitioned primary index
- Row-partitioned primary index (PPI)

Partitioned primary indexes fall into two general categories, depending on whether they have a single partitioning expression or multiple partitioning expressions:

- Single-level
- Multilevel

These can be mixed in any cross-dimensional combination, so the following complete specifications for a primary index can all be defined in Teradata Database:

- Unique nonpartitioned primary index
- Nonunique nonpartitioned primary index
- Unique single-level partitioned primary index
- Unique multilevel partitioned primary index
- Nonunique single-level partitioned primary index
- Nonunique multilevel partitioned primary index

## Primary Index Dimensions

Primary indexes are defined over two orthogonal dimensions: unique versus nonunique and partitioned versus nonpartitioned.

### Unique Versus Nonunique Dimension

This choice raises two issues.

- The possibility of not distributing the rows for a table evenly across the AMPs
- The necessity of checking for duplicate rows in a SET table with a NUPI (unless there is also a USI defined on the table).

The most likely scenario causing an uneven row distribution would be if you did not select a *nearly* unique column to be a nonunique primary index, in which case the distribution of rows for the table across the AMPs or across hash values can be uneven (sometimes the words “lumpy” or “skewed” are used to describe this kind of distribution).

Do not interpret this to mean that NUPIs necessarily, or even usually, cause the distribution of rows across the AMPs to be skewed. Particularly when the number of rows in a table is very large, it is possible to achieve an even distribution of rows with a NUPI, even if it is not nearly unique.

Skewed distributions also occur for *unique* primary indexes whenever the number of rows in a table is small relative to the number of AMPs in the configuration, particularly when the following situations occur.

- There are fewer rows than AMPs.
- There are very few unique values (not more than the number of AMPs), even when there are many rows.

Excessively uneven distribution of rows causes performance problems, particularly for large tables, and should be avoided.

Depending on table size and other factors, duplicate row checks on NUPI SET tables can have a significant negative impact on performance.

## Partitioned Versus Nonpartitioned Dimension

Row-partitioned primary indexes are defined for tables that are frequent targets of range or row-partition-based queries. A common example of this is queries that involve date ranges.

As for a primary index, partitioning is specified either with the CREATE TABLE statement when the table is created or with the ALTER TABLE statement after the table has already been created. The rows of a table having a PPI are hashed to the same AMPS they would be hashed to if the primary index were not partitioned. Once assigned to an AMP, PPI rows are further assigned to, and stored in rowhash order within, their respective row partitions.

Nonpartitioned primary indexes behave identically with the exception that they are assigned to one single row partition (number 0) after they are hashed to their AMP.

Nonpartitioned primary index rows are stored in rowhash order on their respective AMPS.

See [Chapter 9: “Primary Indexes and NoPI Objects,”](#) for additional details about partitioned and nonpartitioned primary indexes.

## Secondary Index Types

- Unique secondary index (USI)
- Nonunique secondary index (NUSI):
  - Hash-ordered on all columns.  
This is the default ordering when you do not specify an ORDER BY clause.
  - Hash-ordered on a single column with no ALL option.
  - Value-ordered on a single column with no ALL option.
  - Hash-ordered on a single column with the ALL option.
  - Value-ordered on a single column with the ALL option.

See the description of the CREATE INDEX statement in *SQL Data Definition Language* for details.

## Join Index Types

There are two basic types of join index:

- Multitable join indexes are defined for join queries that are performed frequently enough to justify defining a prejoin on the joined columns.
- Single-table join indexes are defined to facilitate joins by hashing a frequently joined subset of base table columns to the same AMP as the table rows to which they are frequently joined. This enhanced geography eliminates BYNET traffic as well as often providing a smaller sized row to be read and joined.

You can define a join index with either a nonpartitioned or row-partitioned primary index, but keep in mind that you can define a join index with row-partitioned primary index only if it is not row compressed. See [“Join Index Applications” on page 504](#). You can also define a join index with column partitioning (and no primary index) with optional row partitioning.

The MultiLoad and FastLoad utilities do not support target tables with join indexes. The workaround for this is to FastLoad new rows into an empty table and then use either a

INSERT ... SELECT or MERGE request to populate the indexed table (see *SQL Data Manipulation Language* for details).

For additional information, see “[Join Indexes](#)” on page 215 and [Chapter 11: “Join and Hash Indexes.”](#)

## Hash Indexes

Hash indexes are similar to single-table join indexes, though their SQL definition syntax is more similar to that of secondary indexes.

The MultiLoad and FastLoad utilities do not support target tables with hash indexes. The workaround for this is to FastLoad new rows into an empty table and then use either a INSERT ... SELECT or MERGE request to populate the indexed table (see *SQL Data Manipulation Language* for details).

Teradata recommends using single-table join indexes rather than hash indexes.

For additional information, see “[Hash Indexes](#)” on page 217 and [Chapter 11: “Join and Hash Indexes.”](#)

## Related Topics

For more information on...	See in this book...	See in <i>SQL Data Definition Language...</i>
Primary indexes	<a href="#">Chapter 9: “Primary Indexes and NoPI Objects.”</a>	CREATE TABLE
Secondary indexes	<a href="#">Chapter 10: “Secondary Indexes.”</a>	<ul style="list-style-type: none"><li>CREATE INDEX</li><li>CREATE TABLE</li></ul>
Join and hash indexes	<a href="#">Chapter 11: “Join and Hash Indexes.”</a>	<ul style="list-style-type: none"><li>CREATE JOIN INDEX</li><li>CREATE HASH INDEX</li></ul>

## Advantages of Indexes

Indexes reduce the time it takes to retrieve rows from a database. The faster the retrieval, the better.

## Disadvantages of Indexes

- Teradata Database must update index subtables each time an indexed column value in the base table is updated or deleted, or whenever a new row is inserted.  
This is only a consideration for secondary, join, and hash indexes in the Teradata Database environment. The more secondary, join, and hash indexes you have defined for a table, the larger the potential update maintenance downside becomes.

Because of this, secondary, join, and hash indexes are generally inappropriate for OLTP situations, though some limited use of secondary and sparse join indexes (when the sparse index is defined on columns that are rarely updated) might be appropriate.

- All Teradata Database secondary, join, and hash indexes are stored in subtables, so they exert a burden on system storage space.
- When fallback is defined for a table, a further storage space burden is created because secondary index subtables are also duplicated whenever fallback is defined for a table.

You can optionally specify fallback on join and hash indexes as well. Join and hash indexes do *not* default to fallback if their underlying base tables are defined with fallback.

For these reasons, it is extremely important to use EXPLAIN request modifiers to determine optimum data manipulation statement syntax and index usage before putting those statements (and indexes) to work in a production environment.

## Indexes and Partitioning

Typically, retrieving or updating data involves a relatively small number of rows. Without indexes and partitions, Teradata Database would need to scan every row in the table to find the rows of interest. Indexes and partitions can speed data access by providing an alternative, more direct path to the data of interest. In some cases, a query can be fulfilled even more quickly by accessing the index alone, without requiring access to the indexed table at all.

Because CPUs are becoming faster at a far greater rate than disk storage access rates, appropriate indexing and partitioning can be an important factor in altering the CPU-to-I/O ratio. In fact, it is sometimes possible to render previously I/O-bound systems CPU-bound or nearly CPU-bound through optimal indexing and partitioning of database tables.

The Teradata Database parallel architecture makes indexing and partitioning an aid to better performance, not a crutch necessary to ensure adequate performance. Full-table scans are not something to be feared in the Teradata Database environment. This means that unplanned, ad hoc queries that characterize the data warehouse process, and that often are not supported by indexes, perform very effectively for Teradata Database using full-table scans.

The classic index for a relational database is itself a file made up of rows having two parts.

- A (possibly unique) column in the referenced table.
- A pointer to the location of that row in the base table (if the index is unique) or pointers to all possible locations of the rows (if the index is nonunique).

Because Teradata Database is a massively parallel architecture, it requires an efficient method of distributing and retrieving its data. For all database objects except NoPI and column partitioned objects, that method is hashing. All Teradata Database indexes are based, at least partially, on the rowhash of column values rather than directly on the column values. Rows for a table with a primary index and unique secondary indexes are stored by an AMP in hash order. Nonunique secondary, join, and hash indexes can be stored by an AMP in hash order or in value order of a column. Value order may make an index more useful for satisfying range conditions.

## Selectivity of Indexes and Partitioning

An index that retrieves many rows is said to have low selectivity. By definition, an index with low selectivity typically accesses one or more rows per data block from the table on which it is defined. If the average number of rows accessed per data block is appreciably higher than 1, you should consider dropping the index. However, this process is value-dependent, so you should evaluate the performance of the index in several different situations before dropping it.

An index that retrieves few rows is said to be highly selective. A highly selective index is one that does not access all of the data blocks for the table on which it is defined. An index that is intended to be highly selective should access significantly fewer than an average of 1 row per data block.

The more highly selective an index or partitioning is, the more useful it is for enhancing performance.

In some conditions, it is possible to link several nonunique secondary indexes with low selectivity together by bit mapping them. The result is effectively a highly selective index and a dramatic reduction in the number of table rows that must be accessed. See “[NUSI Bit Mapping](#)” on page 479 for further discussion of linking secondary indexes with low selectivity into a highly selective unit using bit mapping.

## Rowhash and RowID

Primary-indexed Teradata Database table rows are self-indexing based on their primary index and so require no additional storage space.

When a row is inserted into a table, the file system stores the 32-bit rowhash value of the primary index in place with the row.

Rows inserted into a NoPI table or column-partitioned table (see “[NoPI Tables, Column-Partitioned Tables, and Column-Partitioned Join Indexes](#)” on page 280) are allocated to the AMPs differently (see “[Row Allocation for Teradata Parallel Data Pump](#)” on page 237), and it is the hash bucket number for the assigned AMP with a 44-bit uniqueness value rather than the rowhash value with a 32-bit uniqueness value that is stored in their rowID.

A rowID always includes an internal partition number, which is 0 if there is no partitioning, and compressed in some cases when its value is 0.

Because rowhash values are not necessarily unique (see “[Hash Bucket Number](#)” on page 225, the AMP software also produces a unique 32-bit numeric value (called the uniqueness value, see “[Uniqueness Value](#)” on page 227) that it appends to the rowhash value, forming a unique rowID. The value assigned depends on the uniqueness values that have already been assigned. The system assigns uniqueness values in ascending numerical order. For rows in a table, uniqueness is achieved by combining the internal partition number (0 for a non-PPI table), rowhash value and uniqueness value, in that order.

This rowID makes each row in a table uniquely identifiable, even if it is otherwise a duplicate of one or more other rows. Duplicate rows can only occur for MULTISET tables that have no uniqueness constraints. You are strongly discouraged from creating such tables when they are primary-indexed tables.

Note the following things about rowIDs.

- The rowID for a given row can change if the value of a primary index or partitioning column for that row changes.

The rowID can also change for a given row of a column-partitioned table if the row is updated.

- RowIDs can be reused after they are no longer associated with a row.

This means that while a rowID uniquely identifies a row at any particular time, a rowID that once identified a row later might not be associated with any row, or might be associated with a different row because the original row it was associated with might either have taken a different rowID or have been deleted.

The first row having a specific internal partition number and rowhash value that is inserted into a table on the AMP that owns the hash bucket value within the row hash is always assigned a uniqueness value of 1. Additional table rows for that AMP that have the same internal partition number and rowhash value are assigned uniqueness values in a numerically increasing order. The rows are stored on disk, sorted in ascending order of rowID.

To determine uniqueness violations for UPIs or duplicate rows for SET NUPI tables, Teradata Database scans all rows in the rowhash. A duplicate row is defined as a row that matches one or more other rows in a relation exactly.

For a UPI PPI or SET NUPI PPI, Teradata Database scans only the rows with the same internal partition number and rowhash value to search for uniqueness violations or duplicate rows, respectively, not *all* the rows in other partitions that have the same rowhash value.

This scan terminates by reading the last row in the rowhash, and the uniqueness value of that last row is the highest current value. When the system next needs to assign a uniqueness value, it increments the value for the last row read by one.

Uniqueness values are not reused except for the special case in which the row with the highest uniqueness value within a rowhash is deleted from a table.

## Rowhash Value and RowID for NoPI Tables

Because neither nonpartitioned nor column-partitioned NoPI tables have a primary index on which to base the rowhash value for a row, Teradata Database generates a random rowhash value for both nonpartitioned and column-partitioned NoPI table rows based on different values, depending on whether the row is inserted using a simple insert (including array inserts) or an INSERT-SELECT. You can also load nonpartitioned NoPI tables, but *not* column-partitioned tables, using the Fast Load utility, which also generates the rowhash for such tables differently (see “[Row Allocation for FastLoad Operations Into Nonpartitioned NoPI Tables](#)” on page 238).

For simple SQL insert operations, Teradata Database uses the system-generated value of the Query ID to uniquely identify requests as input to the hashing algorithm to determine its destination AMP. The rowhash value generated by the hashing algorithm ensures that rows are ordinarily sent to a different AMP from the AMP selected for the previous request, which balances the distribution of rows among the AMPS as much as is possible without hashing on a primary index value. The randomly generated hash bucket values are as nonunique as

possible to avoid excessive skewing of the row distribution. See “[Row Allocation for Teradata Parallel Data Pump](#)” on page 237 for details of this operation.

For simple insert operations, there are two possible cases.

- A single row is to be dispatched.

This is the case for a standard INSERT operation.

- Multiple rows are to be dispatched.

This is the case for the Array INSERT operations used by the Teradata Parallel Data Pump utility.

For both cases, the system copies the generated rowhash value into the rowID of each row (see “[Base Table Row Format](#)” on page 740). Because the system uses the generated rowhash value only once per request, Teradata Database copies the same rowhash value into the rowID of *all* the rows processed by an Array INSERT operation, which means they are all sent to the same AMP, very possibly in the same step.

For a column-partitioned or nonpartitioned NoPI table, the higher order 20 bits of the hash value is a hash bucket that specifies the AMP for a referencing rowID. The system selects the hash bucket in increasing order as one defined for the AMP per the NoPI hash map on which the row is inserted.

Teradata Database assigns rows randomly to AMPS or copies them locally. The hash bucket is not computed using a hash of the primary index columns because there is no primary index to hash. Because the full 32-bit hash value is not used for a NoPI database object, the uniqueness value is 44 bits and is used as a row number, beginning at one and incrementing by one for each row appended to that partition and hash bucket.

A 44-bit uniqueness value enables a maximum of 17,592,186,044,415 rows per hash bucket. If the maximum for a hash bucket is exceeded, Teradata Database sets the hash bucket bits to the next bucket number for the AMP per the NoPI hash map, and resets the uniqueness value to 1. If the table or join index is not partitioned, the internal partition number is 0.

For a column-partitioned database object, the rowID is the same as for a NoPI table except that the object is column partitioned and, optionally, row partitioned. Therefore, the internal partition number is nonzero.

Join indexes, hash indexes, and secondary indexes have referencing rowIDs whether there is a primary index or not, or whether the underlying table or join index is partitioned or not. For a rowID that references a column-partitioned object, the column partition number (within the internal partition number) is set to 1 by convention and for consistency. A referencing rowID indicates a specific table row, not necessarily to a specific physical row. A rowID can easily be adjusted by adding a constant delta on dereferencing to a desired column partition number in order to access a physical row containing a specific column partition value for the table row.

The rowID for a table row of a NoPI table without column partitioning is as follows.

Internal Partition Number = 0, Hash Bucket = x, Uniqueness = n

The rowID for a table row of a CP table is as follows.

Internal Partition Number = 1, Hash Bucket = x, Uniqueness = n

The Dispatcher distributes nonpartitioned and column-partitioned NoPI table rows from an external source to the AMPs based on a randomly generated rowhash value. This means that if there is a large number of rows to distribute, their assignment eventually balances among the AMPs.

If the source table data for an INSERT ... SELECT statement is skewed, the target table data is skewed when you are loading an nonpartitioned NoPI or column-partitioned table. This is because the rows selected by an INSERT ... SELECT request are not redistributed across the AMPs, but are copied locally to the same AMP. To avoid this problem, you can specify an appropriate HASH BY clause with the SELECT subquery (see *SQL Data Manipulation Language* for information about the syntax and usage of the HASH BY clause in a SELECT request).

Unlike rows loaded into an nonpartitioned NoPI table using Teradata Parallel Data Pump array inserts, rows loaded into an nonpartitioned NoPI table using the FastLoad utility are not parsed, but are instead sent directly to an AMP. Such rows are then hashed by the AMP software on their AMP vproc ID plus a counter value using a randomization algorithm that is different from the standard hashing algorithm. This generates the final rowhash value used for distribution. The receiving AMP determines the hash bucket and uniqueness value to be used as part of its rowID.

**Note:** You cannot use the FastLoad utility to load rows directly into a column-partitioned table. You can FastLoad rows into a staging table and then use an INSERT ... SELECT statement to transfer them into the column-partitioned table.

## Index Hash Mapping

The rows for primary-indexed tables are distributed across the AMPS using a hashing algorithm (see “[Teradata Database Hashing Algorithm](#)” on page 225). The hashing algorithm computes a rowhash value based on the value of the primary index. The rowhash is a 32-bit value containing either a 16-bit hash bucket number with a 16-bit remainder or a 20-bit hash bucket number with a 12-bit remainder (see “[Teradata Database Hashing Algorithm](#)” on page 225).

The number of hash buckets for a Teradata system can be set to either 65,536 or 1,048,576, depending on the settings for the CurHashBucketSize and NewHashBucketSize parameters in DBS Control (see *Utilities: Volume 1 (A-K)* for details about the DBS Control utility. Also see “[Number of Hash Buckets Per System](#)” on page 229). The number of hash buckets for a system is a function of hash bucket size, with 16-bit buckets producing 65,536 buckets per system and 20-bit buckets defining 1,048,576 buckets per system (see “[Number of Hash Buckets Per System](#)” on page 229).

The hash buckets are distributed as evenly as possible among the AMPs on a system (see “[Hash Maps](#)” on page 229).

The BYNET interface maintains a hash map for each AMP, an index of which hash buckets are assigned to which AMPs, that is used in combination with the hash bucket number in a row's rowhash to determine the owning AMP. Row assignment is performed in a manner that

ensures as equal a distribution of table rows as possible among all the AMPs in the configuration.

## Partitioned and Nonpartitioned Indexes and Tables

A row-partitioned index assigns rows to a particular partition within an AMP based on one or more user-defined partitioning expressions. Only primary indexes can be row-partitioned. Any index other than a partitioned primary index is, by default, a nonpartitioned index.

It is also possible to partition table and join index data using column partitions and no primary index. In this case, the table or join index can also optionally be row-partitioned. See “[Column-Partitioned Tables and Join Indexes](#)” on page 285 for more information about column-partitioned tables and join indexes.

## Using EXPLAIN Request Modifiers to Determine the Usefulness of Indexes

The selection of indexes to support a request is *not* under user control. Additionally, you cannot specify resource options for Teradata Database indexes, nor can you control index locking.

Teradata SQL data manipulation language statements do not provide a method for specifying indexes as hints or pragmas in their syntax. The only references made to indexes in the Teradata SQL dialect concern their definition, *not* their use. See *SQL Request and Transaction Processing* for more information about why Teradata SQL does not use hints.

The implications of this behavior include the following:

- It is critically important to collect statistics regularly on all indexed columns, all frequently joined columns, and all columns frequently specified in query predicates to ensure that the Optimizer has access to current information on the demographics of the database so it can know how to best optimize any query or update made to the database. Maintaining fresh statistics is the only way to tune index requests in Teradata Database.

For additional information concerning collecting and maintaining accurate database statistics, see “[COLLECT STATISTICS \(Optimizer Form\)](#)” in *SQL Data Definition Language*. You can also use the Teradata Viewpoint Stats Manager portlet or see the *Automated Statistics Management* Teradata Orange Book, 541-0009628.

- It is equally important to build your SQL queries and updates in such a way that you know their performance will be most optimal.

Apart from good logical database design, there are two ways to ensure that you are accessing your data in the most efficient manner possible.

- Use the EXPLAIN request modifier or the Visual Explain tool (see “[Comparing EXPLAIN Request Modifier Reports Graphically](#)” on page 202) to test various candidate requests and to note which indexes are used by the Optimizer in their execution (if any) as well as examining the relative length of time required to complete the request.
- Use the Teradata Index Wizard client utility to recommend and validate sets of secondary or single-table nonsparse join indexes, or both, for a particular query workload. See *Teradata Index Wizard User Guide* for further information.

EXPLAIN request modifier output provides you with the following basic information.

- The step-by-step access method the Optimizer would use to execute the specified request given the current set of table statistics it has to work with.
- The *relative* and worst case time estimates to perform the specified request.

While you cannot rely on the reported statement execution time as an absolute, you *can* rely on it as a relative means for comparison with other candidate requests against the same tables with the same statistics defined.

Even within the bounds of a constant system configuration a plan can change significantly because of shifting data demographics. The only way to track these shifts and to ensure that the Optimizer is building the best possible plans is to collect statistics frequently on your index columns, frequently joined columns, and columns frequently specified as query predicates and then run new EXPLAIN request modifiers to generate reports to allow you to determine if the new statistics affect whether your indexes are used or not.

Extensive information about how to use the EXPLAIN request modifier is contained in “EXPLAIN Request Modifier” in *SQL Data Manipulation Language*.

Information about using the Visual Explain utility is contained in *Teradata Visual Explain User Guide*.

## Comparing EXPLAIN Request Modifier Reports Graphically

Teradata Database provides a sophisticated tool for analyzing and comparing the EXPLAIN request modifier reports generated by semantically identical, but syntactically different, requests or by the same request performed on different hardware platforms and configurations or under different software releases. This tool, called Visual Explain, provides facilities for displaying and comparing EXPLAIN request modifier reports graphically.

Information about using the Visual Explain utility is contained in *Teradata Visual Explain User Guide*.

## Analyzing EXPLAIN Request Modifier Reports Using SQL

SQL provides another powerful facility for analyzing the plans the Optimizer creates for various requests. This feature, called the Query Capture Facility, permits a DBA to capture access plan information from the query optimizer using the SQL utility statements COLLECT DEMOGRAPHICS, DUMP EXPLAIN, INSERT EXPLAIN, and BEGIN QUERY LOGGING (see *SQL Data Definition Language* for more information about these statements) and then analyze the captured data using the Teradata Index Wizard or standard SQL DML statements.

The captured plans are stored in a relational database called a Query Capture Database (QCD) that you can create for that purpose. QCD is a user database whose space is drawn from available permanent disk space: it is not a set of system tables found in a fixed location.

See *SQL Request and Transaction Processing* for more information on the Query Capture facility and the definitions of its tables.

Note that the Visual Explain tool also uses these tables for its analyses.

See the online help files for the Visual Explain tool and *Teradata Visual Explain User Guide* for more information about its capabilities.

## Target Level Emulation

A related facility, Target Level Emulation, permits you to capture the environmental cost data and dynamic AMP sampled statistics of a production system and then analyze various queries and query workloads in a test environment that emulates your production system.

See *SQL Request and Transaction Processing* and *Teradata System Emulation Tool User Guide* for more information on this tool.

## Related Topics

For additional performance-related information about how to use the plan reports produced by the EXPLAIN request modifier to optimize the performance of your databases, consult the following books.

Topics	Reference
<ul style="list-style-type: none"><li>Where the data reported by explaining an SQL request comes from.</li><li>Meaning of the more commonly used EXPLAIN text phrases and the structure of the Query Capture Database tables.</li></ul>	<i>SQL Request and Transaction Processing</i>
Syntax for the EXPLAIN request modifier, information about its use, and examples.	<i>SQL Data Manipulation Language</i>
<ul style="list-style-type: none"><li>How the Teradata Visual Explain utility displays Optimizer white tree information symbolically rather than textually.</li><li>How the white tree information is captured in the Teradata Query Capture Database, how to administer the Query Capture Database, and how to use the utility to compare different plans for the same SQL request by varying the physical database design and index usage.</li><li>Option for comparing different plans textually as well as graphically.</li></ul>	<i>Teradata Visual Explain User Guide</i>
How to use the Teradata System Emulation Tool (TSET) to vary and test various database object definitions, statistics, cost parameters, and cost profiles on a test machine before exporting the final definitions to a production system.	<i>Teradata System Emulation Tool User Guide</i>

## Using Indexes to Enhance Performance

The following table summarizes how the Optimizer can use indexes to enhance query performance.

Index Type	Standard Use
Primary	Satisfy an equality on an IN condition in a join
Unique primary	Ensure the fastest access to single rows
Nonunique primary	<ul style="list-style-type: none"><li>To perform a single-AMP row selection or join process</li><li>To avoid sorting or redistributing rows</li></ul>
Unique secondary	Process requests that specify equality constraints
Unique primary to match values in one table with index values in another	Ensure optimal join performance.
Composite index only	Optimal processing of requests that employ equality constraints for all fields that comprise the index.
Bitmapped NUSI	Process requests when equality or range constraints involving multiple NUSIs are applied to very large tables.

For smaller tables, the Optimizer uses the index estimated to have the fewest rows per index value.

Using appropriate secondary indexes for the table can increase retrieval performance, but the tradeoff is that update performance can decrease because of the need to update secondary index subtables.

## Determining Index Usage

To determine if the Optimizer will use an index to process a request, specify a WHERE clause for the query of interest based on an index value, any covering column in the index, or both, and then specify the EXPLAIN request modifier to determine whether the specified index value affects path selection.

When you specify an indexed column in a request, semantically equivalent queries specified using different syntax can cause the Optimizer to generate different access plans. For example, the Optimizer might generate different access paths based on the following forms, and depending on the expression, one form might be better than the other.

Form 1.        (A OR B)  
                  AND  
                  (C OR D)

Form 2        (A AND C)  
                  OR  
                  (A AND D)

```

    OR
    (B AND C)
    OR
    (B AND D)

```

In expressions involving both AND and OR operators, the Optimizer generates the access path based on the form specified in the query. The Optimizer does not attempt to rewrite the code from one form to another to find the best path. Consider the following condition.

```

( NUSI = 7
OR NUSI = 342)
AND (X = 3
OR X = 4)

```

In this case, Form 1 is optimal, because the access path consists of two NUSI SELECT operations with values of 7 and 342. The Optimizer applies (X=3 OR X=4) as a residual condition. If the Optimizer uses Form 2, the access path consists of 4 NUSI SELECT operations.

In the following condition, the collection of (NUSI\_A, NUSI\_B) comprises a NUSI. Form 2 is optimal because the access path consists of 4 NUSI SELECT operations, whereas the Form 1 access path requires a full-table scan.

```

(NUSI_A = 1
OR NUSI_A = 2)
AND (NUSI_B = 3
OR NUSI_B = 4)

```

Assume an expression involves a single-column comparison using an IN clause such as the following

```
column IN (value_1, value_2, ...)
```

The Optimizer rewrites that expression to the following form.

```

column = value_1
OR column = value_2
OR column = value_m

```

Therefore, the Optimizer generates the same access path for either form. However, if an expression involves a multi-column comparison using an IN clause, such as in the following query, then the Optimizer rewrites the expression to form b.

```

a.      (column_1 IN ( value_1
                      OR value_2
                      OR value_n)
        AND column_2 IN ( value_3
                      OR value_4
                      OR value_n))

b.      ( column_1 = value_1
          OR column_1 = value_2
          OR column_n = value_m)
        AND
        ( column_2 = value_3
          OR value_n)

```

Notice that the rewritten form differs from the following condition, which is in Form 2).

```
c.      ( column_1 = value_1
```

```
        AND column_2 = value_3)
OR (    column_1 = value_2
        AND column_2 = value_4)
OR      column_n = value_m
```

For more information about the EXPLAIN request modifier, see *SQL Request and Transaction Processing* and *SQL Data Manipulation Language*.

## Keys and Indexes

### Definition of a Key

There are several different meanings for the term *key* in relational database management. When reading this manual, you should always assume that if the term key is used without a qualifying adjective, the reference is to the primary key, or PK, for a table.

The following table defines the various uses of key in the context of relational database management in general or Teradata Database in particular. Note that in every case, a key is an identifier and not a retrieval mechanism. In every case, you should read the word *field* to mean “one field or several fields in combination.” The expression *column set* is used throughout this manual in the same way; to indicate that the table object under discussion, whether it be a key or index, consists of one or more columns.

Term	Definition
Alternate key	Any candidate key not selected to be the primary key for a table.
Candidate key	A field that uniquely identifies a row.  Every normalized table has at least one candidate key, and most have <i>only</i> one. When multiple candidate keys are identified during logical design, one is selected to be the primary key and the candidate keys not selected to be the primary key are then referred to as alternate keys.
Child key	A foreign key that is a primary key or alternate key in another table.
Foreign key	A field that corresponds to the primary key or alternate key of a different, but related, table.  Foreign keys are used to maintain referential integrity. They are often useful as NUPIs when you want rows from two or more related tables to hash to the same AMP.
Grouping key	A field used to group the rows of a table in a particular way.
Hash key	A field used to compute the rowhash value for a row.

Term	Definition
Natural key	<p>The representation of a real world tuple identifier in a relational database. For example, a common identifier of employees in a corporation is a unique employee number. An employee is assigned an employee number whether that information is stored within the database or not.</p> <p>Natural keys are sometimes confused with intelligent keys, but they are very different concepts.</p> <p>Compare with “Surrogate key” in the last row of this table.</p>
Order key	A field used to store the rows of a table in a particular order.
Parent key	A primary key or alternate key that is a foreign key in another table.
Primary key	<p>A field that uniquely identifies a row.</p> <p>A primary key can never be null.</p>
Search key	A field used to match a corresponding field in a searched table.
Sort key	A field used to sort the rows of a table. Note that the sorted table in this case is not necessarily a base table; for example, it can be an intermediate result such as a spool file.
Surrogate key	<p>An arbitrary, system-generated, simple numeric key. Often used when a natural key is otherwise difficult or impossible to define for a table or when the situation demands a non-composite primary key, but no natural non-composite exists.</p> <p>Compare with “Natural key” in the seventh row of this table.</p>

## Definition of an Index

Teradata Database uses the term *index* in two different ways, neither of which is a true index.

- Primary and secondary indexes are rowhash values by default, though they can also be locally value-ordered in some cases. See “[Indexing and Hashing](#)” on page 220.
- Join and hash indexes are inaccessible tables that can substitute for base tables in various queries, particularly queries that require join processing. In these cases, the index is said to cover the query.

Multitable join indexes are often denormalized prejoins of frequently joined tables that the Optimizer can substitute in a query instead of making the join dynamically.

Single-table join indexes and hash indexes tend to be used either as virtual vertically-partitioned tables or, in the case of single-table join indexes, as a version of a base table that is distributed to the AMPs using a different primary index than its parent table. This is useful for redistributing base table rows in a way that facilitates their being joined to rows from other tables that are distributed on the same primary index values. In this way, a single-table join index functions as a version of a hashed NUSI.

A join index can be used to substitute for the base table if it has been defined using any of the following components.

- The UPI of the underlying base table
- The keyword ROWID

You can only specify ROWID in the outermost SELECT of the CREATE JOIN INDEX statement. See “CREATE JOIN INDEX” in *SQL Data Definition Language Detailed Topics*.

- The NUPI of the underlying base table and the keyword ROWID
  - The NUPI of the underlying base table and the USI of the underlying base table
- It is preferable to specify NUPI and ROWID over NUPI and USI.

In the most general definition possible, an index is a column or combination of columns in a table used to access its data in the most high-performing means possible.

The primary index for a table is frequently defined on the same column set that is identified as the primary key during logical database design, but nonunique, non-primary key columns can also be used as the primary index for a table or join index. You can also define tables and some join indexes *not* to have a primary index (see “[NoPI Tables, Column-Partitioned Tables, and Column-Partitioned Join Indexes](#)” on page 280).

## Differences Between Keys and Indexes

The following table summarizes the differences between keys and indexes using the primary key and primary index for purposes of comparison.

Primary Key	Primary Index
Component of logical data model.  Primary keys are also used to declare and maintain referential integrity constraints in the physical database, as noted by the next row in the table. A primary key is implemented as a unique primary or secondary index in the physical Teradata Database.	Component of physical data model.
Used to maintain referential integrity.	Used to distribute and retrieve rows.
Must be unique.	Can be nonunique.
Uniquely identifies each row.	Distributes table rows across the AMPs.  Might or might not uniquely identify each row depending on whether it is defined as a UPI or a NUPI.  The rows of NoPI tables and column-partitioned tables and join indexes are distributed across the AMPs differently because they do not have a primary index.
Values can never be changed.	Values can be changed, but as a general rule should not be.

Primary Key	Primary Index
Cannot be wholly or partly null.	Can be wholly or partly null. A single-column primary index can apply to no more than one row per table. A multicolumn primary index can have a row for any combination of nulls and values, but cannot have the same combination for more than one row. Although you <i>can</i> have a wholly or partly null primary index, there are countless reasons why you should <i>not</i> . See <a href="#">Chapter 13: “Designing for Missing Information”</a> for a discussion of why nulls should be avoided when possible in relational databases.
Does not imply an access path.  This does not mean that the primary key for a table is never used as an access or join path by the Optimizer.	Defines row distribution to the AMPs and an access path.
Not required for physical table definition.	Required for physical table definition with the following exceptions. <ul style="list-style-type: none"> <li>• NoPI tables</li> <li>• Column-partitioned tables and join indexes</li> </ul> These database objects must <i>not</i> have a primary index.

## SQL and Indexes

### Basic I/O Required by SQL DML Requests

The five basic SQL data manipulation language statements, SELECT, INSERT, DELETE, UPDATE, and MERGE, can all use available indexing and partitioning to access and manipulate table rows. Alternatively, the Optimizer applies a full-table scan if that is more efficient or if there is no applicable index for the request.

The following table summarizes the I/O types required by these four statements.

Statement	Types of I/O Required
SELECT	Read
INSERT	Read and Write
DELETE	
UPDATE	
MERGE	

## Role of the WHERE Clause in SQL DML Requests

SELECT, DELETE, and UPDATE requests can all specify a WHERE clause. The WHERE clause filters row selection, limiting access to only those rows or row ranges specified in the clause. It is equivalent to the RESTRICT (or SELECT) operator of the relational algebra.

The MERGE statement uses an ON clause predicate to filter row selection.

Using a carefully constructed WHERE or ON clause to limit the number of rows processed by a DML statement enhances throughput markedly. If no WHERE clause is specified, then all rows participate in the selection, deletion, or update activities specified by a particular DML statement.

**Note:** The WHERE CURRENT OF clause available for DELETE and UPDATE statements in embedded SQL and stored procedures acts on preselected spool rows, so it does not do I/O for each individual row in the cursor.

## How the Optimizer Uses the WHERE and ON Clauses

The Optimizer uses WHERE or ON clause predicates as the foundation on which to build its access and join plans. Many predicates, or search conditions as they are often called, invoke the selection of indexes or row partition elimination by the Optimizer if the appropriate indexes or row partitioning have been defined on the columns specified by the predicate condition.

Specify these conditions whenever possible because they promote optimal performance and are selected by the Optimizer whenever table statistics, environmental cost parameters, and other demographics suggest they would decrease response time.

## Predicates That Invoke Indexes

The following search conditions typically invoke indexes if they are defined on the columns specified in a WHERE clause predicate.

- column\_name = value
- column\_name IS NULL
- column\_name IN (subquery)
- column\_name IN (constant\_list)
- table\_1.column\_x = table\_1.column\_y
- table\_1.column\_x = table\_2.column\_x
- search\_condition\_1 AND ... search\_condition\_n
- search\_condition\_1 OR ... search\_condition\_n
- column\_name = ANY
- column\_name = SOME
- column\_name = ALL
- any range condition satisfied by a value-ordered index

## Predicates That Invoke Full-Table Scans

The following search conditions typically invoke a full-table scan of the base table or of a secondary index subtable when used as a HAVING or WHERE clause condition because they do not specify a specific value or set of values.

- nonequality comparisons
- column\_name IS NOT NULL
- column\_name NOT IN (subquery)
- column\_name NOT IN (constant\_list)
- column\_name BETWEEN value\_1 AND value\_2
- join\_condition\_1 OR join\_condition\_2
- NOT (condition)
- table\_1.column\_x \* expression = value
- table\_1.column\_x \* expression = table\_1.column\_y
- column\_1 || column\_2 = value
- column\_name LIKE expression
- INDEX (column\_name)
- SUBSTRING (column\_name)
- SUM (numeric\_expression)
- MINIMUM (numeric\_expression)
- MAXIMUM (numeric\_expression)
- AVERAGE (numeric\_expression)
- COUNT (value\_expression)
- DISTINCT
- ANY
- ALL
- no WHERE clause specified for query

Some Teradata Database indexes can optionally be stored in the order of their values as well as in the more standard hash order. If a value-ordered index is defined on the column that satisfies a range predicate from this list, then a full-table scan is not necessary to satisfy the condition and the Optimizer merely scans a subset of the value-ordered index subtable instead. Furthermore, for tables and join indexes defined with a partitioned primary index, row partition elimination can limit a full-table scan to just those partitions that are not eliminated.

## Primary Indexed Tables, NoPI Tables, and Column-Partitioned Tables

### Purpose of Primary Indexes

Teradata Database distributes tables horizontally across all AMPs on a system.

The system assigns primary-indexed table rows to AMPs based on the value of their primary index (see “[Row Allocation for Primary-Indexed Tables](#)” on page 235). The determination of which hash bucket, and hence which AMP the row is to be stored on, is made solely on the rowhash value of its primary index.

You can also create tables and join indexes that do not have a primary index. These tables are referred to as NoPI tables, column-partitioned tables, and column-partitioned join indexes. Nonpartitioned NoPI tables are generally used as staging tables for FastLoad and Teradata Parallel Data Pump Array INSERT bulk data loads. Teradata Database uses a slightly different mechanism to assign NoPI table rows to their AMPs (see “[Row Allocation for Teradata Parallel Data Pump](#)” on page 237).

Many retrievals from primary-indexed tables also use the primary index, though others might use a secondary index, a hash or join index, a full-table scan, or a mix of several different index types. Retrievals from NoPI tables often benefit from using secondary or join indexes to avoid full-table scans.

Note that Teradata does *not* use the term primary index in the same way it is commonly used to describe a clustered index in an indexing system based on B+ trees.

## Mandatory Use of Primary Indexes

Except for NoPI tables, column-partitioned tables, column-partitioned join indexes (see “[NoPI Tables, Column-Partitioned Tables, and Column-Partitioned Join Indexes](#)” on [page 280](#)), and global temporary trace tables, each Teradata Database table, hash, and join index must have a primary index.

If you fail to explicitly define a primary index when you create a table or join index, Teradata Database uses one of several situation-dependent default strategies to define a primary index or not to define a primary index for the table or join index. See “[Primary Index Defaults](#)” on [page 263](#) for details. Note that Teradata recommends to explicitly specify a primary index or specify no primary index rather than depend on the system-determined default.

## Restrictions on Primary Indexes

The following restrictions apply to all primary indexes.

- A maximum of one primary index can be defined on a table.  
You cannot define a primary index for a global temporary trace table, a NoPI table, or a column-partitioned table. See “[CREATE GLOBAL TEMPORARY TRACE TABLE](#)” and “[CREATE TABLE](#)” in *SQL Data Definition Language Detailed Topics*.
- No more than 64 columns can be specified for a primary index definition.
- You cannot include columns having XML, BLOB, CLOB, Period, ARRAY, VARRAY, or JSON data types in any primary index definition.
- You cannot compress the values of primary index columns.
- You cannot compress the values of the partitioning columns defined in row partitioning.
- You cannot column partition the table.

## Primary Index Storage

Primary indexes are stored in-line with the row they index as a rowhash value and row uniqueness ID (see “[Rowhash and RowID](#)” on [page 197](#)). If a table is defined with row partitioning, the system also stores the internal partition number of the row.

Because of this, you cannot drop or change a primary index when the table is nonempty. Instead, you must drop the entire table definition and then recreate it with the new primary index.

The primary index for a table is stored in hash-order for a standard index or in hash-order within partition for a row-partitioned index.

## Related Topic

For additional usage information and design tips about primary indexes, see [Chapter 9: Primary Indexes and NoPI Objects](#).

## Secondary Indexes

While nonunique secondary indexes provide faster set selection (the term *set selection* refers to the subset of rows returned by a query that does not select all the rows from a table, and is typically used to describe a *multirow* subset that is returned because of a NUSI condition in a request), unique secondary indexes are useful for retrieving single rows, particularly when a base table either has a nonunique primary index (see “[Nonunique Primary Indexes](#)” on [page 265](#)) or does not have a primary index, as is the case for NoPI tables, column-partitioned tables, column-partitioned join indexes (see “[NoPI Tables, Column-Partitioned Tables, and Column-Partitioned Join Indexes](#)” on [page 280](#)).

Not all queries are based on primary index retrievals. It is common for WHERE clause search criteria to be based on columns other than those making up the primary index for a table, and such occasions often benefit greatly if a secondary index is defined for the column defined for the search criterion.

Similarly, secondary indexes, along with join indexes, are the only way a single row or a row subset can be retrieved from an NoPI table, column-partitioned table, or column-partitioned join index without a full-table or full-column scan for a column-partitioned table (see “[NoPI Tables, Column-Partitioned Tables, and Column-Partitioned Join Indexes](#)” on [page 280](#)).

Secondary indexes are frequently selected by the Optimizer when a search condition cannot be satisfied with a primary index retrieval. The Optimizer also selects secondary indexes for query plans when they completely or partially cover a query.

Many retrievals use the primary index, though others might use a secondary index, a hash or join index, a full-table scan, scan of a subset of row partitions, a full-column scan of a column-partitioned table, or a mix of several different index types.

Note that Teradata does *not* use the term secondary index in the same way it is commonly used to describe a nonclustered index in an indexing system based on B+ trees.

## Mandatory Use of Secondary Indexes

Secondary indexes are never required and generally are not recommended for tables that are accessed by OLTP applications.

They are highly recommended for decision support applications that return multiple rows by design because they often provide superior performance to full-table scans.

While UPIs are supported for FastLoad and MultiLoad, USIs are not. As a general rule, therefore, a table that is frequently updated using these utilities should not be defined with USIs. Otherwise, you must drop the USIs before the load operation begins and then recreate them after the load completes. You can often substitute a MERGE request for a MultiLoad job, and MERGE *does* support loading into target tables defined with USIs (see “[MERGE](#)” in *SQL Data Manipulation Language*). Note that you can create a USI for a table explicitly (see “[Using Unique Secondary Indexes to Enforce Row Uniqueness](#)” on [page 457](#) for details) or implicitly with a PRIMARY KEY or UNIQUE constraint when the constraint is not implemented as a UPI (for details, see “[PRIMARY KEY Constraints](#)” on [page 654](#), “[UNIQUE Constraints](#)” on [page 656](#), and “[Primary Index Defaults](#)” on [page 263](#)).

Note, too, that FastLoad does not support NUSIs (secondary indexes, if needed, must be dropped or not created before the load and created after the FastLoad). See [Chapter 10: “Secondary Indexes,”](#) for details.

## Restrictions on Secondary Indexes

The following restrictions apply to all secondary indexes.

- Excluding the primary index, you can define a maximum of 32 indexes on a table. These 32 indexes can be any combination of secondary, hash, and join indexes, *including* the system-defined secondary indexes used to implement PRIMARY KEY and UNIQUE constraints.  
Each multicolumn NUSI that specifies an ORDER BY clause counts as two consecutive indexes in this calculation.  
You cannot define secondary indexes for a global temporary trace table. See “CREATE GLOBAL TEMPORARY TRACE TABLE” in *SQL Data Definition Language Detailed Topics*.
- No more than 64 columns can be included in a secondary index definition.
- You cannot include columns having XML, BLOB, CLOB, ARRAY, VARRAY, Period, or JSON data types in any secondary index definition. Only nonunique secondary indexes are supported on columns having a Geospatial data type.

## Types of Secondary Index

A secondary index can be defined as unique or nonunique. There are several subtypes of nonunique secondary indexes (see [Chapter 10: “Secondary Indexes,”](#) for details).

## Secondary Index Storage

Secondary indexes are stored in secondary index subtables.

Teradata Database distributes unique secondary index rows on the rowhash for the index. The determination of which hash bucket the row belongs to, and hence which AMP the row is to be stored on, is made on the value of the unique secondary index. Because the rowhash bucket for a secondary index is usually different from the rowhash bucket for the referenced row, unique secondary indexes are generally stored on a different AMP than the row they point to.

Nonunique secondary indexes are not hash-distributed and are always stored on the same AMP as the rows they point to.

Unique secondary indexes are always stored in hash order on an AMP, while nonunique secondary indexes can be stored either in hash order or in value order on an AMP, depending on their intended use.

## Related Topic

For additional usage information and design tips about secondary indexes, see [Chapter 10: “Secondary Indexes,”](#)

# Join Indexes

Join indexes have several uses, including the following.

- Define a prejoin table on frequently joined columns (with optional aggregation) without denormalizing the database.
- Create a full or partial replication of a base table with a primary index on a foreign key column table to facilitate joins of very large tables by hashing their rows to the same AMP as the large table.

You can also use a column-partitioned join index to create a full or partial replication of a base table to facilitate various manipulations of the data.

- Define a summary table without denormalizing the database.

You can define a join index on one or several tables. Single-table join index functionality is an extension of the original intent of join indexes, hence the confusing adjective join used to describe single-table join indexes.

Join indexes, along with secondary indexes, are the only way a single row or a row subset can be retrieved from either an NoPI table or a column-partitioned table without a full-table scan, scan of subset of row partitions, or full-column scan for a table (see “[NoPI Tables, Column-Partitioned Tables, and Column-Partitioned Join Indexes](#)” on page 280).

Join indexes, like hash indexes, are not indexes in the usual sense of the word. They are system-maintained tables that cannot be accessed directly using DML requests.

The Optimizer includes a multitable join index in a plan in the following situations.

- The index covers all or part of a join query, thus eliminating the need to redistribute rows to make the join.
- A query requests that one or more columns be aggregated, and an appropriate aggregate join index exists, thus eliminating the need to perform the aggregate computation.

The Optimizer includes a join index in a plan in the following situations.

- The index covers all or part of a join query, thus eliminating the need to redistribute rows to make the join. A join index can be used for a partial cover only if it has been defined using any of the following components.
  - The UPI of the underlying base table.
  - The keyword ROWID.

You can only specify ROWID in the outermost SELECT of the CREATE JOIN INDEX statement. See “[CREATE JOIN INDEX](#)” in *SQL Data Definition Language Detailed Topics*.

- The NUPI of the underlying base table and the keyword ROWID in a join on a NUPI column.
- The NUPI of the underlying base table or the USI of the underlying base table.

It is preferable to specify NUPI and ROWID over NUPI and USI.

In this case, the Optimizer uses the ROWID, base table UPI or NUPI, or base table USI to join the join index rows with their underlying base table rows to pick up the base table columns necessary to complete the cover.

- A query requests aggregation on a column set and an existing single-table aggregate join index covers the specified aggregation, thus eliminating the need to perform the aggregate computation.

## Mandatory Use of Join Indexes

Join indexes are never required and are rarely recommended for tables that are accessed frequently by OLTP applications. They are highly recommended for decision support applications because they often provide superior performance to large table joins and aggregate computations. See [Chapter 11: “Join and Hash Indexes,”](#) for details.

## Restrictions on Join Indexes

The following restrictions apply to all join indexes.

- Excluding the primary index, you can define a maximum of 32 indexes on a table. These 32 indexes can be any combination of secondary, hash, and join indexes.  
Each multicol NUSI that specifies an ORDER BY clause counts as two consecutive indexes in this calculation.  
You cannot define join indexes for a global temporary trace table. See “CREATE GLOBAL TEMPORARY TRACE TABLE” in *SQL Data Definition Language Detailed Topics*.
- No more than 64 columns from a table in a join index definition.
- No more than 128 columns can be defined for a row compressed join index, 64 each for the fixed and variable parts.
- There is no limit on how many total columns can be defined in an uncompressed join index other than system restrictions on the amount of SQL text required to define them.
- Although the Optimizer substitutes only one multitable join index per referenced table in a query, it also considers additional single-table join indexes for inclusion in the join plan after the optimal multitable join index has been substituted and evaluated for the plan.
- You cannot include columns having XML, BLOB, CLOB, Period, ARRAY, or VARRAY data types in any join index definition. Only the portions of JSON data that are extracted may be used as part of a join index.

Additionally, you cannot create a join index that has a partitioned primary index if that join index uses row compression (see [“Compression Types Supported by Teradata Database”](#) on page 695).

## Types of Join Index

There are four basic join index types defined over two orthogonal dimensions: single or multitable and simple or aggregate. These are better thought of as ways to group functionality than as different types of join index.

- Single-table simple

You can also create a column-partitioned version of a single-table simple join index.

- Single-table aggregate
- Multitable simple
- Multitable aggregate

All join index types can also be sparse, meaning that only the set of rows that satisfy a WHERE clause restriction in the index definition are included in the index (see “[Sparse Join Indexes](#)” on page 556).

See [Chapter 11: “Join and Hash Indexes”](#) and [SQL Data Definition Language](#) for further information.

## Join Index Storage

For the most part, join index storage is identical to standard base table storage except that the rows of join indexes (as distinguished from column value compression. See “[Compression Types Supported by Teradata Database](#)” on page 695 for definitions of the two types of compression) can be compressed. Join index rows are hashed on their primary index. Join index tables can be indexed, and their indexes are stored just like primary and secondary indexes for any other base table.

The major difference in storage between join indexes and base tables is the manner in which the repeated field values of a join index are stored. Repeated field value storage is too complex to describe in this summary. See [Chapter 11: “Join and Hash Indexes”](#) under “[Join Index Storage](#)” on page 598 for details about how repeated field values are stored.

When possible, join indexes also inherit the multivalue compression characteristics of their underlying base tables (see “[Default Column Multivalue Compression for Join Index Columns When the Referenced Base Table Column Is Compressed](#)” on page 501).

See “[Compression Types Supported by Teradata Database](#)” on page 695 for a comparison of row and multi-value compression.

A join index can be stored either in hash-order or in value-order, depending on its intended use. Value-ordered join indexes are restricted to certain data types and field lengths. See “[CREATE JOIN INDEX](#)” in [SQL Data Definition Language](#) for details.

## Related Topic

For additional usage information and design tips about join indexes, see [Chapter 11: “Join and Hash Indexes”](#).

## Hash Indexes

Hash indexes are used for the same purposes as single-table join indexes. Because single-table join indexes are more flexible in their definition options, and are essentially equivalent in function to hash indexes, you should use equivalent single-table join indexes in preference to hash indexes.

The principal difference between hash and single-table join indexes are listed in the following table.

Hash Index	Single-Table Join Index
Cannot have secondary indexes.	Can have secondary indexes.
Rows are partitioned on the hash of the primary index of its base table unless otherwise explicitly specified.	Rows are partitioned on an explicitly defined primary index.
Column list cannot contain aggregate functions.	Column list can contain aggregate functions.
CREATE HASH INDEX syntax is similar to the syntax for CREATE INDEX.	CREATE JOIN INDEX syntax specifies a subquery, making it superficially similar to the syntax for CREATE VIEW.
Cannot be specified for a NoPI table, a column-partitioned table, or a global temporary trace table.	Can be specified for a NoPI table or column-partitioned table.

Hash indexes create a full or partial replication of a base table with a primary index, usually on a foreign key column table to facilitate joins of very large tables by hashing them to the same AMP.

You can define a hash index on one table only. With the exception of not allowing aggregates, the functionality of hash indexes is the same as that of single-table join indexes.

Hash indexes, like join indexes, are not indexes in the usual sense of the word. They are tables that you cannot access directly using DML requests.

The Optimizer includes a hash index in a query plan when the index covers all or part of a join query, thus eliminating the need to redistribute rows to make the join. In the case of partial query covers, the Optimizer uses certain implicitly defined elements in the hash index to join it with its underlying base table to pick up the base table columns necessary to complete the cover.

## Mandatory Use of Hash Indexes

Hash indexes are never required.

Instead, you should use single-table join indexes as needed.

## Restrictions on Hash Indexes

The following restrictions apply to all hash indexes.

- Excluding the primary index, you can define a maximum of 32 indexes on a table.  
Each multicol NUSI that specifies an ORDER BY clause counts as 2 consecutive indexes in this calculation.

These 32 indexes can be any combination of secondary, hash, and join indexes, *including* the system-defined secondary indexes used to implement PRIMARY KEY and UNIQUE constraints.

You cannot define hash indexes for global temporary trace tables, for NoPI tables, or for column-partitioned tables. See “[NoPI Tables, Column-Partitioned Tables, and Column-Partitioned Join Indexes](#)” on page 280), “CREATE GLOBAL TEMPORARY TRACE TABLE,” “CREATE HASH INDEX,” and “CREATE TABLE” in *SQL Data Definition Language Detailed Topics*.

- You cannot define a partitioned hash index.
- You cannot include columns having XML, BLOB, CLOB, Period, ARRAY, VARRAY, or JSON data types in any hash index definition.

## Hash Index Storage

For the most part, hash index storage is identical to base table storage except that hash index rows are, in most cases, row compressed automatically. Hash index rows are hashed on their primary index (which is always defined as nonunique). Hash index tables can be indexed explicitly, and their indexes are stored just like nonunique primary indexes for any other base table. Unlike join indexes, hash index definitions do not permit you to specify secondary indexes.

The major difference in storage between hash indexes and base tables is the manner in which the repeated field values of a hash index are stored. Repeated field value storage is too complex to describe in this summary. See [Chapter 11: “Join and Hash Indexes”](#) under “[Join Index Storage](#)” on page 598 for details about how repeated field values are stored. A hash index can be stored either in hash-order or in value-order, depending on its intended use.

## Related Topic

For additional usage information and design tips about hash indexes, see [Chapter 11: “Join and Hash Indexes”](#).

## Reference Indexes

A Reference Index is an internal structure that the system creates whenever a referential integrity constraint is defined between tables using a PRIMARY KEY or UNIQUE constraint on the parent table in the relationship and a REFERENCES constraint on a foreign key in the child table. Teradata Database does not create or maintain REFERENCE indexes for batch referential integrity relationships.

The index row contains a count of the number of references in the child, or foreign key, table to the PRIMARY KEY or UNIQUE constraint in the parent table.

A maximum of 64 referential constraints can be defined for a table. Similarly, a maximum of 64 tables can reference a single table in a referential constraint. Therefore, there is a maximum of 128 Reference Index descriptors that can be stored in the table header per table.

However, the Reference Index subtable for a table stores only the Reference Indexes that define its relationship with its child tables, so only 64 Reference Indexes are stored in the subtable per base table.

Apart from capacity planning issues, Reference Indexes have no user visibility.

## Related Topic

For information about reference index sizing issues, see “[Reference Index Sizing Equation](#)” on page 867.

# Indexing and Hashing

Hashing and indexing are two different methods for facilitating data access.

Teradata Database does not use true indexing. What we refer to as indexes are either rowhash values or, in the case of join indexes, data tables that substitute for base tables in join queries.

## Indexing Mechanisms

In the case of indexing, a data value (or values, if the index is composite) from a row acts as an index key to that row. The index associates the index key with a relative row address that reports the location of the row on disk. Indexes are stored in order of their index key values and are said to be value-ordered.

Think of an index on a database table as you would think of the index to a book. A book index typically provides two data values: a character string indicating the indexed topic and a page number or numbers indicating the location of that information in the book.

If the topic is unique, then there is only one page number listed for it. If the topic is nonunique, then there might be several pages listed for it.

To find a piece of information in a book, you can either consult the index, find the topic you need, and then turn directly to the appropriate page (think of this as an indexed retrieval) or you can start reading the book from the beginning and stop only when you find the information you seek (think of this as a full-table scan).

## Hashing Mechanisms

In the case of hashing, an index key data value is transformed by a mathematical function called a hash function to produce an abstract value not related to the original data value in an obvious way. Hashed data is assigned to hash buckets, which are memory-resident routing structures that correspond in a 1:1 manner to the relationship a particular hash code or range of hash codes has with an AMP location.

Hashing is similar to indexing in that it associates an index key with a relative row address.

Hashing differs from indexing in the following ways.

- There is no obvious correspondence between a hash code and the location of the row it refers to, even though the hash code identifies the location of the row.
- Different data values can and often do produce identical hash codes. This results in what is called a hash collision, because any rows having the identical rowhash value would be stored in the same disk location if there were no means for preventing that from happening.

Ordinarily, the rows for primary-indexed tables in Teradata Database are distributed, or horizontally row partitioned, among the AMPs based on a hash code generated from the primary index, or primary hash key, for the table (see “[Row Allocation for Primary-Indexed Tables](#)” on page 235). The only exceptions to this are rows from global temporary trace tables, NoPI tables, column-partitioned tables, and column-partitioned join indexes (see “[Row Allocation for Teradata Parallel Data Pump](#)” on page 237 and “[Row Allocation for FastLoad Operations Into Nonpartitioned NoPI Tables](#)” on page 238).

Because Teradata Database hash functions are formally mature and mathematically sound, rows with unique primary indexes are always distributed in a uniformly random fashion across the AMPs, even when there is a natural clustering of key values. This behavior both avoids nodal hot spots and minimizes the number of key comparisons required to perform join processing when two rows have the same rowhash value.

All primary indexes are hashed for distribution, and while the stored value for the primary index is retained as data, it is the hashed transformation of that value that is used for distribution and primary index retrieval of the row. The primary indexes of hash and join indexes can, in some circumstances, be stored in value-order (there are restrictions on the data type and column width of any value-ordered primary index for a hash or join index: see “[CREATE HASH INDEX](#)” and “[CREATE JOIN INDEX](#)” in *SQL Data Definition Language* for details), but their rows are still distributed to the AMPs based on the hash of their primary index value.

Nonpartitioned and column-partitioned NoPI table rows that are inserted into Teradata Database using Teradata Parallel Data Pump ARRAY insert processing are distributed among the AMPs based on a hash code generated from their Query ID (see “[Row Allocation for Teradata Parallel Data Pump](#)” on page 237).

Nonpartitioned NoPI table rows that are FastLoaded onto a system are distributed among the AMPs using a randomization algorithm that is different from the standard hashing algorithm (see “[Row Allocation for FastLoad Operations Into Nonpartitioned NoPI Tables](#)” on page 238).

Unique secondary indexes are also hashed and distributed to their appropriate index subtables based on that hash value, where they are stored in rowID order. Rows stored in this way are said to be hash-ordered. Non-primary index retrievals usually benefit from defining one or more secondary hash keys on a table.

To summarize, although Teradata Database uses the term *index* for these values, they are really hash keys, not indexes.

## Tradeoffs Between Hashing and Indexing

This topic compares the relative advantages and disadvantages of hashing and indexing and concludes that hashing is the better methodology for the Teradata massively parallel architecture.

### Advantages of Hashing Over Indexing

Why use hashing instead of indexing? The following list of advantages explains why hashing is superior to indexing for the Teradata parallel architecture.

- For the majority of queries, hashing provides consistently better performance because rows are always distributed evenly across the AMPs, providing a consistently balanced workload across parallel units. Because Teradata Database uses B\*-trees on a fixed-length rowID rather than on the column values themselves, there are never more than two levels of the B\*-tree, so there is no need to reorganize because the trees never become unbalanced and unordered. This is true for standard operation. System reconfiguration *is* necessary whenever you add nodes or AMPs to your system.

The Teradata file system B\*-tree is implemented on the master index, cylinder index, and data block descriptor structures (see “[Master Index](#)” on page 246, “[Cylinder Index](#)” on page 247, and “[Data Block](#)” on page 248).

B+ trees on variable-length values can become unbalanced and then require a lengthy reorganization.

- Primary indexes are not stored in an index subtable. Instead, the rowhash for an individual row is stored directly as part of the row header for the row. This direct access to row data offers the following advantages over indexing of the same data.
  - Enhanced performance because of fewer I/Os
  - Decreased storage overhead because there are no primary index subtables
  - Hashing is far better suited for the parallel database architecture pioneered by Teradata than is indexing. Because each table in a Teradata database is distributed horizontally across the AMPs on a row-by-row basis, the storage and retrieval algorithms and indexes used by standard file systems for storing and retrieving data are not as efficient as the distributed parallel Teradata architecture.

Because of its hashed data placement and parallel architecture, Teradata Database requires only a one-time decision on the primary index for each table: the system takes care of everything else.

Compare this with other relational products where a DBA must perform tasks such as determining keys and locations and coding space allocation for each partition before even beginning to create a table. Then the DBA must embed the partitioning assignments into the CREATE TABLE statement and forever after monitor and maintain partition sizes and perform periodic table reorganizations.

- By carefully defining primary index columns on frequently used join constraints, you can force rows that are frequently joined to be co-located on the same AMP. This is critical to join performance, because rows must be on the same AMP if they are to be joined. By co-

locating join-constrained rows, you avoid the need to redistribute the rows from one or more tables across the BYNET to the AMP where they will be joined with rows located on that AMP.

## Disadvantages of Other Row Partitioning Methods

There are several problems with traditional data placement schemes, particularly in a VLDB environment.

- Data set partitioning guarantees hot spots and interconnect log jams.
- This method requires the DBA to batch data as it arrives in the database and then manually control how it is partitioned. While it is possible to maintain a balanced data load with these scheme, it is *not* possible to maintain a balanced processing load.

Table rows are received, batched, and partitioned serially. This means that the newest data is always clustered together. Because a high percentage of data warehouse processing involves comparing current data with historical data to detect trends, this results in the majority of users attempting to access the same, co-located, data simultaneously.

- VLDB data warehouse environments are typically configured as massively parallel or clustered processing units. This requires some sort of network interconnect channel.

There is no way to know whether rows from tables that are to be joined are co-located or not, and there is no way to ensure that they are. This means that table joins in a data set partitioning situation typically need to transmit vast quantities of data across their interconnect channels in order to join tables, severely reducing system throughput in the process.

- Pure range partitioning is not a useful method for most data warehouse applications. When data access is repeatedly based on the same constraints, then range partitioning is an excellent choice. Data can be partitioned into multiple sets based on the repeatedly used constraints.

As the following list of potential problems indicates, there are more problems with pure range partitioning in a massively parallel data warehouse environment than there are solutions provided.

- Business data is rarely well-balanced. Because of this, the distribution of the data must be carefully analyzed in order to determine how best to partition the data across network nodes.
- Some way must be found to ensure that commonly joined table rows are co-located in order to minimize traffic on the interconnect channels.
- Some method of balancing table data across multiple nodes must be determined.
- Because the demographics of data always change as a function of time, these considerations must be revisited repeatedly to ensure continued smooth performance.

Notice that all these issues require intensive intervention on the part of the DBA.

Distribution and other demographic data must be collected and analyzed (and an inexpensive, yet reliable method of doing the collection and analysis must be found and implemented), algorithms must be discovered or developed and then tested, and every aspect of the data must be monitored continually.

The Teradata Database solution to range partitioning is the partitioned primary index, which hashes table rows to the AMPs using the same rowhash method that assigns nonpartitioned primary index rows, but adds the ability to further assign those rows to user-defined range partitions. See “[Row-partitioned and Nonpartitioned Primary Indexes](#)” on page 266 for further information about partitioned primary indexes.

- Random partitioning can have disadvantages.

Random, or round-robin, partitioning is formally related to hash partitioning. Unlike hashing, which uses an algorithm with known partitioning properties to distribute table rows, random partitioning uses a random number generator to distribute rows. The resulting distribution is even, but unrepeatable. This method makes it impossible to know where a table row is stored, so it can never be accessed directly.

Random partitioning may cause data to be redistributed for join and aggregate processing, resulting in suboptimal system performance.

- Schema partitioning almost always requires extensive redistribution of table rows for join and aggregate processing in a networked configuration.

This method, which is a scheme to assign specific table groups (“schemas”) to specific physical processors or nodes, has proven useful for optimizing the retrieval performance of specific tables in small, single node systems.

When applied to a multiple node parallel environment, however, its deficiencies are readily apparent.

- Most join cases require all rows to be redistributed across the interconnect for each query, reducing system throughput in the process.
- Node balancing problems are always made worse when schema partitioning is used for anything other than very small tables.

## Advantages of Indexing Over Hashing

There are two categories of query that often perform better with value-ordered indexing.

- Range queries
- Retrievals having selection criteria that involve only part of a multicolumn hash key

First consider range queries. The following SQL SELECT statement requests all rows from the parts table with a part number ranging between 3517 and 3713, inclusive. The primary key for the table is part\_number.

```
SELECT part_number, part_description
  FROM parts_table
 WHERE part_number BETWEEN 3517 AND 3713;
```

Depending on the configuration and the database manager, such a query might perform better if executed against a table that uses a value-ordered primary index key rather than a hash-ordered hash key.

To enhance the ability of hash keys to retrieve range query data, Teradata Database provides two mechanisms: the capability to store index rows in the order of their index values and the capability to store base table rows within partitions, including range partitions, after they have been hashed to an AMP. See “CREATE INDEX,” “CREATE JOIN INDEX,” and “CREATE

“TABLE” in *SQL Data Definition Language* and “CASE\_N” and “RANGE\_N” in *SQL Functions, Operators, Expressions, and Predicates*.

The only way to avoid the partial hash key issue is to have a thorough understanding of your applications and data demographics before you define your indexes. In other words, if your applications will use partial key selection criteria, either define the index on those frequently retrieved columns or define a secondary index on them. The optimal solution is very dependent on the individual circumstances and there is no single correct way to design for this particular situation.

## Teradata Database Hashing Algorithm

The Teradata Database hashing algorithms are proprietary mathematical functions that transform an input data value of any length into a 32-bit value referred to as a rowhash, which is then used to assign a row to an AMP.

Primary-indexed rows are allocated to the AMPs based on the hashed value of their primary index (see “[Row Allocation for Primary-Indexed Tables](#)” on page 235).

Rows that are inserted into nonpartitioned or column-partitioned NoPI tables using Teradata Parallel Data Pump ARRAY INSERTs are not allocated to the AMPs based on the value of their primary index, but on the value of the Query ID for their request (see “[Row Allocation for Teradata Parallel Data Pump](#)” on page 237).

Rows that are inserted into nonpartitioned or column-partitioned NoPI tables using INSERT ... SELECT operations are appended AMP-locally to their target table, so there is no need to hash them to a destination AMP.

The rows that are FastLoaded into nonpartitioned NoPI tables (see “[NoPI Tables, Column-Partitioned Tables, and Column-Partitioned Join Indexes](#)” on page 280) are assigned to their destination AMPs using a different hashing algorithm than the hashing algorithm used for primary-indexed rows and rows inserted into an nonpartitioned or column-partitioned NoPI table using Teradata Parallel Data Pump ARRAY INSERT operations. FastLoaded rows are hashed on a combination of their AMP vproc ID and an internal counter rather than on a Query ID (see “[Row Allocation for FastLoad Operations Into Nonpartitioned NoPI Tables](#)” on page 238 for details).

You can select from among several hashing algorithms to use with your system, depending on the character sets your site uses. The only difference among the available hashing algorithms is how they are optimized to handle different character sets such as those used by double-byte Asian languages and European languages that use various diacritical marks. You *cannot* define multiple hashing algorithms for the same system.

### Hash Bucket Number

The first 16 or 20 bits of the rowhash are a hash bucket number, which is used to define the number of the hash bucket to which the hashed row belongs. The number of hash buckets for a Teradata system depends on the DBS Control parameters CurHashBucketSize and

NewHashBucketSize. See *Utilities: Volume 1 (A-K)* and “[Hash Maps](#)” on page 229 for more information on hash bucket sizes.

The remaining 16 (or 12) bits are a remainder from the operation of the hash function on the original input value. There is no advantage to a 16-bit hash bucket number, and you should use 20-bit numbers unless you have a good reason not to. The next several pages explain the reasons for this.

The values input to the hashing algorithm are the column set values that constitute an index on the table in question.

The following graphics illustrate the structure of a Teradata Database hash bucket number and the remainder (modulo) of the hashing operation on the primary index value for systems with 65,536 hash buckets and 1,048,576 hash buckets, respectively.

Hash Bucket Number	16-bit Remainder
1094A082	
Hash Bucket Number	12-bit Remainder
1094A083	

A 32-bit rowhash provides 4.2 billion possible rowhash values, which reduces hash collisions to a level that eliminates their impact on retrieval performance for all intents and purposes.

Because the number of possible index column values for a given domain can easily exceed that number of values, further information, called the *uniqueness value*, is required to make each row uniquely identifiable. This situation occurs for tables hashed on a nonunique primary index, where different rows can have the same primary index value, and so the same rowhash value, and also for tables hashed on a unique primary index if there is a hash collision.

The expectation is that all new customers will configure their systems to use the larger number of buckets no matter how many AMPs their system has. However, if your system currently uses a 16-bit hash bucket size, you must decide when it is most advantageous for you to convert to the larger number of buckets.

This decision is based on the following factors.

- Whether the number of AMPs configured on the system results in a large enough imbalance to mandate a change to a 20-bit hash bucket size.
- When there is sufficient down time available to you to run the reconfiguration or to perform a sysinit and restore your data from an archive.

There is no simple formula to determine when this conversion should be done. The following table shows the percentage of imbalance that can be expected for a given number of AMPs. As

the number of AMPs increases, the percentage of imbalance also increases. A system with 1,000 or more AMPs should consider converting to a 20-bit hash bucket size.

Number of AMPs	Number of Hash Buckets	Low Number of Buckets Per AMP	High Number of Buckets Per AMP	Percentage Imbalance
100	65,536	655	656	0.15
200	65,536	327	328	0.31
300	65,536	218	219	0.46
400	65,536	163	164	0.61
500	65,536	131	132	0.76
750	65,536	87	88	1.14
1,000	65,536	65	66	1.52
2,000	65,536	32	33	3.03
3,000	65,536	21	22	4.55
4,000	65,536	16	17	5.88
5,000	65,536	13	14	7.14
6,000	65,536	10	11	9.09
7,000	65,536	9	10	10.00
8,000	65,536	8	9	11.11
9,000	65,536	7	8	12.50
10,000	65,536	6	7	14.29

## Uniqueness Value

Teradata Database uses an additional 32-bit system-generated uniqueness value to ensure the uniqueness of any rowID (see “[Hash Collisions](#)” on page 228).

Base table row distribution to the AMPs for primary-indexed tables is always determined by the rowhash value for the primary index on the table, while index subtable values are hashed on the value of the secondary index they represent.

For NoPI tables and column-partitioned tables, the rowhash value for rows loaded by Teradata Parallel Data Pump or FastLoad is always assigned by the destination AMP.

Hashing on Query ID or hashing as done by FastLoad determine the destination AMP, not the hash bucket.

See “[Row Allocation for Teradata Parallel Data Pump](#)” on page 237 and “[Row Allocation for FastLoad Operations Into Nonpartitioned NoPI Tables](#)” on page 238.

If the primary index for a table is row-partitioned, then its partitioning expression controls the order in which the rows are stored once they have been hashed to an AMP, but the selection of the AMP a row hashes to is controlled only by its rowhash value.

Retrieval operations use the primary index value for a row plus any existing secondary, join, or hash indexes that the Optimizer determines would reduce the cost of the activities required to retrieve the row.

Primary index operations use only the rowhash value, while secondary index operations use the entire rowID value.

## Hash Collisions

The principal problem faced by hashing is hash collisions: situations in which the rowhash value for different rows is identical, making it difficult for a system to discriminate among the hash synonyms when one unique row is requested for retrieval from a set of hash synonyms.

### Minimizing Hash Collisions

To minimize this problem, Teradata Database defines 4.2 billion hash values. The AMP software adds a system-generated 32-bit uniqueness value to the rowhash value. The resulting 64-bit value prefixed with an internal partition number is called the rowID, and this value uniquely identifies each row in a system, making a scan to retrieve a particular row among several having the same rowhash a trivial task. A scan must check each of the rows to determine if it has the searched for value and not another value that has the same rowhash value.

The following graphics illustrate the structure of a Teradata Database rowID for a PPI table on systems defined to have 65,536 hash buckets and 1,048,576 hash buckets, respectively.

64-bit Internal Partition Number	16-bit Hash Bucket Number	16-bit Remainder	32-bit Uniqueness Value
----------------------------------	---------------------------	------------------	-------------------------

1094B056

64-bit Internal Partition Number	20-bit Hash Bucket Number	12-bit Remainder	32-bit Uniqueness Value
----------------------------------	---------------------------	------------------	-------------------------

1094B084

If the rowID were for an nonpartitioned table, the internal partition number is 0 and is compressed in the rowID in the row header of the row and in a referencing rowID in a secondary index. For a table with 2-byte partitioning, the internal partitioning is compressed to two bytes.

# Hash Maps

A hash map is a mechanism for determining which AMP a row is stored on, or, in the case of the Open PDE hash map, the destination AMP for a message sent by the Parser subsystem. Each cell in the map array corresponds to a hash bucket, and each hash bucket is assigned to an AMP. Hash map entries are maintained by the BYNET.

## Number of Hash Buckets Per System

A hash map defines the correspondence between a hash bucket and an AMP, mapping specific hash buckets to specific AMPs. There are either 65,536 or 1,048,576 hash buckets in the map, divided as evenly as possible among the AMPs on the system (see “[The Problem of Skew](#)” on [page 230](#) for information about the importance of distributing rows evenly among the AMPs).

The size of a hash bucket is related to the number of hash buckets available to the system, as the following table indicates.

FOR systems with this many hash buckets ...	THE size of each bucket is this many bits ...
65,536	16
1,048,576	20

The number of hash buckets for a system is defined by the following DBS Control parameters (see *Utilities: Volume 1 (A-K)*).

- **CurHashBucketSize**

This parameter specifies the number of bits used to identify a hash bucket in the current system configuration. You cannot change it.

The default width for new installations and for systems that are already using 20-bit hash buckets is 20 bits.

- **NewHashBucketSize**

The System Initializer (sysinit) and Reconfiguration (reconfig) utilities use this parameter to modify the number of bits used to identify a hash bucket. If the values for CurHashBucketSize and NewHashBucketSize differ, either sysinit or reconfig reads the value for NewHashBucketSize and converts the hash bucket size to that width.

Specify this value *only* when you want to change the system configuration. The reconfig utility or the sysinit utility copies the value of NewHashBucketSize over the current value for CurHashBucketSize when the reconfig or sysinit job completes.

The default value for new installations and for systems that are already using 20-bit hash buckets is 20 bits.

If you upgrade a system that currently uses 16-bit hash buckets, the system continues to use 16-bit hash buckets unless you set the value for NewHashBucketSize to 20 and then run either the System Initializer utility or the Reconfiguration utility to change it. For more information on sysinit and reconfig, see *Support Utilities*.

If you want your system to continue to use 16-bit hash buckets, be sure the NewHashBucketSize parameter is set to 16 before you run either the sysinit or reconfig utility.

## Hash Map Consistency Across Configurations

Hash maps are consistent across configurations that have the same number of hash buckets. Any two systems having the same number of AMPs and hash buckets also have identical primary hash maps. Fallback hash maps often differ somewhat from system to system depending on how their AMPs are clustered and on their development across time.

### The Problem of Skew

Each node in a Teradata system has a set of five database hash maps, a NoPI hash map, and an Open PDE hash map, making a total of 7 hash maps. Each of these hash maps performs a different function. Teradata Database hash maps have either 65,536 or 1,048,575 entries, one for each hash bucket (see “[Number of Hash Buckets Per System](#)” on page 229). This number was carefully selected to balance the requirement that the hash maps be memory-resident with the need to have enough buckets to avoid the skewing problems seen when an uneven number of buckets are assigned to each AMP. Unless the number of AMPs in the system divides evenly into 65,536 or 1,048,575, some skewing is unavoidable.

As an example of how a relatively small number of hash buckets quickly leads to significant skew, consider the following case. Suppose there were an imaginary Teradata system that had only 3,643 hash buckets rather than 65,536 hash buckets. Now suppose this was a large system having 200 AMPs. The result of having so few hash buckets is that some AMPs have 18 buckets and others have 19. This might seem like a small difference, but a quick computation tells a different story. The distribution of hash buckets maintained by this system guarantees a 5.26% skew of rows across the AMPs as indicated by the following calculation.

$$\left(\frac{1}{19}\right) \times (100) = 5.26\% \text{ skew}$$

The following table shows how the number of hash buckets per AMP varies with the number of AMPs in the configuration. Note that configurations with a number of AMPs that does not divide evenly into 65,536 spread the remainder buckets among the AMPs as evenly as possible. Increasing the number of hash buckets to 1,048,575 permits a still more even distribution of buckets among the AMPs.

Number of AMPs in Configuration	Number of Hash Buckets/AMP	Number of AMPs With This Many Hash Buckets	Percent Skew
4	16,384	4	0.0000
5	13,107	4	0.0076
	13,108	1	

Number of AMPs in Configuration	Number of Hash Buckets/AMP	Number of AMPs With This Many Hash Buckets	Percent Skew
11	5,957	2	0.0170
	5,958	9	
60	1,092	11	0.0900
	1,093	4	

Similarly, a system with 1,000 AMPs has 65 hash buckets on some AMPs and 66 hash buckets on others. In this particular case, the AMPs having 66 hash buckets also perform 1.5% more work than those with 65 hash buckets, a skew percentage of 1.515. The work per AMP imbalance increases as a function of the number of AMPs in the system for those cases where 65,536 is not evenly divisible by the total number of AMPs.

There is no calculation to determine when a system needs to convert from 16-bit hash buckets to 20-bit hash buckets, but from the figures in the following table, you can see that a system with 1,000 or more AMPs should seriously consider converting to 1,048,575 hash buckets. The table indicates how a system with 65,536 hash buckets affects the skew of row distribution for tables as a function of the number of AMPs on the system.

Number of AMPs	Low Number of Hash Buckets Per AMP	High Number of Hash Buckets Per AMP	Imbalance Percentage
100	655	656	0.15
200	327	328	0.31
300	218	219	0.46
400	163	164	0.61
500	131	132	0.76
750	87	88	1.14
1,000	65	66	1.52
2,000	32	33	3.03
3,000	21	22	4.55
4,000	16	17	5.88
5,000	13	14	7.14
6,000	10	11	9.09
7,000	9	10	10.00
8,000	8	9	11.11
9,000	7	8	12.50

Number of AMPs	Low Number of Hash Buckets Per AMP	High Number of Hash Buckets Per AMP	Imbalance Percentage
10,000	6	7	14.29

Note that there is also a relationship between the number of hash buckets on a system and the percentage of disk space that is wasted on AMPs that have fewer hash buckets than others. This is referred to as the hash imbalance problem, where some AMPs own  $n$  hash buckets for a table, while others own  $n-1$ . The one bucket difference translates into wasted disk storage.

This relationship can be expressed as a percentage of wasted disk space as follows.

$$\text{Totalwastedspace}_{n \text{ buckets}} = \frac{\text{NumAMPs} - \text{NumAMPs with } n+1 \text{ buckets}}{\text{NumAMPs}} \times \text{Space}_{n+1 \text{ AMP}}$$

where:

Equation element ...	Specifies the ...
Total wasted space <sub>n buckets</sub>	percentage of total wasted space on AMPs that control $n$ hash buckets.
NumAMPs	There are $N-m$ such AMPs on any system, where $N$ is the number of AMPs on the system and $m$ is the number of AMPs on the system having $n+1$ hash buckets.
NumAMPs with $n+1$ buckets	number of AMPs in the system that own $n+1$ hash buckets.
Space <sub><math>n+1</math> AMP</sub>	<p>space occupied by one hash bucket on an AMP that owns <math>n+1</math> hash buckets.</p> <p>This value is determined using the following equation.</p> $\text{Space}_{n+1 \text{ AMP}} = \frac{1}{n+1}$ <p>where <math>n+1</math> is number of hash buckets owned by an AMP with <math>n+1</math> hash buckets.</p>

The percent wasted disk space for 65,536 versus 1,048,576 hash buckets for a given number of system AMPs decreases greatly for systems having the larger number of hash buckets.

An additional advantage of a larger number of hash buckets is the finer granularity it provides for rowhash-level locking (see *SQL Request and Transaction Processing* for details about locking levels), which both improves the performance of primary index retrievals and

decreases the probability of rowhash-level locking conflicts. This is particularly true for very large tables.

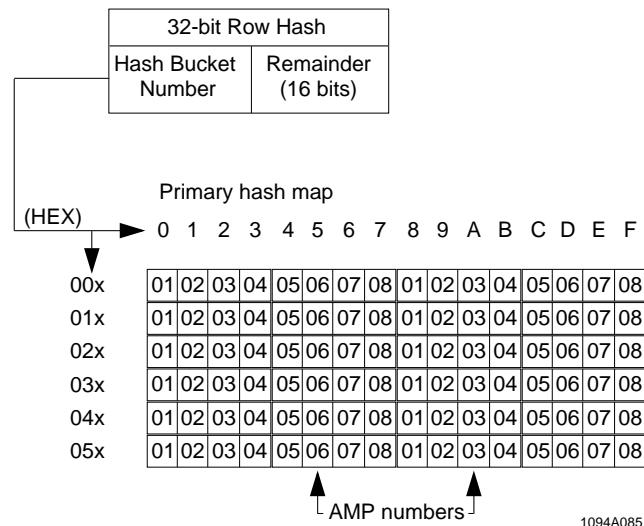
## Definitions for the Six Database Hash Maps

The 6 hash maps maintained on each node in a Teradata system are defined as follows.

- The current configuration primary map

This map maintains the hash bucket-AMP correspondences for the primary copies of all rows on the system.

The following graphic is a logical diagram of the primary hash map for a system with 65,536 hash buckets. Physical hash maps do not look like this.



- The current configuration fallback map

This map maintains the hash bucket-to-AMP correspondences for the fallback copies of all rows from tables defined with fallback. The fallback hash map is not identical to the current configuration primary hash map.

- The reconfiguration primary map

This map indicates which primary rows migrate to which AMPs during a reconfiguration.

- The reconfiguration fallback map

This map indicates which fallback row copies migrate to which AMPs during a reconfiguration.

- The NoPI hash map

- The bit map hash map

This map is used to optimize hash-related queries on the AMPs (see “[NUSI Bit Mapping](#)” on page 479).

Another hash map, called the Open PDE hash map, is maintained by the Parallel Data Extensions system to indicate the destination for Parser messages that are addressed by hash bucket number.

The increased size of the hash maps for systems with 1,048,576 hash buckets increases the amount of memory consumed by them, as indicated by the following table.

Hash Map	Size for 16-Bit Bucket	Size for 20-Bit Bucket
DBS primary	128 KB	2 MB
DBS fallback	128 KB	2 MB
DBS reconfiguration primary	128 KB	2 MB
DBS reconfiguration fallback	128 KB	2 MB
DBS NoPI	128 KB	2 MB
DBS bitmap	64 KB	1 MB
Open PDE	4 MB	4 MB

These changes result in a total increase of memory consumption of approximately 8.5 MB for a system configured with 20-bit hash buckets. If the system is configured with 16-bit hash buckets, increased memory consumption is restricted to the 4 MB used for the Open PDE hash map.

The differences in the number of hash buckets for 16-bit buckets and 20-bit buckets is documented in the following table. These numbers do not change if the number of AMPs in the system increases.

Bucket Size (Bits)	Number of Hash Buckets per AMP
16	4
20	64

## Hash Map Lookup

When a row is hashed, the resulting hash bucket number points to a cell entry in a hash map. This entry defines the AMP to which the row hashes.

While the BYNET uses only the hash bucket number to distribute a row to its assigned AMP, the receiving AMP uses the entire rowID.

# Hash-Based Table Partitioning to AMPs

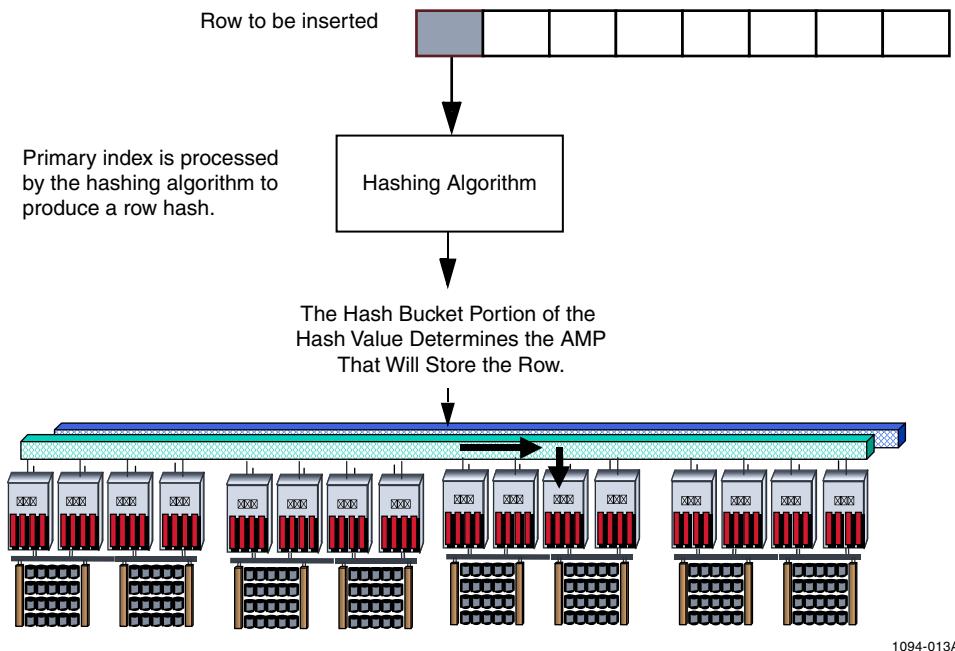
All Teradata Database tables are partitioned, or distributed, horizontally by default. This means that the individual rows of each Teradata Database table are assigned to different AMPs in a way that balances their distribution equally in order to facilitate parallel processing.

Teradata Database uses the hash bucket portion of the computed rowhash value for each row to determine which AMP it should be assigned to (see “[Teradata Database Hashing](#)

[Algorithm](#) on page 225 and [“Row Allocation for Teradata Parallel Data Pump” on page 237](#) for information on how this is done for nonpartitioned and column-partitioned NoPI tables). If the primary index for a table is row-partitioned, then further processing is required to compute which row partition it belongs to on its assigned AMP (see [“Row-Partitioned Row Allocation” on page 242](#)). See [“Row Allocation for FastLoad Operations Into Nonpartitioned NoPI Tables” on page 238](#) for a description of how Teradata Database FastLoads rows into nonpartitioned NoPI tables.

## Row Allocation for Primary-Indexed Tables

The process for assigning a particular primary-indexed table row to an AMP is depicted in the following graphic. The logical process involved in this assignment is described in the table following the graphic.



1094-013A

The logical process involved in row assignment for a table SQL insert operation is roughly the following.

- 1 The INSERT request passes its syntactical, lexical, and security constraints.
- 2 A row header is prepended to each row that includes information about whether the table into which it is to be inserted is partitioned.
- 3

IF the row has ...	THEN the ...
a primary index	primary index is submitted to the hashing algorithm for processing.

IF the row has ...	THEN the ...
no primary index	Query ID for the request is submitted to the hashing algorithm for processing (see “ <a href="#">Row Allocation for Teradata Parallel Data Pump</a> ” on <a href="#">page 237</a> ).

4

IF the row has a...	THEN ...
a primary index	the hashing algorithm produces a rowhash value. Part of that value (the high-order 16 or 20 bits, depending on a system setting) is used as the hash bucket number.
no primary index	the hashing algorithm produces a rowhash value consisting of the hash bucket number and a remainder.  Teradata Database uses this value only to assign a row to an AMP. Once the row reaches its AMP, the AMP software generates a different hash bucket number for it that is stored in its rowID.

- 5 The value of the hash bucket number is checked against the hash map and the destination AMP for the row is determined.
- 6 The Dispatcher sends the row set across the BYNET to its destination AMP.
- 7 Check uniqueness. If a unique primary or secondary index exists, it will be used to ensure uniqueness. If neither exists and no UNIQUE specifier exists on a column (which implicitly creates a unique primary or secondary index), and the table is a SET table, then check for uniqueness by scanning the rows having the same hash value.
- 8 The AMP uses both partition number, if it exists, and the hash code to determine where the row should be stored. The partition number provides the higher-order grouping attribute.

IF the primary index is this type ...	THEN AMP software ...
Nonpartitioned	Assigns the next uniqueness value. When available unique values are exhausted, the next hash bucket owned by this AMP is substituted.  When Teradata Database does duplicate row checks, it considers nulls to be equal, while in SQL conditions, null comparisons evaluate to UNKNOWN.  See “ <a href="#">Inconsistencies in How SQL Treats Nulls</a> ” on <a href="#">page 674</a> for information about how nulls are interpreted by commercially available relational database management systems.

IF the primary index is this type ...	THEN AMP software ...
Partitioned	<ol style="list-style-type: none"> <li>1 Uses the partitioning expressions for the PPI to calculate the internal partition number for the row.</li> <li>2 The internal partition number determines the row partition to which the row is assigned.</li> <li>3 Assigns the next uniqueness value.</li> <li>4 Go to Stage 9.</li> </ol>

- 9 The file system assigns the row to the appropriate data block and stores it in order of its internal partition number, hash value, and uniqueness value.

See “[Row-Partitioned Row Allocation](#)” on page 242 for more information and for information on PPIs.

The row is then stored in the table in rowID order.

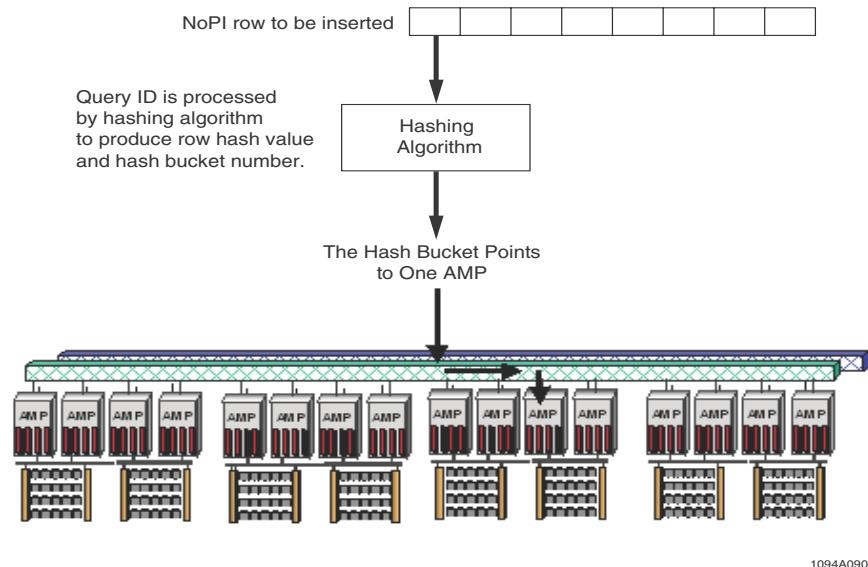
## Row Allocation for Teradata Parallel Data Pump

Because NoPI tables and column-partitioned tables have no primary index, Teradata Database must build their rowhash value differently than it does for primary-indexed rows, generating a random rowhash value in such a way that for a new request, data is usually sent to a different AMP than the AMP the previous request distributed its rows to. This method attempts to balance the distribution of rows across the AMPS as much as is possible without having a primary index to hash. The Query ID for the request is used for this purpose because it uses the PE vproc ID in its high digits and a counter-based value in its low digits, which supports both single-session and multiple-sessions INSERT operations equally well.

Rows in a NoPI table or column-partitioned table are not hashed based on a primary index, so Teradata Database can distribute them to any AMP as desired. To maximize this benefit, Teradata Parallel Data Pump array inserts on a NoPI table or column-partitioned table always pack as many rows as there are in the request from the client into the same AMP step. The rows are sent to the same AMP in one insert step. The distribution of the rows in a Teradata Parallel Data Pump job that are to be inserted into a NoPI table or column-partitioned table uses a random generator that determines the AMP destination for each insert step (see next paragraph). The random generator choose a different AMP from the one that the previous request distributed rows to, spreading the rows across the AMPS in a way that avoids skewing.

Teradata Database creates a random rowhash value to use for the distribution step, using the value of the query ID for the request as input to the randomization algorithm. The logical process involved in this assignment is described in the table following the graphic in the topic “[Row Allocation for Primary-Indexed Tables](#)” on page 235.

Refer to the following graphic for a pictorial illustration of the row allocation process for Teradata Parallel Data Pump array inserts into NoPI and column-partitioned table rows.



There are two cases for a simple insert operation.

- Only a single data row is processed.
- Multiple data rows are processed for a Teradata Parallel Data Pump array INSERT request.

For both cases, the system copies the randomly generated rowhash value into the rowID of each row being processed as soon as it has been built. A randomly generated rowhash value is generated once per request so that in the case of a Teradata Parallel Data Pump array insert, the same rowhash value is used for all of the rows in the request. That means that Teradata Database sends all rows for that request to the same AMP, possibly in the same step.

The source spool for rows that are inserted into NoPI tables and column-partitioned tables by INSERT ... SELECT operations are appended AMP-locally to their target table, so there is no need to hash them to a destination AMP.

You should consider the following recommendations for Teradata Parallel Data Pump array inserts into nonpartitioned and column-partitioned tables.

- If the order of data application is not important, set the SERIALIZATION option to OFF.
- Do not set the number of sessions for a Teradata Parallel Data Pump job to a larger value than the number of AMPS on the system.

## Row Allocation for FastLoad Operations Into Nonpartitioned NoPI Tables

When you use the FastLoad utility to bulk load rows into an nonpartitioned NoPI table, Teradata Database accesses the configuration map and builds a list of system AMPS, including offline AMPS.

A round robin protocol sends the first buffer of rows to its own AMP. The next buffer is assigned to the next highest AMP, and so on. When the highest AMP is reached, Teradata Database wraps the list around and selects the lowest AMP again. For a down AMP, Teradata Database automatically redirects the buffer to the fallback owner of the assigned hash bucket,

and the receiver inserts the rows into the fallback subtable. Down AMP recovery occurs when the down AMP comes back online.

To prevent row redistributions for NoPI tables during reconfiguration and restore operations, AMPs build a separate hash map to be used specifically for NoPI tables.

## Row Allocation for FastLoad Operations Into Column-Partitioned Tables

You cannot use the FastLoad utility to bulk load rows directly into a column-partitioned database table. Instead, use FastLoad to load the rows into a staging table and then use an INSERT ... SELECT request to copy them into the target column-partitioned table.

## Row Allocation for INSERT Operations Into Nonpartitioned NoPI Tables and Column-Partitioned Tables

Column-partitioned tables and join indexes are primarily used for running analytics and undertaking data mining analyses where they are loaded with an INSERT ... SELECT request and then remain fairly static, with occasional minor maintenance. When the table is or its row partitions are deleted, the anticipated scenario starts over again. Column-partitioned tables are not intended for OLTP or other activities in which a table is frequently updated. While you can insert individual rows into NoPI tables and column-partitioned tables, singleton inserts into these types of tables should not be undertaken frequently.

As is true for all insert operations into nonpartitioned NoPI tables and column-partitioned tables, Teradata Database randomly distributes the individual rows or blocks of rows to the AMPs or copies them locally because they do not have a primary index that can be used to hash-distribute them across the AMPs.

The anticipated usage for column-partitioned tables is roughly 88% large INSERT ... SELECT operations or array-INSERTs into an empty table or empty row partitions, 2% singleton INSERTs, 7% updates, and 3% deletions.

## Row Allocation for INSERT ... SELECT Operations Into NoPI Tables and Column-Partitioned Tables

Because NoPI tables and column-partitioned tables do not have a primary index, Teradata Database randomly distributes their individual rows or blocks of rows to the AMPs or copies them locally rather than hash-distributing them in the way that rows are allocated for primary-indexed tables.

A common way to load multiple rows into NoPI tables and column-partitioned tables is to use INSERT ... SELECT requests. Besides just loading rows into such tables, you can also specify the HASH BY and LOCAL ORDER BY clauses to distribute and sort the rows from the SELECT subquery of an INSERT ... SELECT request before inserting them into a NoPI or column-partitioned table. You specify these options either individually or together within a request, but you can only specify them if the target table or underlying table of the target view does not have a primary index (see *SQL Data Manipulation Language* for details about the syntax and usage of the HASH BY and LOCAL ORDER BY options with INSERT ... SELECT requests).

The HASH BY option redistributes the selected rows by the hash value you specify. You can follow a HASH BY specification with a LOCAL ORDER BY option that orders the rows locally and then inserts them locally into the target table or underlying table of the target view. This is useful if the result of the SELECT subquery does not provide an even distribution.

If the target table or underlying table of the target view is also column-partitioned, you can use these options to distribute equal values of a column to the same AMP, which might enable effective autocompression of the column partitions with the columns on which the hash value is calculated.

Specifying HASH BY RANDOM redistributes the data blocks of the selected rows randomly, and is a handy way to redistribute rows when there is no particular column set on which to hash distribute them. Distributing data blocks is more efficient than distributing individual rows and generally provides a very even distribution; however, distributing individual rows sometimes provides a more even distribution.

HASH BY RANDOM(1,2000000000), for example, is a useful function to specify for cases where you want to redistribute individual rows rather than data blocks of rows.

Assume the following definitions for the examples that follow.

```
CREATE TABLE orders (
    o_orderkey      INTEGER NOT NULL,
    o_custkey       INTEGER,
    o_orderstatus   CHARACTER(1) CASESPECIFIC,
    o_totalprice    DECIMAL(13,2) NOT NULL,
    o_ordertsz     TIMESTAMP(6) WITH TIME ZONE NOT NULL,
    o_comment       VARCHAR(79))
UNIQUE INDEX (o_orderkey),
PARTITION BY COLUMN;

CREATE TABLE orders_staging AS orders
WITH NO DATA
NO PRIMARY INDEX;

CREATE TABLE tpi (
    a INTEGER,
    b INTEGER,
    c INTEGER)
PRIMARY INDEX (a);

CREATE TABLE tnopil (
    a INTEGER,
    b INTEGER,
    c INTEGER)
NO PRIMARY INDEX;

CREATE TABLE tnopi2 (
    a INTEGER,
    b INTEGER,
    c INTEGER)
NO PRIMARY INDEX;
```

The following INSERT ... SELECT request selects rows from the NoPI table *orders\_staging* and redistributes them randomly as data blocks before copying them locally into the

column-partitioned table *orders*. This request does not specify a LOCAL ORDER BY clause to order the selected rows locally before inserting them into the target table *orders*.

```
INSERT INTO orders
SELECT *
FROM orders_staging
HASH BY RANDOM;
```

In contrast, the following INSERT ... SELECT request selects rows from the NoPI table *orders\_staging* and redistributes them randomly as individual rows before copying them locally into the column-partitioned table *orders*. This request does not specify a LOCAL ORDER BY clause to order the selected rows locally before inserting them into the target table *orders*.

```
INSERT INTO orders
SELECT *
FROM orders_staging
HASH BY RANDOM(1,2000000000);
```

The following INSERT ... SELECT request uses the HASH BY specification with *o\_totalprice* from the column-partitioned table *orders*. This corresponds to the fourth expression in the select list of the SELECT subquery, which is *o\_totalprice*. Teradata Database distributes the spool generated for the SELECT on the value of *o\_totalprice* rather than the value of *o\_totalprice* from *orders\_staging*.

```
INSERT INTO orders
SELECT o_orderkey, o_custkey, o_orderstatus, o_totalprice,
       o_ordertsz, o_comment
FROM orders_staging
HASH BY o_totalprice;
```

Specifying a LOCAL ORDER BY clause orders the selected rows locally on their internal combined partition number (which is always 1 for column-partitioned tables) using the partitioning of the target if it is row-partitioned and then on any specified ordering expressions before inserting them locally into the target. If the target table or view is also column-partitioned, a LOCAL ORDER BY specification can provide more effective autocompression of its column partitions with the columns on which the ordering is done.

For example, the following INSERT ... SELECT request redistributes the rows selected from *orders\_staging* by the hash value of *o\_orderkey* to more evenly distribute them. Teradata Database then orders the selected rows locally to enable better run length compression on the *o\_ordertsz* column partition before copying the data locally into the column-partitioned target table *orders*.

```
INSERT INTO orders
SELECT *
FROM orders_staging
HASH BY o_orderkey
LOCAL ORDER BY o_ordertsz;
```

The HASH BY and LOCAL ORDER BY clauses are also useful for INSERT ... SELECT operations on nonpartitioned NoPI tables. The following example selects rows from the primary-indexed table *tpi* and redistributes them randomly as data blocks before locally coping them into the NoPI table *tnopi\_1*.

```
INSERT INTO tnopi_1
```

```
SELECT *
FROM tpi
HASH BY RANDOM;
```

In contrast, the following example selects rows from the primary-indexed table *tpi* and redistributes them randomly as individual rows before locally copying them into the NoPI table *tnopi\_1*.

```
INSERT INTO tnopi_1
SELECT *
FROM tpi
HASH BY RANDOM(1, 2000000000);
```

The next example selects rows from the nonpartitioned NoPI table *tnopi\_1* and redistributes them randomly as data blocks before locally copying them into the NoPI table *tnopi\_2*.

```
INSERT INTO tnopi_2
SELECT *
FROM tnopi_1
HASH BY RANDOM;
```

In contrast, the following example selects rows from the nonpartitioned NoPI table *tnopi\_1* and redistributes them randomly as individual rows before locally copying them into the NoPI table *tnopi\_2*.

```
INSERT INTO tnopi_2
SELECT *
FROM tnopi_1
HASH BY RANDOM(1, 2000000000);
```

## Row-Partitioned Row Allocation

If the primary index for your table is row partitioned, the following events occur when a row is processed for storage.

- 1 The row is hashed to its AMP (see “[Hash-Based Table Partitioning to AMPS](#)” on page 234) by the hash values of its primary index column set.
- 2 Its internal partition number is calculated by the AMP using software stored in the table header that computes the internal partition number from the row partitioning expressions defined for the table.
- 3 The file system inserts the row in logical order by internal partition number and hash value/uniqueness value.

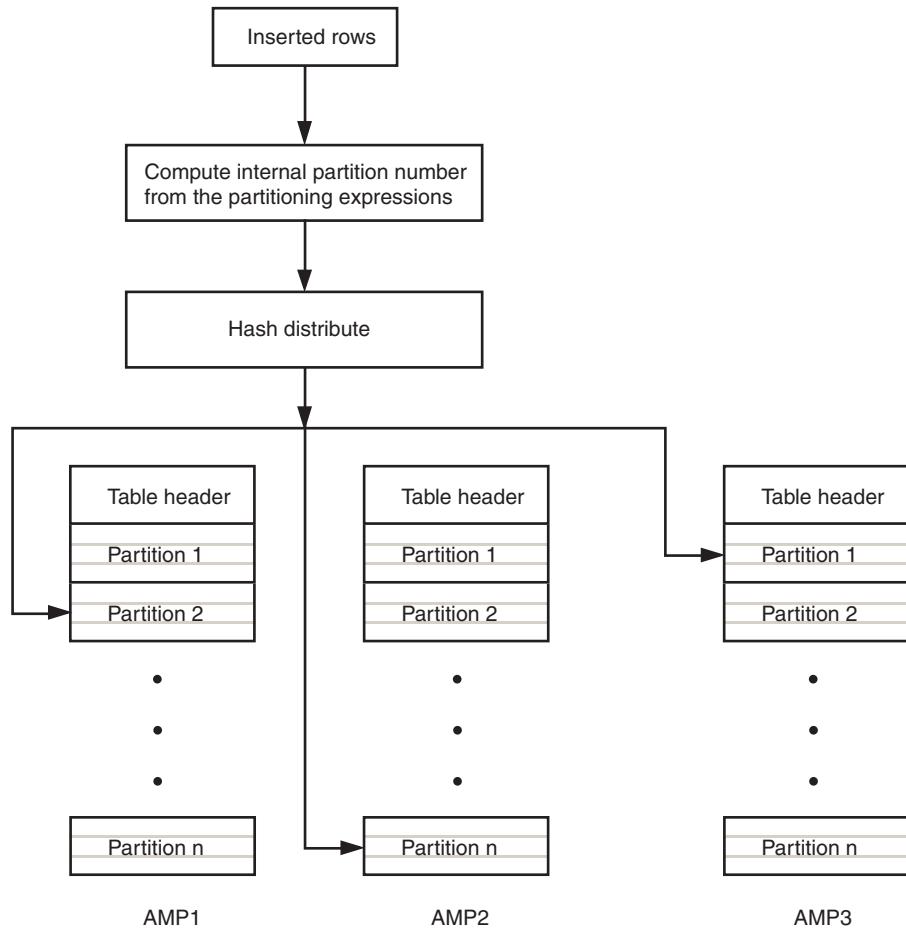
If there are *n* row partitioning expressions, the rows are effectively grouped as follows:

- a The value of the first row partitioning expression
- b The value of the second row partitioning expression
- ...
- c The *value of the n<sup>th</sup>* row partitioning expression

These row partitioning expression values are combined into a single combined partition that is mapped to an internal partition number that preserves their sequential order.

The final ordering is by the hash value of the primary index columns and its associated uniqueness value (see “[Uniqueness Value](#)” on page 227).

The following diagram provides a high-level picture of the process for a PPI table at the point where the rows have arrived at their assigned AMPs.



1094C012

## Hash-Related Functions

Teradata SQL provides several functions that you can use to analyze the hashing characteristics of your existing indexes and candidate indexes. These functions are documented fully in *SQL Functions, Operators, Expressions, and Predicates*.

The following table summarizes the hash-related functions. Note that they can be nested.

This function ...	Returns the ...
HASHAMP	number of the AMP associated with an expression.
HASHBAKAMP	number of the backup AMP associated with an expression.

This function ...	Returns the ...
HASHBUCKET	<p>number of the hash bucket associated with a hash row expression.</p> <p>The range of possible values returned by HASHBUCKET varies depending on how your system is configured for hash bucket size.</p> <ul style="list-style-type: none"><li>For systems with 16-bit hash buckets, the possible values returned by HASHBUCKET range from 0 - 65,535.</li><li>For systems with 20-bit hash buckets, the possible values returned by HASHBUCKET range from 0 - 1,048,575.</li></ul>
HASHROW	hexadecimal rowhash value for an expression.

**Note:** Unity Director does not support these hash-related functions.

## Using HASHROW

HASHROW is especially useful for evaluating the following things.

- Row distribution efficiency of proposed primary index column.
- Relative efficiency of various data types for proposed or existing primary index columns.

As shown in the following example set, you can use the output from HASHROW to compute the average number of rows that cause hash collisions.

### Example 1

This example uses the HASHROW function on a unique primary index.

Assume that the *emp\_no* column is the UPI for the *employee* table. To find the number of hash collisions generated by the values in *emp\_no*, submit the following request.

```
SELECT(COUNT(*) (FLOAT)) / (COUNT(DISTINCT HASHROW(emp_no)))
FROM employee;
```

If there are no hash collisions, the result ratio is close to 1. The larger the returned value, the more hash collisions occur and the less efficient the analyzed column expression analyzed is as a primary index.

If a proposed primary index contains non-unique columns, perform the following check.

- Submit the request in the example against the proposed primary index column expression and save the result.
- Resubmit the request against a column expression where each index value is unique and save the result.
- Compare the result step 1 against the result of step 2.

The comparison should indicate clearly which index generates fewer hash collisions.

### Example 2

The following example uses the HASHROW function against a the same primary index column analyzed in “[Example 1](#)” on page 244, but retyped as DECIMAL(2,2).

- Determine how many hash collisions occur with the existing data type by submitting the same query used in “[Example 1](#)” on page 244.

```
SELECT (COUNT(*) (FLOAT)) / COUNT(DISTINCT HASHROW (emp_no))
FROM employee;
```

- Determine whether hash collisions would be reduced by changing the data type to DECIMAL(4,3) by submitting the following two requests (both are required because of the restrictions on the DISTINCT option).

```
SELECT COUNT(DISTINCT emp_no (DECIMAL(4,3)))
FROM employee;
```

```
SELECT COUNT(DISTINCT HASHROW(emp_no (DECIMAL(4,3))))
FROM employee;
```

- Divide the numbers returned by the two requests to calculate the average number of hash collisions that occur with the new data type.
- Compare this average with the result of the first request to determine if the change reduces the collision count.

### Example 3

When used together, HASHAMP, HASHBUCKET, and HASHROW indicate the distribution of primary rows among the AMPs.

For example, the following query shows the row distribution of the *employee* table based on the primary index *emp\_no*, from which you can quickly see whether distribution is even or uneven.

```
SELECT HASHAMP (HASHBUCKET(HASHROW(emp_no))), COUNT(*)
FROM employee
GROUP BY 1
ORDER BY 1;
```

IF the distribution is ...	THEN ...
uneven	<p>try one of the following.</p> <ul style="list-style-type: none"> <li>Change the primary index to a column set that uniquely identifies each row.</li> <li>Add a column to the primary index to force uniqueness.</li> <li>Change the data type of one or more index columns (see “<a href="#">Hashing and Data Types</a>” on page 250).</li> </ul>
even	<p>create your production table defining the primary index based on the most high-performing column set. For example, either of the following examples where preliminary performance results are based on trials on a test system.</p> <ul style="list-style-type: none"> <li>A primary index based on a column that provides the best performance.</li> <li>A primary index based on the same column, but having a particular data type that provides the best performance (see “<a href="#">Hashing and Data Types</a>” on page 250).</li> </ul>

# Hashed Row Access

## Master Index

The master index is a memory-resident file system data structure that contains pointers to every cylinder index on a given AMP in entries referred to as Cylinder Index Descriptors, or CIDs. When a cylinder does not contain data, it is not listed in the master index.

Each master index entry contains the following data sorted on table ID and rowhash.

- Lowest table ID in the cylinder
- Lowest partition/rowhash/uniqueness value on the cylinder (associated with the lowest table ID)

The partition value is 0 for NPPI tables and nonpartitioned NoPI tables.

- Highest table ID in the cylinder
- Highest partition rowhash (*not* the uniqueness value) value on the cylinder (associated with the highest table ID)

The partition value is 0 for NPPI tables and nonpartitioned NoPI tables.

- Cylinder number

The file system uses the table ID and rowID (or partition, partition/rowhash combination, or rowhash value) for the row being retrieved to scan the master index for the desired cylinder number.

As an example of how a table ID and rowhash are used to scan the master index, consider the following query.

```
SELECT *
  FROM employee
 WHERE emp_no = 7225;
```

For this example, assume that the decimal rowhash value for 7225, the primary index value for the row in question, is 266 and the table ID for the *employee* table is 100.

Consider the following set of contrived master index entries, assuming all tables are NPPI tables, so their partition number is 0.

Lowest Table ID	Starting RowID	Highest Table ID	Highest Rowhash Value	Cylinder Number
...	...	...	...	...
078	0 + 58234 + 0001	081	0 + 58332	204
100	0 + 00017 + 0001	100	0 + 00073	037
100	0 + 00102 + 0001	100	0 + 00687	169
100	0 + 00785 + 0001	100	0 + 00991	777
100	0 + 01087 + 0001	100	0 + 01116	802

Lowest Table ID	Starting RowID	Highest Table ID	Highest Rowhash Value	Cylinder Number
100	0 + 03662 + 0003	100	0 + 03712	117
100	0 + 04123 + 0001	100	0 + 04255	888
100	0 + 05974 + 0001	100	0 + 06785	753
100	0 + 07353 + 0001	100	0 + 07834	477
123	0 + 00543 + 0001	123	0 + 00786	529
...	...	...	...	...

The actual cylinder number under the column heading Cylinder Number is a 16-digit hexadecimal value, but to make this example more readable, smaller integer values are used.

The process for finding the required cylinder block is as follows.

- 1 The file system performs a binary search on the master index using the target tableID and rowID as the search key.
- 2 This results in locating the cylinder ID for cylinder number 169.

## Cylinder Index

The cylinder index is a disk-resident file system file structure that contains substructures referred to as Subtable Reference Descriptors (SRDs) and Data Block Descriptors (DBDs). The structure of SRDs and DBDs is the same for any system configurations. Each disk cylinder has its own cylinder index. The cylinder index contains pointers to all the data blocks and free sectors on the cylinder it represents.

If AMP memory is large enough, the cylinder index usually remains cached. This normally means that the only I/O required for a primary index lookup is for the data block that contains the row.

Each cylinder index entry contains the following data sorted on table ID and rowhash.

- Table ID for the table stored in the block (rows from different subtables are never mixed in the same data block)
- Lowest rowID value in the block
- Highest partition/rowhash value in the block
  - The partition is 0 for NPPI tables and nonpartitioned NoPI tables.
- Sector number for that block
- Number of sectors in the block

The file system uses the table ID and rowID value for the row being retrieved to scan the cylinder index for the beginning sector number and sector count where the desired block is stored.

Following through with the query from “[Master Index](#) on page 246”, consider the following set of fictional cylinder index entries for cylinder 169. Note that the  $+ n$  values for the rowIDs indicate the uniqueness value.

Table ID	Starting RowID	Highest Partition and Rowhash Value	Beginning Sector Number	Sector Count
100	0 + 00058 + 0001	0 + 00163	1010	9
100	0 + 00184 + 0001	0 + 00329	0789	6
100	0 + 00363 + 0001	0 + 00994	0530	4
100	0 + 01035 + 0001	0 + 01114	0056	5
100	0 + 01185 + 0002	0 + 01190	1138	8
100	0 + 01221 + 0001	0 + 02072	0117	16
100	0 + 02123 + 0001	0 + 02365	0888	7
...	...	...	...	...

The process for finding the required data block is as follows.

- 1 The file system performs a binary search on the cylinder index using the table ID and rowID as the search key.
- 2 This locates the data block that begins with sector 0789 with a sector count of 6.

## Data Block

The data block is a disk-resident file system structure that contains one or more rows from the same subtable. Depending on the configuration of your system, the file system aligns data blocks on either 4K boundaries (aligned configurations) or on 512 byte boundaries (unaligned configurations). Rows from different subtables are never mixed within the same data block, and rows are wholly contained within a single data block. Rows are *never* split between multiple data blocks. If an entire row does not fit within an existing data block, the file system splits the block into as few new data blocks as are required so that no rows span data blocks. This data block splitting process is automatic.

This blocking strategy ensures the following things.

- No more than 511 bytes (or 4095 bytes for aligned configurations) ever remain unused within a data block.

This assumes that the DBS Control parameter `PermDBAllocUnit` is not altered from its default value.

- A data block with logically adjacent rows can be located immediately by looking in the cylinder index without searching through any chains.
- No reorganization because of block split fragmentation is ever required.

Block splits never create overflow blocks, so all blocks are always immediately addressable from the cylinder index.

The data block is the physical I/O unit for the Teradata file system. Data block sizes range between 512 bytes and approximately 1MB.

You can set the default maximum multirow data block size in several ways.

- Set a system-wide default using the DBS Control utility.
- Set or change the block size at the individual table level by establishing a value using the DATABLOCKSIZE parameter of the CREATE TABLE and ALTER TABLE statements, respectively.

If a row being stored is larger than the effective maximum size for a data block, the row is stored in a data block of its own. But a row cannot exceed the absolute maximum size data block supported by the Teradata Database.

Blocks within an individual table can vary in size, and the file system adjusts their sizes dynamically as required. You can also define a merge block ratio for each table that enables the file system to merge small data blocks into a single large data block when a full-table modification operation occurs on a table. If a specific merge block ratio is not defined for a table, the system default merge block ratio is used unless the merge block feature has been disabled using DBS Control.

Data blocks have a standard header identifying the rows in the data block, the table the rows belong to and information about the space available in the block. The header for data blocks that are compressed at the block level is not compressed. Such data blocks also contain the following additional information:

- The first rowid(16 bytes)
- The last rowid (16 bytes)
- The compressed size of the data block (2 bytes)
- The compression algorithm used (1 byte)
- The checksum produced by the compression process (4 bytes)

See *SQL Data Definition Language* for syntax and usage of the ALTER TABLE and CREATE TABLE statements. The MERGEBLOCKRATIO option for the ALTER TABLE and CREATE TABLE statements is only valid for permanent tables and permanent journal tables. It does not apply to volatile tables or to global temporary tables.

Although the rows within a data block are not stored in any particular physical order, a structure called the Row Reference Array, which contains 2-byte pointers to the first word in each row, stores its data in ascending order of rowID.

The logical storage sequence for rows in an *employee* file might look something like the following table, assuming that *employee* is an NPPI table.

Partition Number	RowID		User Data		
	Rowhash	Uniqueness ID	Emp Number	Last Name	First Name
0	7332	0001	6149	Smith	John
0	7332	0002	6171	Chang	Wei-hee
0	7332	0003	7777	Misawa	Mitsuharu
0	8431	0001	6333	Messiaen	Jean-Luc
0	9758	0001	5010	Wanz	Hans
0	9758	0002	6214	Kulthum	Muhammad
	...	...	...	...	...

## Hashing and Data Types

Though it might not be apparent at first glance, the data type of a column or its precision can affect the way it is hashed, and this property can be an important consideration for database design.

### Hashing On Bit Patterns, Not Values

The Teradata Database hashing algorithm operates on the bit patterns of the data on which it operates: the internal representation of the data, not its external value.

For example, a numeric value stored as a DECIMAL number with one precision produces a different hash code than the same numeric value stored as a DECIMAL value with a different precision (see “[Data Type Considerations](#)” on page 822). Additionally, a number typed as an integer requires only 4 bytes of storage, while its decimal representation might take as many as 8 bytes, depending on the size of the number.

### Considerations for Indexing

You should evaluate the following considerations when assigning data types to index columns.

- Suitability of the type family for indexing
  - Numeric types are best because they are compact and often tend toward uniqueness in business data.
  - Character and byte types are poor because they tend not to be compact and because character and byte data naturally tends to be non-unique.
  - XML, BLOB, CLOB, Period, ARRAY, VARRAY, and JSON data type columns cannot be used to define any type of index.

- Hash collisions
  - “[Example 2](#)” on page 244 and “[Example 3](#)” on page 245 under the topic “[Hash-Related Functions](#)” address this issue.
- Storage space

Greater space requirements for a column translate to fewer data rows per block, which results in slower full-table scans.
- Join and predicate conversions

Any time tables are joined on a column, whether indexed or not, the columns must have the same data type in both tables. If the types do not agree, one must be converted, which results in a performance hit. The same is true for comparisons made in WHERE clause predicates.

# Teradata Database Index Comparisons

The following table summarizes the similarities and differences among Teradata Database primary, secondary, join, and hash indexes.

Attribute	Primary Index	Secondary Index	Join Index	Hash Index
Must be defined	Yes  If the table is not a NoPI, a column-partitioned NoPI, or a global temporary trace table. If the join index is not a column-partitioned join index.	No  This excludes system-defined USIs for PRIMARY KEY and UNIQUE constraints that are required for those constraints if they are not Referential Constraint-associated.	No	No
	No  If the table or join index is a NoPI, a column-partitioned NoPI, or a global temporary trace table.			
Maximum per table	1	32  The combined number of secondary, hash, and join indexes defined on a table cannot exceed 32.  This limit includes the system-defined secondary indexes used to implement PRIMARY KEY and UNIQUE constraints for tables.  Any PRIMARY KEY or UNIQUE NOT NULL constraint that is not defined on a primary index column set is implemented as a USI.  A multicolumn NUSI that specifies an ORDER BY clause counts as two consecutive indexes in this calculation.		

Attribute	Primary Index	Secondary Index	Join Index	Hash Index
Minimum per table	1 If the table has a primary index.	0		
	0 If the table is a NoPI or column-partitioned table. If the join index is a column-partitioned join index.			
Maximum number of columns	64		64 You can specify 64 columns per referenced base table in a multitable join index definition if the index is not row compressed.	
Unique type supported	Yes			No
Nonunique type supported	Yes			
Affects base table row partitioning	Yes	No		
Affects index subtable row distribution	Not applicable.  There are no primary index subtables.  Primary indexes are stored in-line as row data.	Unique	Yes	Not applicable
		Nonunique	No	Nonpartitioned and PPI join and hash index rows are distributed on their primary indexes, just like base table rows.  Column-partitioned NoPI join index rows are not distributed on their primary index value because they have no primary index.

Attribute	Primary Index	Secondary Index	Join Index	Hash Index
Can be partitioned within an AMP	Yes  This is only true for row partitioning.	No	Yes  The primary index for a join index can be a PPI if the join index is not row compressed.  Row compressed join indexes cannot have a partitioned primary index.  Column-partitioned NoPI join indexes cannot have a primary index, but they can be row-partitioned.	No
Can be created and dropped dynamically	No, if the table is populated. Yes, if the table is not populated.	Yes		
Enhances access performance	Yes			
Supports multiple data types	Yes  XML, BLOB, CLOB, ARRAY, VARRAY, Period, and JSON data types are not supported for <i>any</i> kind of index.  Geospatial data types are only supported for NUSIs.			
Stored in a separate file structure	No	Yes		
Maintenance processing overhead	No	Yes		

## Table Access Summary

The following table characterizes 9 access methods available to the Optimizer for accessing tables. Definitions for the access method abbreviations used in that table are provided in the following table.

Access Method Abbreviation	Definition
UPI	Unique Primary Index
UPPI	Unique Partitioned Primary Index If you row-partition a unique primary index, all the partitioning columns must be included in the index definition.
NUPI	Nonunique Primary Index
NUPPI	Nonunique Partitioned Primary Index
USI	Unique Secondary Index
NUSI	Nonunique Secondary Index
JI	Join Index
HI	Hash Index
FTS	Full-Table Scan With or without partition elimination.

For many table join operations, a join index is often a more efficient means for accessing data. See “[Join Indexes](#)” on page 215 and [Chapter 11: “Join and Hash Indexes.”](#)

### Table Access Summary

Access Method	Relative Efficiency	Number of AMPs Accessed	Number of Rows Returned	Spool Space Required by Query?
UPI	Very efficient.	1	1	No.
UPPI	Very efficient.  The relative efficiency is the same for single-level and multilevel PPI tables.	1	1	No.
NUPI  Non-unique PPI	Efficient when selectivity is high and skew is low and if limited to one internal row partition by row partition elimination.  Otherwise, performance degrades as a function of the number of internal row partitions that must be accessed.  The relative efficiency is the same for single-level and multilevel PPI tables.	1	Multiple.	Depends. <ul style="list-style-type: none"><li>• If a query returns a single response, then no spool is required.</li><li>• If a query returns more rows than can be handled by a single response, then spool is required.</li></ul>
USI	Very efficient.  USI access is normally the only way to access a single row in a NoPI or column-partitioned table without a full-table scan; however, if a NUSI on a NoPI or column-partitioned table only indexes a single row, then it can also be used.	2  The maximum number of AMPs accessed for a USI row retrieval is two. If a base table row and its USI row happen to hash to the same AMP, then the retrieval only accesses one AMP.	1	No.

Table Access Summary

Access Method	Relative Efficiency	Number of AMPs Accessed	Number of Rows Returned	Spool Space Required by Query?
NUSI	<p>Efficient when the number of rows accessed is less than the number of data blocks in the table.</p> <p>NUSI access is the only way to access a selected multirow set in a NoPI table, column-partitioned table, or column-partitioned join index without a full-table scan unless an appropriate join index exists.</p>	<ul style="list-style-type: none"> <li>One if the NUSI is defined on the same column set as the NUPPI.</li> <li>All otherwise.</li> </ul>	Multiple.	Yes.
Join index	Very efficient.	<ul style="list-style-type: none"> <li>If the join index is accessed by its primary index, the number of AMPs accessed is 1.</li> <li>If the join index is not accessed by its primary index, the number of AMPs accessed is all.</li> </ul>	<ul style="list-style-type: none"> <li>If the join index is accessed by its primary index, multiple rows are accessed</li> <li>If the join index is not accessed by its primary index, all AMPs are accessed using a full-table scan.</li> </ul>	<p>Depends.</p> <ul style="list-style-type: none"> <li>If the join index covers the query, no spool is required.</li> <li>If the join index partially covers the query and its definition specifies either a ROWID, the UPI for the base table, or a USI on that table, spool is required.</li> </ul> <p>You can only specify ROWID in the outermost SELECT of the CREATE JOIN INDEX statement. See “CREATE JOIN INDEX” in <i>SQL Data Definition Language Detailed Topics</i>.</p>

Table Access Summary

Access Method	Relative Efficiency	Number of AMPs Accessed	Number of Rows Returned	Spool Space Required by Query?
Join index with a nonunique PPI	<p>Efficient when selectivity is high and skew is low and if limited to one internal partition by row partition elimination.</p> <p>More efficient if there is row or column partition elimination.</p> <p>Otherwise, performance degrades as a function of the number of internal row partitions that must be accessed.</p>	1	Multiple.	<p>Depends.</p> <ul style="list-style-type: none"> <li>If a query returns a single response, no spool is required.</li> <li>If a query returns more rows than can be handled by a single response, spool is required.</li> </ul>
Hash index	<p>Very efficient.</p> <p>You cannot define a hash index on a NoPI table.</p>	Depends on the primary index.	Multiple.	<p>Depends.</p> <ul style="list-style-type: none"> <li>If the hash index covers the query, no spool is required.</li> <li>If the hash index partially covers the query, spool is required.</li> </ul>
Full-table scan	<p>Efficient because the system touches each row and data block only once.</p> <p>If an nonpartitioned NoPI table or column-partitioned NoPI table has no secondary or join indexes, a full-table scan is the only way its rows can be accessed.</p> <p>This also applies to a column-partitioned NoPI join index that has no secondary indexes.</p>	All.	All.	<p>Yes.</p> <p>Can require spool as large as the table being scanned.</p>

# Index Selection Summary

The following list of factors illustrates how highly multidetermined the selection of Teradata Database indexes is. These factors are examined in the chapters on individual types of indexes that follow this one.

## Nonspecific Factors

- Degree of normalization of the database
- How the Optimizer might use the index
- Is the table designed as a staging table for batch loading operations?
- Is the table designed as a sandbox table to contain rows until an appropriate primary index can be determined for their permanent table?
- Table type indexed
  - Major entity
  - Minor entity
  - Subentity

## Primary Index Partitioning Type

- Nonpartitioned
- Row-partitioned
  - Single-level
  - Multilevel

## Column Data Type Factors

Index columns cannot have any of the following data types.

- XML
- BLOB
- CLOB
- BLOB-based UDT
- CLOB-based UDT
- ARRAY/VARRAY

Geospatial columns only support non-unique secondary indexes.

## Space Utilization Factors

- How much space does the index occupy?
- Type of data protection specified

## Demographic Factors

- Cardinality of the table
- Number of distinct column values
- Maximum rows per value
- Columns most frequently used to access table rows
- Are rows most commonly accessed by values or by a join?
- Degree of skew of column values

## Application Factors

In which application environment are rows most commonly accessed?

- Decision support
- OLTP
- Event queues
- Tactical queries
- Ad hoc queries
- Range queries
- Batch reporting
- Batch maintenance

## Transaction Factors

- How are transactions written?
- How are transactions parcelled?
- What levels and types of locking does a transaction require?
- How long does the transaction hold locks?

## DML Factors

- Number of DELETE operations and when they occur
- Number of INSERT operations and when they occur. This includes INSERT operations made by a MERGE request.
- Number of UPDATE operations and when they occur. This includes UPDATE operations made by a MERGE request.

## Tables and Index Types that Cannot Have a Primary Index

- Nonpartitioned NoPI tables
- Column-partitioned tables (with or without row partitioning)
- Column-partitioned join indexes (with or without row partitioning)
- Global temporary trace tables

# CHAPTER 9 Primary Indexes and NoPI Objects

---

This chapter describes Teradata Database primary indexes: unique versus nonunique and row partitioned versus nonpartitioned.

The process of selecting a primary index is given heavy emphasis. Other topics describe primary index access to rows, the various performance considerations that pertain to primary indexes, special considerations for nonunique primary indexes relating to duplicate row checking, and space utilization parameters for primary indexes.

The main emphasis of the chapter is primary index features dealing with user data tables, but the considerations for selecting the column set to be used for the primary index for a base data table are largely identical to those for selecting the column set to be used as the primary index for a join index. Any join index-specific considerations for primary indexes are documented in [Chapter 11: “Join and Hash Indexes.”](#)

The chapter also covers tables and join indexes that do not have a primary index. These database objects are referred to as NoPI objects.

NoPI objects come in two forms.

- Nonpartitioned NoPI objects
- Column-partitioned NoPI objects (with or without row partitioning)

## Primary Indexes

Teradata Database tables can have 0 or 1 primary index. If you do not define either PRIMARY INDEX or NO PRIMARY INDEX explicitly when you create a table, then a default primary index for a table may be defined or the table may default to have no primary index (see [“Primary Index Defaults” on page 263](#) for details).

Use the CREATE TABLE statement to create primary indexes (see “CREATE TABLE” in *SQL Data Definition Language*).

Note that Teradata does *not* use the term primary index in the same way it is commonly used to describe a clustered index in an indexing system based on B+ trees.

Data accessed using a primary index is always a one-AMP operation because a row and its primary index are stored together in the same structure (see [“Base Table Row Format” on page 740](#)). This is true whether the primary index is unique or nonunique, and whether it is row partitioned or nonpartitioned.

## Purposes of the Primary Index

- To define the distribution of the rows to the AMPs.

With the exception of NoPI tables and column-partitioned tables and join indexes, Teradata Database distributes table rows across the AMPs on the hash of their primary index value (see “[Teradata Database Hashing Algorithm](#)” on page 225). The determination of which hash bucket, and hence which AMP the row is to be stored on, is made solely on the value of the primary index.

The choice of columns for the primary index affects how even this distribution is. An even distribution of rows to the AMPs is usually of critical importance in picking a primary index column set.

- To provide access to rows more efficiently than with a full-table scan.

If the values for all the primary index columns are specified in a DML statement, single-AMP access can be made to the rows using that primary index value.

With a row partitioned primary index, faster access is also possible when all the values of the partitioning columns are specified or if there is a constraint on partitioning columns.

Other retrievals might use a secondary index, a hash or join index, a full-table scan, or a mix of several different index types.

- To provide for efficient joins.

If there is an equijoin constraint on the primary index of a table, it may be possible to do a direct join to the table (that is, rows of the table might not have to be redistributed, spooled, and sorted prior to the join).

- To provide a means for efficient aggregations.

If the GROUP BY key is on the primary index of a table, it is often possible to perform a more efficient aggregation.

## Restrictions on Primary Indexes

- No more than one primary index can be defined on a table.

You can also define base data tables that do not have a primary index (see “[NoPI Tables, Column-Partitioned Tables, and Column-Partitioned Join Indexes](#)” on page 280 and “[Column-Partitioned Tables and Join Indexes](#)” on page 285).

You cannot define a primary index for a global temporary trace table. See “CREATE GLOBAL TEMPORARY TRACE TABLE” in *SQL Data Definition Language* for details.

- No more than 64 columns can be specified in a primary index definition.
- Primary index columns cannot be defined on columns that have XML, BLOB, CLOB, BLOB-based UDT, CLOB-based UDT, XML-based UDT, Period, ARRAY, VARRAY, VARIANT\_TYPE, Geospatial, or JSON data types.
- You cannot define a primary index for a NoPI table.
- You cannot define a primary index for column-partitioned tables and join indexes.
- Primary index columns cannot be defined on row-level security constraint columns.
- You cannot specify multivalue compression for primary index columns.

## Primary Index Dimensions

Primary indexes are defined across two orthogonal dimensions:

- Unique versus nonunique (see “[Unique and Nonunique Primary Indexes](#)” on page 264).
- Row partitioned versus nonpartitioned (see “[Row-partitioned and Nonpartitioned Primary Indexes](#)” on page 266).

A partitioned primary index can be row-partitioned on anywhere from 1 to 62 levels for:

- Base tables that are not queue tables
- Global temporary tables
- Volatile tables
- Uncompressed join indexes

The following types of tables and indexes cannot be partitioned:

- Queue tables
- Row-compressed join indexes
- Hash indexes
- Journal tables

See “[Row-partitioned Primary Indexes](#)” on page 267, “[Single-Level Partitioning](#)” on page 362, and “[Multilevel Partitioning](#)” on page 372.

For hash and join indexes *only*, the primary index can also be value-ordered.

## Primary Index Defaults

You should always explicitly specify a primary index or explicitly specify no primary index for your tables.

At the least, you should specify a PRIMARY KEY constraint on the column set you want to define as the primary index if you do not specify an explicit primary index.

Note that Teradata Database *never* creates a partitioned primary index by default.

There are no defaults for the primary index of a temporary table, which you must always declare explicitly.

### Exceptions to the PrimaryIndexDefault Setting

Use the PrimaryIndexDefault field in DBS Control to determine whether Teradata Database will automatically create a primary index for a table created without the PRIMARY INDEX, NO PRIMARY INDEX, PRIMARY KEY, and UNIQUE modifiers. For more information on the PrimaryIndexDefault field, see *Utilities: Volume 1 (A-K)*.

No matter how the PrimaryIndexDefault parameter is set, if you do not specify either an explicit PRIMARY INDEX clause, an explicit NO PRIMARY INDEX clause, or an explicit PARTITION BY clause, Teradata Database creates the table with a unique primary index only

if it is also defined with either a PRIMARY KEY constraint or a UNIQUE column constraint. In this case, an error occurs for a temporal table since it cannot have unique primary index.

You can specify no more than one primary index when you create a table with these exceptions:

- You cannot define a primary, or any other index, for global temporary trace tables regardless of the setting of PrimaryIndexDefault. See “CREATE GLOBAL TEMPORARY TRACE TABLE” in SQL Data Definition Language Detailed Topics.
- You cannot define a primary index for NoPI tables or column-partitioned tables. For column-partitioned tables, the default primary index specification is NO PRIMARY INDEX regardless of the setting of PrimaryIndexDefault.

See “[NoPI Tables, Column-Partitioned Tables, and Column-Partitioned Join Indexes](#)” on page 280 and “[Column-Partitioned Tables and Join Indexes](#)” on page 285 for information about these types of tables.

## Guidelines for NoPI Tables

If you specify an explicit PRIMARY KEY constraint, UNIQUE constraint, or both for an explicitly defined NoPI table, Teradata Database redefines those constraints as USIs, as you can see if you submit a SHOW TABLE request for such a table.

## Problematic Selection of a Primary Index Column Set

When the selection of a primary index column set is problematic, consider either of the following possible solutions.

- Create it *without* a primary index.  
You can load rows into a NoPI table while you continue to determine an appropriate primary index. Such a table is sometimes referred to as a *sandbox* table.
- Use an identity column as the primary index for the table (see “[Identity Columns](#)” on page 818 and “[Surrogate Keys](#)” on page 91).

# Unique and Nonunique Primary Indexes

You can define a primary index to be either unique or non-unique. The choice of the columns to use for the primary index may depend on how its rows are most frequently accessed. The uniqueness of a primary index depends on the whether or not the data in the columns is unique.

This topic describes the differences between the two types as well as their relative advantages and disadvantages.

## Unique Primary Indexes

Major entities and subentities are typically assigned unique primary indexes. When a subentity has been defined in the logical model, it typically uses the same unique primary

index in the physical model as the major entity it is associated with. This ensures that related rows from the two tables always hash to the same AMP.

Consider the following *employee* table derived from a major entity in the logical database design. The primary key is also defined as the unique primary index for the table.

**employee**

employee_ID	employee_name	home_address
PK		
UPI		
6149	Joe Smith	3 Homestead Way
6171	Wei-hee Chan	44 Fifth Avenue
7049	Yuka Maeda	1000 Chestnut Lane
...	...	...

You should always specify the unique primary index column set as NOT NULL.

## Nonunique Primary Indexes

Minor entities are typically assigned nonunique primary indexes defined on the same column as the major entity with which they are associated.

Consider the following *employee\_phone* table derived from a minor entity in the logical database design.

The *employee\_phone* table is related to the *employee* table by its foreign key, *employee\_id*, which is the primary key of the *employee* table. Note that the primary index for this table is defined only on the *employee\_id* attribute, which is only half the primary key. This makes the primary index nonunique by definition.

The advantage gained by this is that both *employee* and *employee\_phone* rows have the same primary index and hash to the same AMP. This means that joins on these tables, which are a frequent occurrence, do not require redistribution of table rows across the BYNET.

You could have defined a NUPI on *phone\_number* because there should not be many duplicate entries for that field (husband-wife-child groupings and roommates being likely examples), but you would lose the advantage of hashing related rows to the same AMPS.

employee_phone		
employee_ID	phone_number	phone_remarks
PK		
FK		
NUPI		
6149	555-1234	Home land line. Not after 20:00.
6149	555-9315	Cell phone.
6149	555-8357	Pager.
6171	555-5678	Any time.
7049	555-9012	Never call home number.
...	...	...

Employee 6149, Joe Smith, has three different telephone numbers (rows shaded in red) where he can be reached away from the office.

## Polyinstantiation

You can create a USI for a row-level security-protected table as a composite of a row-level security constraint column and the columns of a NUPI for the table. This property can be used to implement polyinstantiation.

Polyinstantiation is a property that enables a relation to contain multiple rows with the same primary key value, where the multiple instances are distinguished by their security levels, where a security level is defined by a row-level security constraint column.

For this property not to violate the relational model, the security level instances would need to be defined as components of a composite primary key.

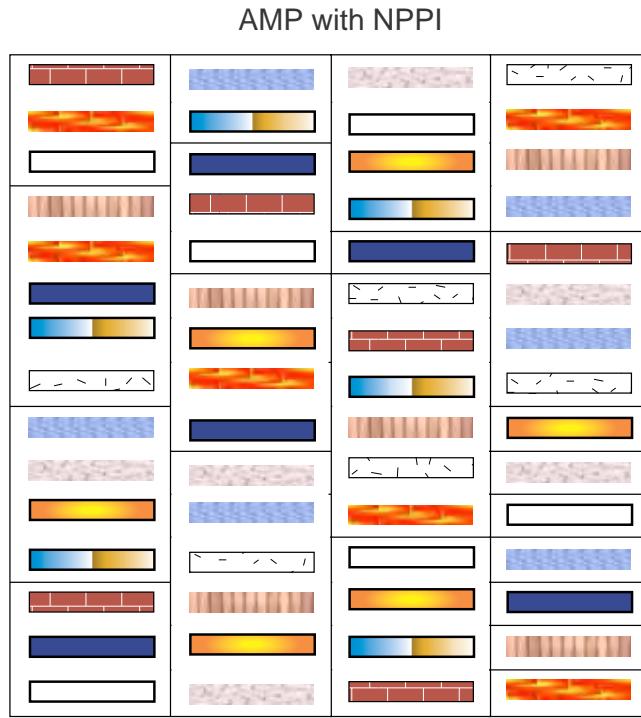
## Row-partitioned and Nonpartitioned Primary Indexes

A primary index can be either row-partitioned or nonpartitioned. Primary indexes for hash and join indexes can also either be hash- or value-ordered, while primary indexes for all other table types are hash-ordered *only*.

The decision to define which of the two choices for a table depends on how its rows are most frequently accessed (see “[Row-Partitioned and Nonpartitioned Primary Index Access for Typical Operations](#)” on page 403 and “[Single-Level Partitioning Case Studies](#)” on page 370). This topic describes the differences between the two types as well as their relative advantages and disadvantages.

## Nonpartitioned Primary Indexes

The nonpartitioned primary index is the standard Teradata Database primary index. When a table is created with a nonpartitioned primary index, its rows are hashed to the appropriate AMPs and stored there in row hash order, as illustrated by the following graphic.



GG02A014

Rows within the same box have the same row hash value. Rows with the same color pattern have a common value for a column that is not the primary index (but might be one of the columns in a multi-column primary index).

## Row-partitioned Primary Indexes

A row-partitioned primary index is defined to be either single-level or multilevel depending on how many partitioning expressions are defined for the partitioning in the CREATE TABLE or ALTER TABLE statements.

A row-partitioned primary index is an extension to the nonpartitioned primary index. You can define both single-level and multilevel partitioning for global temporary and volatile tables, for base tables (but not queue tables or hash indexes), and for noncompressed join indexes.

**Note:** You can also row-partition tables and join indexes that do not have a primary index if the object is column-partitioned. See “[Column-Partitioned Tables and Join Indexes](#)” on [page 285](#) for more information about column partitioning.

Optimal partitioning expressions are typically coded using CASE\_N or RANGE\_N expressions based on exact numeric, character or DateTime columns. DateTime expressions

can include the BEGIN and END bound functions and the DATE, CURRENT\_DATE, and CURRENT\_TIMESTAMP functions.

The test value you specify with a RANGE\_N function must result in a BYTEINT, SMALLINT, INTEGER, BIGINT, DATE, TIMESTAMP(n), TIMESTAMP(n) WITH TIME ZONE, CHARACTER, VARCHAR, GRAPHIC, or VARCHAR(n) CHARACTER SET GRAPHIC data type.

Support for the BEGIN and END bound functions also includes support for the IS [NOT] UNTIL\_CHANGED and IS [NOT] UNTIL\_CLOSED functions.

You can specify any valid Date or Time element for the IS [NOT] UNTIL\_CHANGED and IS [NOT] UNTIL\_CLOSED functions in partitioning expressions based on the CASE\_N function.

You cannot specify multi-value compression for the partitioning columns of a partitioning expression.

Row-partitioning optimizes range queries while also providing efficient primary index join strategies. Analyze your range query optimization needs carefully because there are performance tradeoffs between specific range access improvements and possible decrements for primary index accesses and joins and aggregations on the primary index that occur as a function of the number of populated row partitions.

When a primary-indexed table or join index is created with row-partitioning, its rows are hashed to the appropriate AMPs based on the primary index, and then assigned to their computed internal partition number based on the value of the partitioning expressions defined by the user when the table was created or altered. The evaluation of a partitioning expression determines the partition number for a row. Once a row has been dispatched to its home AMP, Teradata Database converts the partition numbers from each partitioning level to an internal partition number that is stored in the rowID.

The value of a PARTITION or PARTITION#*Ln* is determined by reading the partition number field in the rowID and then converting that back to the appropriate combined partition number or the specified partitioning level whenever you submit a query that specifies PARTITION or PARTITION#*Ln* in its select list.

In other words, the value you see as a PARTITION number when you select the PARTITION (or PARTITION#*Ln*) is actually not a column that is stored with the other columns of a table row, but is a virtual column whose value is determined by decoding the bit pattern stored in the internal partition number of the rowID.

The partitioning maxima using different types of partitioning expressions are listed in the following table. These maxima apply to both row-partitioned tables and join indexes and to column-partitioned tables and join indexes (which may also have row partitioning).

Partitioning Parameter	Maximum Value
Combined partitions for a table with 2-byte partitioning.	65,535
Combined partitions for a table with 8-byte partitioning.	9,223,372,036,854,775,807

Partitioning Parameter	Maximum Value
Valid range for the number of partitioning levels for a table or join index with 2-byte partitioning.	1 - 15
Valid range for the number of partitioning levels for a table or join index with 8-byte partitioning.	1 - 62
RANGE_N  Number of ranges plus NO RANGE [OR UNKNOWN] and UNKNOWN partitions, with 2-byte partitioning.	65,535
RANGE_N  Number of ranges plus NO RANGE [OR UNKNOWN] and UNKNOWN partitions (whether or not specified), with 8-byte partitioning.	9,223,372,036,854,775,807
CASE_N  Number of conditions.	Bound by the request text size and other limits.

When you have a partitioned table, the bits stored in the internal partition number of the rowID represent the combined partition number of the row, which is determined by combining the partitioning expressions of a PARTITION BY clause into a single expression. The expression that combines the partitioning expressions of multilevel partitioning into a single combined partition number is called a combined partitioning expression.

The terms internal partition number and combined partition number are defined as follows.

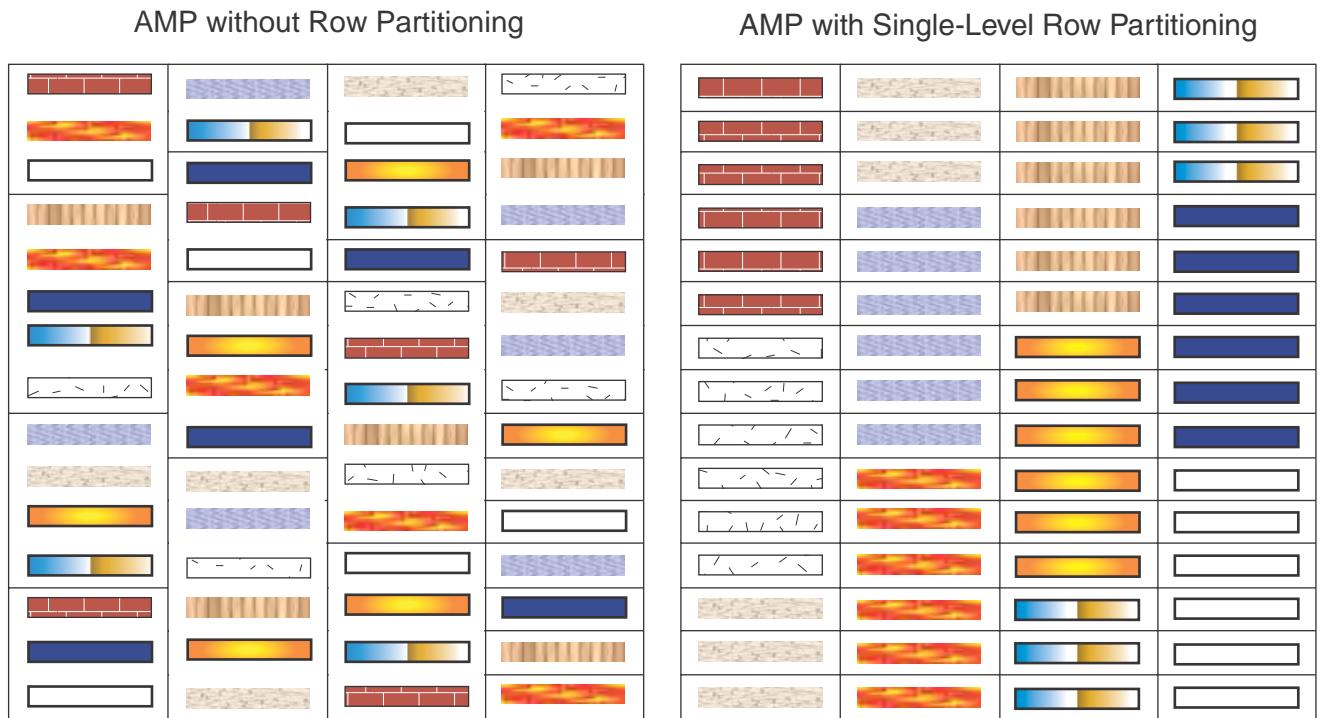
Term	Definition
Internal partition number	A value Teradata Database calculates from the combined partition number.  The internal partition number is used to number partitions internally and is stored in the rowID.  The internal and combined partition numbers can be identical if no modification is needed.  For single-level partitioning only, partitions for the NO RANGE [OR UNKNOWN], NO CASE [OR UNKNOWN], and UNKNOWN options are placed at fixed internal partitions, followed by partitions for range and conditions following.  Modification is required to retain existing internal partition numbers after an ALTER TABLE request that drops or adds ranges or partitions. For a table without partitioning, the internal partition number is always zero.

Term	Definition
Combined partition number	<p>The value computed for the partitioning expressions for a row in a partitioned table.</p> <p>For a table without partitioning, the combined partition number is always 0.</p> <p>Combining the results for a table with single-level partitioning, the combined partition number is the same as the value of the single partitioning expression.</p> <p>When you specify the system-determined columns PARTITION or PARTITION#Ln in a request, Teradata Database converts the internal partition number stored in the rowID into the appropriate combined partition number or the partition number of the specified partitioning level and returns that value to the requestor.</p>

Teradata Database groups partitioned rows into partition groups on an AMP first by their internal partition number, then by row hash value within each internal partition (if there is a primary index, otherwise by the assigned hash bucket), and then by uniqueness value.

The following graphic also illustrate the difference between a nonpartitioned table and a partitioned table. Both tables contain the same data for the year 2006. Each colored rectangle represents a row. Rows with the same color are in the same month. Rows in the same box have the same row hash based on *order\_number*, which is the primary index column.

For a nonpartitioned primary-indexed table, the rows are ordered only by their hash value. For a partitioning that is partitioned by month, the rows are first ordered by month based on *order\_date* and then ordered by their hash values for the primary index column. Note that within an internal partition there are fewer rows per hash value.



1094-002A

If a SELECT query specifies values for all the primary index columns, the AMP that contains those rows can be determined, and only one AMP needs to be accessed. If the query conditions are not specified on the partitioning columns, then each internal partition can be probed to find the rows based on the hash value, assuming there is no usable alternative index. If conditions are also specified on the partitioning columns, then row partition elimination might further reduce the number of partitions to be probed on that AMP (see *SQL Request and Transaction Processing* for information about row partition elimination).

If a SELECT query does *not* specify the values for all the primary index columns, an all-AMP, full-table scan is required for a table with no partitioning when there is no usable alternative index. However, with partitioning, if conditions are specified on the partitioning columns, row partition elimination might reduce what would otherwise be an all-AMP, full-table scan to an all-AMP scan of only the internal partitions that are not eliminated. The extent of row partition elimination depends on the partitioning expressions, the conditions specified in the query, and the ability of the Optimizer to recognize such opportunities.

Suppose a query only requests orders dated August, 2010. For a nonpartitioned table, the whole table is read to determine which rows are from August because the August rows are scattered throughout the table. For the partitioned table, on the other hand, row partition elimination can be used to exclude all the internal partitions that do not contain activity for August. Because all the August rows can be grouped together in the partitioned table, it becomes possible to position directly to the first row for August, then read sequentially until a September row is found, at which point the system stops reading rows. This query requested about  $\frac{1}{12}$  of the rows, so only about  $\frac{1}{12}$  of the partitioned table must be read. If you partition

by day and change the query to make the ending date August 2nd, an even smaller subset of the table is all that must be read.

When rows for current day activity are inserted later tonight into the nonpartitioned table, they are scattered throughout the entire table, so the average hits per block value is very low. For the partitioned table on the other hand, the inserts are clustered in a smaller area. This provides a better locality of reference. If the partitioning is by week or by day, then the inserts are clustered even more tightly, and the hits per block value is very high.

If you want to delete the oldest month of data, the task is a full-table scan for the nonpartitioned table, but for the partitioned table the rows are cluster together, so it is possible to delete them all with a high number of average hits per block. Transient journaling is not done for each of the rows, further enhancing performance when all the rows of an internal partition are deleted as the last action of a transaction.

You can see that if the table is partitioned on order date, there is no need for a NUSI on order date, and if you had one for the nonpartitioned table, you could drop it when you converted the primary index for the table to a partitioned primary index.

You should consider defining a table with partitioning to support either of the following workload characteristics:

- A majority of the requests in the workload specify range predicates on some column, particularly a date column, of the candidate partitioned table.
- A majority of the queries in the workload specify an equality predicate on some column of the candidate partitioned table, and that column is either:
  - Not the only column in the primary index column set
  - or
  - Not a primary index column.

In addition to these two workload characteristics, one of the following sets of characteristics of the primary index should also be considered. One of these three cases is always true.

- The primary index is used for the following applications.
  - Primarily or exclusively to distribute rows evenly across the AMPs.
  - Rarely, if ever, to access rows or to join tables.
  - To access rows using a condition specified on a column that is suitable for partitioning the table.

or

- The primary index, defined *with* the entire set of partitioning columns, is used for the following applications.
  - To distribute rows evenly across the AMPs.
  - To access rows directly or as a table join condition.

or

- The primary index, defined *without* the entire set of partitioning columns, is used for the following applications.

- To distribute rows evenly across the AMPs.
- To access rows directly or as a table join condition.

In this case, you must pay particular attention to weighing and optimizing the performance tradeoffs the situation makes possible.

The following factors characterizes the general performance of partitioned tables with respect to nonpartitioned tables:

- Partitioned access by means of an equality constraint on *all* primary index and partitioning columns is as efficient as the same access made by means of a nonpartitioned primary index.
- Partitioned access on an equality constraint on the primary index columns and an equality or other constraint on the partitioning columns that limits access to a single partition, produces performance as efficient as that made with a nonpartitioned primary index.
- Partitioned access with a constraint on the partitioning columns can approach that of nonpartitioned primary index access when all the primary index columns are specified with equality constraints on an expression that does not reference any columns, depending on the extent of row partition elimination (an indirect measure of the number of internal partitions that must be probed before the desired row is found) required.
- Partitioned access via an equality condition on the primary index that neither includes all of the partitioning columns nor makes a constraint on the partitioning columns might not be as efficient as access by means of a nonpartitioned primary index depending on the number of internal partitions that contain data, but is generally more efficient than a full-table scan.
- Partitioned access when not all of the primary index columns are specified with equality constraints, but there are conditions specified on the partitioning columns that limit the number of internal partitions that must be scanned is more efficient than access by a nonpartitioned primary index, which requires a full-table scan.

Specifically, these are the major performance characteristics of row-partitioned tables with respect to nonpartitioned tables:

- Inserting a large number of rows into a partitioned table can be much faster than the same insert operation into an identical table that it is defined without partitioning.

To optimize load performance, the first partitioning expression of the table preferably should match the row arrival pattern of the load utility, which is most often based on transaction date. In this case, for example, the first partitioning expression should be based on a DATE or TIMESTAMP column. It is generally, but not always, true that if you do not match the table partitioning with the arrival pattern of the load utility, you will not achieve much of an optimization by partitioning the table.

The following design considerations are also important for this partitioning performance characteristic:

- Performance improvements decrease as a function of the number of secondary indexes defined on the target table in a manner similar to the behavior of a nonpartitioned table when increasingly more secondary indexes are defined on it.

- Performance improvements are greater as a function of how finely the granularity is defined for the partitions. For example, partitions defined on weekly intervals are typically better than those defined on monthly intervals.
- Accessing rows using conditions on partitioning columns can be faster than the same access to the same rows when the table is defined with a nonpartitioned primary index.  
The performance of SELECT, UPDATE, and DELETE operations is improved when queries specify conditions on the partitioning columns of the table because such predicates cause a higher average number of hits per data block.

The following design considerations are also important for this partitioned performance characteristic:

- The degree of improvement in the performance of select, update, and delete operations is proportional to the extent of row partition elimination that can be realized (see *SQL Request and Transaction Processing* for details about row partition elimination).  
In other words, the more internal partitions that can be excluded by conditions specified on the partitioning columns for a partitioned table, the better.
- Performance improvements are generally greater as a function of how finely the granularity is defined for the internal partitions.
- Depending on the number of internal partitions that contain data, accessing rows by means of an equality constraint on their primary index can be slower for a partitioned table than for the equivalent nonpartitioned table when you neither include all of the partitioning columns in the primary index definition nor specify a constraint on those columns in the query.

The following design considerations are important for this partitioned performance characteristic:

- The more coarse the granularity of the partitioning you define, or the more internal partitions eliminated, the less performance degradation you are likely to experience.
- Defining an appropriate secondary index on the partitioned table can sometimes minimize the performance degradation of primary index access.
- Joins can be different for partitioned and nonpartitioned tables that are otherwise equivalent, but the effect of the different join strategies that arise cannot be predicted easily in many cases.

The join plan the Optimizer pursues depends on the picture it has of the data demographics based on collected statistics, dynamic-AMP samples, and derived statistics. The usual recommendation applies here: check EXPLAIN reports to determine the best way to design your indexes to achieve the optimal join geography.

The following design considerations are important for this partitioned performance characteristic:

- Primary index-to-primary index joins are more likely to generate different join plans than other partitioned-nonpartitioned join comparisons.  
To minimize the potential for performance issues in making primary index-to-primary index joins, consider the following guidelines:
  - Partition the two tables identically if possible.

- A coarser granularity of partitions is likely to be superior to a finer partition granularity.
- Examine your EXPLAIN reports to determine which join methods the Optimizer is selecting to join the tables.  
 Rowkey-based merge joins are generally better than joins based on another family of join methods.
- Efficient row partition elimination can often convert joins that would otherwise be poor performers into good performers.
- The most likely candidate for poor join performance is found when you are joining a partitioned table with a nonpartitioned table, the partitioned table partitioning column set is not defined in the nonpartitioned table, there are no predicates on the partitioned table partitioning column set, and there are many internal partitions defined for the partitioned table.
- The need for secondary indexes is often different for partitioned and nonpartitioned tables that are otherwise equivalent.

Several opposing scenarios present themselves for resolving the issues that might arise from the existence or absence of secondary indexes:

You probably should ...	IF ...
drop an existing secondary index on the partitioning column set of a partitioning expression	<p>it is not required to enforce the uniqueness of the partitioning column set of a partitioning expression. In practice, it is unlikely that uniqueness would be required in this case.</p> <p>This has the following benefits:</p> <ul style="list-style-type: none"> <li>• The partitioning itself might provide performance benefits similar to those realized by the secondary index.</li> <li>• General system performance is enhanced because there is no need to maintain the secondary index subtable.</li> <li>• Not having a secondary index subtable realizes considerable disk savings.</li> </ul>
add a new secondary index on the primary index column set	<p>one or more columns of the partitioning column set is not also a member of the primary index column set and the primary index values are unique.</p> <p>In this situation, you must define a USI on the primary index column set to enforce its uniqueness because you cannot define a partitioned primary index to be unique unless its definition contains all of the partitioning columns.</p> <p>In some cases, the addition of the USI results in worse performance.</p>

When you are doing an analysis of whether a table should have partitioning or not, always weigh the costs of a given strategy set against its benefits carefully.

You must consider all of the following factors when making your analysis of a partitioning expression.

- Would the proposed workloads against the table be better supported by a partitioning expression based on a CASE\_N, RANGE\_N, or some other expression?
- Should the partitioning expression specify a NO CASE, NO CASE OR UNKNOWN, NO RANGE, NO RANGE OR UNKNOWN, or UNKNOWN option?

If the test value in a RANGE\_N expression can never be null, there is no need for an UNKNOWN or NO RANGE OR UNKNOWN partition.

If a condition in a CASE\_N expression can never be unknown, there is no need for an UNKNOWN or NO CASE OR UNKNOWN partition.

- Should the table be partitioned on only one level or on multiple levels?
- If the partitioning expression specifies CURRENT\_DATE functions, CURRENT\_TIMESTAMP functions, or both, how should the expression be configured to minimize problems deriving from reconciling to a new current date or current timestamp value?
- The query workloads that will be accessing the partitioned table.

This factor must be examined at both the specific, or particular, level and at the general, overall level.

Among the factors to be considered are the following:

- Performance
  - Does a nonpartitioned table perform better than a partitioned table for the given workload and for particularly critical queries?
  - Is one partitioning strategy more high-performing than others?
  - Do other indexes such as USIs, NUSIs, join indexes, or hash indexes improve performance?
  - Does a partitioning expression cause significant row partition elimination for queries to occur or not?
- Access methods and predicate conditions
  - Is access to the table typically made by primary index, secondary index, or some other access method?
  - Do typical queries specify an equality condition on the primary index and include the complete partitioning column set?
  - Do typical queries specify a *nonequality* condition on the primary index or the partitioning columns?
- Join strategies
  - Do typical queries specify an equality condition on the primary index column set (and partitioning column set if they are not identical)?
  - Do typical queries specify an equality condition on the primary index column set but not on the partitioning column set?
  - Do typical query conditions support row partition elimination?
- Data maintenance
  - What are the relative effects of a partitioned table versus a nonpartitioned table with respect to the maintenance workload for the table?

- If you must define a USI on the primary index to make it unique, how much additional maintenance is required to update the USI subtable?
- Frequency and ease of altering partitions
  - Is a process in place to ensure that ranges are added and dropped as necessary?
  - Does the partitioning expression permit you to add and drop ranges?
  - If the number of rows moved by dropping and adding ranges causes large numbers of rows to be moved, do you have a process in place to instead create a new table with the desired partitioning in place, then INSERT ... SELECT or MERGE the source table rows into the newly created target table with error logging?
  - If you want to delete rows when their partition is dropped, have you specified NO RANGE, NO RANGE OR UNKNOWN, or UNKNOWN partitions?
- Backup and restore strategies.

See [Teradata Archive/Recovery Utility Reference](#) for details of how partitioning might affect your archive and restore strategies.

See [“Single-Level Partitioning Case Studies” on page 370](#) for a set of design scenarios that evaluate the performance effects of several different partitioned primary indexes.

See [“Single-AMP Queries and Partitioned Tables” on page 905](#) and [“All-AMP Tactical Queries and Partitioned Tables” on page 909](#) for specific design issues related to partitioned table support for tactical queries.

## Partitioning Expression Data Type Considerations

**Note:** In addition to the following data type rules and restrictions, be aware that you cannot specify a row-level security constraint column in a partitioning expression.

The partitioning expressions you can define for a partitioned table have certain restrictions regarding the data types you can specify within them and with respect to the data type of the result of the function.

The following table summarizes these restrictions.

Data Type	PARTITION BY		
	RANGE_N	CASE_N	Expression
• ARRAY • VARRAY	N	N	N
BIGINT	Y	X	I
BLOB	N	N	N
BYTE	X	X	X
BYTEINT	Y	X	I
CHARACTER	Y	X	I

Data Type	PARTITION BY		
	RANGE_N	CASE_N	Expression
CLOB	N	N	N
DATE	Y	X	I
<ul style="list-style-type: none"> <li>• DECIMAL</li> <li>• NUMERIC</li> <li>• NUMBER (exact form)</li> </ul>	X	X	I
<ul style="list-style-type: none"> <li>• DOUBLE PRECISION</li> <li>• FLOAT</li> <li>• REAL</li> <li>• NUMBER (approximate form)</li> </ul>	X	X	I
GRAPHIC	N	X	N
INTEGER	Y	X	Y
INTERVAL YEAR	X	X	I
INTERVAL YEAR TO MONTH	X	X	X
INTERVAL MONTH	X	X	I
INTERVAL DAY	X	X	I
INTERVAL DAY TO HOUR	X	X	X
INTERVAL DAY TO SECOND	X	X	X
INTERVAL SECOND	X	X	X
LONG VARCHAR	Y	X	I
LONG VARCHAR CHARACTER SET GRAPHIC	N	N	N
PERIOD	N	X	N
The BEGIN and END bound functions are valid in a partitioning expression when they are defined on a valid PERIOD data type column and the result can be cast implicitly to a numeric data type.			
SMALLINT	Y	X	I
TIME	X	X	X
TIME WITH TIME ZONE	X	X	X
TIMESTAMP	Y	X	X
TIMESTAMP WITH TIME ZONE	Y	X	X
UDT	N	N	N

Data Type	PARTITION BY		
	RANGE_N	CASE_N	Expression
VARBYTE	X	X	X
VARCHAR	Y	X	I
VARCHAR(n) CHARACTER SET GRAPHIC	N	N	N
• XML • XMLTYPE	N	N	N

The following table explains the abbreviations used in the previous table.

Symbol	Definition
I	Valid for a partitioning expression. If the type is also the data type of the result, then it must be such that it can be cast to a valid INTEGER or BIGINT value.
N	Not valid for a partitioning expression. If the partitioning expression is defined using a CASE_N function, then this type is <i>not</i> valid for the CASE_N condition.
X	Valid for a partitioning expression, but cannot be the data type of the result of the expression. If the partitioning expression is defined using a CASE_N function, then this type <i>is</i> valid for the CASE_N condition.
Y	Valid for a partitioning expression and valid as the data type of the result of the partitioning expression.

## Partitioning a Table or Join Index That Does Not Have a Primary Index

Besides partitioning tables or join indexes on their rows, you can also partition them on their columns. Column partitioning enables Teradata Columnar. Column partitioning is only allowed when a table or join index has no primary index (see “[NoPI Tables, Column-Partitioned Tables, and Column-Partitioned Join Indexes](#)” on page 280).

Column partitioning is a physical database design choice that is not suitable for all workloads. For example, column partitioning is usually not suitable for workloads that often select both a significant number of rows and project many columns from a table. However, column partitioning might be suitable if a request selects a significant number of rows, but projects only a few columns, or conversely, if a request projects many columns, but only selects a small number of rows.

Column partitioning is especially suitable for the case where both a small number of rows are selected and only a few columns are projected. See “[Column-Partitioned Tables and Join Indexes](#)” on page 285 for more information about how table and join indexes can be

partitioned on their columns and how column-partitioned tables and join indexes can be applied optimally.

**Note:** A column-partitioned table or join index can also have 1 or more row partitioning levels.

## Using Partition-Level BARestore

Backup, Archive, Restore (BAR) for table partitions allows backup and restore of selected table partitions for tables with a partitioned primary index. Performance improves when a BAR operation involves partitions of a table rather than the entire table.

# NoPI Tables, Column-Partitioned Tables, and Column-Partitioned Join Indexes

A NoPI object is a table or join index that does not have a primary index and always has a table kind of MULTISET.

The basic types of NoPI objects are:

- Nonpartitioned NoPI tables
- Column-partitioned tables and join indexes (these may also have row partitioning)

The chief purpose of NoPI tables is as staging tables. FastLoad can efficiently load data into empty nonpartitioned NoPI staging tables because NoPI tables do not have the overhead of row distribution among the AMPs and sorting the rows on the AMPs by rowhash.

Nonpartitioned NoPI tables are also critical to support Extended MultiLoad Protocol (MLOADX). A nonpartitioned NoPI staging table is used for each MLOADX job.

The optimal method of loading rows into a column-partitioned table from an external client is to use FastLoad to insert the rows into a staging table, then use an INSERT ... SELECT request to load the rows from the source staging table into the column-partitioned target table.

You can also use Teradata Parallel Data Pump array INSERT operations to load rows into a column-partitioned table.

Global temporary trace tables are, strictly speaking, also a type of NoPI table because they do not have a primary index, though they are generally not treated as NoPI tables.

Because there is no primary index for the rows of a NoPI table, its rows are not hashed to an AMP based on their primary index value. Instead, Teradata Database either hashes on the Query ID for a row (see “[Row Allocation for Teradata Parallel Data Pump](#)” on page 237), or it uses a different algorithm to assign the row to its home AMP (see “[Row Allocation for FastLoad Operations Into Nonpartitioned NoPI Tables](#)” on page 238).

Teradata Database then generates a RowID for each row in a NoPI table by using a hash bucket that an AMP owns (see “[Rowhash Value and RowID for NoPI Tables](#)” on page 198 and “[Row](#)

[Allocation for Teradata Parallel Data Pump](#)” on page 237). This strategy makes fallback and index maintenance very similar to their maintenance on a PI table.

Global temporary tables, volatile tables, and temporal tables can be defined as NoPI tables. Column-partitioned tables and column-partitioned join indexes must be defined without a primary index. See “[Column-Partitioned Tables and Join Indexes](#)” on page 285 for details about column partitioning and NoPI tables and join indexes.

## INSERT... SELECT into CP and NoPI Tables

When the target table of an INSERT ... SELECT request is a column-partitioned or NoPI table, Teradata Database inserts the data from the source table locally into the target table, whether it comes directly from the source table or from an intermediate spool file. This is very efficient because it avoids a redistribution and sort. However, if the source table or the resulting spool is skewed, the target column-partitioned or NoPI table can also be skewed. In this case, you can specify a HASH BY clause to redistribute the data from the source before Teradata Database executes the local copy operation.

Consider using hash expressions that provide good distribution and, if appropriate, improve the effectiveness of autocompression for the insertion of rows into the target table.

Alternatively, you can specify HASH BY RANDOM to achieve good distribution if there is not a clear choice for the expressions to hash on.

When inserting into a column-partitioned table, also consider specifying a LOCAL ORDER BY clause with the INSERT ... SELECT request to improve the effectiveness of autocompression.

## Uses for Nonpartitioned and Column-Partitioned Tables

Nonpartitioned NoPI tables are particularly useful as staging tables for bulk data loads. When a table has no primary index, its rows can be dispatched to any given AMP arbitrarily, so the system can load data into a staging table faster and more efficiently using FastLoad or Teradata Parallel Data Pump array INSERT operations. You can only use FastLoad to load rows into a NoPI table when it is unpopulated, not partitioned, and there are no USIs.

You must use Teradata Parallel Data Pump array INSERT operations to load rows into NoPI tables that are already populated. If a NoPI table is defined with a USI, Teradata Database checks for an already existing row with the same value for the USI column (to prevent duplicate rows) when you use Teradata Parallel Data Pump array INSERT operations to insert rows into it.

By storing bulk loaded rows on any arbitrary AMP, the performance impact for both CPU and I/O is reduced significantly. After having been received by Teradata Database all of the rows can be appended to a nonpartitioned or column-partitioned NoPI table without needing to be redistributed to their hash-owning AMPS.

Because there is no requirement for such tables to maintain their rows in any particular order, the system need not sort them. The performance advantage realized from NoPI tables is achieved optimally for applications that load data into a staging table, which must first

undergo a conversion to some other form, and then be *redistributed* before they are stored in a secondary staging table or the target table.

Using a nonpartitioned NoPI table as a staging table for such applications avoids the row redistribution and sorting required for primary-indexed staging tables. Another advantage of nonpartitioned NoPI tables is that you can quickly load data into them and be finished with the acquisition phase of the utility operation, which frees client resources for other applications.

Both NoPI and column-partitioned tables are also useful as so-called sandbox tables when an appropriate primary index has not yet been defined for the primary-indexed table they will eventually populate. This use of a NoPI table enables you to experiment with several different primary index possibilities before deciding on the most optimal choice for your particular application workloads.

## Rules and Limitations for NoPI and Column-Partitioned Tables

The rules and limitations for NoPI tables are the same as those for primary-indexed tables with the following exceptions:

- You cannot create a nonpartitioned NoPI join index.  
You can create a column-partitioned join index (with or without row partitioning).
- You cannot create a NoPI or column-partitioned:
  - Queue table
  - Error table
  - SET table

The default table type for NoPI and column-partitioned tables in both Teradata and ANSI/ISO session modes is always MULTISET.
- Global temporary trace tables  
Global temporary trace tables do not have a primary index by default; however, you are not allowed to specify the NO PRIMARY INDEX option when you create a global temporary table.
- If none of the clauses PRIMARY INDEX (*column\_list*), NO PRIMARY INDEX, or PARTITION BY are specified explicitly in a CREATE TABLE or CREATE JOIN INDEX request, whether the table or join index is created with or without a primary index generally depends on whether a PRIMARY KEY or UNIQUE constraint is specified for any of the columns and on the setting of the DBS Control parameter PrimaryIndexDefault (see “[Primary Index Defaults](#)” on page 263 and *Utilities: Volume 1 (A-K)* for details and exceptions).
- Neither NoPI tables nor column-partitioned tables can specify a permanent journal.
- Nonpartitioned NoPI tables cannot specify an identity column.  
Column-partitioned tables can specify an identity column.
- Hash indexes cannot be defined on NoPI or column-partitioned tables.
- SQL UPDATE (Upsert Form) requests cannot update either a NoPI or a column-partitioned target table.

- SQL MERGE requests cannot update or insert into either a NoPI or a column-partitioned target table.  
SQL MERGE requests can update or insert into a primary-indexed target table from a NoPI or column-partitioned source table.
- You cannot load rows into either a nonpartitioned NoPI or a column-partitioned table using the MultiLoad utility.

You can define all of the following commonly used features for both NoPI and column-partitioned tables:

- Fallback
- Secondary indexes
- Join indexes
- PRIMARY KEY and UNIQUE column constraints
- CHECK constraints
- FOREIGN KEY constraints
- Triggers
- XML, BLOB, and CLOB columns.

**Note:** Because there is normally only one row hash value per AMP for NoPI tables, there is also a limit of approximately 256M rows per AMP for NoPI tables that contain columns typed as XML, BLOB, or CLOB.

You can define any of the following table types as NoPI tables:

- Nonpartitioned base tables
- Column-partitioned base tables (with or without row partitioning)
- Column-partitioned, single-table, non-aggregate, noncompressed join indexes (with or without row partitioning)
- Nonpartitioned global temporary tables
- Nonpartitioned volatile tables

## Manipulating Nonpartitioned NoPI Table Rows

After a nonpartitioned NoPI staging table has been populated with rows, you should execute one of the following types of DML request to move the nonpartitioned NoPI staging table source rows to a primary-indexed or column-partitioned target table.

- INSERT ... SELECT
- MERGE (for a primary-indexed target table only)
- UPDATE ... FROM (for a primary-indexed target table only)

For these cases with a primary-indexed target table, Teradata Database reads the rows in the nonpartitioned NoPI source table and then redistributes them in the same way it redistributes them from a primary-indexed table to their hash-owning AMPS based on the primary index of the target table (see “[Row Allocation for Teradata Parallel Data Pump](#)” on page 237 and “[Row](#)

[“Allocation for Primary-Indexed Tables” on page 235](#)). For a column-partitioned target table, the rows are locally copied, unless the INSERT... SELECT specifies a HASH BY clause.

You can use the following DML statements to manipulate or retrieve nonpartitioned NoPI table rows prior to moving them to their target table.

- DELETE
- INSERT
- SELECT
- UPDATE

See *SQL Data Manipulation Language* for details of the limitations of these statements when they are used with NoPI tables.

You *cannot* use UPDATE (Upsert Form) requests to change data in either a NoPI or a column-partitioned table.

Without a primary index, there can be no single-AMP primary index access to table rows. However, you can create both unique and nonunique secondary indexes (see [Chapter 10: “Secondary Indexes”](#)) and join indexes (see [Chapter 11: “Join and Hash Indexes”](#)) on NoPI tables, and with appropriate secondary indexes defined, you can specify those indexes in your query conditions in such a way as to avoid full-table scans. You cannot specify a join index in a query condition, but if an appropriate join index exists, the Optimizer can use it to create the access or join plan for a request.

You cannot use the MultiLoad utility to load rows into either NoPI or column-partitioned tables.

You can use FastLoad to load rows into an empty nonpartitioned NoPI table, but not to load rows into a column-partitioned table or NoPI table.

Keep in mind that secondary and join indexes can slow the loading of rows into a NoPI table using Teradata Parallel Data Load array INSERTs. FastLoad is more efficient than Teradata Parallel Data Pump for loading rows into an empty nonpartitioned NoPI table because Teradata Database processes each Teradata Parallel Data Pump request as a separate transaction.

If you use FastLoad to load rows into a nonpartitioned NoPI table, you cannot create any secondary or join indexes, CHECK constraints, referential integrity constraints, or triggers on the table until after the load operation has completed because FastLoad cannot load rows into a table defined with any of those features. You *can* use FastLoad to load rows into an empty nonpartitioned NoPI tables that is defined with row-level security constraints, however.

## Related Topics

Topic	Reference
Column-partitioned tables and join indexes	<a href="#">“Column-Partitioned Tables and Join Indexes” on page 285</a>

Topic	Reference
Using both NoPI and column-partitioned tables	"CREATE TABLE" in <i>SQL Data Definition Language Detailed Topics</i>
NoPI tables	Teradata Database Orange Book 541-0007565B02 <i>No Primary Index (NoPI) Table User's Guide</i> by Tam Ly
Column-partitioned tables and join indexes	Teradata Database Orange Book 541-0009036A02 <i>Teradata Columnar</i> by Paul Sinclair and Carrie Ballinger

## Column-Partitioned Tables and Join Indexes

This topic describes the structure of column-partitioned tables and column-partitioned join indexes, comparing them with nonpartitioned NoPI tables.

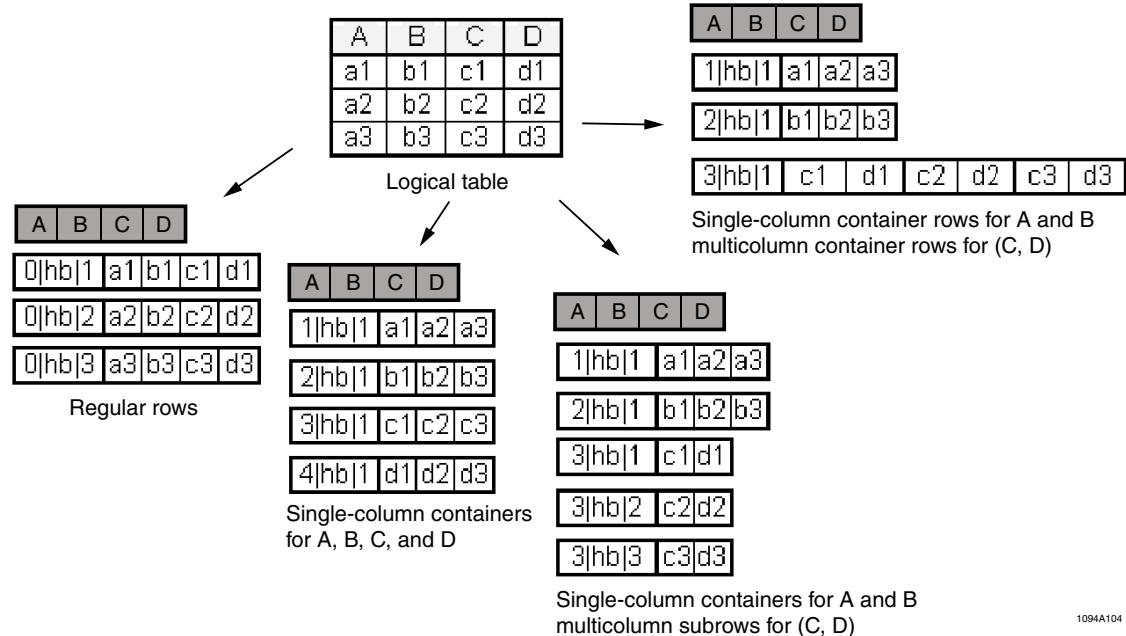
When autocompression methods are described, they are described only to indicate how their use affects the content and size of the row header. Because Teradata Database determines whether to use autocompression and which autocompression methods to use for a container, they are not documented any further in the Teradata Database documentation library.

### Options for Storing a NoPI Table

The following graphic shows four options for storing a NoPI table in Teradata Database, where the darkened row for each table represents a table header, one on each AMP, for a physical table. The table header defines the layout and other information about the physical table.

Moving from left to right, the structures represent the layouts of four different methods of NoPI storage.

- Traditional Teradata Database row storage without partitioning
- A column-partitioned table with single-column containers (COLUMN format), one for each column
- A column-partitioned table with single-column containers for columns A and B; multicolmn subrows (ROW format) for columns C and D
- A column-partitioned table with single-column containers for columns A and B; multicolmn containers for columns C and D



The topics that follow compare various configurations of nonpartitioned NoPI and column-partitioned tables.

## About Column-Partitioned Tables

If a request requires multiple columns to return the requested result set, the Optimizer query plan includes assembling projected column values from selected table rows together to form the result rows. This can be combined with row partition elimination to further reduce the data that must be accessed to satisfy a query.

Column partitioning has various system-applied compression techniques available to it that can be employed automatically to reduce the storage requirements for a table. These compression methodologies can reduce the I/O required to process queries and, when combined with row partitioning, the number of compression opportunities might increase.

The following CREATE TABLE request creates a table definition with four column partitions, two of which are internal partitions that Teradata Database creates for all column-partitioned tables.

The theoretical maximum number of column partitions for this table is 65,534, including the two column partitions for internal use, and the maximum column partition number is 65,535. The difference is caused by there always being at least one unused column partition number available for altering a column partition.

However, this table can actually have no more than 2,050 column partitions because the total number of columns for a table cannot exceed 2,048. To have 2,050 column partitions, each user-defined column partition could have only one column.

```
CREATE TABLE sales_1 (
    storeid          INTEGER NOT NULL,          /* Partition 1 */
    productid        INTEGER NOT NULL,
    salesdate        DATE FORMAT 'yyyy-mm-dd' NOT NULL,
```

```

        totalrevenue DECIMAL(13,2)),
        (totalsold INTEGER, /* Partition 2 */
        topsalesperson INTEGER,
        note VARCHAR(256)))
NO PRIMARY INDEX -- This clause is optional.
PARTITION BY COLUMN; -- Defines 2 multicolumn partitions plus 2
-- additional column partitions for internal
-- use.
    
```

The following example creates a table definition for a column-partitioned table with row partitioning.

The COLUMN specification in the PARTITION BY clause defines 7 column partitions: 5 column partitions based on the column grouping in the column definition list plus 2 additional column partitions for internal use.

The maximum number of column partitions is 779 including the 2 column partitions for internal use. This means that it is possible to add 770 column partitions to this table. The maximum column partition number is 780.

The RANGE\_N function in the partitioning expression defines 48 row partitions with the option to add up to 36 row partitions for a maximum of 84 row partitions. The maximum row partition number is 84.

The maximum combined partition number for this table is 65,520, which is computed from the following product: 780\*84. This is the product of the maximum partition number of each partitioning level. The table has 2-byte partitioning because the number of combined partitions is not greater than 65,535.

If you were to create this table using multicolumn partitions, you could add columns up to the maximum of 2,048. You could also specify the ADD option to specify a higher number of column partitions that could be added, but that would result in the table using 8-byte partitioning rather than 2-byte partitioning.

```

CREATE TABLE sales_2 (
        store_id      INTEGER NOT NULL,
        product_id    INTEGER NOT NULL,
        sales_date    DATE FORMAT 'yyyy-mm-dd' NOT NULL,
        total_revenue DECIMAL(13,2),
        (total_sold   INTEGER,
        top_salesperson INTEGER,
        note          VARCHAR(256)))
NO PRIMARY INDEX, -- This clause is optional.
PARTITION BY (COLUMN, -- Defines 4 single-column partitions and
-- 1 multicolumn partition plus 2
-- additional column partitions for internal
-- use.
        RANGE_N(salesdate BETWEEN DATE '2007-01-01'
                AND     DATE '2010-12-31'
                EACH INTERVAL '1' MONTH)
        ADD 36);
    
```

You can use column partitioning to improve query performance through column partition elimination.

You can use row partitioning to improve query performance through row partition elimination, which reduces the need to access all of the rows in a table. Teradata Database uses

efficient methods to reconstruct qualifying rows from the projected columns of the column partitions that are not eliminated.

Advantages of column partitioning include a simple CREATE TABLE syntax with default autocompression, the ability of the Optimizer to perform column partition elimination, and to facilitate efficient access data from column partitions. Teradata Database manages the column partitions so that the table appears to you as a single object.

## Example: Nonpartitioned NoPI Table

The following SQL text is the table definition for a nonpartitioned NoPI table. Note that this table does *not* have column partitioning.

```
CREATE TABLE orders (
    order_num      INTEGER NOT NULL,
    order_date     DATE NOT NULL,
    item_num       INTEGER COMPRESS NULL,
    item_desc      CHARACTER(30) COMPRESS NULL)
NO PRIMARY INDEX;
```

Because NoPI tables are always multiset tables, Teradata Database does not do duplicate row checks. Teradata Database randomly distributes rows or blocks of rows to the AMPs or locally copies them, as is the case for an INSERT ... SELECT request, instead of hash distributing them it does for a table with a primary index. In the case of a NoPI table, Teradata Database appends the rows for an AMP to the portion of that table owned by that AMP.

The table header and user data would look something like this.

Table Header	Row ID			Presence Bits	Orders			
	Partition Number	Hash Bucket	Uniqueness		OrderNum	OrderDate	ItemNum:N	ItemDesc:N
User Data Rows	0	n	1	1,1	100	07-29-2009	756	Hammer
	0	n	2	1,1	100	07-29-2009	124	Screwdriver
	0	n	3	1,1	100	07-29-2009	437	1-Inch Nails
	0	n	4	1,1	290	08-04-2009	110	Shovel
	0	n	5	0,1	290	08-04-2009	null	Rack
	0	n	6	1,1	450	09-01-2009	815	Light Bulb
	0	n	7	1,1	501	09-15-2009	437	1-Inch Nails
	0	n	8	0,1	530	09-15-2009	null	Drill
	0	n	9	1,1	625	10-03-2009	815	Light Bulb
	0	n	10	1,0	719	10-07-2009	756	null
	0	n	11	1,1	768	11-06-2009	756	Hammer

where:

Table element ...	Specifies ...
Partition Number	<p>the internal partition number indicating the column partition and row partitions for this row.</p> <p>Because there is no partitioning for this example, the partition number for every row in the table is 0.</p> <p>Bits in the flag byte of a physical row indicate no partitioning so a 2-byte or 8-byte partition number of 0 is implied, but does not actually exist in a physical row of a NoPI table.</p>
Hash Bucket	a 20-bit (or 16-bit) hash bucket value.
n	<p>the lowest numbered hash bucket for this AMP from the NoPI hash map.</p> <p>This value increases to the next hash bucket from the NoPI hash map for this AMP if the uniqueness overflows.</p> <p>Teradata Database does not actually compute n by hashing the values of any columns, but whatever AMP the row is randomly sent initially uses its lowest numbered hash bucket from the NoPI hash map.</p>
Uniqueness	<p>a uniqueness value.</p> <ul style="list-style-type: none"> <li>The value is 48 bits for a 16-bit hash bucket.</li> <li>The value is 44 bits for a 20-bit hash bucket.</li> </ul> <p>If the uniqueness value overflows, it resets to 1.</p>
N	a nullable column with COMPRESS NULL.
Presence Bits	<p>null compression with a presence indicator bit.</p> <p>If the presence bit is 1, a value is present; otherwise, no value is present. This means the field is null, and the column takes no space because COMPRESS NULL is specified.</p> <p>No presence bits exist if the column is defined as NOT NULL.</p> <p>In the example, only 2 presence bits are shown per row because only 2 columns in the table are nullable.</p>

## Example: Column-Partitioned Table With No Autocompression

The following SQL text is the table definition for the *orders* column-partitioned table with no autocompression.

```

CREATE TABLE orders (
    (order_num    INTEGER NOT NULL),
    (order_date   DATE NOT NULL),
    (item_num     INTEGER COMPRESS NULL),
    (item_desc    CHARACTER(30) COMPRESS NULL CHARACTER SET LATIN)
)
NO PRIMARY INDEX
PARTITION BY COLUMN NO AUTO COMPRESS;

```

NO AUTO COMPRESS can be omitted if the default settings are changed to NO AUTO COMPRESS. Because a column-partitioned table is also a multiset table, Teradata Database does not do any duplicate row checks. Column-partitioned tables are also similar to

nonpartitioned NoPI tables in that Teradata Database randomly distributes rows or blocks of rows to the AMPs or are copied locally as is done for an INSERT ... SELECT request instead of being hash-distributed as is done for a table with a primary index. Teradata Database appends each column partition value of a row for an AMP to the column partition or to the combined partition if multilevel partitioning is being used on that AMP.

This example examines a column-partitioned table with four column partitions (plus two for internal use) with one column per user-specified partition. The user-specified column partitions are not autocompressed and they all have COLUMN format.

COLUMN format is a series of consecutive column partition values stored in a container, where the rowID of the container is the standard rowID consisting of an internal partition number, a hash bucket number, and a uniqueness value associated with the first column partition value in the container. The internal partition number in this case represents the column and row partition numbers, if any.

The rowIDs associated with subsequent column partition values are implied by their position in the container. The rowIDs associated with the column partition values in the container, which are explicit for the first and implied for the remainder, have the same internal partition number and hash bucket, and the uniqueness increases by one for each column partition value represented by the container.

The file system stores a container based on the rowID in its row header as is currently done for a physical row representing a row in a table that is not column-partitioned.

A column partition has as many containers as needed to hold all of the column partition values of the table columns included in that partition. Teradata Database stores the column partition values in the inserted row order within a container, and the containers are in row-insertion order within a combined partition. To simplify the example, only two containers are shown for each of the column partitions, and each container only represents a small number of column-partition values.

Containers for different column partitions can have a different number of values represented per container. In this example, the first container for *item\_desc* represents 5 column values, while the first container for each of the other columns represent 6 column values. The second container for *item\_desc* starts at uniqueness value 6, while the second container for each of the other columns starts at uniqueness value 7.

Because no rows are deleted, the internal delete column partition is empty. Note that the first uniqueness value for a container plus the number of column partition values represented by that container does not equal the first uniqueness value of the next container if the intervening containers are deleted or if the internal partition number or hash bucket changes for the next container.

A null column in the table takes no place in the column value list.

The internal delete column partition, partition number 65,535 is empty in this case because the table is column-partitioned.

The table header and user data for this example would look something like this.

Table Header	Row ID			Presence Bits	Orders				
	Partition Number	Hash Bucket	Uniqueness		OrderNum:1,NAC	OrderDate:2,NAC	ItemNum:3,NAC,N	ItemDesc:4,NAC,N	
Column Values →									
OrderNum	1	n	1		100	100	100	290	290
	1	n	7		501	530	625	719	768
OrderDate	2	n	1		07-29-2009	07-29-2009	07-29-2009	08-04-2009	08-04-2009
	2	n	7		09-15-2009	09-15-2009	10-03-2009	10-07-2009	11-06-2009
ItemNum	3	n	1	1,1,1,1,0,1	756	124	437	110	815
	3	n	7	1,0,1,1,1	437	815	756	756	
ItemDesc	4	n	1	1,1,1,1,1	Hammer	Screwdriver	1-Inch Nails	Shovel	Rack
	4	n	6	1,1,1,1,0,1	Light Bulb	1-Inch Nails	Drill	Light Bulb	Hammer

where:

Table element ...	Specifies ...
Partition Number	<p>a 2-byte or 8-byte internal partition number indicating the column partition and row partitions for a container.</p> <p>Bits in the flag byte of a physical row indicate the partitioning, which in this case is 2-byte partitioning, and a non-0 2-byte part of the internal partition number exists in a physical row of this column-partitioned table.</p>
Hash Bucket	a 20-bit (or 16-bit) hash bucket value.
n	<p>the lowest numbered hash bucket for this AMP from the NoPI hash map.</p> <p>This value increases to the next hash bucket for this AMP if the uniqueness overflows.</p> <p>Teradata Database does not compute n by hashing the values of any columns, but whatever AMP the row is randomly sent initially uses its lowest numbered hash bucket from the NoPI hash map.</p>
Uniqueness	<p>a uniqueness value.</p> <ul style="list-style-type: none"> <li>The value is 48 bits for a 16-bit hash bucket.</li> <li>The value is 44 bits for a 20-bit hash bucket.</li> </ul> <p>If the uniqueness value overflows, it resets to 1.</p>
NAC	no autocompression.
N	a nullable column with COMPRESS NULL.
Presence Bits	<p>null compression with a presence indicator bit.</p> <p>If the presence bit is 1, a value is present; otherwise, no value is present. This means the field is null, and the column takes no space because COMPRESS NULL is specified.</p> <p>No presence bits exist if the column is defined as NOT NULL.</p> <p>In the example, only 2 presence bits are shown per row because only 2 columns in the table, <i>item_num</i> and <i>item_desc</i>, are nullable.</p>

### Example: Column-Partitioned Table With Autocompression

The following SQL text is the table definition for a column-partitioned table with autocompression.

```
CREATE TABLE orders (
    order_num    INTEGER NOT NULL,
    order_date   DATE NOT NULL,
    item_num     INTEGER,
    item_desc    CHARACTER(30) CHARACTER SET LATIN)
NO PRIMARY INDEX
PARTITION BY COLUMN AUTO COMPRESS;
```

AUTO COMPRESS can be omitted as long as the default settings have not been changed to NO AUTO COMPRESS. With the exception of the column partitions being autocompressed

in this example, it is identical to the example used in “[Example: Column-Partitioned Table With Autocompression](#)” on page 292.

A column field in the table takes no place in the column value list.

Values in *italic* typeface are in the local value list dictionary for the container.

The internal delete column partition, partition number 65,535 in this case because the table is only column-partitioned, is empty.

Table Header	Row ID			ACT	Autocompression Bits	Orders				
	Partition Number	Hash Bucket	Uniqueness			OrderNum:1NAC	OrderDate:2,AC	ItemNum:3,AC,N	ItemDesc:4,AC,N	
Column Values →										
OrderNum	1	n	1	R2 TT01 2,0	1111,1010,0110	100	290	450		
	1	n	7	TT01 2,0	10,10,10,10,10	501	530	625	719	768
OrderDate	2	n	1	R2	11,10,01	07-29-2009	08-04-2009	09-01-2009		
	2	n	7	PV 3,0,1	001,001,010,011,100	10-03-2009	10-07-2009	11-06-2009	09-15-2009	
ItemNum	3	n	1	P	1,1,1,1,0,1	756	124	437	110	815
	3	n	7	PV 3,0,1	010,000,011,001,100	437	815	756		
ItemDesc	4	n	1	TL01 4,’	0110,1011,1100, 0110,0100	Hammer	Screwdriver	1-Inch Nails	Shovel	Rack
	4	n	6	PV 3,0,1 TL01 4,’	0011010,010,1100, 0110101,0011010, 0000000,1000110	1-Inch Nails	Drill	Hammer	<i>Light Bulb</i>	

where:

Table element ...	Specifies ...
Partition Number	a 2-byte or 8-byte internal partition number indicating the column partition and row partitions for a container.  Bits in the flag byte of a physical row indicate the partitioning, which in this case is 2-byte partitioning, and a non-0 2-byte part of the internal partition number exists in a physical row of this column-partitioned table.
Hash Bucket	a 20-bit (or 16-bit) hash bucket value.
n	the lowest numbered hash bucket for this AMP from the NoPI hash map.  This value increases to the next hash bucket for this AMP if the uniqueness overflows.  Teradata Database does not compute n by hashing the values of any columns, but whatever AMP the row is randomly sent initially uses its lowest numbered hash bucket from the NoPI hash map.
Uniqueness	a uniqueness value. <ul style="list-style-type: none"> <li>• The value is 48 bits for a 16-bit hash bucket.</li> <li>• The value is 44 bits for a 20-bit hash bucket.</li> </ul> If the uniqueness value overflows, it resets to 1.
AC	autocompression for containers of the column
N	a nullable column.
ACT	autocompression type.

The internal delete column partition, column partition 65,535, is empty because the table is column-partitioned.

The following bullets explain the autocompression used for each container in this example.

- First *OrderNum* container.

Teradata Database applied 2 autocompression techniques to this container.

- The first two bits of each set of 4 autocompression bits are for R 2 (run length) autocompression, where the unsigned value of the two bits is the run length.
- The second two bits are for TT01 2,0 (trim high-order bytes that are zero) autocompression.

The number of sets of bits is the same as the number of values in this container since nulls do not need to be represented by this container.

This container represents 6 values, with the uniqueness for rowIDs starting at 1 and continuing up to 6. Because the first value, 100, has a run length of 3, the second value, 290, has a run length of 2, and the last value, 450, has a run length of 1.

The TT01 bits for the first value, 100, means that the value 100 is stored in the container as a single byte, and the 3 high-order 0 bytes are stripped.

The TT01 bits for the second value, 290, means that the value is stored as two bytes, and the 2 high-order 0 bytes are stripped.

The third value, 450, means the value is stored as 2 bytes, and the 2 high-order bytes are stripped, even though the data type is INTEGER which normally requires 4 bytes.

- Second *OrderNum* container.

Teradata Database applied only TT01 2,0 autocompression to this container because it has no runs.

There is one set of bits for each value, but in this case, there are only 2 bits per value because run length bits are not included. The container represents a sequence of 5 values, with the uniqueness value for rowIDs starting at 7 as indicated by the rowID of the container and continuing up to 11.

Each of the 5 values is stored as 2 bytes and the 2 high-order 0 bytes are stripped from each value.

- First *OrderDate* container.

Teradata Database applied R 2 (run length) autocompression to this container.

The number of sets of bits is the same as the number of values in this container because it does not need to represent nulls.

The container represents 6 values, with the uniqueness value for rowIDs starting at 1 and continuing up to 6. Because the first value, 07-29-2009, has a run length of 3, the second value, 08-04-2009, has a run length of 2, and the last value, 09-01-2009, has a run length of 1.

- Second *OrderDate* container.

Teradata Database applied PV 3,0,1 (single null and local value list) autocompression to this container.

In this case, there are 3 bits for each value represented, but only 3 values and 1 local value list dictionary value are stored in the container.

The container represents a sequence of 5 values, with the uniqueness value for rowIDs starting at 7 and continuing up to 11. The autocompression bits for the first 2 values indicate that their values are not present and to use the first entry in the local value list dictionary as their values.

The autocompression bits for the next 3 values indicate the value is present, not compressed, and an index to the value in the list of values.

- First *ItemNum* container.

Teradata Database applied P (single null) autocompression to this container.

There is one bit for each of the 6 values represented, but only 5 values in the container because ItemNum is null for the fifth row of the table and a value is not present.

Note that a null takes no space even though COMPRESS NULL is not specified because autocompression is applied. It is also possible that space for the column partition value could be in the container even if COMPRESS is specified because the system determines the compression used when autocompression is in effect.

- Second *ItemNum* container.

Teradata Database applied PV 3,0,1 (single null and local value list) autocompression to this container.

In this case, there are 3 bits for each of the 5 values represented but only 2 values and 1 local value list dictionary value are stored in the container.

This container represents a sequence of 5 values (with the uniqueness for rowIDs starting at 7 and continuing to 11). The autocompression bits for the first and third values indicate that the values are present (437 and 815, respectively), not compressed, and that there is an index to the value in the list of values.

The bits for the second value indicate that it is null and not present. The bits for the fourth and fifth values indicate that their values are not present and that Teradata Database should use the first entry in the local value list dictionary as their values.

- First *ItemDesc* container.

Teradata Database applied TL01 4,' ' (trim trailing single-byte space for fixed-length data type) autocompression to this container.

In this case, there are 4 bits for each of the 5 values represented and 5 values (stripped of trailing spaces) in the container.

The 4 bits for a value indicate its length after trimming trailing spaces. The uncompressed value can be obtained from the value in the container by adding trailing spaces as needed to a fixed length of 30. In this case, the first container represents only 5 values, but the total number of values represented by both containers of *ItemDesc* is 11, the same as the total for the other columns.

- Second *ItemDesc* container.

Teradata Database applied PV 3,0,1 (single null and local value list) and TL01 4,' ' (trim trailing single-byte space for fixed-length data type) autocompression to this container.

In this case, there are 7 bits for each of the 6 values represented, but only 3 values and 1 local value list dictionary value are stored in the container.

The first 3 bits indicate whether the value is null (00) and not present, is not present and to use the entry in the local value-list dictionary (01) as its value, or present (010, 011, 100, 110, 111 indicating an index in the list of values). The other 4 bits indicate the length of the value, if present, after trimming trailing spaces. The length is 0000 if the field is null, so it is ignored.

This container represents a sequence of values, with the uniqueness for rowIDs starting at 6 as indicated by the rowID of the container and continuing to 11. The autocompression bits for the first and fourth values indicate that their values (Light Bulb) are not present and to use the entry in the local value list dictionary as their values.

The bits for the second, third, and sixth values (1-inch Nails, Drill, and Hammer, respectively) indicate that the values are present, but stripped of trailing spaces. The bits for the fifth value indicate that it is null and not present.

The stripped values can be extended with spaces to their fixed length of 30 when decompressing the values.

The types of autocompression that Teradata Database chooses to apply to the containers for a table are chosen on a container by container basis, as this example demonstrates. Teradata Database could just as easily apply other autocompression types, depending on the context for

each container to which autocompression was applied. Optionally, some column partitions can be set for AUTO COMPRESS and some for NO AUTO COMPRESS by specifying AUTO COMPRESS or NO AUTO COMPRESS as appropriate for each column partition to override the default.

## Column Partitioning Performance

You can exploit column partitioning to improve the performance of some classes of workloads. The general performance impact of column partitioning is summarized in the following bullets.

- You should see a significant I/O performance improvement for requests that access a variable small subset of the columns and rows of a column-partitioned table or join index. This includes accessing columns to respond to both predicates and projections.

For example, if 20% of the data in rows is required to return the result set for a query, the I/O for a column partitioned table should be approximately 20% of the I/O for the same table without column partitioning.

Column partitioning can further reduce I/O depending on the effectiveness of autocompression and row header compression.

Additional I/O might be required to reassemble rows if many columns are projected or specified in query predicates.

- You might see a negative impact on the performance of queries that access more than a small subset of the columns or rows of a column-partitioned table or join index.
- You might see a relative increase in spool size compared to the size of a source column-partitioned table.

When data from an autocompressed column-partitioned table is spooled, Teradata Database does not carry the autocompression over to the spool because the spool is row-oriented. This can lead to a large spool relative to the compressed data in the column-partitioned table.

Because user-specified compression *is* carried over to the spool, applying user-specified compression to the column-partitioned table might be beneficial if spool usage in terms of space consumption and I/O operations becomes an issue.

- You might see a reduction in CPU usage for column-partitioned tables.

At the same time, consumption of CPU resources might increase to process autocompression, decompression, and containers.

With a reduction in I/O and a possible increase in CPU, workloads can change from being I/O-bound to being CPU-bound, and the performance of CPU-bound workloads might not improve, and might even be worse, with column partitioning.

- You might see a negative performance impact on INSERT operations for a column-partitioned table or join index, especially for single-row inserts, and less so for block-at-a-time bulk insert operations such as array inserts and INSERT ... SELECT operations.

For an INSERT ... SELECT operation, the CPU cost increases as the number of column partitions increases because there is a cost to split a row into multiple column partitions.

Teradata Database scans the source  $\text{CEILING}(\frac{\text{number\_user\_spec\_col\_part}}{\text{number\_available\_col\_part\_contexts}})$  times, where:

Equation element ...	Specifies the number of ...
<i>number_user_spec_col_part</i>	user-specified column partitions.
<i>number_available_col_part_contexts</i>	available column partition contexts.

This might be less of a factor to consider than the increase in CPU consumption as the number of column partitions is increased.

You must understand the potential tradeoffs when you consider the number of column partitions you create for a table. For example, workloads that contain a large number of INSERT operations can benefit from a table with fewer column partitions when it comes to CPU usage, but creating the table with columns in their own individual partitions might be more optimal for space usage and decreasing the number of I/Os, so you must determine an appropriate balance among the factors that you can finely tune.

For example, a good candidate for column partitioning is a table where the workloads that access it are heavily query-oriented, and the benefits gained from column partitioning, even though the partitioning increases the CPU cost to load the data, a good tradeoff.

Value compression and autocompression can have a negative impact on CPU consumption for workloads that tend to insert many rows, similar to the impact that is seen with a nonpartitioned table that has compression. Because Teradata Database applies autocompression to every column partition by default, this can cause a significant increase in CPU consumption compared to multi-value compression, which might only be selectively applied to columns.

However, compression can significantly reduce space usage and decrease the I/O operations required for the INSERT operations and for subsequent requests, making the tradeoff between increased CPU consumption and decreased I/O operations a factor that must be considered.

Be aware that FastLoad and MultiLoad are not supported for column-partitioned tables.

- You might see a negative performance impact for UPDATE operations that select more than a small subset of rows to be updated. Because updates are done as a DELETE operation followed by an INSERT operation, Teradata Database needs to access all of the columns of rows selected for update.

The cost of performing updates that access many rows when the table is column-partitioned might not be acceptable, and if it is not, you should avoid column partitioning the table. For these kinds of UPDATE operations, doing an INSERT ... SELECT request into a copy of the table might be a better alternative.

- Take care not to over-partition tables.

In extreme cases of over-partitioning, a column-partitioned table might be as much as 22 times larger than a table that is not column-partitioned.

Row partitioning of a column-partitioned table might result in over-partitioning such that only a few values with the same combined partition number occur and, so only a few values are included in each of the containers, which reduces the effectiveness of row header compression.

When increasingly more containers are required to respond to a request, each supporting fewer column partition values, the advantage of row header compression is lost. In addition, more I/O is required to access the same amount of useful data. A data block might contain a mix of eliminated and non-eliminated combined partitions for a query. But to read the non-eliminated combined partitions, the entire data block must be read and, therefore, eliminated combined partitions in the data block are also being read unnecessarily.

Over-partitioning may exist when populated combined partitions have fewer than 10 data blocks. With 10 data blocks per combined partition, 10% of the data that Teradata Database reads is not required by a request. As the number of data blocks decreases, in increasingly large quantity of unneeded data is read. In the worst case, even if there is column partition elimination, all the data blocks of the table must be read.

The magnitude of performance changes when accessing a column-partitioned table or join index, both positive and negative, can range over several orders of magnitude depending on the workload. You should not column-partition a table when performance is so severely compromised that you cannot offset the reduced performance with physical database design choices such as join indexes.

These and other impacts of column partitioning are described in more detail in the topics that follow.

## Cases Where Positive Performance Effects Are Most Likely To Occur

The greatest improvement in the performance of querying column-partitioned tables occurs when a request specifies a highly selective predicate on a column in a single-column partition of a column-partitioned table with hundreds or thousands of columns and only a small number of them are projected by the request.

## Cases Where Negative Performance Effects Are Most Likely To Occur

The greatest decrement in the performance of querying column-partitioned tables occurs when the following conditions take place.

- Most or all of the columns in a column-partitioned table are projected by a request.
- The request is not selective.
- The table being queried has thousands of column partitions.
- The retrieval performance for a column-partitioned table or join index is not good when the number of column partition contexts that are available is significantly fewer than the number of column partitions that must be accessed to respond to the query. Note that there are at least eight available column partitions contexts. Depending on your memory configuration, there may be more available contexts.

When this happens, consider the following remedies.

- Reconfigure the table or join index to decrease the number of column partitions that need to be accessed by combining column partitions so there are fewer of them.
- If there is enough memory available, increase the setting for the DBS Control parameter PPICacheThrP.
- The table being queried has enough row-partitioned levels that there are very few physical rows in populated combined partitions, and the physical rows contain only one or a few column partition values.

## Autocompression

When you create a column-partitioned table or join index, Teradata Database attempts to use one or more methods to compress the data that you insert into the physical rows of the object unless you specify the NO AUTO COMPRESS option at the time you create it or NO AUTO COMPRESS is set as the default. The process of selecting and applying appropriate compression methods to the physical containers of a column-partitioned table or join index is referred to as *autocompression*.

Autocompression is most effective for a column partition with a single column and COLUMN format.

Teradata Database only applies autocompression to column partitions with COLUMN format, and then only if it reduces the size of a container. Teradata Database autocompresses column partitions by default with the following requirements.

- Minimal CPU resources are required to decompress the data for reading.
- Teradata Database does not need to decompress many values to find a single value.

Teradata Database applies autocompression for a physical row on a per container basis. For efficiency, the system may use the autocompression method chosen for the previous container, including not using autocompression, if that is more effective. Containers in a column partition might be autocompressed in different ways. In most cases, the data type of a column is not a factor, and Teradata Database compresses values based only on their byte representation. As a general rule, the only difference that needs to be considered is whether the byte representation is fixed or variable length.

For some values there are no applicable compression techniques that can reduce the size of the physical row, so Teradata Database does not compress the values for that physical row, but otherwise the system attempts to compress physical row values using one of the autocompression methods available to it. When you retrieve rows from a column-partitioned table, Teradata Database automatically decompresses any compressed column partition values as is necessary.

Because the selection of autocompression methods used for a container is made by Teradata Database and not by users, the methods that autocompression can select from to compress the data in a column partition are not documented in the Teradata user documentation library.

## Examples

Consider the following PARTITION BY clause as an example of what happens with excess partitions. Assume that *o\_custkey* has the INTEGER data type, *o\_orderdate* has the DATE data

type, and there are 8 columns defined for the table. Teradata Database defines 8-byte partitioning because the maximum combined partition number before adding excess partitions to a level is 462,000 ((8+2+1)\*500\*84), which is greater than the maximum combined partition number for 2-byte partitioning, which is 65,335.

```
PARTITION BY (COLUMN,
    RANGE_N(o_custkey BETWEEN 0
        AND      499999
        EACH 1000),
    RANGE_N(o_orderdate BETWEEN DATE '2003-01-01'
        AND      DATE '2009-12-31'
        EACH INTERVAL '1' MONTH) )
```

The partitioning for this database object has the following characteristics.

- The number of column partitions defined for level 1 is 10, including two internal use column partitions and assuming a single column per partition.  
The maximum number of column partitions is 20, meaning that 10 additional column partitions could be added.  
The maximum column partition number is 21.
- The number of row partitions defined for level 2 is 500.  
Teradata Database adds any excess partitions to level 2, so level 2 has a maximum of 5,228,668,955,133,092 row partitions.  
The default for level 2 is ADD 5228668955132592.
- The number of row partitions defined for level 3 is 84, which is the same number of partitions defined for level 3.  
Because this is not the first row partitioning level that does not specify an ADD option, the default is ADD 0.

The implication of all this is that the partitioning specified by the preceding PARTITION BY clause is equivalent to the following PARTITION BY clause.

```
PARTITION BY (COLUMN ADD 10,
    RANGE_N(o_custkey BETWEEN 0
        AND      499999
        EACH 1000)
    ADD 5228668955132592,
    RANGE_N(o_orderdate BETWEEN DATE '2003-01-01'
        AND      DATE '2009-12-31'
        EACH INTERVAL '1' MONTH)
    ADD 0 )
```

The adjusted maximum combined partition number is 9,223,372,036,854,774,288.

The following example is a full CREATE TABLE request that creates the column-partitioned table named *t1*.

```
CREATE TABLE t1 (
    a01 INTEGER, a02 INTEGER, a03 INTEGER, a04 INTEGER, a05 INTEGER,
    a06 INTEGER, a07 INTEGER, a08 INTEGER, a09 INTEGER, a10 INTEGER,
    a11 INTEGER, a12 INTEGER, a13 INTEGER, a14 INTEGER, a15 INTEGER,
    a16 INTEGER, a17 INTEGER, a18 INTEGER, a19 INTEGER, a20 INTEGER,
    a21 INTEGER, a22 INTEGER, a23 INTEGER, a24 INTEGER, a25 INTEGER,
    a26 INTEGER, a27 INTEGER, a28 INTEGER, a29 INTEGER, a30 INTEGER,
```

```

a31 INTEGER, a32 INTEGER, a33 INTEGER, a34 INTEGER, a35 INTEGER,
a36 INTEGER, a37 INTEGER, a38 INTEGER, a39 INTEGER, a40 INTEGER,
a41 INTEGER, a42 INTEGER, a43 INTEGER, a44 INTEGER, a45 INTEGER,
a46 INTEGER, a47 INTEGER, a48 INTEGER, a49 INTEGER, a50 INTEGER,
a51 INTEGER, a52 INTEGER, a53 INTEGER, a54 INTEGER, a55 INTEGER,
a56 INTEGER, a57 INTEGER, a58 INTEGER, a59 INTEGER, a60 INTEGER,
a61 INTEGER, a62 INTEGER, a63 INTEGER, a64 INTEGER, a65 INTEGER,
a66 INTEGER, a67 INTEGER, a68 INTEGER, a69 INTEGER, a70 INTEGER,
a71 INTEGER, a72 INTEGER, a73 INTEGER, a74 INTEGER, a75 INTEGER,
a76 INTEGER, a77 INTEGER, a78 INTEGER, a79 INTEGER, a80 INTEGER,
a81 INTEGER, a82 INTEGER, a83 INTEGER, a84 INTEGER, a85 INTEGER,
a86 INTEGER, a87 INTEGER, a88 INTEGER, a89 INTEGER, a90 INTEGER,
a91 INTEGER, a92 INTEGER, a93 INTEGER, a94 INTEGER, a95 INTEGER,
a96 INTEGER, a97 INTEGER)
NO PRIMARY INDEX
PARTITION BY (RANGE_N(a2 BETWEEN 1
                     AND      48
                     EACH 1,
                     NO RANGE, UNKNOWN)
              ADD 10,
              COLUMN,
              RANGE_N(a3 BETWEEN 1
                     AND      50
                     EACH 10));

```

The partitioning for this table has the following characteristics.

- The number of partitions defined for level 1 is 50 and, initially, the maximum number of partitions and the maximum partition number for this level is  $(50+10) = 60$  because there is an ADD 10 clause for this level.
- The number of partitions defined for level 2 is 99: 97 user-specified partitions plus 2 for internal use.

Because there is a level of row partitioning without an ADD clause, the column partitioning level has a default of ADD 10, so the maximum number of partitions for this level is 109, with a maximum column partition number of 110.

- The number of partitions defined for level 3 is 5. Because there is no ADD clause specified for this level and it is the first row partitioning level without an ADD clause, assuming a default of ADD 0, the maximum combined partition number before adding excess partitions to a level is  $(60 \times 110 \times 5)$ , or 33,000, which is not greater than 65,535, so this partitioning consumes 2 bytes in the row header.

The actual maximum number of partitions for this level is the largest number that would not cause the maximum combined partition number to exceed 65,535. This value

is  $\text{FLOOR}(\frac{65535}{60 \times 110}) = 9$ . Therefore, this level has a default of ADD 4, and the maximum number of partitions for the level is 9.

The adjusted maximum combined partition number for the table is  $(60 \times 110 \times 9)$ , or 59,400.

Any remaining excess partitions can be added to level 1, meaning that the maximum number of partitions for level 1 can be increased such that the maximum combined partition number does not exceed 65,535.

This value is  $\text{FLOOR}(\frac{65535}{110 \times 9}) = 66$  and the ADD 10 clause for level 1 can be replaced by ADD 16.

The implication of all this is that the partitioning specified by the preceding CREATE TABLE request is equivalent to the following CREATE TABLE request.

```

CREATE TABLE t1 (
    a01 INTEGER, a02 INTEGER, a03 INTEGER, a04 INTEGER, a05 INTEGER,
    a06 INTEGER, a07 INTEGER, a08 INTEGER, a09 INTEGER, a10 INTEGER,
    a11 INTEGER, a12 INTEGER, a13 INTEGER, a14 INTEGER, a15 INTEGER,
    a16 INTEGER, a17 INTEGER, a18 INTEGER, a19 INTEGER, a20 INTEGER,
    a21 INTEGER, a22 INTEGER, a23 INTEGER, a24 INTEGER, a25 INTEGER,
    a26 INTEGER, a27 INTEGER, a28 INTEGER, a29 INTEGER, a30 INTEGER,
    a31 INTEGER, a32 INTEGER, a33 INTEGER, a34 INTEGER, a35 INTEGER,
    a36 INTEGER, a37 INTEGER, a38 INTEGER, a39 INTEGER, a40 INTEGER,
    a41 INTEGER, a42 INTEGER, a43 INTEGER, a44 INTEGER, a45 INTEGER,
    a46 INTEGER, a47 INTEGER, a48 INTEGER, a49 INTEGER, a50 INTEGER,
    a51 INTEGER, a52 INTEGER, a53 INTEGER, a54 INTEGER, a55 INTEGER,
    a56 INTEGER, a57 INTEGER, a58 INTEGER, a59 INTEGER, a60 INTEGER,
    a61 INTEGER, a62 INTEGER, a63 INTEGER, a64 INTEGER, a65 INTEGER,
    a66 INTEGER, a67 INTEGER, a68 INTEGER, a69 INTEGER, a70 INTEGER,
    a71 INTEGER, a72 INTEGER, a73 INTEGER, a74 INTEGER, a75 INTEGER,
    a76 INTEGER, a77 INTEGER, a78 INTEGER, a79 INTEGER, a80 INTEGER,
    a81 INTEGER, a82 INTEGER, a83 INTEGER, a84 INTEGER, a85 INTEGER,
    a86 INTEGER, a87 INTEGER, a88 INTEGER, a89 INTEGER, a90 INTEGER,
    a91 INTEGER, a92 INTEGER, a93 INTEGER, a94 INTEGER, a95 INTEGER,
    a96 INTEGER, a97 INTEGER)
NO PRIMARY INDEX
PARTITION BY (RANGE_N(a2 BETWEEN 1
                      AND      48
                      EACH 1,
                      NO RANGE, UNKNOWN)
              ADD 16,
              COLUMN ADD 10,
              RANGE_N(a3 BETWEEN 1
                      AND      50
                      EACH 10)
              ADD 4);

```

Now the maximum combined partition number is  $(66 \times 110^9)$ , or 65,340. The maximum number of combined partitions is  $(66 \times 109^9)$ , or 64,746. The number of combined partitions is  $(50 \times 99^5)$ , or 24,750 and you can alter the table to add additional partitions to each of the partitioning levels.

## Autocompression and Spools

A spool generated from a table with autocompression does not have any autocompression, though it might have inherited user-specified compression based on the rules for spool inheriting the compression characteristics of its parent. This means a very large spool relative to the size of the column partitions can be generated if those column partitions are highly compressed by the autocompression and there is little or no selectivity.

## Autocompression Interactions With User-Specified Compression Methods

Teradata Database applies user-specified compression for columns within a multicolumn column partition value, but Teradata Database applies autocompression to a column partition value as a whole.

Teradata Database might decide not to use one or more of the user-specified compression techniques with autocompression if it determines that other autocompression techniques, including none, provide better compression for a container.

When the system constructs a container, it first applies any user-specified compression. When the container reaches a certain size limit, which is defined internally, Teradata Database examines the container to determine what autocompression techniques and user-specified compression can be used to reduce its size. After compressing the container, the system can append additional column partition values to it using the method of autocompression it has determined to be optimal and any user-specified techniques until a container size limit is reached, at which time Teradata Database constructs a new container.

As an alternative to the previous method, the system may decide that it is more efficient to reuse the compression from the previous container and apply that compression when constructing a container.

If you specify block-level compression, which is not an autocompression technique, for a column-partitioned table or join index, it is applied for data blocks independently of whether Teradata Database applies autocompression.

## Checking the Effectiveness of Autocompression

To check the effectiveness of autocompression, use the SHOWBLOCKS command of the Ferret utility to compare the size of the data or a sample of the data stored with and without compression. You should also compare the performance with and without compression to measure the benefits of each.

SHOWBLOCKS is documented in *Utilities: Volume 1 (A-K)*.

## Using the NO AUTO COMPRESS Option

You can override the autocompression default by specifying the NO AUTO COMPRESS option. This option ensures that Teradata Database uses the specified null compression, multi-value compression, or algorithmic compression for the column partition when you create the column-partitioned table or join index.

If you specify NO AUTO COMPRESS for a column partition, or if the column partition has ROW format, Teradata Database always applies any compression techniques that you specify to every container or subrow of the column partition. For column partitions with COLUMN format, Teradata Database applies row header compression, which is applied to column partitions that have COLUMN format. To disable row header compression, specify ROW format when you create a column-partitioned table or join index.

There is some overhead in determining whether or not a physical row should be compressed and, if so, what compression techniques to use. If Teradata Database determines that there is no appropriate technique to compress the physical rows of the column partition, or if you determine that the compression techniques used do not effectively compress the column partition, you can eliminate this overhead by specifying the NO AUTO COMPRESS option for the column partition.

## Anticipated Workload Characteristics for Column-Partitioned Tables and Join Indexes

The expected scenarios for column-partitioned tables and join indexes are those where the partitioned table data is loaded with an INSERT ... SELECT request with possibly some minor ongoing maintenance, and then the table is used to run analytics and data mining, at which point the table or row partitions are deleted, and the scenario then begins over again. This is referred to as an insert once scenario. Column-partitioned tables are not intended to be used for OLTP- or tactical query-type activities.

The design point for data maintenance of column-partitioned database objects is roughly 88% INSERT ... SELECT or array INSERT operations into empty table or row partitions, 2% other inserts operations, 7% update operations, and 3% delete operations.

Most requests that access a column-partitioned table or join index are expected to be selective on a variable subset of columns, or to project a variable subset of the columns, where the subset accesses fewer than 10% of the column partitions for any particular request.

The expected number of column partitions that need to be accessed for requests should preferably not exceed the number of available column partition contexts as it does in the following EXPLAIN of a SELECT request. If it does, there may be undesirable negative impact on performance in some cases.

Note that the count of 21 for the number of column partitions includes the 20 selected partitions in the SELECT statement and the delete column partition from the column-partitioned table. 21 exceeds the number of available column partition contexts, so 20 column partitions (including the delete column partition) of the column-partitioned table are merged into the first subrow column partition of the column-partitioned merge spool. The remaining column partition that needs to be accessed from the column-partitioned table is copied to the second subrow column partition in the column-partitioned merge spool (for a total of 2 subrow column partitions). This reduces the number of column partitions to be accessed at one time (which is limited by the number of available column-partition contexts). The result is then retrieved from the two subrow column partitions of the column-partitioned merge spool.

```
EXPLAIN SELECT a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, r,
           s, t, u
  FROM /*CP*/ t1;
*** Help information returned. 13 rows.
*** Total elapsed time was 1 second.
```

### Explanation

- 
- 1) First, we lock a distinct PLS."pseudo table" for read on a RowHash to prevent global deadlock for PLS.t1.
  - 2) Next, we lock PLS.t1 for read.
  - 3) We do an all-AMPS RETRIEVE step from 21 column partitions (20 contexts) of PLS.t1 using covering CP merge Spool 2 (2 subrow partitions and Last Use) by way of an all-rows scan with no residual conditions into Spool 1 (all\_amps), which is built locally on the AMPS. The size of Spool 1 is estimated with low confidence to be 2 rows (614 bytes). The estimated time for this step is 0.03 seconds.
  - 4) Finally, we send out an END TRANSACTION step to all AMPS involved in processing the request.  
-> The contents of Spool 1 are sent back to the user as the result of statement 1. The total estimated time is 0.03 seconds.

A column-partitioned table can be used as a sandbox table where data can be added until an appropriate indexing method is determined. Requests that access a small, but variable, subset of the columns might run more efficiently against a column-partitioned table compared to a NoPI table without column partitioning.

## General Performance Guidelines for Column Partitioning

Use these guidelines as a starting point, but understand that they might not be suitable for all workloads. As you gain experience using column partitioning, you might find that alternate choices are more appropriate in some cases.

Note the following general points about optimizing the performance of column-partitioned tables and join indexes.

- Keep in mind that column partitioning is not optimal for all query types and workloads.
- As is always true, you should test and prove any physical database design, preferably first on a test system with a valid sample of the data and then on the production system with the full data before releasing the design into the production environment.

This includes testing all of the following items before putting a column-partitioned table or join index into production use.

- Queries
- Workloads
- Bulk data loads
- Maintenance
- Archive, restore, and copy performance

Also be sure to check the space usage of the tables.

The performance guidelines for column-partitioned tables and join indexes are divided into the following subgroups.

- General performance guidelines for column partitioning
- Guidelines for queries, contexts, and table maintenance
- Guidelines for partitioning column-partitioned tables and join indexes
- Guidelines for specifying table and column attributes for column-partitioned tables
- Guidelines for specifying compression for column-partitioned tables and join indexes
- Guidelines on I/O operations, CPU usage, and disk space usage for column-partitioned tables and join indexes
- Guidelines for collecting statistics on column-partitioned tables and join indexes

## Guidelines for Queries, Contexts, and Table Maintenance for Column Partitioning

- The optimal application for column-partitioned tables is large fact tables or call detail tables that are accessed frequently by analytic workloads.

Workloads that rely heavily on joins and aggregations based on a primary index are not suited for column partitioning because column-partitioned tables do not have a primary index.

A join index might provide acceptable performance for requests that depend on primary index access, while column partitioning the base table could enable superior performance for other classes of workloads.

Use column partitioning for tables accessed by analytic queries where the tables are refreshed using bulk loading techniques periodically with possibly some interim modifications.

Do *not* use column-partitioned tables for highly volatile data, for cases in which a table is lightly populated with rows, or if the data is to be used in tactical query workloads.

- To facilitate query performance, you should make sure that you follow at least one, if not both, of the following guidelines.
  - Ensure that the number of column partitions accessed by a request does not exceed the number of available column partition contexts.
  - Write the request in such a way that it is highly selective.

Ideally, you should ensure that *both* of these guidelines are followed.

Requests in workloads that heavily access column-partitioned tables and do not conform to this recommendation should be minimized, because their performance is likely to be degraded.

To support acceptable performance for queries that have the desired characteristics, you should employ physical database design options like secondary and join indexes when possible. Before you employ indexes, you must understand that additional maintenance costs are always incurred because Teradata Database must update index subtables any time the base table columns they are defined on are updated.

- Because neither nonpartitioned NoPI tables nor column-partitioned tables have a primary index, they are not good choices if typical queries run against them would benefit from primary-indexed access. In this case, consider indexing the table using secondary or join indexes once its initial configuration is determined not to achieve adequate performance for its typical workloads.
- Always measure the INSERT cost for tables that are candidates for column partitioning. Do not use column partitioning when the increased cost of inserting data is not acceptable or is not offset by improved performance in the query workload.
- Because you should not use column-partitioned tables for highly volatile data, apply UPDATE operations to column-partitioned tables sparingly.
- Perform DELETE operations on a column-partitioned table either for the entire table or for entire row partitions.
- The Optimizer uses the DBS Control parameter PPICacheThrP to determine the number of available file contexts that can be used at one time to access a partitioned table.

If the number of available file contexts determined by PPICacheThrP is less than 8, then 8 file contexts are available. If the number of file contexts specified by PPICacheThrP is more than 256, then 256 file contexts are available. For a column-partitioned database

object, Teradata Database uses the number of file contexts as the number of available column partition contexts.

A file context can be associated with each column partition context for some operations, but in other cases, Teradata Database might allocate a buffer to be associated with each column partition context.

The ideal number of column partition contexts should be at least equal to the number of column partitions that need to be accessed by a query; otherwise, performance can degrade since not all the needed column partitions can be read at one time. Performance and memory usage can be impacted if PPICacheThrP is set too high, because too high a setting can lead to memory thrashing or even a system crash.

At the same time, the benefits of partitioning can be lessened if PPICacheThrP is set unnecessarily low, and performance might degrade significantly. The default setting for this parameter is expected to be optimal for most workloads; however, after monitoring performance and memory usage, you might need to adjust the setting to obtain the best balance.

- You should periodically refresh or append new rows to a column-partitioned table, or to the row partitions of the column-partitioned table, using INSERT ... SELECT requests that move large quantities of data.

Date or timestamp can partitioning may help to improve column-partitioned table maintenance.

See “[Bulk Loading a Column-Partitioned Table](#)” on page 318 for more information.

## Guidelines for Partitioning Column-Partitioned Tables and Join Indexes for Column Partitioning

- While you can define column partitioning at any level of multilevel partitioning, in most cases you should follow these guidelines when you configure the partitioning for a column-partitioned table.
  - Code the column partitioning level first and follow the column partitioning level with any row partitioning levels.
  - If you do not code the column partition at the first level, code it as the second level after DATE or TIMESTAMP row partitioning.

Some considerations that might lead to putting the column partitioning at a lower level are the following.

- Potential improvements for cylinder migration.
- Block compression effectiveness.
- If you specify row partitioning as part of a multilevel column partitioning for a table, consider specifying the ADD option for the any partitioning levels that might need to increase their number of partitions in the future.
- Unless you have a good reason not to, you should use the defaults when you specify PARTITION BY COLUMN. Do not override the defaults without first giving the reasons for doing so serious consideration

- For columns that are often specified in queries, but where the specific set of columns specified varies from request to request, you should create single-column partitions for the frequently specified columns.

- Use ROW format for wide column partitions because it has less overhead than a container that holds one or a few values.

If Teradata Database does not assign ROW format for a column partition, but you have determined that ROW format is more appropriate because it decreases space usage, specify ROW explicitly.

- Use COLUMN format for narrow column partitions, especially if autocompression is effective.

If Teradata Database does not assign COLUMN format for a multicolumn partition, but COLUMN is user-determined to be more appropriate (decreases space usage, etc.), specify COLUMN explicitly.

You might need to specify COLUMN format explicitly for a multicolumn partition that contains a column with a VARCHAR, VARCHAR(n) CHARACTER SET GRAPHIC, or VARBYTE data type and defined with a large maximum value, but where values are actually very short in most cases. This is because the system-determined format might be ROW based on the large maximum length.

## Guidelines for Specifying Table and Column Attributes for Column Partitioning

- Both nonpartitioned NoPI tables and column-partitioned tables are created with a MULTISET table type by default.  
Neither nonpartitioned NoPI tables nor column-partitioned tables can be created as a SET table. As a result, Teradata Database does not perform duplicate row checks for either.
- Use the system default settings for the table-level option DATABLOCKSIZE and the DBS Control parameter PermDBSize for a column-partitioned table unless performance analysis indicates otherwise.
- The settings for the table-level option FREESPACE and the DBS Control parameter FreeSpacePercent might require adjustment for a column-partitioned table or join index with small internal partitions, as might be the case if a table or join index is also row partitioned, particularly if you add data incrementally to the table or index.

These options specify the amount of space on each cylinder that is to be left unused during load operations, and the reserved free space allows table data to expand on current table cylinders, preventing or delaying the need for additional table cylinders to be allocated, preventing or delaying data migration operations associated with new cylinder allocations.

If you do a large INSERT ... SELECT operation and internal partitions are either large or empty, little or no free space is needed. Keeping new table data physically close to existing table data and avoiding data migrations, can improve overall system performance.

- Follow these column attribute guidelines when you create your column-partitioned tables.
  - Specify NOT NULL for columns that should not be null.

Because nullable columns can significantly decrease the effectiveness of autocompression, you should avoid them unless you have a sound reason for specifying them.

- Specify column-level CHECK constraints when you can.  
CHECK constraints are valid for both NoPI tables and column-partitioned tables.
- Specify UNIQUE and PRIMARY KEY constraints when you can.  
UNIQUE and PRIMARY KEY constraints are valid for both NoPI tables and column-partitioned tables.
- Specify foreign key constraints whenever they are applicable.  
Foreign key constraints are valid for both NoPI tables and column-partitioned tables.
- Many considerations for NoPI tables that are not column-partitioned also apply to column-partitioned tables because they are also NoPI tables.

The following features are valid for both NoPI tables and for column-partitioned tables:

- Fallback
- Unique secondary indexes
- Non-unique secondary indexes
- Join indexes
- Reference indexes
- The following SQL statements and Teradata Tools and Utilities utility are not allowed on nonpartitioned NoPI tables and column-partitioned tables:
  - UPDATE (Upsert Form)
  - MERGE
  - MultiLoad

For a primary-indexed table, Teradata Database generates the hash value of a row from the values of the columns that constitute the primary index. This hash value determines to which AMP a row is sent and stored. Although neither NoPI tables nor column-partitioned tables have a primary index, each row still must be assigned to a hash bucket, and the bucket number to which the row is assigned is generated internally.

This approach allows fallback and index maintenance to work as they would if the table had a primary index.

- You cannot specify permanent journaling for a column-partitioned table or a NoPI table.
- The following features are *not* shared by nonpartitioned NoPI tables and column-partitioned tables.
  - You cannot create a column-partitioned global temporary or volatile table.  
A nonpartitioned NoPI table can be created as a global temporary or volatile table.
  - A column-partitioned table can have an identity column, while a nonpartitioned NoPI table cannot.
  - The setting of the DBS Control PrimaryIndexDefault parameter does not affect the default primary index specification if PARTITION BY is specified. The default behavior for creating a column-partitioned table without specifying a primary index is always NO PRIMARY INDEX.

- You are not required to specify NO PRIMARY INDEX when you create a column-partitioned table, but you may need to specify NO PRIMARY INDEX for a nonpartitioned NoPI table, depending on the DBS Control settings.

The default behavior for CREATE TABLE for a column-partitioned table is NO PRIMARY INDEX if you do not specify a primary index. The setting of the PrimaryIndexDefault DBS Control parameter does not affect this behavior.

- XML, BLOB, and CLOB are valid data types for both nonpartitioned NoPI tables and column-partitioned tables.

**Note:** There is a limit of 256M rows per rowkey per AMP with XML, BLOB, and CLOB data types. NoPI tables normally have only one hash value on each AMP, so the effective limit on the number of rows per AMP is approximately 256M.

The exact number is 268,435,455 rows per rowkey per AMP.

For column-partitioned tables, these limits are per column partition:hash bucket combination rather than rows per hash value.

See the following topics for detailed performance guidelines for specifying compression, I/O, CPU usage, and storage operations, and collecting statistics for column-partitioned tables and join indexes.

- “[Guidelines for Specifying Compression for Column-Partitioned Tables and Join Indexes](#)” on page 311
- “[Guidelines on Optimizing I/O Operations, CPU Usage, and Disk Space Usage for Column-Partitioned Tables and Join Indexes](#)” on page 312
- “[Guidelines for Collecting Statistics on Column-Partitioned Tables and Join Indexes](#)” on page 313

## Guidelines for Specifying Compression for Column-Partitioned Tables and Join Indexes

- If autocompression is effective for a column, put that column into a single-column partition even if it is not frequently accessed.
- If autocompression on the individual columns or subsets of columns is not effective, you should group columns into a column partition when those columns are always or frequently accessed together by queries or are infrequently accessed.
- If autocompression is not effective for a column partition, specify NO AUTO COMPRESS for the column partition (with COLUMN format) to avoid the overhead of checking for autocompression opportunities when there are none to exploit.

Autocompression is most effective for single-column partitions with COLUMN format, less so for multicolumn partitions (especially as the number of columns increases) with COLUMN format, but not effective for column partitions with ROW format.

- Specify multivalue compression, algorithmic compression, or both for known high-occurrence values for columns where compression can be effective. For example, if you know that the data for a column is limited to a few values, you should specify multi-value compression for those values.

Specify an appropriate algorithmic compression for Unicode columns that contain a substantial number of compressible characters. The ASCII script (U+0000 to U+007F) of Unicode is compressible with the UTF8 algorithm.

## Guidelines on Optimizing I/O Operations, CPU Usage, and Disk Space Usage for Column-Partitioned Tables and Join Indexes

- The primary intent of column partitioning is to reduce the number of I/O operations undertaken by a query workload. The following factors can all contribute to reducing the I/O required by query workloads on column-partitioned tables.
  - Column partition elimination
  - Row partition elimination for multilevel partitioned tables and join indexes
  - Highly selective query predicates

Other factors such as those from the following list can also play a role in reducing the number of I/O operations required to resolve a query.

- Autocompression
- Row header compression
- User-specified multivalue and algorithmic compression

Trading I/O for CPU might enhance the performance of many workloads on an I/O-bound system.

See “Comparing the Number of I/O Operations Required to Answer the Same SELECT Request” on page 313 for more information about I/O operations and column-partitioned tables.

- A secondary intent of column partitioning is to reduce the amount of disk space consumed by table and join index storage. This is particularly effective when Teradata Database can apply row header compression and autocompression to column-partitioned table and join index data.
- Although column partitioning is designed to reduce the number of I/O operations required to process workloads, it is not intended to reduce the CPU usage of queries on column-partitioned tables.

While there are cases where CPU usage decreases for queries made on a column-partitioned table, CPU usage can also increase for some functions such as INSERT operations undertaken on a column-partitioned table.

For a CPU bound system, column partitioning might not provide any benefit, and might even degrade performance. An exception is the case where a subset of the workload that is I/O bound, even if overall the system is CPU bound, in which case column partitioning could be beneficial. Experiment with running your CPU-bound workloads against both nonpartitioned tables and column-partitioned tables to determine what the differences are.

## Guidelines for Collecting Statistics on Column-Partitioned Tables and Join Indexes

- Collect statistics regularly on the columns and indexes of a column-partitioned table just as you would for any other tables.
- Always collect statistics on the system-derived PARTITION column.

## Comparing the Number of I/O Operations Required to Answer the Same SELECT Request

The purpose of this topic is to compare the number of I/O operations required to perform the identical query against a nonpartitioned primary-indexed table, a PPI table, a single-level column-partitioned table, and a multilevel column-partitioned table.

Each of the tables examined contains the same 4 million rows of data and each is probed using the identical SELECT request. The only difference is the configuration of the table containing the 4 million rows.

The documented I/O figures are only approximations to demonstrate the relative differences in I/O operations for the same SELECT request executed on the same table, the only difference among the table versions being whether they are indexed or not and the type of partitioning used, if any.

The first table, *io\_pi*, has a nonpartitioned primary index and has the following definition.

```
CREATE TABLE io_pi (
    a INTEGER,
    b INTEGER,
    c CHARACTER(100),
    d INTEGER,
    e INTEGER,
    f INTEGER,
    g INTEGER,
    h INTEGER,
    i INTEGER,
    j INTEGER,
    k INTEGER,
    l INTEGER)
PRIMARY INDEX (a)
```

You can see the general organization of the table from a few sampled rows.

a	b	c	d	e	f	g	h	i	j	k	l
1	5	a	3	9	9	4	6	2	7	4	5
2	9	q	5	4	6	3	8	5	1	1	2
3	1	d	1	1	3	3	4	7	8	2	9
4	8	m	7	3	9	4	1	4	2	8	6
5	3	f	2	2	4	7	3	1	5	7	2
6	6	r	1	8	2	8	3	4	2	5	1

a	b	c	d	e	f	g	h	i	j	k	l
7	2	e	0	5	1	6	4	3	9	9	7
8	4	u	9	0	1	2	7	6	6	0	3
9	2	d	3	7	5	1	2	6	3	3	8
...	...	...	...	...	...	...	...	...	...	...	...

Now submit the following SQL request, which reads all 4 million rows in *io\_pi*.

```
SELECT SUM(f)
FROM io_pi
WHERE b BETWEEN 4 AND 7;
```

To read the required rows, Teradata Database used 9,987 I/O operations.

The next table is a partitioned primary index table with the following definition.

```
CREATE TABLE io_ppi (
    a INTEGER,
    b INTEGER,
    c CHARACTER(100),
    d INTEGER,
    e INTEGER,
    f INTEGER,
    g INTEGER,
    h INTEGER,
    i INTEGER,
    j INTEGER,
    k INTEGER,
    l INTEGER)
PRIMARY INDEX (a)
PARTITION BY RANGE_N(b BETWEEN 1
                      AND      9
                      EACH     1));
```

After loading *io\_ppi* with the same 4 million rows, you submit the same request against *io\_ppi* that you had earlier submitted against *io\_pi*.

Now submit the following SQL request, which reads only 4 row partitions in *io\_ppi*.

```
SELECT SUM(f)
FROM io_ppi
WHERE b BETWEEN 4 AND 7;
```

Teradata Database must only read 4 row partitions, but still took 4,529 I/O operations to read those four partitions.

The next table is a single-level column-partitioned table with the following definition.

```
CREATE TABLE io_cp_col_nopi (
    a INTEGER,
    b INTEGER,
    c CHARACTER(100),
    d INTEGER,
    e INTEGER,
    f INTEGER,
    g INTEGER,
```

```

    h INTEGER,
    i INTEGER,
    j INTEGER,
    k INTEGER,
    l INTEGER)
PARTITION BY COLUMN;

```

Now submit the following request, which reads only 2 column partitions in *io\_cp\_nopi*.

```

SELECT SUM(f)
FROM io_cp_col_nopi
WHERE b BETWEEN 4 AND 7;

```

Teradata Database must read only 2 column partitions, but still took 281 I/O operations to read those 2 partitions.

The next table is a multilevel column-partitioned table with the following definition.

```

CREATE TABLE io_cp_rowcol_nopi (
    a INTEGER,
    b INTEGER,
    c CHARACTER(100),
    d INTEGER,
    e INTEGER,
    f INTEGER,
    g INTEGER,
    h INTEGER,
    i INTEGER,
    j INTEGER,
    k INTEGER,
    l INTEGER)
PARTITION BY (COLUMN,
    RANGE_N(b BETWEEN 1
        AND      9
        EACH     1));

```

Now submit the following SQL request, which reads only 4 row partitions of 2 column partitions in *io\_cp\_rowcol\_nopi*.

```

SELECT SUM(f)
FROM io_cp_rowcol_nopi
WHERE b BETWEEN 4 AND 7;

```

Teradata Database only needed to perform 171 I/O operations to return the result set for this request.

Summarizing this example set, each table in the set contains 4 million rows, and each must solve the identical SELECT request. The following table summarizes the number of I/O operations each table required to return the requested answer set.

Partitioning Used	Number of I/O Operations Required
None.	9,987
Table has a nonpartitioned primary index.	
PRIMARY INDEX	

Partitioning Used	Number of I/O Operations Required
Table has a single-level partitioned primary index. <pre>PRIMARY INDEX PARTITION BY RANGE_N(b BETWEEN 1                       AND      9                       EACH     1)</pre>	4,529
Table has no primary index and single-level column partitioning. <pre>PARTITION BY COLUMN</pre>	281
Table has a no primary index and multilevel column partitioning. <pre>PARTITION BY (COLUMN,                RANGE_N(b BETWEEN 1                            AND      9                            EACH     1))</pre>	171

The conclusion to draw from the data in these examples is that column and row partition elimination is an effective performance optimization technique, and the more partitions that can be eliminated when responding to a request, the better.

For this particular query workload, the multilevel column-partitioned table requires the fewest I/O operations. You should not interpret this to mean that multilevel column partitioning is a universally optimal performance enhancement strategy.

## Locks and Concurrency When Accessing a Column-Partitioned Table

There is normally only one hash bucket per AMP that is used for the rows in a NoPI table or column-partitioned table. Teradata Database selects this hash bucket from the NoPI hash map that the AMP owns. The first row going into the NoPI table or column-partitioned table (or combined partition) begins with a uniqueness value of 1. The uniqueness value increases as more rows are inserted into the table.

The Reconfig and Restore/Copy client utilities can increase the number of hash buckets per AMP in a NoPI or column-partitioned table, but the number of hash buckets is generally low. A NoPI or column-partitioned table can be viewed as a very highly nonunique NUPI table in term of hash buckets. Because of this property, you need to be aware that a row-hash lock on a NoPI or column-partitioned table on an AMP can lock many, and usually all, of the rows in that table on that AMP.

Single-row INSERT requests and multirow INSERT requests such as Teradata Parallel Data Pump array INSERT operations use row-hash locks that can block any other reader and writer of the table until the request completes.

You can specify a LOCKING FOR ACCESS request modifier for a reader to avoid a lock conflict with a writer. For multiple writers into the same NoPI table or column-partitioned table, the Lock Manager grants the lock to one writer at a time. Because of this, you should not have multiple writers running concurrently on the same AMP. This is something that can

happen, for example, when the number of sessions in a Teradata Parallel Data Pump array INSERT job is greater than the number of AMPs in your system.

## Selecting Rows From a Column-Partitioned Table

Without an index being defined on a column-partitioned table, any SELECT request on the table results in at least one full-column partition scan up to a full-table scan, which can be expensive if there are not at least as many available column partition contexts as column partitions that need to be accessed.

Because you can define USIs, NUSIs, and join indexes on a column-partitioned table, you can implement indexing methods that are appropriate to facilitate the performance of your applications. Compared to queries that are unable to use a primary index for access even if a table has a primary index, column-partitioning a table can facilitate far better performance for queries on that table over a full-table scan when not all of the column partitions need to be accessed.

Because Teradata Database supports fallback for column-partitioned tables, you can view a column-partitioned table that has fallback when there are down AMPs on your system.

Consider the following when selecting data from a column-partitioned table.

- Without a primary index, there is no single-AMP, primary index access path; therefore, at least one full-column partition scan must be performed on all AMPs unless the Optimizer chooses a secondary index or join index access path.

The best practice is that if you need a high transaction rate for queries run against a table, a column-partitioned table without indexes is probably not a good choice.

- Secondary indexes are valid for column-partitioned tables, and secondary index access paths for SELECT requests work the same for a column-partitioned table as they do for a primary-indexed table.

Retrieving the needed column partition values can be more expensive than retrieving a row using a secondary index. For example, if  $n$  column partition values are needed for a query result set, the number of I/O operations needed to retrieve those rows could be  $n$  times as many as would be required for a primary-indexed table, and even higher if the primary-indexed table can take better advantage of the FSG cache.

- You should design the queries you intend to run on your column-partitioned tables to have one to a few predicates that are very selective in combination, and the number of available column partition contexts should be at least equal to the number of accessed column partitions.
- Avoid using unselective queries that retrieve most or all the columns from a column-partitioned table.

For example, the following SELECT request can be very expensive if there are not enough available column partition contexts for *cp\_table* and there is not an alternative access method such as a join index without column partitioning:

```
SELECT *
FROM cp_table;
```

## Joins With a Column-Partitioned Table, Join Index, or Spool

Teradata Database can directly access a column-partitioned table, join index, or spool using a dynamic hash join or product join. This means the column-partitioned object must be joined to a duplicated spool, and that spool must be relatively small for the join to be efficient.

The system can also directly access a column-partitioned database object using a RowID join. In this case, the rowIDs must come from a secondary index or from a join index if the table is column-partitioned, on the column-partitioned object, or from a previous retrieve or join to the column-partitioned object.

Other joins methods are possible, but to use those methods, Teradata Database must first construct the selected rows from the column partitions and then spool them, possibly with a redistribution operation and local AMP sort or duplication to all AMPs. This might be a reasonable plan if only a few rows are selected or if only a few columns are needed from the column-partitioned object.

If a join index is applicable, the Optimizer can make use of the index without actually having to join the tables.

## Bulk Loading a Column-Partitioned Table

The expected method of populating a column-partitioned table is an INSERT ... SELECT request from one or more source tables.

If the data is from an external source, you can use FastLoad to load the data to a staging table and then populate the column-partitioned table with an INSERT ... SELECT request. You can also populate a column-partitioned table from an external source using a Teradata Parallel Data Pump array INSERT to insert data into the column-partitioned table. However, Teradata Parallel Data Pump is not expected to be as efficient as a FastLoad and INSERT ... SELECT request.

You should rarely update column-partitioned tables using single-row INSERT requests because such requests can cause a large degradation in performance by needing to append a column partition value to each of the column partitions.

Inserting data into a column-partitioned table is expected to gain some efficiency over a primary indexed-table because the data is just appended to the end of a table or row partition, and the rows are not required to be in any particular order. However, this can be negatively offset for a column-partitioned table by the need to transform rows into columns.

Bulk inserts using an INSERT ... SELECT request or Teradata Parallel Data Pump array INSERT can minimize this impact for column-partitioned tables because they can apply the transformation of multiple rows to columns as a set instead of individually. Teradata Parallel Data Pump array inserts can group as many data rows as are allowed in an AMP step because the rows are not required to be sent to a specific AMP.

The transformation from rows to columns performs better if the number of column partitions (not including the delete column partition) does not exceed the number of available column partition contexts; otherwise, Teradata Database must make additional scans of the source data.

For INSERT ... SELECT (without a HASH BY clause) requests into a NoPI or column-partitioned target table, Teradata Database does not redistribute data from the source table, but instead locally appends it to the target NoPI table. Keep in mind that this results in skew in the target table if the source is skewed. The source can be skewed either after a series of joins or after applying single-table predicates. You can use the HASH BY option to redistribute the rows to a NoPI table or column-partitioned table to avoid skewing by specifying RANDOM or by choosing good expressions to hash on.

Fallback and index maintenance for a column-partitioned table are done the same way as fallback and index maintenance for a primary-indexed table.

The following list of considerations for inserting data into a NoPI table apply equally well to a column-partitioned table.

- Because they do not have a primary index, Teradata Database does not hash rows based on any column in a NoPI table. However, the system still generates a rowID for each row in a NoPI table. The process is that Teradata Database selects a hash bucket from the NoPI table hash map that an AMP owns. It then uses that hash bucket to generate a rowID. This strategy helps make fallback and index maintenance comparable to the maintenance on a primary-indexed table.
- For single-statement INSERT requests, multistatement INSERT requests, and array INSERT operations into a NoPI table, Teradata Database sends the rows to the AMPS through a random generator. This random generator is designed in such a way that for a new request, data is generally sent to a different AMP from the one to which the previous request sent data. The concept is to balance the data among the AMPS as much as possible without the use of a primary index.
- For INSERT ... SELECT requests into a column-partitioned or nonpartitioned NoPI target table, the SELECT component of the request can be either a simple SELECT (retrieving all data from the source table) or a complex SELECT (retrieving data from one or more source tables). Spooling the source table before inserting the new rows can be avoided for a simple SELECT in some cases.
- Teradata Database can send data to any AMP in complete blocks prior to inserting the rows into a target column-partitioned table. No data redistribution is required for each individual row. This is particularly beneficial for Teradata Parallel Data Pump array INSERT.
- The row hash for each row in a column-partitioned table is internally controlled and generated, so rows are always appended at the end of the table (if the table does not also have row partitioning) or combined partition (if the table also has row partitioning), and never inserted in a middle of a row hash. The result is that the rows need not be sorted if the column-partitioned table does not have row partitioning.

There are several performance advantages for bulk INSERT operations.

- Inserting data into a column-partitioned table is somewhat more efficient than inserting into a primary-indexed table because the data is appended to the end of a table or row partition and the rows are not required to be in any particular order.

This can be negatively offset for a column-partitioned table by the need to transform from rows to columns.

Bulk insert operations using an INSERT ... SELECT request or Teradata Parallel Data Pump array INSERT operation can minimize this negative offset for column-partitioned tables because they can apply the transformation of multiple rows to columns as a set instead of individually. Teradata Parallel Data Pump array INSERT operations can group as many rows as allowed in an AMP step because the rows are not required to be sent to a specific AMP.

The transformation from rows to columns performs better if the number of column partitions (not including the DELETE column partition) does not exceed the number of available column partition contexts; otherwise, Teradata Database must perform additional scans of the source data.

- If there is no row partitioning for the table, there is no need to sort the data.

Although Teradata Parallel Data Pump array INSERT operations do not have an explicit sort phase of rows in the array that is being inserted into a primary-indexed table, the file system does an implicit memory-resident sort of the data. This sort work is not done when data is appended to the end of the table for a nonpartitioned NoPI or column-partitioned table that does not have row partitioning.

- For INSERT ... SELECT operations that do not specify a HASH BY clause into NoPI and column-partitioned target tables, Teradata Database does not redistribute the data from the source table. Instead, Teradata Database appends the data locally into the target table.

This causes skew in the target table if the source is skewed, which can happen either after a series of joins or after applying single-table predicates. You can specify the HASH BY option to redistribute the rows to a NoPI or column-partitioned table to avoid skewing by specifying RANDOM or by specifying good hashing expressions.

- Data can be sent to and stored on any AMP.

Another performance advantage is that there is no requirement that a NoPI or column-partitioned table row must be stored on any particular AMP. This is very useful for Teradata Parallel Data Pump array INSERT operations because data sent in a buffer from Teradata Parallel Data Pump can all be combined into the same step going to the same AMP for insertion.

In contrast to this, Teradata Database generally splits the into multiple steps targeting the destination AMP for primary-indexed table data. The performance impact increases as the number of AMPs in the system increases. With fewer steps to process, the CPU and I/O burdens on the system are both reduced.

For single-row INSERT operations, the performance for a NoPI table compared to a primary-indexed table is not significantly different. Appending a row to a NoPI table is somewhat more efficient than inserting a row into a primary-indexed table in the middle of a hash value, but the end of transaction processing, including the flushing of the WAL log, remains the same. Column-partitioned tables impose the additional performance burden of transforming the row into the column partitions.

There is a potential disadvantage if the distribution of inserted rows to the AMPs is skewed. In most cases, the method of inserting data into both NoPI tables and column-partitioned tables leads to a fairly even distribution of the rows. If the distribution of rows to the AMPs is skewed, you can populate the table using the HASH BY option with

an INSERT ... SELECT request (see *SQL Data Manipulation Language* for information about the HASH BY option).

Teradata Database handles an INSERT ... SELECT request in one of the following ways depending on the source table, the target table, and the number of available column partition contexts.

IF there are ...	THEN Teradata Database reads the source table rows ...
sufficient column partition contexts available for the target column-partitioned table	a block-at-a-time after they are spooled (if that is required), and then it builds, for each row in the block and then for each column partition, a column partition value and appends it to the last container (if there is no remaining container, Teradata Database starts a new one) of the corresponding internal partition or, if the column partition has ROW format, the subrow is written out.
insufficient available column partition contexts available for the target column-partitioned table	for one set of column partitions and then reads again for another column partition until all the column partition values are appended.  This eliminates multiple writes of the last container of internal partitions at the cost of multiple reads of the source, but reading is much less expensive than writing.

## Setting the Optimal Teradata Parallel Data Pump SERIALIZE Option

Setting the SERIALIZE option of the Teradata Parallel Data Pump utility to ON in the BEGIN LOAD statement serves two purposes.

- To order the application of data.

The SERIALIZE option applies rows in the order that they occur in the input data source. You do this by using the KEY option to specify the primary index of the table to force rows with the same primary index value to go into the same session.

- To avoid hash lock contention.

The SERIALIZE option forces rows with the same primary index value to go into the same session, which reduces hash lock contention among multiple sessions.

The SERIALIZE option is mostly important for NUPI tables, especially with highly non-unique data. There is some additional CPU for the Teradata client when SERIALIZE is set to ON.

For a NoPI table or a column-partitioned table, the traditional hash lock contention issue no longer applies because the table does not have a primary index. If the order of data application is not important to you, you should set SERIALIZE to OFF.

For a NoPI table or column-partitioned table, Teradata Database generally uses one hash value for as many rows as there are that fill up the 44-bit uniqueness values for a combined row partition. A row hash lock on a NoPI table or column-partitioned table usually locks all the rows on an AMP because it is frequently true that the rows on an AMP all have the same row hash value in their rowID. Typically, multiple Teradata Parallel Data Pump sessions running on the same AMP on the same NoPI table or column-partitioned table block one other.

Therefore, you should keep the number of Teradata Parallel Data Pump sessions to the number of AMPs in your system.

## Maintenance Cost for Column-Partitioned Tables and Join Indexes

Although rows in a NoPI tables and column-partitioned tables are not hashed by a primary index, they are all assigned a unique row identifier that includes an internally generated hash bucket number. This enables fallback and index maintenance on nonpartitioned NoPI and column-partitioned tables to work very much the same as they do for primary-indexed tables, so the maintenance cost for fallback and indexes for the two table types is usually comparable.

There might be some degradation for index maintenance if there are insufficient available column partition contexts for collecting index values from multiple column partitions.

Deleting or updating a row in a column-partitioned table requires at least one column partition scan of all row partitions that are not eliminated if there is row partitioning, but there is no applicable secondary index defined for the table.

A column-partitioned table is not intended to undergo a significant amount of maintenance after being initially populated. If there is a need for a significant amount of maintenance, you should verify that performance is acceptable, and if it is not, you should not use a column-partitioned table for the workload.

## Column-Partitioned Table as a Source Table in a Data Moving Request

Column-partitioned tables are not usually expected to be used as source tables; however, there might be situations where a column-partitioned table is the source for an insert into a target primary-indexed table.

- Target primary-indexed table.

Operations such as INSERT ... SELECT, MERGE, UPDATE ... FROM, and DELETE ... FROM for a target PI table can run slower when the source is a column-partitioned table compared to the case when a source primary-indexed table with the same primary index as the target.

This is because with a column-partitioned source table, selected rows must be constructed from the referenced columns of the column partitions, redistributed, and sorted locally on the AMPs. Copying data from one primary-indexed table to another primary-indexed table with a different primary index or different partitioning requires redistribution and AMP-local sorting.

- Target NoPI table.

Operations such as INSERT ... SELECT, MERGE, UPDATE ... FROM, and DELETE ... FROM for a target NoPI table can run slower when the source is a column-partitioned table compared to the case when the source table is either a nonpartitioned NoPI table or a primary-indexed table.

This is because with a column-partitioned source table, selected rows must be reconstructed from the referenced columns of the column partitions.

- Target column-partitioned table.

Operations such as INSERT ... SELECT, MERGE, UPDATE ... FROM, and DELETE ... FROM for a target column-partitioned table can run slower when the source is a column-partitioned table compared to the case when the source table is either a NoPI or a primary-indexed table.

This is because with a source column-partitioned table, selected rows might need to be constructed from the referenced columns of the column partitions and often require spooling the source column-partitioned table to reconstruct rows followed by reading the rows and converting to the column partitioning of the target column-partitioned table.

In some cases when the source and target have the same column definitions and partitioning, spooling can be avoided by a direct copy of the source table to the target table based on physical rows.

In some cases, even if the tables have different column partitioning and there are sufficient available column-partition contexts, spooling can be avoided.

## Deleting Rows From a Column-Partitioned Table

Consider the following information before deleting data from a column-partitioned table.

- A DELETE ALL request or an unconstrained DELETE request takes the fastpath DELETE for any table. If specified within an explicit transaction in Teradata session mode, the DELETE request must be the last statement of the last request of the transaction.  
Similarly, if the column-partitioned table is also row-partitioned, Teradata Database can do a fastpath DELETE. For these cases, Teradata Database recovers the space that had been used by the deleted rows.

A *fastpath* optimization is one that can be performed faster if certain conditions are met. For example, in some circumstances DELETE and INSERT operations can be performed faster if they can avoid reading the data blocks and avoid transient journaling.

- For all other cases, a DELETE request uses a scan or an index on a column-partitioned table. In this case, the rows are marked as deleted in the delete column partition *without recovering the space*, but both LOB space and index space *is* recovered.

Because of this, you should only delete rows in this manner for a relatively small number of rows and you should use the form of DELETE request described in the previous bullet to delete large amounts of rows.

The space is recovered from the column-partitioned table when all the rows are deleted at the end of a transaction or when the entire row partition that contains the deleted rows is deleted at the end of a transaction.

## Updating a Column-Partitioned Table

Consider the following information before updating data in a column-partitioned table.

- An UPDATE request uses a scan, an index, or a rowID spool to access a column-partitioned table and select the qualifying rows for the update.
- An UPDATE request is processed in the following way.

- a Selects the rows to be updated.
- b Transforms columns to rows.
- c Deletes the old row without recovering the space and marks its delete bit in the delete column partition.

**Note:** Both LOB space and index space *is* recovered.

- d Reinserts the updated rows, transforming them from rows to columns and appending the column values to their corresponding combined partitions.

Teradata Database recovers the space from the column-partitioned table when it deletes all of the rows at the end of a transaction or when it deletes the entire row partition that contains the deleted rows at the end of a transaction.

- Teradata Database also updates the columns in the column-partitioned table that are used in a secondary or join index.

## Inserting Rows Into a Column-Partitioned Table Using Teradata Parallel Data Pump

Teradata Parallel Data Pump array INSERT operations use a specialized SQL client-server protocol that enables multiple data parcels to be passed in a single request. When Teradata Database detects the presence of multiple data parcels in a request, it iterates the execution of that request for each data parcel by binding its user data references on each iteration to the data record passed in the corresponding data parcel.

There are two main performance benefits from using Teradata Parallel Data Pump array INSERT operations.

- Request text size

The request text in an iterated request such as a Teradata Parallel Data Pump array INSERT request specifies one single instance of an INSERT statement, so it is independent of the number of times the INSERT operation is iterated.

When you preface a DML request with a USING request modifier, it only specifies the columns that are referenced by a single iteration. Therefore, the size of the request text for an iterated request can be greatly reduced when compared to an equivalent multistatement INSERT request. The smaller request text reduces the cost of building the request from the client and sending it to the server. The smaller request text also allows more data parcels per request and requires less cache space in the server and helps reduce the cost of searching the cache.

- Multirow INSERT

For a Teradata Parallel Data Pump array INSERT request, Teradata Database can group multiple rows together by AMP ownership into a single AMP step. The processing of the rows is optimized by performing the inserts in the same step instead of multiple steps. This optimizes the following operations.

- Reduces the CPU path because fewer steps are generated and sent or received between the PE and the AMP.
- Reduces I/O because more rows can be inserted with one single I/O call.

The performance impact is directly dependent on the number of rows that can be grouped together in the same AMP step, which depends on the following factors.

- The total number of data records in the request

A higher PACK factor causes there to be more rows in a request, which in turn leads to an increase in the average number of rows in the same step.

- The system configuration, particularly the number of AMPS

Smaller systems with fewer AMPS increase the average number of rows in the same step with the same PACK factor. As the number of AMPS increases, the average goes down.

- The clustering of inserts with same NUPI values

Inserts into NUPI table with many duplicate NUPI values make the average increase because rows with the same NUPI value all go to the same AMP.

Because rows in both NoPI tables and column-partitioned tables are not hashed based on a primary index, they can be dispatched to any AMP as desired. To maximize this benefit, a Teradata Parallel Data Pump array INSERT on a nonpartitioned NoPI table or a column-partitioned table always packs as many rows as there are in the request from the client into the same AMP step. These rows are sent to the same AMP in one INSERT step. This is independent of the system configuration and the clustering of data.

The sending of the rows in a Teradata Parallel Data Pump array INSERT job that are to be inserted into either a NoPI table or a column-partitioned table uses a random generator that determines the destination AMP for each INSERT step. The random generator is designed to choose a different AMP from the one that the previous request sent data to ensure that data is spread across the AMPS as equally as it can be to avoid skew.

With a constant PACK factor, the number of rows that are packed into the same AMP step on a primary-indexed table depends on the number of AMPS in the system and how the data is clustered. As the number of AMPS increases and the clustering of data decreases, the number of rows that are packed into the same AMP step decreases.

FOR a table with this kind of primary index ...	Data clustering ...
unique	is generally minimal.
non-unique	can be high, in which case the number of rows that are packed into the same AMP step is also high and is independent of the number of AMPS in the system.

Therefore, a Teradata Parallel Data Pump array INSERT has the most benefit when comparing performance on NoPI or column-partitioned tables with a UPI table on large systems with many AMPS. A Teradata Parallel Data Pump array INSERT does not have as much benefit when comparing performance on a NoPI or column-partitioned table against the performance of a NUPI table with high data clustering.

IF a column-partitioned table has ...	THEN ...
row partitioning	<ol style="list-style-type: none"> <li>1 the rows to be inserted must be sorted in memory.</li> <li>2 the rows must be split into column partition values that are appended to the end of the internal combined partitions to which they belong.</li> </ol>
does not have row partitioning	<p>any improvement in performance can be offset by the processing that is required to do the following.</p> <ol style="list-style-type: none"> <li>1 Split the rows to be inserted into column partition values.</li> <li>2 Append those values to the ends of the corresponding column partitions.</li> </ol>

The performance of transforming rows into columns and appending column partition values is affected by the following factors.

- Whether there are at least as many available column partition contexts as column partitions.  
 Remember that the Optimizer uses the setting of the DBS Control parameter PPICacheThrP (or the cost profile constant PPICacheThrP field) to determine the number of available column partition contexts that can be used at one time to append column partition values to their column partitions.
- The number of column partition values inserted into a combined partition at one time.  
 The more column partition values inserted, the better.

## Effect of Skewed Data on Column-Partitioned Tables

For a primary-indexed table, Teradata Database hashes each row based on the value of its primary index. The hash value determines to which AMP the row is dispatched and stored. However, for a NoPI table or column-partitioned table, Teradata Database sends rows to the AMPs randomly, using single-row INSERT requests, array INSERT requests, and Teradata Parallel Data Pump requests.

This randomization typically is not a problem for data skewing, and the data nearly always balances its distribution of rows among the AMPs. However, there are several operations on NoPI tables and column-partitioned tables that can lead to data skewing.

- **INSERT ... SELECT** requests into a NoPI target table or column-partitioned target table  
 When the target table of an **INSERT ... SELECT** is a column-partitioned/NoPI table, data coming from the source table, whether it is directly from the source table or from an intermediate spool, is locally inserted into the target column-partitioned/NoPI table. Performance wise, this is very efficient since it avoids a redistribution and sort. But if the source table or the resulting spool is skewed, the target column-partitioned/NoPI table is also skewed; in this case, a HASH BY clause can be used to redistribute the data from the source before the local copy.

For the expressions to hash on, consider expressions that provide good distribution and, if appropriate, improve the effectiveness of autocompression for the insertion of rows back

into the column-partitioned table. Alternatively, use a HASH BY RANDOM clause for a good distribution if there is not a clear choice for the expressions to hash on and for more efficient redistribution.

When inserting into a column-partitioned table, also consider use of an LOCAL ORDER BY clause to improve the effectiveness of autocompression.

- Down AMP Recovery Processing

When an AMP in a cluster goes down, Teradata Database reroutes the data dispatched to the down AMP for insertion to a fallback AMP instead.

When the down AMP comes back online, recovery begins and copies the data from the fallback AMPS for the down AMP that it missed while it was down. The process synchronizes data on all of the AMPS before you can open the table for more work.

If a cluster has more than 2 AMPS, there are multiple fallback AMPS for each AMP in the cluster.

For a NoPI or column-partitioned table, Teradata Database sends the data randomly to the AMPS for insertion with a randomly-generated hash value. This hash value is not the hash value that is stored in the row. When an AMP is down in a cluster, multiple fallback AMPS can receive data on behalf of the down AMP. Because there is normally one hash bucket on an AMP that is used for all of the rows in a NoPI or column-partitioned table, when the down AMP comes back online, the rows from the fallback AMPS have the same hash bucket and would create a skewing problem if they were sent to the previously down AMP as part of the recovery.

Instead, there is only one fallback AMP for each AMP in a cluster for NoPI and column-partitioned tables.

Multiple AMPS can still receive rows on behalf of a down AMP, but only one AMP that puts the data of the down AMP into a fallback data subtable, and that is the data goes to the down AMP when it comes back up.

Other AMPS put the data in their primary data subtable. This can be a problem when there are more than 2 AMPS in a cluster and there is a down AMP while a lot of data is inserted into a nonpartitioned or column-partitioned table, and the data in that NoPI or column-partitioned table can be skewed when the down AMP comes back online. 2-AMP clusters do not have this problem.

If a column-partitioned or NoPI table becomes skewed and that table is populated from other source tables, one option to handle the problem is to delete all the rows in the table and then repopulate the table from its source tables using an INSERT ... SELECT request with a HASH BY clause as needed to provide good distribution.

If this option is not available to you, you can use an INSERT ... SELECT request with a HASH BY clause to move the data from the skewed column-partitioned or table into another column-partitioned or NoPI table and then drop the original column-partitioned or NoPI table.

When you insert rows into a column-partitioned table, you should always consider specifying a LOCAL ORDER BY clause to improve the effectiveness of autocompression.

- Reconfiguration Server Utility

The Reconfiguration utility has a similar issue to that of the Restore and Copy client utilities when data is moved to a different configuration, particularly a configuration with more AMPs, which is normally the case when you expand your system.

Although rows in a column-partitioned or NoPI table are not hashed based on their primary index values, and the AMPs on which the rows reside are determined arbitrarily, each row does have a rowID with a hash bucket that is owned by the AMP storing that row.

Teradata Database redistributes the rows in a column-partitioned or NoPI table using the Reconfiguration utility by sending each row to the AMP that owns the hash bucket in that row based on the new configuration map. As is true for the Restore and Copy utilities, Reconfiguration can skew the distribution of rows for a column-partitioned or nonpartitioned NoPI table by relocating them on a different configuration. And the more AMPs there are in a configuration, the fewer hash buckets there are to go around for the table.

It is also possible that multiple hash buckets can be redistributed to the same AMP. With fewer AMPs, it is guaranteed that some AMPs have multiple hash buckets. Because of this, significantly more space can be required for a column-partitioned or NoPI table than is required for a comparable primary-indexed table.

To avoid this problem, you can use an `INSERT ... SELECT` request to move the column-partitioned or NoPI table rows into a primary-indexed table before the Reconfiguration and delete all the rows from the column-partitioned or NoPI table.

After the Reconfiguration completes, you can use an `INSERT ... SELECT` request to move the rows from the primary-indexed table back into the column-partitioned or NoPI table. For the primary index of the primary-indexed table, consider using one that provides good distribution and, if appropriate, improves the effectiveness of autocompression for the insert back into the column-partitioned or NoPI table.

If the column-partitioned or NoPI table is populated from other source tables, you can also delete all the rows from the column-partitioned or NoPI table and then after the Reconfiguration completes, repopulate the column-partitioned or NoPI table from the source tables using an `INSERT ... SELECT` request with a `HASH BY` clause as needed to provide good distribution.

If you do neither, and the row distribution for the column-partitioned or NoPI table is skewed after the Reconfiguration, use an `INSERT ... SELECT` request with a `HASH BY` clause from the skewed column-partitioned or NoPI table into another column-partitioned or NoPI table and then drop the original column-partitioned or NoPI table.

When inserting rows into a column-partitioned table, always consider using a `LOCAL ORDER BY` clause to improve the effectiveness of autocompression.

- **Restore and Copy Client Utilities**

When the AMP configuration, the hash function of the source system, and the target system are part of the same restore or copy operation, there is no data redistribution involved. As a result, the rows that were originally stored on an AMP are restored or copied to that same AMP.

When this happens, the data demographics are identical between the source and target systems. The data demographics for a primary-indexed table can change when the AMP

configuration or the hash function of the source system and the target system is different. This is because Teradata Database rehashes the data in this case, and you restore or copy it to a different AMP on the target system from that on the source system.

For a column-partitioned or NoPI table, there is no rehash even if the hash function is different between the target and source systems because there is no primary index on which to do a rehash. The hash bucket in a row coming from the source system does not change when it is stored on the target system, so as long as there is no change in AMP configuration, rows in a column-partitioned or NoPI table are restored or copied to the same AMP as they were on for the source system.

A problem can occur when there is a change in AMP configuration. Because there is normally one hash bucket used on an AMP for a column-partitioned or NoPI table, when the target system has more AMPs than the source system there are not enough hash buckets in the table to go to all of the AMPs, so some AMPs do not have any data after a restore or copy operation. This skews the column-partitioned or NoPI table rows and can affect performance negatively. Furthermore, multiple hash buckets go to the same AMPs as the result of a restore or copy operation when the target system has fewer AMPs than the source system. If the restore or copy operation is to a target system with fewer AMPs than the source system, this might or might not happen. The result is that much more space could be required for a column-partitioned or NoPI table than a primary-indexed table.

To avoid this problem, you can use an `INSERT ... SELECT` request to move the column-partitioned or NoPI table rows into a primary-indexed table for archive, and do *not* archive the column-partitioned or NoPI table. Then after restoring the rows, use an `INSERT ... SELECT` request to move the rows from the primary-indexed table into a column-partitioned or NoPI table.

Consider using a primary index that provides good row distribution for the primary-indexed table and consider a primary index that provides good distribution and, if appropriate, improves the effectiveness of autocompression for the insertion back into the column-partitioned or NoPI table.

If the column-partitioned or NoPI table is populated from other source tables, you can also archive the source tables (*not* the column-partitioned or NoPI table) and then after restoring it, populate the column-partitioned or NoPI table from the source tables using an `INSERT ... SELECT` request with a `HASH BY` clause as needed to provide good distribution.

If the column-partitioned or NoPI table is part of the archive, for example, if the archive were made prior to knowing system configuration or hash would be different, do *not* restore the column-partitioned or NoPI table, and after restoring the source tables, populate the column-partitioned or NoPI table.

If you do neither is done and the column-partitioned or NoPI table is restored in a skewed state, use an `INSERT ... SELECT` request with a `HASH BY` clause from the skewed column-partitioned or NoPI table into another column-partitioned or NoPI table and then drop the original column-partitioned or NoPI table.

When you insert rows into a column-partitioned table, you should always consider using a `LOCAL ORDER BY` clause to improve the effectiveness of autocompression.

## Operations and Utilities for Column-Partitioned Tables

The following tables provide a comprehensive list of the operations and utilities that you might consider using with column-partitioned tables. The tables include information such as whether the utility or operation is supported for column-partitioned tables, possible substitutions for utilities and operations that Teradata Database does not support for column-partitioned tables, and various usage information about the operation or utility as it applies to nonpartitioned NoPI tables.

The first table lists the SQL operations you might use for column-partitioned tables.

SQL Statement Name	Support for Column-Partitioned Tables	Usage Notes
DELETE	Supported	See “ <a href="#">Deleting Rows From a Column-Partitioned Table</a> ” on page 323.
INSERT	Supported	While INSERT operations into column-partitioned tables are supported, you should not use single-row INSERT requests to add rows to column-partitioned tables frequently because such requests can cause a large degradation in performance by needing to transform a row into a column and then appending a column partition value to each of the column partitions.  See “ <a href="#">Bulk Loading a Column-Partitioned Table</a> ” on page 318.
INSERT ... SELECT	Supported	See “ <a href="#">Bulk Loading a Column-Partitioned Table</a> ” on page 318.
MERGE	Not supported	The UPDATE component of a MERGE request is required to fully specify the primary index.  Because column-partitioned and nonpartitioned NoPI tables do not have a primary index, Teradata Database cannot support MERGE requests for NoPI tables.  Use UPDATE and INSERT requests instead.
UPDATE	Supported	See “ <a href="#">Updating a Column-Partitioned Table</a> ” on page 323.
UPDATE (Upsert Form)	Not supported	The UPDATE component of an Upsert request is required to fully specify the primary index.  Because column-partitioned and nonpartitioned NoPI tables do not have a primary index, Teradata Database cannot support UPSERT requests for NoPI tables.

The second table lists the Teradata Tools and Utilities operations you might use for and column-partitioned tables.

TTU Utility Name	Support for Column-Partitioned Tables	Usage Notes
CheckTable	Supported	<p>The LEVEL HASH check, which is done with either an explicit LEVEL HASH command or implicitly in a LEVEL THREE command, works differently on a primary-indexed table and a NoPI or column-partitioned table.</p> <ul style="list-style-type: none"> <li>For a primary-indexed table, the check regenerates the row hash for each data row based on the primary index values and then compares with the rows on disk.</li> <li>For NoPI tables and column-partitioned tables, the check looks at the row hash value for each data row and verifies that the hash bucket that is part of the row hash correctly belongs to the AMP.</li> </ul>
FastExport	Supported	You can export the data in NoPI tables and column-partitioned tables the same way you can export data rows for a primary-indexed table
FastLoad	Not supported	<p>Use INSERT ... SELECT or Teradata Parallel Data Pump requests instead. These are normally used to populate a column-partitioned table.</p> <p>You can also use FastLoad to load data into a staging table and then use an INSERT ... SELECT request to populate the column-partitioned table</p>
MultiLoad	Not supported	Use INSERT ... SELECT and Teradata Parallel Data Pump requests instead. These are normally used to populate a column-partitioned table.
Reconfiguration	Supported	<p>Reconfiguration processing for a NoPI table or column-partitioned table is similar to the Reconfiguration processing that is done for a primary-indexed table.</p> <p>The main difference is that a NoPI table or column-partitioned table normally has one hash bucket per AMP, which is like a very skewed non-unique NUPI table.</p> <p>Because of this, when you reconfigure a system to have more AMPS, there might be some AMPS that do not have any data for a NoPI table or column-partitioned table.</p> <p>As a result, Reconfiguration can cause data skewing for both NoPI and column-partitioned tables.</p>
Restore and Copy	Supported	<p>Restore and Copy processing for a NoPI or column-partitioned table are very similar to the processing used by those utilities for a primary-indexed table.</p> <p>The main difference is that a NoPI table or column-partitioned table normally has one hash bucket per AMP, which is like a very nonunique NUPI table.</p> <p>Because of this, when you restore or copy the data from a NoPI table to a different configuration that has more AMPS, there might be some AMPS that do not have any data.</p> <p>As a result, Restore and Copy can cause can cause data skewing for both NoPI and column-partitioned tables.</p>

TTU Utility Name	Support for Column-Partitioned Tables	Usage Notes
TableRebuild	Supported	<p>Table Rebuild processing for a NoPI or column-partitioned table is the same as the Table Rebuild processing for a primary-indexed table.</p> <p>The table must have fallback protection for Table Rebuild to be able to rebuild it. Rows in a NoPI table or column-partitioned table have a row hash value, so Teradata Database can rebuild them the same way it rebuilds rows for a primary-indexed table.</p>

## Storage and Other Overhead Considerations for Partitioning

### Implications for Storage of Partitioning

The storage implications of partitioned tables and join indexes are minimal.

- Each row in a partitioned table or join index has either 2 or 8 additional bytes in its row header to store its internal partition number.
- An empty partition occupies no space on disk, and a populated partition occupies only 2, 4, or 8 bytes more per row than the same rows would if they belonged to a nonpartitioned table because a partition is just an ordering of the rows on an AMP.
- There is no additional overhead accrued by inserting the first row into a partition.  
Another way to say this is that there is no action required to create a partition. Rows are just inserted in their partition/row hash/uniqueness order.
- There is no additional overhead accrued by deleting the last row of a partition.  
In other words, there is no additional action required to drop a partition. The last row of a partition is just deleted in the same way as any other row.

### Storage Considerations for Column-Partitioned Tables and Join Indexes

Column-partitioned tables and join indexes are always stored in packed64 format.

A table with column partitioning has similar space usage and guidelines as a table with only row partitioning except that you must take the impact of column compression or expansion into account.

If the flag byte that precedes the first presence byte for a physical row has the first high-order and third bits both not set, the physical row is in a nonpartitioned table or join index and does not contain an internal partition number because the internal partition number for any nonpartitioned database object it is assumed to be 0.

If the first bit is set and third bit is not set, the physical row is in a table or join index that has 2-byte partitioning and that physical row contains a 2-byte internal partition number.

If the first bit is not set and the third bit is set, the physical row is in a table or join index that has 8-byte partitioning and the physical row contains an 8-byte internal partition number.

The following table summarizes this information.

Flag Byte Settings for Partitioning and Size of Internal Partition Number		
First Bit	Third Bit	Description
Not set	Not set	Physical row is from a nonpartitioned table or join index.
Set	Not set	Physical row is from a partitioned table or join index that has 2-byte partitioning.
Not set	Set	Physical row is from a partitioned table or join index that has 8-byte partitioning.
Set	Set	Not used and reserved for future use.

Teradata Database stores the 2-byte or 8-byte internal partition number after the first presence byte of a physical row. Even though this information is stored after the hash value:uniqueness for a column-partitioned table or after the hash bucket:uniqueness for a nonpartitioned NoPI table, the file system orders physical rows according to their internal partition number first, followed by the hash value:uniqueness.

The first presence byte in a container is not used as a presence byte; instead, it is used to indicate information about autocompression for the container.

A column-partitioned table or join index can be much larger than an otherwise equivalent, but not column-partitioned, table or join index if there are few physical rows in populated combined partitions or there are many column partitions with ROW format and the subrows are narrow.

A container can only contain values of rows that have the same internal partition number and hash bucket value. The increased size occurs because of the increased number of physical rows and the overhead of the row header for each physical row.

However, a column-partitioned table or join index is usually *smaller* than a table or join index that is not column-partitioned if COLUMN format is used for narrow column partitions and the table or join index is not over-partitioned because of the reduced number of physical rows and autocompression.

## Column Partitions With COLUMN Format

A column partition with COLUMN format packs column partition values into a physical row, or container, up to a system-determined limit. The column partition values must be in the same combined partition to be packed into a container.

The row header occurs once for a container instead of there being a row header for each column partition value.

The format for the row header is either 14 or 20 bytes and consists of the following fields.

- Length
- rowID

The rowID of the first column partition value is the rowID of the container.

- Flag byte
- First presence byte

The rowID of a column partition value can be determined by its position within the container. If many column partition values can be packed into a container, this row header compression can greatly reduce the space needed for a column-partitioned object compared to the same object without column partitioning.

If Teradata Database can only place a few column partition values in a container because of their width, there can actually be an increase in the space needed for a column-partitioned object compared to the object without column partitioning. In this case, ROW format might be more appropriate.

If Teradata Database can only place a few column partition values because the row partitioning is such that only a few column partition values occur for each combined partition, there can be a very large increase in the space needed for a column-partitioned object compared to the object without column partitioning. In the worst case, the space required for the column-partitioned object can be as much as 24 times larger.

In this case, consider altering the row partitioning to allow for more column partition values per combined partition or removing column partitioning.

If the container has autocompression, 2 bytes are used as an offset to compression bits, 1 or more bytes indicate the autocompression types and their arguments, if any, for the container, 1 or bytes, depending on the number of column partition values, of autocompression bits, 0 or more bytes are used for a local value-list dictionary depending the autocompression type, and 0 or more bytes are used for present column partition values.

If the container does not have autocompression either because you specified NO AUTO COMPRESS for the column partition or because no autocompression types are applicable for the column partition values of the container, Teradata Database uses 0 or more bytes for column partition values.

The byte length of a container is rounded up to a multiple of 8.

The formats for single-column and multicolmum partitions with COLUMN format differ slightly, as listed by the following bullets.

- A single-column partition with COLUMN format consists only of single-column containers. Each container represents a series of column values.
- A multicolmum partition with COLUMN format consists of multicolmum containers. Each container represents a series of column partition values where the column partition value is made up of a set of values, one for each column in the partition.

See “[Row Structure for Containers \(COLUMN Format\)](#)” on page 759 for more information.

## Column Partitions With ROW Format

A physical row of a column partition with ROW format is called a subrow. Subrows have the same format as traditional Teradata Database rows, but only include the columns defined for the column partition. Note that currently there is no autocompression for subrows.

If Teradata Database uses ROW format for the column partitions of a column-partitioned object is such that many narrow column partition values occur, there can be a very large increase in the space needed for a column-partitioned object compared to the object without column partitioning. In the worst case, the space required for the column-partitioned object can be as much as 24 times larger.

Because of this, you should not specify ROW format for narrow column partitions. In the worst case, each column partition value can have a row header (14 or 20 bytes plus 6 or more bytes needed for a container) for each column partition value as compared to the same object without column partitioning. An object that is not column-partitioned only has a row header (14 or 20 bytes) for each regular row.

ROW format is useful for wide column partitions where one or only a few values fit in a container, and there is neither much benefit nor a negative impact from row header compression. ROW format provides quicker and more direct access to a specific column partition value than with COLUMN format because when a container has COLUMN format, Teradata Database must locate the column partition value within the container.

Depending on the autocompression types used for a container, access can be as simple as indexing into the container or it might require a sequential access through bits indicating how a value is compressed or sequential access through the column partition values, or both, to position to the specific column partition value to be accessed.

See “[Row Structure for Subrows \(ROW Format\)](#)” on page 767 for more information.

## Memory Limitations and Partitioning

### Error Messages for Memory Limitations for Partitioning

You might encounter several different memory limitations when working with partitioned tables. The following table documents the relevant error messages.

Message Number	Problem	Possible Cause	Remedy
3708	Table header size limit exceeded	Too many columns in the table.	If possible, reduce the number of columns in the table.
		Too many distinct values compressed in the table.	If possible, reduce the number of compressed values.
		Partitioning expressions are too complex.	Simplify the partitioning expressions or reduce their number. If possible, use RANGE_N instead of CASE_N.

Message Number	Problem	Possible Cause	Remedy
3710	Parser memory size limit exceeded.  The limit is variable and is determined by the value of the MaxParseTreeSegs field in DBS Control.	Current values for DBS Control parameters do not allocate enough parser memory to process the request.	<p>Change the values for the following DBS Control fields to the specified settings:</p> <ul style="list-style-type: none"> <li>If the problem occurs with partitioned tables that do not have either hash or join indexes, change the value for MaxParseTreeSegs to this value.                     <ul style="list-style-type: none"> <li>2000 for byte-packed format systems</li> <li>4000 for byte-aligned format systems</li> </ul> </li> <li>If the problem occurs with partitioned tables that have either hash indexes or join indexes or both, change the value for MaxParseTreeSegs to this value.                     <ul style="list-style-type: none"> <li>2000 for byte-packed format systems</li> <li>4000 for byte-aligned format systems</li> </ul> </li> </ul> <p>You might find that your query workloads that require these values need to be increased still further.</p> <p>See <i>Utilities: Volume 1 (A-K)</i> for information about how to change the value for MaxParseTreeSegs in the DBS Control record.</p>
	Parser memory size limit exceeded.	Partitioning expressions are too complex.	<p>Simplify the partitioning expressions or reduce their number.</p> <p>If possible, use RANGE_N instead of CASE_N.</p>
3891	Check if the partitioning CHECK constraint text size limit is exceeded.  The partitioning CHECK constraint text is derived from the partitioning expressions for the partitioning.	Text for the partitioning expressions in the partitioning definition is too long.	<p>Reduce the partitioning constraint text to 16,000 or fewer characters.</p> <p>30 characters of the 16,000 character limit for table constraints, partitioning constraints, and named constraints apply to the following constraint for a single-level partitioning: CHECK ((partitioning_expression) BETWEEN 1 AND 65535). This defines the limit for the partitioning expression text to be <math>6,000 - 30 = 15,970</math> characters.</p> <p>For multilevel partitioning, with a minimum partitioning constraint of CHECK /*nn*/ partitioning_expression_1 IS NOT NULL AND partitioning_expression_2 IS NOT NULL, the constraint is a minimum of 48 characters in length, so the minimum limit for the partitioning expression text is <math>16,000 - 48 = 15,952</math>. Each additional partition beyond 2 subtracts another 19 characters from the minimum of 15,952.</p>
3930	The dictionary cache is full.	Dictionary cache is too small.	<p>Increase size of dictionary cache to 1 MB.</p> <p>Because the default size of the dictionary cache is 1 MB, the only reason you might need to do this is if the DBA has reduced the size of the dictionary cache to something less than its default.</p>

## PPICacheThrP DBS Control Parameter

The PPICacheThrP performance parameter of the DBS Control utility (see *Utilities: Volume 1 (A-K)*) specifies the percentage value the system uses to calculate the cache threshold used in operations dealing with multiple partitions. The value is specified in units of 0.10% and can range between 0 - 500 (0% and 50.0%).

**Note:** Cache thresholds can also be controlled by the Optimizer cost profile parameter constant named PPICacheThrP. If the parameter is set, it overrides the setting for the PPICacheThrP DBS Control parameter. Only technical support personnel can change the cost profile settings for your system.

PPICacheThrP also specifies the percentage value to use for calculating the number of memory segments that can be allocated to buffer the appending of column partition values to column partitions for column-partitioned tables.

The sum of the sizes of this memory minus some overhead divided by the size of a column partition context determines the number of available column partition contexts. If there are more column partitions in a target column-partitioned table than available column partition contexts, multiple passes over the source rows are required to process a set of column partitions where the number of column partitions in each set equals the number of available column partition contexts. In this case, there is only one file context open, but each column partition context allocates buffers in memory.

The Optimizer also uses the DBS Control parameter PPICacheThrP to determine the number of available file contexts that can be used at a time to access a partitioned table.

IF PPICacheThrP determines the number of available file contexts to be ...	THEN Teradata Database considers this many file contexts to be available ...
< 8	8
> 256	256

Teradata Database uses the number of file contexts as the number of available column partition contexts for a column-partitioned table.

**Note:** Teradata Database might associate a file context with each column partition context for some operations, and in other cases it might allocate a buffer with each column partition context.

Ideally, the number of column partition contexts should be at least equal to the number of column partitions that need to be accessed by a request. Otherwise, performance can degrade because not all of the needed column partitions can be read at one time.

Performance and memory usage can be impacted if PPICacheThrP is set too high, which can lead to memory thrashing or a system crash. At the same time, the benefits of partitioning can be lessened if the value for the DBS Control parameter PPICacheThrP is set unnecessarily low, causing performance to degrade significantly.

The default is expected to be applicable to most workloads, but you might need to make adjustments to get the best balance.

**Note:** The DBS Control setting for PPICacheThrP is overridden by the Optimizer cost profile constant PPICacheThrP if that parameter is set in an applicable cost profile. Only technical support personnel can change the cost profile settings for your system.

The PPI Cache Threshold (PCT) is defined as follows.

For byte-packed format systems, or byte-aligned format systems where file system cache per AMP is less than 100 MB.

$$PCT = \left( \sum \text{file\_system\_cache} \right) \times \left( \frac{\text{PPICacheThrP}}{1,000} \right)$$

where:

Equation element ...	Specifies the ...
$(\sum \text{file\_system\_cache})$	total size of file system cache per AMP.
PPICacheThrP	cache threshold percentage specified by the PPICacheThrP parameter.

For 64-bit systems where the file system cache per AMP is greater than 100 MB.

$$PCT = \frac{100 \text{ MB} \times \text{PPICacheThrP}}{1,000}$$

where:

Equation element ...	Specifies the ...
PPICacheThrP	cache threshold percentage specified by the PPICacheThrP parameter.

The default is 10, which represents 1.0%. Increasing the PCT to a higher percentage is likely to cause memory contention problems. *Never change this value without first performing a thorough analysis of the impact of the change on your query workloads.*

When performing operations on partitioned tables, Teradata Database processes a subset of populated, non-eliminated partitions together instead of handling them one at a time. Teradata Database maintains a context that defines the current position within a partition for each partition being processed. In the case of multilevel partitioning, the term partition refers to a *combined* partition.

## How Teradata Database Controls Memory Usage by Partitions

The system associates each data block that contains a non-eliminated partition with each context. Whenever possible, the set of data blocks containing the partitions being processed is kept in memory to improve its processing.

If enough memory is not available to contain all the relevant data blocks, some of those blocks must be swapped to disk. While a minimum amount of swapping is acceptable, excessive swapping degrades system performance. The function of the PPICacheThrP parameter is to control the amount of partitioning data block swapping.

PPICacheThrP does this by controlling the memory usage of partitioned operations. Larger values improve the performance of partitioned operations as long as the following restrictions are observed:

- Data blocks are kept memory-resident.  
If they must be swapped to disk, performance might begin to degrade.
- The number of contexts does not exceed the number of populated, non-eliminated partitions being processed.  
In this case, increasing the value of PPICachThrP does not improve performance because each partition has a context, and additional contexts would not be used.

Note that the maximum PPI cache that can be allocated is 100 MB regardless of the setting for PPICacheThrP.

Teradata Database uses the value of PCT for the following operations on a partitioned table.

- Joins
- Aggregation
- Merge spooling

The equations described in the following bullets define the maximum number of partitions and contexts to process at a time based on the available PCT as determined from PPICacheThrP. If there are fewer populated non-eliminated partitions, then the system partitions all table partitions simultaneously.

The following bullets describe the partitioning operations that use PCT.

- Joins on a partitioned table.

If a join is performed on the primary index of partitioned table or spool and the other table or spool is not partitioned, the maximum number of partitions processed at one time from the partitioned table or spool is equal to the following calculation:

$$\text{Maximum number of partitions processed} = \text{MAX}(\text{MIN}\left(\frac{\text{PCT}}{(\text{est\_avg\_datablock\_size} + 12K)}, 256\right), 8)$$

If the join is made on the primary index of two partitioned relations, then the maximum number of partitions processed at a time from the relations is calculated as follows:

$$\text{Maximum number of partitions processed} = (\text{MAX}(\text{MIN}(f1, 256) + \text{MIN}(f2, 256), 16))$$

where:

$$\frac{(f1 \times db1) + (f2 \times db2)}{2} \leq \text{PPI Cache Threshold}$$

For each partition that is being processed, one data block is memory-resident. The definitions for the variables in this equation are as follows:

Equation element ...	Specifies the ...
f1	number of partitions to be processed at one time from the left relation in the join as determined by the Optimizer.
f2	number of partitions to be processed at one time from the right relation in the join as determined by the Optimizer.
db1	estimated average datablock size of the left relation in the join.
db2	estimated average datablock size of the right relation in the join.

- Aggregation of a partitioned table.

If the system aggregates on the primary index of a partitioned table, the maximum number of partitions processed at one time from the table is calculated as follows:

$$\text{Maximum number of partitions processed} = \left( \text{MAX}(\text{MIN}\left(\frac{\text{PCT}}{\text{(estimated avg datablock size} + 12K)}, 256\right), 8) \right)$$

Note that for each partition that is being processed, one data block is memory-resident. You should decrease the value of PPICacheThrP if memory contention occurs during these types of operations.

# Advantages and Disadvantages of Partitioned Primary Indexes

Advantages	Disadvantages
<ul style="list-style-type: none"> <li>Row partition elimination enables large performance gains to be realizable, and these are visible to end users. For this particular optimization, more populated partitions are generally better than fewer populated partitions.</li> <li>Batch inserts and updates can run faster if the partitioning schema matches the data arrival pattern. This optimization is visible only to the DBA and operations staff.                     <ul style="list-style-type: none"> <li>Finer granularity of partitions is generally better than a coarser granularity: daily is better than monthly.</li> <li>The largest performance improvements occur when there are no secondary indexes.</li> </ul> </li> <li>Teradata Parallel Data Pump inserts and updates can benefit from more FSG cache hits because of the increased locality of reference when a target table is partitioned on transaction date.  In this case, a finer partition granularity is generally better than a coarser partition granularity.</li> <li>Inserts into empty partitions are not journaled.  This optimization is only invoked if the table has no referential integrity constraints.</li> </ul>	<ul style="list-style-type: none"> <li>Partitioned table rows are each 2 or 8 bytes wider than the equivalent nonpartitioned table row. Partitioned table rows are 4 bytes wider if multivalue compression is specified for the table.  The extra 2 or 8 bytes are used to store the internal partition number for the row.</li> <li>You cannot define the primary index of a partitioned table to be unique unless the entire partitioning column set is part of the primary index definition.  You <i>can</i> define a USI on the primary index columns to enforce uniqueness if the complete partitioning column set is not a component of the primary index definition; however, that adds different performance issues to the equation.</li> <li>Primary index-based row access can be degraded if the partitioning column set is not a component of the primary index.                     <ul style="list-style-type: none"> <li>If you can define a secondary index on the primary index column set, then performance is independent of the number of partitions.</li> <li>If you cannot, or have not, defined a secondary index on the primary index, then having fewer partitions is better than having more partitions, whether achieved by means of the table definition itself or by row partition elimination during query processing.</li> </ul> </li> </ul>
<ul style="list-style-type: none"> <li>Delete operations can be nearly instantaneous when the partitioning column set matches the retention policy, there is no secondary index defined on the partitioning column set, and the delete is the last statement in the transaction.  You can delete all of the rows in a partition if you want to do so. In this case, there is no journaling of rows if no secondary index is defined on the partitioning column set.</li> <li>These properties might permit you to drop a secondary index or join index on the partitioning column set.</li> </ul>	<ul style="list-style-type: none"> <li>Joins of partitioned tables to nonpartitioned tables with the same primary index can be degraded. To combat this, observe the following guidelines:                     <ul style="list-style-type: none"> <li>Identically partition all tables to be joined with the same primary index when possible and then join them on the partitioning columns.</li> <li>Fewer partitions, whether achieved by means of the table definition itself or by row partition elimination, are often better than more partitions for the nonpartitioned-to-partitioned join scenario.</li> </ul> </li> </ul>

## Effects of Partition Cardinality On Query Performance

Multilevel partitioning typically defines a large number of partitions for the combined partitioning expression.

If there is a large number of populated partitions for the combined partitioning expression (high partition cardinality-note that the term cardinality here does not refer to the number of rows per partition, but the number of partitions themselves), then performance of primary index access, joins, and aggregations on the primary index can be degraded. This can be extremely critical given that the maximum number of combined partitions that can be defined is  $2^{63}-1$ .

Therefore, multilevel partitioning may be an appropriate choice when these operations are rarely done without also obtaining significant row partition elimination.

Note, however, that having a large number of row partitions making up the combined partitioning expression reduces the number of data blocks the system must process when it needs to access only a small number of row partitions due to row partition elimination.

## Selecting the Partitioning Granularity

The key guideline for determining the optimum granularity for the partitions of a partitioned table or join index is the nature of the workloads that most commonly access the object. The higher the number of partitions you define for an object, the more likely an appropriate range query against the table will perform more quickly, given that the partition granularity is such that the Optimizer can eliminate all but one partition or a few partitions.

On the other hand, it is generally best to avoid specifying too fine a partition granularity. For example if query workloads never access data at a granularity of less than one month, there is no benefit to be gained by defining partitions with a granularity of less than one month. Furthermore, unnecessarily fine partition granularity is likely to increase the maintenance load for a partitioned table, which can lead to degradation of overall system performance. In the end, even though too fine a partition granularity itself does not itself introduce performance degradations, the underlying maintenance on such a table *can* indirectly degrade performance.

## Determining the Maximum Number of Partitions for a Partitioned Database Object

When you define a partitioned database object, the question of how many partitions to define per partitioning level must be addressed.

If the table or join index is defined with multiple partitioning levels, you should explicitly specify the number of partitions per partitioning level using the ADD option (see “CREATE TABLE” in *SQL Data Definition Language*). If you do not, Teradata Database reserves any

excess partitions for partitioning level 1, and these partitions cannot be added to any other partitioning level once the table or join index is created.

The maximum number of partitions that are available for the level 1 partition from excess partitions depends on whether you have defined the object with 2-byte partitioning or with 8-byte partitioning.

The maximum number of partitions for level 1 for 2-byte partitioning is the following.

$$\text{Maximum number of partitions}_{\text{level 1}} = \left\lfloor \frac{65,535}{\left( \prod_{i=2}^n d_i \right)} \right\rfloor$$

The maximum number of partitions for level 1 for 8-byte partitioning is the following.

$$\text{Maximum number of partitions}_{\text{level 1}} = \left\lfloor \frac{9,223,372,036,854,775,807}{\left( \prod_{i=2}^n d_i \right)} \right\rfloor$$

where:

Equation element ...	Specifies
$\lfloor \rfloor$	the floor, or lower truncated value, of the expression.
$d_i$	the number of partitions defined at level $i$ .

- For a column-partitioning level, if
    - You do not specify an ADD clause for the column-partitioning level
    - There is also row partitioning
      - AND
      - At least one of the row-partitioning levels does not specify an ADD clause
- Then the maximum number of partitions for the column-partitioning level is the number of column partitions defined plus 10.
- For a column-partitioning level, if
    - You do not specify an ADD clause for the column-partitioning level
    - There is also row partitioning
    - All of the row-partitioning levels specify an ADD clause
      - AND
      - Using the number of column partitions defined plus 10 as the maximum number of column partitions, the partitioning would be 2-byte partitioning
- Then the maximum number of partitions for the column-partitioning level is the largest value that does not cause the partitioning to be 8-byte partitioning.

Otherwise, the maximum number of partitions for the column-partitioning level is the largest value that does not cause the maximum combined partition number to exceed 9,223,372,035,854,775,807.

For single-level partitioning, the maximum combined partition number is the same as the maximum partition number for the partitioning level.

- For a column-partitioning level, the maximum partition number is 1 more than the maximum number of partitions for that level. This is so that at least 1 unused column partition number is always available for altering a column partition.
- For each row-partitioning level without an ADD clause in level order, the maximum number of partitions for the row-partitioning level is the largest value that does not cause the partitioning to be 8-byte partitioning if using the number of row partitions defined as the maximum for this and any lower row-partitioning level without an ADD clause the partitioning would be 2-byte partitioning, is the largest value that does not cause the partitioning to be 8-byte partitioning.

Otherwise, the maximum number of partitions for this level is the largest value that does not cause the maximum combined partition number to exceed 9,223,372,035,854,775,807.

- The maximum number of partitions for a row-partitioning level is at least 2.

Otherwise, Teradata Database aborts the request and returns an error to the requestor. This error occurs when only one partition is defined for a row-partitioning level with an ADD 0 or with no ADD option and the maximum is not increased to at least 2.

- Having a maximum number of partitions for a partitioning level that is larger than the number of defined partitions for that level might enable the number of defined partitions for that level to be increased using an ALTER TABLE request.

The maximum number of partitions for a level cannot be increased.

- For a row-partitioning level, the maximum partition number is the same as the maximum number of partitions for that level.
- The maximum combined partition number is the product of the maximum partition number determined for each partitioning level. The value cannot exceed 9,223,372,036,854,775,807; otherwise, Teradata Database aborts the request and returns an error to the requestor.

For single-level partitioning, the maximum combined partition number is the same as the maximum partition number for the partitioning level.

- The number of combined defined partitions is the product of the number of defined partitions for each partitioning level.
- The maximum number of combined partitions is the product of the maximum number of partitions determined for each partitioning level.

For single-level partitioning, the maximum number of combined partitions is the same as the maximum number of partitions for the partitioning level.

If there is column partitioning, the maximum number of combined partitions is smaller than the maximum combined partition number or it is the same.

- If the maximum combined partition number is greater than 65,535, the partitioning is 8-byte partitioning; otherwise, it is 2-byte partitioning.

- Single-level column partitioning with no ADD option is 2-byte partitioning.
- The rules listed in the following table apply to 2-byte partitioning and to 8-byte partitioning for both row-partitioned and column-partitioned tables and join indexes.

For this partitioning ...	The maximum number of partitions for the first level is increased to ...	And the maximum number of partitioning levels is ...
2-byte	<p>the largest value that does not cause the maximum combined partition number to exceed 65,535 if it does not already do so.</p> <p>If there is at least one level with an explicit ADD clause, there is at least one level that consists solely of a RANGE_N function with BIGINT data type, or there is column-partitioning, this is repeated for each of the other levels, if any, from the second level to the last level.</p>	15
8-byte	<p>the largest value that does not cause the maximum combined partition number to exceed 9,223,372,036,854,775, 807 if it does not already do so.</p> <p>This is repeated for each of the other levels from the second level to the last level.</p>	62

You can alter any level to have between 1 and the maximum number of partitions allowed for that partitioning level.

## Assigning Excess Combined Partitions to Partitioning Levels

Teradata Database assigns leftover partitions to different partitioning levels depending the following set of rules.

IF the partitioning is ...	THEN Teradata Database adds as many leftover combined partitions as possible to the ...
single-level	first and only row or column partitioning level and the ADD value is overridden if one is specified.
multilevel and ...	
all the row partitioning levels have an ADD clause, but there is a column partitioning level <i>without</i> an ADD clause	column-partitioning level, which does not need to be at the first partitioning level.
a column partitioning level and at least one of the row partitioning levels does not specify an ADD clause, including the case where none of the row partitioning levels specify an ADD clause	<p>first row-partitioning level without an ADD clause after using a default of ADD 10 for the column-partitioning level.</p> <p>This is repeated for each of the other row partitioning levels without an ADD clause in partitioning level order.</p>

IF the partitioning is ...	THEN Teradata Database adds as many leftover combined partitions as possible to the ...
a column partitioning level has an ADD clause and at least 1 of the row partitioning levels does not specify an ADD clause	first row partitioning level without an ADD clause. This is repeated for each of the other row partitioning levels without an ADD clause in partitioning level order
there is no column partitioning level and at least 1 of the row partitioning levels does not have an ADD clause, including the case where none of the row partitioning levels specify an ADD clause	
all the partitioning levels specify an ADD clause or after applying leftover combined partitions	first row or column partitioning level and the ADD clause for the first partitioning level, if specified, is overridden. If there at least 1 level that specifies an explicit ADD clause, there is at least 1 level that consists only of a RANGE_N function with BIGINT data type, there is column partitioning, or the partitioning is 8-byte, this is repeated for each of the other levels from the second level to the last level.

## Usage Recommendations For Partitioning

### Basing Partitioning on a Numeric Column

This is the form that is the easiest to code, but is rarely the optimal choice.

If a table has a *division\_number* column defined as an integer with values between 1 and 65535, the following CREATE TABLE fragment can be used to establish partitioning by the division number:

```
CREATE TABLE ...
PRIMARY INDEX (invoice_number)
PARTITION BY division_number;
```

If the *division\_number* column for an attempted INSERT or UPDATE request has a value of 0 or less, is null, or is greater than 65535, then the request aborts and the system returns an error to the requestor. The row is not inserted or updated.

This clause can also be used if *division\_number* is defined with a CHARACTER data type, provided the values are always digits so that the column can be cast to INTEGER, because the system performs an implicit cast to the INTEGER type whenever necessary.

You can specify as many as 65,535 partitions if there are that many distinct values for the column. If you assume the more likely situation in which a company has, perhaps, four divisions, then there are only four partitions with rows. The table is not any larger because of the empty partitions, though the default Optimizer assumption that there are 65,535 partitions might sometimes mislead it into making suboptimal plan choices.

A disadvantage of this form is that the partitioning cannot be altered unless the table is empty.

## Basing Partitioning on Modulo Partitioning of a Numeric Column

This form uses a single column with a wide range of values, and maximizes the number of partitions. As an example, assume that a telephone company has a table with detailed information about each outgoing telephone call. One of the columns is the originating phone number, defined with the DECIMAL data type. A possible, and perhaps useful, partitioning expression can be defined as follows:

```
CREATE TABLE ...
PRIMARY INDEX (phone_number, call_start)
PARTITION BY phone_number mod 65535 + 1;
```

This partitioning expression, assuming millions of subscribers, populates each of the 65,535 partitions. Some partitions might have more rows than others, because some customers make more phone calls than others, but the distribution among the partitions should be somewhat even. This partitioning expression can improve the performance of a query that examines all phone calls made from a particular number by orders of magnitude by scanning only one partition out of 65,535 instead of the entire table.

Some disadvantages of this form are that the partitioning cannot be altered unless the table is empty, a maximum of 65535 partitions can be defined, and row partition elimination for queries is usually limited to constant or USING value equality conditions on the partitioning column.

## Basing the Partitioning Expression on Two or More Numeric Columns

This form uses arithmetic operations, typically multiplication and addition, on two or more numeric columns with suitably small value ranges. Assume a table with a three-digit product code and a two-digit store number. The store numbers count consecutively from 0, and there are fewer than 65 stores. This table can be partitioned as follows:

```
CREATE TABLE ...
PRIMARY INDEX (store_number, product_code, sales_date)
PARTITION BY store_number * 1000 + product_code;
```

If many queries specify both *store\_number* and *product\_code*, this might be a useful partitioning expression. One downside is that it fails if the number of products grows to the point that a four-digit number is required, or if the number of stores expands beyond 64.

Note that the Optimizer assumes 65,535 partitions even though some might be empty. The table is not any larger because of the empty partitions, though the Optimizer default assumption that there are 65,535 partitions based on the specification might sometimes mislead it into making suboptimal plan choices.

Some disadvantages of this form are that the partitioning cannot be altered unless the table is empty and row partition elimination for queries is usually limited to constant or USING value equality conditions on both of the partitioning columns.

An alternative is to use multilevel partitioning, as demonstrated by the following CREATE TABLE request:

```
CREATE TABLE ...
```

```
PRIMARY INDEX (store_number, product_code, sales_date)
PARTITION BY (RANGE_N(store_number) BETWEEN 0
              AND      64
              EACH     1),
              RANGE_N(product_code)
              BETWEEN 1
              AND      999
              EACH     1));
```

## Basing the Partitioning Expression on a CASE\_N Function

You can use the CASE\_N function to concisely define a partitioning expression for which each partition contains data based on an associated condition. When you specify CASE\_N for single-level partitioning expression, two partition numbers, NO CASE and UNKNOWN, are automatically reserved for specific uses.

The CASE\_N function is patterned after the SQL CASE expression (see *SQL Functions, Operators, Expressions, and Predicates*). The function returns an INTEGER value numbered from 1, indicating which CASE\_N condition first evaluated to TRUE for the particular value. The returned value can map directly to a partition number or be further modified to calculate the partition number.

Assume a table has a *total\_revenue* column, defined as DECIMAL. The table can be partitioned on that column, so that low revenue products are separated from high revenue products. The partitioning expression can be written as follows:

```
CREATE TABLE ...
PRIMARY INDEX (store_id, product_id, sales_date)
PARTITION BY CASE_N (total_revenue < 10000,
                      total_revenue < 100000,
                      total_revenue < 1000000,
                      NO CASE, UNKNOWN);
```

This request defines 5 partitions, conceptually numbered from 1 to 5 in the order they are specified in the partitioning expression.

This partition number ...	Represents ...
1	products with <i>total_revenue</i> less than 10,000.
2	products with <i>total_revenue</i> of at least 10,000, but less than 100,000.
3	products with <i>total_revenue</i> of at least 100,000 but less than 1,000,000.
4 (NO CASE)	any value that does not evaluate to TRUE or UNKNOWN for any previous CASE_N condition, which in this case is <i>total_revenue</i> equal to or greater than 1,000,000.
5 (UNKNOWN)	values for which it is not possible to determine the truth value of a previous CASE_N expression.  For this partitioning condition, a row with a null for <i>total_revenue</i> is assigned to the UNKNOWN partition, because by definition, it is not possible to evaluate whether a null is less than 10,000.

The rows in the NO CASE and UNKNOWN partitions *are* valid, and the system accesses those partitions to process queries unless the query conditions exclude them. By defining NO CASE and UNKNOWN partitions, you can ensure that *any possible value* maps to a partition. In the absence of those partitions, some values result in errors and so are not inserted into the table. In practice, it is probably better to have more than three revenue ranges unless the queries against this table rarely specify narrower revenue ranges.

This example demonstrates that CASE\_N can be used to define complicated partitioning expressions tailored to a specific table and specific query workloads:

```
CREATE TABLE
...
PRIMARY INDEX (col1, col2)
PARTITION BY CASE_N (col3 < 6,
                      col3 >= 8
                      AND col3 < 10
                      AND col4 <> 12,
                      col5 <> 10
                      OR col3 = 20,
NO CASE OR UNKNOWN);
```

Without knowing the meaning and data demographics of the columns, there is no way of knowing whether this partitioning expression is useful.

Note that the NO CASE and UNKNOWN partitions are combined into a single partition in this example.

Unlike the case for the previous partitioning expression examples, the Optimizer knows how many partitions are defined when CASE\_N is used as the partitioning expression and does not have to assume a default number of 65,535 partitions.

There are several disadvantages to this form:

- The partitioning of the table cannot be altered unless it is empty.
- Row partition elimination for queries is often limited to constant or USING value equality conditions on the partitioning columns.
- The Optimizer might not eliminate some row partitions that it possibly could if the partitioning were better conceived.
- As the number of conditions increases, evaluating a CASE\_N function can be costly in terms of CPU cycles, and a CASE\_N might also contribute to causing the table header to exceed its maximum size limit. Therefore, you may need to limit the number of conditions you define in the CASE\_N function to a relatively small number.

See “[Considerations for Basing a Partitioning Expression on a CASE\\_N Function](#)” on [page 349](#) for additional information on this topic.

## Considerations for Basing a Partitioning Expression on a CASE\_N Function

Building a partitioning expression on CASE\_N is a reasonable thing to do only if the following items are *all* true:

- The partitioning expression defines a mapping between conditions and INTEGER numbers.

- There are limited number of conditions (too many conditions can lead to excessive CPU usage or the table header exceeding its maximum size limit).
- Your query workloads against the table use equality conditions on the partitioning columns to specify a single partition.
- You have no need to alter the partitioning of the table.
- You get the plans and data maintenance you need.

## Basing a Partitioning Expression on a RANGE\_N Function

The RANGE\_N function is provided to simplify the specification of common partitioning expressions where each partition contains a range of numeric data, and is especially useful when the column contains a date or timestamp. RANGE\_N returns an INTEGER determined by the first range that includes the column value, numbered from 1, that can be mapped directly to a partition number or can be further modified to calculate the partition number. RANGE\_N is commonly used to define partitioning expressions. When you use a RANGE\_N function to define a single-level partitioning expression, two partition numbers, NO RANGE and UNKNOWN, are reserved for specific uses.

Assume a table with 7 years of order data, ranging from 2001 through 2007. The next partitioning expression creates 84 partitions, one for each month of the period covered by the data:

```
CREATE TABLE
...
PRIMARY INDEX (order_number)
PARTITION BY RANGE_N (order_date BETWEEN DATE '2001-01-01'
                           AND      DATE '2007-12-31'
                           EACH INTERVAL '1' MONTH);
```

Each partition contains roughly the same number of rows, assuming that order volume has stayed roughly constant across the seven year interval. Neither a NO RANGE nor an UNKNOWN partition is defined because this partitioning expression definition is for data that only has dates within the ranges specified in the partitioning expression.

It is frequently desirable to have each partition contain roughly the same number of rows, but it is *not* required. The next example puts the older orders into partitions with coarser granularity, and the newer orders into partitions with finer granularity:

```
CREATE TABLE
...
PRIMARY INDEX (order_number)
PARTITION BY RANGE_N (order_date BETWEEN DATE '1994-01-01'
                           AND      DATE '1997-12-31'
                           EACH INTERVAL '2' YEAR,
                           DATE '1998-01-01'
                           AND      DATE '2000-12-31'
                           EACH INTERVAL '1' YEAR,
                           DATE '2001-01-01'
                           AND      DATE '2003-12-31'
                           EACH INTERVAL '6' MONTH,
                           DATE '2004-01-01'
                           AND      DATE '2007-12-31'
                           EACH INTERVAL '1' MONTH,
```

```
NO RANGE, UNKNOWN);
```

In this example, the more recent data is partitioned more finely than the older data. This can be a good strategy if you know that the older data is rarely accessed except as part of a full-table scan, because it reduces some potential disadvantages by defining a smaller number of partitions. However, maintaining this structure over extended epochs of time is not as simple as maintaining a structure in which each interval covers the same time duration. In this example, the years 2004 through 2007 are partitioned by month. As time passes, and 2004 data becomes older and less frequently referenced, it shall become necessary to repartition the table if the pattern of defining longer time intervals for older data is to be maintained.

It is both easy and fast to add and drop partitions from the ends of a partitioning definition, but repartitioning intervals in the middle partitions requires much more work, and it is usually faster to reload the data. In addition, when a range partition is dropped, any rows in that partition are moved from the dropped range partition to the NO RANGE partition or to an added range partition. Also, when a range is added, rows might need to be moved from the NO RANGE partition to the new range partition.

Some expansion room is allowed for future dates by specifying the final partition as extending to the end of 2007. It is easy to add and drop ranges to the end of a partitioning expression using the ALTER TABLE statement. While you could have specified an ending date far in the future, such as 2099-12-31, it is generally not desirable to define hundreds of partitions that shall not be used for decades.

The example shows intervals of years and months. It is also possible to partition by day (EACH INTERVAL '1' DAY) or by week (EACH INTERVAL '7' DAY). A seven-day interval can start on any day of the week, so if you want to start the weekly intervals on Sunday, for example, the beginning date should be chosen so that it falls on a Sunday. Also, the last range might be less than specified by the EACH clause. For example, suppose that with seven-day intervals the first date falls on Sunday, but the last date also falls on Sunday, in which case the last range spans only one day.

The RANGE\_N NO RANGE clause is comparable to the CASE\_N NO CASE clause, and the UNKNOWN clause has the same meaning for both functions.

As with the CASE\_N partitioning expression, the Optimizer knows how many partitions are defined when RANGE\_N is used as the partitioning expression.

The following example revisits the *division\_number* example, this time using a RANGE\_N function:

```
CREATE TABLE
...
PRIMARY INDEX (invoice_number)
PARTITION BY RANGE_N (division_number BETWEEN 1
                      AND      4
                      EACH     1);
```

This alternate partitioning expression allows you to specify only four partitions instead of 65,535. The Optimizer might be able to choose certain join plans and more accurately cost those plans when the maximum number of partitions is both known and small, making this a better choice than using the column directly. This RANGE\_N partitioning expression also prevents rows with non-valid division numbers from being inserted into the table.

## Considerations for Basing the Partitioning Expression on a RANGE\_N Function

Using the RANGE\_N function to build a partitioning expression offers the following advantages:

- Defining an efficient mapping or ranges between integer (BYTEINT, SMALLINT, INTEGER, BIGINT), character (CHARACTER, GRAPHIC, VARCHAR, VARCHAR(n) CHARACTER SET GRAPHIC), DATE, or TIMESTAMP type and INTEGER numbers.
- Provides more opportunities than other expressions to optimize queries.
- The Optimizer knows the number of defined partitions when you specify RANGE\_N to define the partitioning expression.

For other partitioning expressions, the Optimizer generally assumes a total of 65,535 partitions when statistics have not been collected, which could easily be far more than the number of populated partitions. However, collecting statistics on PARTITION can provide information about which partitions are empty.

- Faster partitioning changes than any other expression using the following ALTER TABLE options.
  - ADD RANGE
  - DROP RANGE

You can optimize the effects of using RANGE\_N in your partitioning expression by observing the following guidelines:

- Reference only a single integer (BYTEINT, SMALLINT, INTEGER, BIGINT), character (CHARACTER, GRAPHIC, VARCHAR, VARCHAR(n) CHARACTER SET GRAPHIC), DATE, or TIMESTAMP column, not expressions.

For example, specifying a simple expression such as  $\frac{x}{10}$  in place of a column name in the RANGE\_N specification, even if the expression references only a single column, can hinder, or even prevent, row partition elimination.

- For equal-sized ranges, always specify an EACH clause.

Note the following collateral facts about equal- and unequal-sized partitions:

- The performance of unequal-sized partitions varies depending on which partitions are accessed.
- Unequal size ranges can prevent fast partitioning changes from being made using the ALTER TABLE statement (see *SQL Data Definition Language*).
- Using the NO RANGE, NO RANGE OR UNKNOWN, or UNKNOWN specifications for a range can negatively affect later ALTER TABLE partitioning strategies.

Do not use these clauses unless you have specific reasons for doing so (see *SQL Functions, Operators, Expressions, and Predicates* for details).

Reasons not to use the NO RANGE, NO RANGE OR UNKNOWN, and UNKNOWN clauses include the following.

- The maintenance and use of partitioned tables is simpler if you do not use these options, and avoiding their use also prevents bad data from being inserted into the table.

- If these partitions are not eliminated by row partition elimination, they can cause negative performance impacts if they contain a large number of rows.
- If a partitioning column is NOT NULL such that test values can never be null, do not specify UNKNOWN or NO RANGE OR UNKNOWN.
- If the specified ranges cover all possible values, do not specify NO RANGE or NO RANGE OR UNKNOWN.

## Basing the Partitioning Expression on a RANGE\_N Character Column

The RANGE\_N function provides a simplified method for mapping a character value into one of a list of specified ranges, and then returning the number of that range. A range is defined by its starting and ending boundaries, inclusively.

If you do not specify an ending boundary, the range is defined by its starting boundary, inclusively, up to, but not including, the starting boundary for the next range.

The EACH clause is not supported for character test values, and Teradata Database returns an error if you specify an EACH clause in a RANGE\_N function with a character test value. Each range specified for expressions character data types maps to exactly an integer range of only one value. Therefore, the number of partitions that can be specified is somewhat limited because of table header limitations and due to the increase in CPU usage to handle a large number of ranges.

You can specify an asterisk for the first starting boundary to indicate the lowest possible value, and you can specify an asterisk for the last ending boundary to indicate the highest possible value.

As with numeric RANGE\_N expressions, options are provided to handle cases when the value does not map into one of the specified ranges or evaluates to UNKNOWN because of a null result, making it impossible to determine into which range the value would map.

The following rules apply to the use of the RANGE\_N function for character data in addition to the rules that exist for numeric data:

- You can specify only one test value, and it must result in an integer (BYTEINT, INTEGER, SMALLINT, BIGINT), character (CHARACTER, GRAPHIC, VARCHAR, VARCHAR(n) CHARACTER SET GRAPHIC), DATE, or TIMESTAMP data type.
- A RANGE\_N partitioning expression can specify the UPPERCASE qualifier and the following functions.
  - CHAR2HEXINT
  - INDEX
  - LOWER
  - MINDEX
  - POSITION
  - TRANSLATE
  - TRANSLATE\_CHK
  - TRIM

- UPPER
- VARCHAR(n) CHARACTER SET GRAPHIC
- Teradata Database aborts the request and returns an error if any of the specified ranges are defined with null boundaries, are not increasing, or overlap. Increasing order is determined using the session collation and the case sensitivity specification for the test value at the time the table is created.

## Using CASE\_N and RANGE\_N in SELECT Requests

You can also use the CASE\_N and RANGE\_N functions in a SELECT request. For example, you might use them to help determine the distribution of rows among partitions for a proposed partitioning expression. For example, before deciding to partition on *division\_number*, you might want to check the resulting distribution with a request like the following.

```
SELECT RANGE_N(division_number) BETWEEN 1
                                AND      4
                                EACH     1) AS p,
          COUNT(*) AS c,
FROM sales
GROUP BY p
ORDER BY p;
```

Another use of RANGE\_N is to determine the number of ranges defined, for example, as follows.

```
SELECT RANGE_N(DATE '2007-12-31') BETWEEN DATE '2001-01-01'
                                              AND      DATE '2007-12-31'
                                              EACH INTERVAL '30' DAY);
```

This query returns the value 86 because the last range is less than 30 days.

The following query over the same data returns the value 84 because the number of days per range varies between 28 and 31, depending on the month and year.

```
SELECT RANGE_N(DATE '2007-12-31') BETWEEN DATE '2001-01-01'
                                              AND      DATE '2007-12-31'
                                              EACH INTERVAL '1' MONTH);
```

The final example returns the value 13 because the last range is only one day in length.

```
SELECT RANGE_N(DATE '2002-01-01') BETWEEN DATE '2001-01-01'
                                              AND      DATE '2002-01-01'
                                              EACH INTERVAL '1' MONTH);
```

See “[Considerations for Basing the Partitioning Expression on a RANGE\\_N Function](#)” on page 352 for additional information on this topic.

## Workload Characteristics, Queries, and Row Partition Elimination

Row partition elimination is most effective in the following situations. You should *always* verify (using the EXPLAIN request modifier) that you are getting the desired results for any plans.

- Row partition elimination is most effective with constant conditions on the partitioning columns.

- When a row partitioning expression is written using something other than the RANGE\_N function or a single column, row partition elimination is most effective when you specify constant equality conditions.
- Row partition elimination can also be effective with equality conditions on USING variables if the conditions specify a single partition.
- Row partition elimination occurs for CURRENT\_DATE and DATE built-in functions for inequality conditions. This does *not* prevent the request from being cached.
- Row partition elimination *might* occur for other built-in functions and USING variables in inequality conditions, and if it does, the action prevents the system from caching the request.
- Multiple ORed equality conditions on the same row partitioning column do *not* invoke partition elimination.

As an alternative, you should try either to use the UNION operator on two SELECT requests or to substitute constants for the USING variables in any inequality conditions.

- Use simple comparison of a partitioning column to a constant, built-in function, or USING variable expression for your query conditions.

For example,

- $d=10$  AND  $d \leq 12$
- $d$  BETWEEN 10 AND 12 AND  $d = 10+1$
- $d$  IN (20, 22, 24) AND  $d = 20$  OR  $d=21$
- $d = :udd$  BETWEEN CURRENT\_DATE-7 and CURRENT\_DATE-1
- Avoid specifying query conditions with expressions or functions constructed on the row partitioning column. For example, use the form in Example 2 rather than the form in Example 1, and the form in Example 4 rather than the form in Example 3.

### Example 1

The predicate in this query is based on an expression constructed on the row partitioning column  $x$ :

```
CREATE TABLE
  ...
  PARTITION BY x;

  SELECT ...
  WHERE x+1 IN (2,3);
```

### Example 2

The predicate in this query is based only on the value of the row partitioning column  $x$ :

```
CREATE TABLE
  ...
  PARTITION BY RANGE_N(x BETWEEN 1
                        AND 65533 ← preferably use max value
                        EACH      1); of x instead of 65,533
```

It is preferable to specify the exact upper limit of the range of  $x$ , if it is less than 65,533, in this CREATE TABLE request rather than 65,533.

```
SELECT ...
WHERE x IN (1,2);
```

### Example 3

The predicate in this query is based only on the value of row partitioning column  $x$  even though the table is partitioned by the expression  $x + 1$ :

```
CREATE TABLE
...
PARTITION BY x+1;

SELECT ...
WHERE x IN (1,2);
```

### Example 4

The predicate in this query is based only on the value of row partitioning column  $x$ :

```
CREATE TABLE
...
PARTITION BY RANGE_N(x BETWEEN 0
                      AND 65532 ← preferably use max value
                      EACH      1);   of x instead of 65532
```

Note that it is preferable to specify the exact upper limit of the range of  $x$ , if it is less than 65,552, in this CREATE TABLE request rather than 65,532.

```
SELECT ...
WHERE x IN (1,2);
```

## Workload Characteristics and Row Partitioning

Consider using row partitioning when your workloads have any of the following characteristics:

- The number of queries in workloads that access the table have a range constraint, particularly a date constraint on some column of the table.
- Queries have an equality constraint on some column of the table, and that column is either not the only primary index column or it is not a primary index column at all.
- If there is a primary index that is used only, or principally, to achieve an even distribution of rows, but not usually for accessing or joining rows, and access is frequently made on a column that is suitable for partitioning.
- If there is a primary index that is used to achieve an even distribution of rows as well as for accessing or joining rows, and columns suitable for partitioning are included in the primary index definition.
- If there is a primary index that is used to achieve an even distribution of rows as well as for accessing or joining rows, but columns suitable for partitioning are *not* included in the primary index definition.

This might be a good candidate for partitioning, but you need to pay particular attention to weighing the performance tradeoffs that often result in this situation.

- Use the RANGE\_N function for a partitioning expression, preferably on a column with a DATE or TIMESTAMP data type, because it generally provides more opportunities for row partition elimination, and the Optimizer knows the exact number of defined row partitions.

For instance, instead of the following row partitioning expression.

```
PARTITION BY column
```

use

```
PARTITION BY RANGE_N(column BETWEEN m
                      AND      n
                      EACH     s)
```

Dates and timestamps are often used in query conditions and therefore makes good candidates for a partitioning expression.

When partitioning on a DATE column, use RANGE\_N with a single overall range divided into ranges of equal size as follows.

```
PARTITION BY RANGE_N(date_column BETWEEN DATE '...'
                      AND      DATE '...'
                      EACH INTERVAL 's' t)
```

- Use DATE constants or TIMESTAMP constants such as DATE '2011-08-06' or TIMESTAMP '2011-08-25 10:14:59' to specify the ranges in a partitioning expression. This is not only easier to read, making it clear that the expression is defined on a date, but it also removes the dependence on the FORMAT used in the implicit conversion to a date. You can also specify TIMESTAMP(*n*) WITH TIME ZONE constants for RANGE\_N partitioning.

For example, you can specify a RANGE\_N-based partitioning expression like the following.

```
RANGE_N(ts BETWEEN TIMESTAMP '2003-01-01 00:00:00+13:00'
          AND      TIMESTAMP '2009-12-31 23:59:59-12:59'
          EACH INTERVAL '1' MONTH)
```

Use an INTERVAL constant in the EACH clause where the variable *t* is DAY, MONTH, YEAR, or YEAR TO MONTH.

Do *not* use INTEGER values or CHARACTER constants for dates in your partitioning expressions, since they can easily be incorrectly specified such that they do convert to the date you expected.

For example, it might seem intuitive to simply partition by the name of a DATE column as follows:

```
PARTITION BY sales_date
```

This form does not produce a syntax error. In fact, it works correctly *only for dates in the early 1900s* and follows the rule of implicit conversion to get an INTEGER partition number, but such a table is not generally useful.

For this case, you should instead use RANGE\_N with a granularity of EACH INTERVAL '1' DAY.

It might also seem intuitive to specify something like the following to indicate that a date column in the primary index is to be partitioned by week:

```
PARTITION BY 7
```

However, this form *does* produce a syntax error.

For this case, use RANGE\_N with a granularity of EACH INTERVAL '7' DAY.

- Consider specifying only as many date ranges as are currently needed plus a few additional ranges for the future.

By limiting ranges to those that are currently needed, you help the Optimizer to better cost plans and also allow for more efficient primary index access, joins, and aggregations when the partitioning column is not included in the primary index.

This is not as important if you collect current PARTITION statistics.

A good guideline is to define 10% or fewer of the partitions to be empty to be able to handle future dates. You should also define enough future ranges to minimize the frequency of ALTER TABLE statements needed to drop and add ranges. However, if you make changes too infrequently, you might forget to alter the table entirely.

Altering the table only once a year is probably not a good idea for the following reasons:

- Because you must create too many empty partitions.
- Because it is too easy to forget to alter the partitioning ranges if you do not do it fairly regularly.
- Because you fail to follow the procedure often enough to prevent problems from occurring when you finally get around to following it.

You must balance these concerns of having *enough* future partitions, but not *too* many.

- RANGE\_N permits a faster partitioning change using ALTER TABLE ... DROP RANGES or ... ADD RANGES.
- Reference a single INTEGER or DATE column in the RANGE\_N function.

Do *not* use an expression such as  $\frac{x}{10}$  in place of a simple column reference in a partitioning expression constructed from a RANGE\_N function even if the expression only references a single column.

- Define ranges of equal size using EACH to specify the granularity of the partition. Multiple ranges, with or without specifying an EACH granularity, require more CPU time to execute and different sized ranges can prevent fast partitioning changes.
- Consider not specifying the NO RANGE, NO RANGE OR UNKNOWN, or UNKNOWN partitions in the RANGE\_N function.

Using these partitions can degrade query performance because queries can be forced to scan data in these partitions unnecessarily. If you specify the NO RANGE, NO RANGE OR UNKNOWN, or UNKNOWN partitions, you can also affect the performance of ALTER TABLE partitioning change negatively because data might need to be moved among the partitions. An UNKNOWN partition is not needed if the partitioning cannot produce rows with unknown partition values (this is often the case when a partitioning column is specified to be NOT NULL).

- Deciding *not* to use a RANGE\_N function for a partitioning expression can be a good choice in the following instances:
  - You do not partition the table or join index on a DATE column.
  - You use constant or USING variable equality conditions on the partitioning columns in a majority of the queries in your workloads to specify a single partition.

If the assumption made by the Optimizer that the table (not based on a CASE\_N partitioning expression) has 65,535 partitions provides good plans in these cases, you do not need to alter partitioning, and the system produces the plans and data maintenance performance your site requires.

## Workload Characteristics and Partitioning

Consider the following points when you define the partitioning for a table or join index:

- At minimum, you must choose a primary index (or specify no primary index) to achieve an even distribution of rows to the AMPs.
- When appropriate, the primary index column set ought to be constructed from columns that are often constrained by equality conditions in queries in order to provide fast access, joins, and aggregations on those columns.

If the partitioning column set is included in the primary index column set, some concerns are not an issue. However, you would not add the partitioning columns to the primary index to avoid these concerns anyway, because there is usually little, if any, benefit in doing so.

- If the partitioning columns are included in the set of primary index columns, then you can specify the primary index to be UNIQUE.

Adding a partitioning column as a primary index column just to make the primary index unique is not effective, however, because it is the *original* set of primary index columns that needs to be unique and possibly used for access, joins, and aggregations.

- If the partitioning columns are also included in the set of primary index columns, it might be a good choice to define many combined partitions for primary index access and joins. Be aware that plan costing can be affected if there are too many empty combined partitions if PARTITION statistics are not collected. Other factors can also lead to having fewer combined partitions.
- If all of the partitioning columns are not included in the primary index column set, the primary index cannot be defined as a UPI.

If the primary index columns must be unique, define a USI on them.

Because MultiLoad and FastLoad do not support USIs, you must use another load strategy such as any of the following:

- Loading the rows using Teradata Parallel Data Pump (see *Teradata Parallel Data Pump Reference*).
- Loading the rows into a staging table followed by an INSERT ... SELECT or MERGE into the target table using error logging (see “CREATE ERROR TABLE” in *SQL Data Definition Language* and “INSERT ... SELECT” and “MERGE” in *SQL Data Manipulation Language*).
- Dropping the USI, loading the rows using MultiLoad or FastLoad, and then recreating the USI (see *Teradata FastLoad Reference* and *Teradata MultiLoad Reference*).
- Including the partitioning columns in the PI, noting the previously documented problems with this approach.

You must evaluate the tradeoffs among these choices carefully.

- If the primary index columns need to be an efficient access path and there are many combined partitions, consider one of these options:
  - Defining a USI on the primary index columns to improve access time.
  - Defining a NUSI on the primary index columns to improve access time.
  - Creating a join index to cover queries made against the table.

- Creating a hash index to cover queries made against the table.
- Defining a USI or NUSI on the primary index columns to improve access time.
- Consider defining fewer combined partitions when a table is also accessed or joined on the primary index.

## Workload Characteristics and Joins

Consider the following points when you write join queries against a partitioned table:

- Specify equijoins on the primary index and partitioning column sets, if possible, in order to prejudice the Optimizer to use efficient RowKey-based joins (see *SQL Request and Transaction Processing*).

Consider including the partitioning column in the nonpartitioned primary index table so you can join on the partition column. This means that, depending on the situation, you might want to consider denormalizing the physical schema to enhance the performance of partitioned table-to-nonpartitioned table joins.

- If you specify an equijoin on the primary index column set, but not on the partitioning column set, the fewer combined partitions that exist after any row partition elimination, the better.

Otherwise, the table might need to be spooled and sorted.

The Optimizer can specify sliding-window joins when there are a small number of participating combined row partitions (see *SQL Request and Transaction Processing*).

- Use RANGE\_N to define fewer partitions and specify conditions on the row partitioning columns to reduce the number of combined row partitions involved in the join by evoking partition elimination.

To ensure that the Optimizer creates good query plans for your partitioned tables, you should always collect PARTITION statistics and keep them current.

If you have not collected PARTITION statistics, the Optimizer does not know whether a combined row partition is empty or not, so it has to assume all defined combined row partitions might have rows with respect to the plan it generates; however, it might choose among several such plans based on the estimated number of populated combined row partitions.

- Dynamic row partition elimination for a product join improves performance when a partitioned table and another table are equijoined on the row partitioning column of the partitioned table (see *SQL Request and Transaction Processing*).

Remember to collect statistics (see “COLLECT STATISTICS (Optimizer Form)” in *SQL Data Definition Language*) on all of the following.

- The primary indexes of both tables.
- The partitioning columns of the partitioned table.
- The column in the nonpartitioned table that is equated to the partitioning column of the partitioned table.
- The system-derived PARTITION column of all partitioned tables (see “[PARTITION Columns](#)” on page 801).

The recommended practice for recollecting statistics is to set appropriate thresholds for recollection using the THRESHOLD options of the COLLECT STATISTICS statement. See “COLLECT STATISTICS in *SQL Data Definition Language* for details on how to do this.

## General Recommendations for Using Row-Partitioned Tables and Join Indexes

To take optimal advantage of row-partitioning as it is used in Teradata Database for row-partitioned tables and join indexes, you must have a thorough understanding of partitioning expressions, the general notion of partitioning, and the specific attributes that partitioning brings to a table. The placement of data on the AMPs and the use of row partition elimination by the system can significantly improve the performance of some queries, while at the same time degrading the performance of other queries. You must consider the impact of partitioning on data maintenance. You must be aware that partitioning increases the size of each row header in a partitioned table or index by either 2 or 8 bytes (partitioned table rows are 4 bytes wider if multivalue compression is specified for the table) and that partitioning also increases the size of each secondary index row by 2 or 8 bytes for each referencing ROWID in the index.

Partitioning is a physical database design consideration, and like any other physical database design issue, it is more likely to work well if you have done a good logical database design first.

You should not focus on any one aspect of partitioning while undertaking the process of creating the physical design for your databases. Each and every one of the following attributes must work well together to ensure the success of your partitioned tables and join indexes.

- The partitioning expression
- Queries

This includes both those queries designed specifically to access the partitioned table and the general class of all queries that are likely to access it.

- Performance, access methods, join strategies, row partition elimination
- Ease of altering the partitioning expression
- Effects of the row partitioning on data maintenance for the table
- Backup and restore operations on the table

A successful partitioning expression is one that takes advantage of row partition elimination, supports ease of partition altering with ALTER TABLE, and has no significant negative impact on data maintenance.

You should always experiment with your intended uses of partitioned tables, considering and analyzing performance tradeoffs between using partitioning or not, various partitioning strategies, and using partitioning along with, or instead of, other indexing such as secondary, hash, and join indexes.

Analyze the maintenance process choices and their performance. Note that additional maintenance is required if you define a USI to enforce uniqueness on a column set of a partitioned table.

Finally, you must always ensure that you are getting the results that you expect. Be sure to review EXPLAIN reports, looking for row partition elimination and rowkey-based joins. Defining a sophisticated partitioning expression is helpful only if the queries in your workloads are able to invoke row partition elimination. Be sure to measure performance for the query workload and for critical queries both before and after creating the partitioned table or join index. Never assume that partitioning will improve the performance of your maintenance workloads, verify it. And always weigh the costs against the benefits.

## Single-Level Partitioning

### Partitioning CHECK Constraints for Single-Level Partitioning

Teradata Database derives a table-level partitioning CHECK constraint from the partitioning expression. The text for this derived partitioning constraint cannot exceed 16,000 characters; otherwise, Teradata Database aborts the request and returns an error to the requestor.

The following diagrams provides the two forms of this partitioning CHECK constraint derived for single-level partitioning. The forms differ depending on whether the partitioning expression has an INTEGER data type or not.

The first form applies to partitioning expressions that do not have an INTEGER type. Call this partitioning constraint form 1.

```
CHECK ((CAST((partitioning_expression) AS INTEGER)) BETWEEN 1 AND max)
```

The second form applies to partitioning expressions that have an INTEGER type. Call this partitioning constraint form 2.

```
CHECK ((partitioning_expression) BETWEEN 1 AND max)
```

where:

Syntax element ...	Specifies the ...
<i>partitioning_expression</i>	partition number returned by the single-level partitioning expression.
<i>max</i>	maximum number of partitions defined by <i>partitioning_expression</i> . <ul style="list-style-type: none"><li>• If the partitioning expression is defined using something other than a RANGE_N or CASE_N function, the value of <i>max</i> is 65535.</li><li>• If the partitioning expression is defined using a RANGE_N function, the value of <i>max</i> is the number of partitions defined by the RANGE_N function.</li><li>• If the partitioning expression is defined using a CASE_N function, the value of <i>max</i> is the number of partitions defined by the CASE_N function.</li></ul>

Multilevel-partitioned tables have a different table-level partitioning CHECK constraint (see “[Multilevel Partitioning](#)” on page 372).

If any one of the following items is true, Teradata Database uses a different form of partitioning constraint:

- A partitioning expression for one or more levels consists solely of a RANGE\_N function with a BIGINT data type.
- An ADD clause is specified for one or more partitioning levels.
- The table has 8-byte partitioning.
- There is a column-partitioning level.
- For other than level 1 of a populated table, the number of partitions for at least one level changes when the table is altered.
- For other than level 1 of an empty table, the number of partitions for at least one level decreases when the table is altered.
- The cost profile constant PartitioningConstraintForm is set to 1.

The format for this partitioning constraint text is as follows. Call this partitioning constraint form 4.

```
CHECK /*nn bb cc*/ partitioning_constraint_1 ...
      [AND partitioning_constraint_n])
```

where:

Syntax element ...	Specifies ...
<i>nn</i>	the number of partitioning levels. <ul style="list-style-type: none"> <li>• For 2-byte partitioning, <i>nn</i> ranges between 01 and 15, inclusive.</li> <li>• For 8-byte partitioning, <i>nn</i> ranges between 01 and 62, inclusive.</li> </ul>
<i>bb</i>	the number of bytes used to store the internal partition number in the row header. <ul style="list-style-type: none"> <li>• For 2-byte partitioning, <i>bb</i> = 2.</li> <li>• For 8-byte partitioning, <i>bb</i> = 8.</li> </ul>
<i>cc</i>	the column partitioning level. <ul style="list-style-type: none"> <li>• If there is no column partitioning, <i>cc</i> = 00.</li> <li>• If there is column partitioning, <i>cc</i> ranges between 01 and <i>nn</i>, inclusive.</li> </ul>
<i>partitioning_expression_i</i>	the partitioning expression at partitioning level <i>i</i> .
<i>partitioning_constraint_i</i>	<ul style="list-style-type: none"> <li>• <i>partitioning_expression_i /*i d+a*/ IS NOT NULL</i> if there is row partitioning at partitioning level <i>i</i>.</li> <li>• <i>PARTITION#Li /*i d+a*/ = 1</i> if there is column partitioning at partitioning level <i>i</i>.</li> </ul>
<i>i</i>	a partitioning level that ranges between 1 and <i>nn</i> , inclusive. Leading zeros are not used for the value of <i>i</i> .

Syntax element ...	Specifies ...
<i>d</i>	<p>the number of currently defined partitions for the level.</p> <p>For a column-partitioned level, this includes the 2 internal column partitions.</p> <p>Leading zeros are not used for the value of <i>d</i>.</p>
<i>a</i>	<p>the number of additional partitions that could be added (which might be 0) or X.</p> <p>X occurs for level 2 and higher if this partitioning constraint form is only being used because the cost profile constant PartitioningConstraintForm is set to 1 to force use of the new constraint form in all cases.</p> <p>If the new constraint form would be used regardless of the setting of PartitioningConstraintForm or this is for level 1, <i>a</i> is an integer number.</p> <p>Leading zeros are not used for the value of <i>a</i>.</p>

Each of the partitioning constraints corresponds to a level of partitioning in the order defined for the table.

The *TableCheck* column of *DBC.TableConstraints* contains the unresolved condition text for a table-level CHECK constraint check or implicit table-level constraint such as a partitioning constraint. The ConstraintType code for such a partitioning constraint is Q for a partitioned object. See *Data Dictionary* for details.

Rows that violate this partitioning CHECK constraint, including those whose partitioning expression evaluates to null, are not allowed in the table.

Assume you have created the following table.

```
CREATE TABLE orders (
    o_orderkey      INTEGER NOT NULL,
    o_custkey        INTEGER,
    o_orderstatus    CHARACTER(1) CASESPECIFIC,
    o_totalprice     DECIMAL(13,2) NOT NULL,
    o_orderdate      DATE FORMAT 'yyyy-mm-dd' NOT NULL,
    o_orderpriority  CHARACTER(21),
    o_clerk          CHARACTER(16),
    o_shipppriority  INTEGER,
    o_comment         VARCHAR(79))
PRIMARY INDEX (o_orderkey)
PARTITION BY (RANGE_N(o_custkey) BETWEEN 0
              AND 49999
              EACH 100))
UNIQUE INDEX (o_orderkey);
```

The partitioning CHECK constraint SQL text that is stored in *DBC.TableConstraints* for this multilevel partitioned primary index is as follows.

```
CHECK (RANGE_N(o_custkey) BETWEEN 0
              AND 49999
              EACH 100)
BETWEEN 1 AND 500
```

Now suppose that you create a column-partitioned version of the *orders* table, *orders\_cp*, with the following definition.

```
CREATE TABLE orders_cp (
    o_orderkey      INTEGER NOT NULL,
    o_custkey       INTEGER,
    o_orderstatus   CHARACTER(1) CASESPECIFIC,
    o_totalprice    DECIMAL(13,2) NOT NULL,
    o_orderdate     DATE FORMAT 'yyyy-mm-dd' NOT NULL,
    o_comment       VARCHAR(79))
NO PRIMARY INDEX
PARTITION BY (RANGE_N o_custkey BETWEEN 0
              AND 100000
              EACH      1), COLUMN)
UNIQUE INDEX (o_orderkey);
```

The partitioning CHECK constraint for this table with 8-byte partitioning is as follows.

```
CHECK /*02 08 02*/ RANGE_N(o_custkey BETWEEN 0 AND 100000 EACH
1) /*1 100001+485440633518572409*/ IS NOT NULL AND PARTITION#L2 /
*2 9+10*/ =1)
```

You could use the following SELECT request to retrieve the level for the column partitioning for each of the objects that have column partitioning in the system.

```
SELECT DBaseId, TVMId, ColumnPartitioningLevel (TITLE
                                              'Column-Partitioning Level')
  FROM DBC.TableConstraints
 WHERE ConstraintType = 'Q'
   AND ColumnPartitioningLevel >= 1
 ORDER BY 1,2;
```

See *Data Dictionary* for details about *DBC.TableConstraints* and its role in recording partitioning metadata.

The maximum size of all partitioning CHECK constraints is 16,000 characters.

## Single-Level Partitioning Example

The following multipart example demonstrates the various properties of different single-level partitioning of the same data.

**Stage 1:** First single-level partitioning of the *orders* table.

```
CREATE TABLE orders (
    o_orderkey INTEGER NOT NULL,
    o_custkey   INTEGER)
PRIMARY INDEX (o_orderkey)
PARTITION BY RANGE_N(o_custkey BETWEEN 0
                      AND 100
                      EACH 10); /* p1 */
```

This definition implies the following information about the partitioning of *orders*:

- Number of partitions in the first, and only, level =  $d_1 = 11$
- Total number of combined partitions =  $d_1 = 11$
- Combined partitioning expression =  $p_1$

If the value of *o\_custkey* is 15, then the following additional information is implied:

- Partition number for level 1 = PARTITION#L1 =  $p_1(15) = 2$ .
- PARTITION#L2 through PARTITION#L15 are all 0.
- Combined partition number = PARTITION =  $p_1(15) = 2$ .

Value of o_custkey	Result of the RANGE_N function Value of PARTITION Value of PARTITION#L1
0 - 9	1
10 - 19	2
20 - 29	3
30 - 39	4
40 - 49	5
50 - 59	6
60 - 69	7
70 - 79	8
80 - 89	9
90 - 99	10
100	11

**Stage 2:** Second single-level partitioning of the *orders* table.

Suppose you then submit the following ALTER TABLE request on *orders*:

```
ALTER TABLE orders
MODIFY PRIMARY INDEX
DROP RANGE BETWEEN 0
AND      9
EACH     10;
```

This alters the partitioning expression to:

```
RANGE_N(o_custkey BETWEEN 10
AND      100
EACH     10);
```

In other words, the table definition after you have performed the ALTER TABLE request is as follows:

```
CREATE TABLE orders (
    o_orderkey INTEGER NOT NULL,
    o_custkey   INTEGER)
PRIMARY INDEX (o_orderkey)
PARTITION BY RANGE_N(o_custkey BETWEEN 10
AND      100
EACH     10); /* p1 */
```

This changes the information implied by the initial table definition about the partitioning of *orders* as follows:

- Number of partitions in the first, and only, level =  $d_1 = 10$
- Total number of combined partitions =  $d_1 = 10$
- Combined partitioning expression =  $p_1$

Now if  $o\_custkey$  is 15, the following additional information is implied:

- Partition number for level 1 = PARTITION#L1 =  $p_1(15) = 1$ .
- PARTITION#L2 through PARTITION#L15 are all 0.
- Combined partition number = PARTITION =  $p_1(15) = 1$ .

The following table indicates the new partition numbers for the various defined ranges for  $o\_custkey$ :

Value of $o\_custkey$	Result of the new RANGE_N function Value of PARTITION Value of PARTITION#L1
10 - 19	1
20 - 29	2
30 - 39	3
40 - 49	4
50 - 59	5
60 - 69	6
70 - 79	7
80 - 89	8
90 - 99	9
100	10

**Stage 3:** Third single-level partitioning of the *orders* table.

Suppose you submit the following ALTER TABLE request on *orders*:

```
ALTER TABLE orders
  MODIFY PRIMARY INDEX
    ADD RANGE BETWEEN 5
      AND 9
      EACH 1;
```

This alters the partitioning expression to"

```
RANGE_N(o_custkey BETWEEN 5
          AND 9
          EACH 1, 10 AND 100
          EACH 10);
```

In other words, the table definition after you have performed the ALTER TABLE request is as follows:

```

CREATE TABLE orders (
    o_orderkey INTEGER NOT NULL,
    o_custkey INTEGER)
PRIMARY INDEX (o_orderkey)
PARTITION BY
RANGE_N(o_custkey BETWEEN 5
          AND      9
          EACH     1, 10 AND 100
          EACH     10); /* p1 */

```

This changes the information implied by the second table definition about the partitioning of *orders* as follows:

- Number of partitions in the first, and only, level =  $d_1 = 15$
- Total number of combined partitions =  $d_1 = 15$
- Combined partitioning expression =  $p_1$

Now if *o\_custkey* is 15, the following additional information is implied:

- Partition number for level 1 = PARTITION#L1 =  $p_1(15) = 6$ .
- PARTITION#L2 through PARTITION#L15 are all 0.
- Combined partition number = PARTITION =  $p_1(15) = 6$ .

The following table indicates the new partition numbers for the various defined ranges for *o\_custkey*.

Value of <i>o_custkey</i>	Result of the new RANGE_N function Value of PARTITION Value of PARTITION#L1
5	1
6	2
7	3
8	4
9	5
10 - 19	6
20 - 29	7
30 - 39	8
40 - 49	9
50 - 59	10
60 - 69	11
70 - 79	12
80 - 89	13
90 - 99	14

Value of o_custkey	Result of the new RANGE_N function Value of PARTITION Value of PARTITION#L1
100	15

Note that this table describes the PARTITION#Ln values for a table having a 2-byte ROWID. If the table had an 8-byte ROWID, there would be as many as 62 partitions.

#### Stage 4: Fourth single-level partitioning of the *orders* table.

Suppose you submit the following ALTER TABLE request on *orders*:

```
ALTER TABLE orders
    MODIFY PRIMARY INDEX
        ADD RANGE BETWEEN 0
            AND      4
            EACH     2;
```

This alters the partitioning expression to be:

```
RANGE_N(o_custkey BETWEEN 0
          AND      4
          EACH     2, 5 AND 9
          EACH     1, 10 AND 100
          EACH     10);
```

In other words, the table definition after you have performed the ALTER TABLE request is as follows:

```
CREATE TABLE orders (
    o_orderkey INTEGER NOT NULL,
    o_custkey INTEGER)
PRIMARY INDEX (o_orderkey)
PARTITION BY RANGE_N(o_custkey BETWEEN 0
                      AND      4
                      EACH     2, 5 AND 9
                      EACH     1, 10 AND 100
                      EACH     10);
```

This changes the information implied by the third table definition about the partitioning of *orders* as follows:

- Number of partitions in the first, and only, level =  $d_1 = 18$
- Total number of combined partitions =  $d_1 = 18$
- Combined partitioning expression =  $p_1$

Now if *o\_custkey* is 15, the following additional information is implied.

- Partition number for level 1 = PARTITION#L1 =  $p_1(15) = 9$ .
- PARTITION#L2 through PARTITION#L15 are all 0.
- Combined partition number = PARTITION =  $p_1(15) = 9$ .

The following table indicates the new partition numbers for the various defined ranges for *o\_custkey*.

Value of o_custkey	Result of the new RANGE_N function Value of PARTITION Value of PARTITION#L1
0 - 1	1
2 - 3	2
4	3
5	4
6	5
7	6
8	7
9	8
10 - 19	9
20 - 29	10
30 - 39	11
40 - 49	12
50 - 59	13
60 - 69	14
70 - 79	15
80 - 89	16
90 - 99	17
100	18

## Single-Level Partitioning Case Studies

This topic presents several case study scenarios to evaluate various approaches to row partitioning a primary index on one level in such a way that optimal benefit is realized.

- The first scenario (see “[Scenario 1](#)” on page 427) is set in a retail sales environment. The effects of row partitioning a sales table on months is examined from several different perspectives.
- The second scenario (see “[Scenario 2](#)” on page 431) is a continuation of the first and attempts to boost the benefits of using a partitioned table further by partitioning on days rather than months. The concept is to investigate how the number of row partitions affects query workload performance.

- The third scenario (see “[Scenario 3](#)” on page 432) is set in a telecommunications environment. The effects of partitioning a telephone call tracking table on call date and phone number are compared and contrasted for the same query workloads.
- The final scenario (see “[Scenario 4](#)” on page 434) illustrates how the decision whether to use row partitioning is often multidetermined, requiring a solution based on numerous, carefully weighted factors.

## Determining an Optimal Partitioning Scheme

An optimal partitioning scheme for any table depends on the anticipated query mix. Use your extended logical data model as the starting point for making the decision, but you should always test a few different scenarios to ensure the best partitioning scheme for your particular application and system configuration.

Note that these scenarios are intended to make specific points about partitioning and are not meant to be taken as industry-specific partitioning recommendations.

The following tables list the results of a test prototype performed to examine the potential improvements of using partitioning instead of not using partitioning for the same table.

Test Description	Reduction in Elapsed Time (percent)
Select all rows with a particular value of the partitioning column (200 partitions with roughly the same number of rows each).	98
Select a month of activity from one partition containing six months of data (11 years of data contained in 40 partitions of unequal size).	97
Delete all rows with a particular value of the partitioning column (200 partitions of equal size).	99
Update one column in each row that has a particular value for the partitioning column (200 partitions of equal size).	98

Test Description	Improvement
Teradata Parallel Transporter update operation to insert a number of rows equal to 1% of the table cardinality into one partition out of 200 total partitions.	More than 10 times faster.
Teradata Parallel Transporter update operation to insert a number of rows equal to 1% of the table cardinality into one partition out of 200 total partitions with one NUSI defined on the table.	More than 6 times faster.

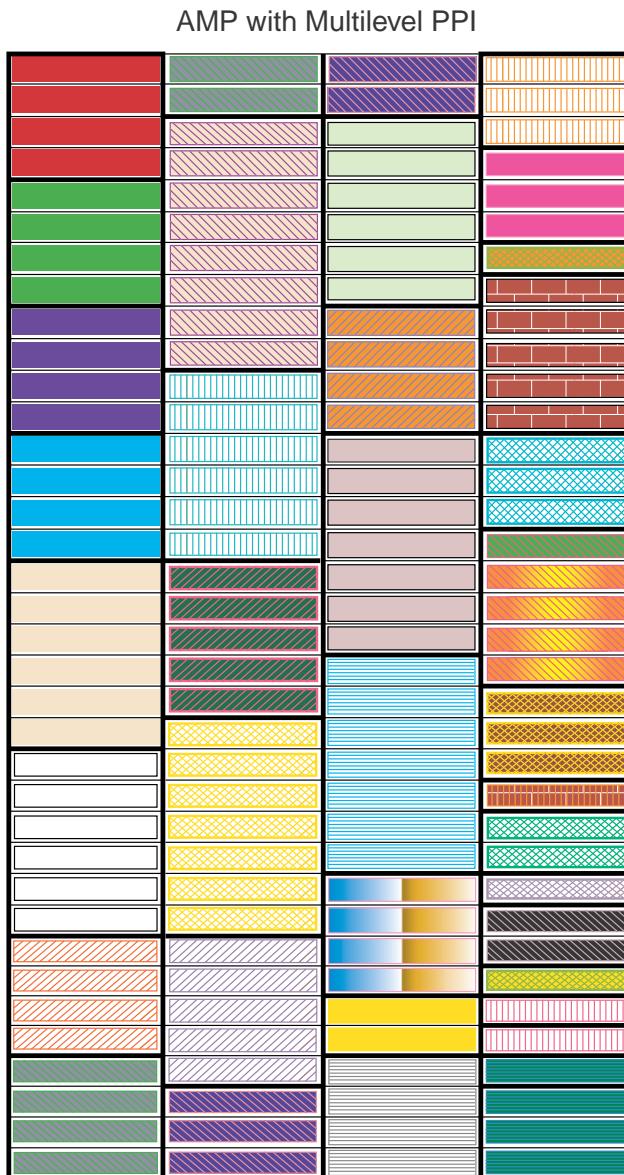
## Multilevel Partitioning

### Row Partitioning Across AMPs for 3 Partitioning Levels

The following graphic, which is best viewed on a color monitor and best printed on a color printer, shows the partitioning of rows based on a table that has 3 partitioning levels based on the table defined in “[Detailed Multilevel Partitioning Example](#)” on page 393.

The only difference between this graphic and the data in the table for the multilevel row partitioning example is that there is only one row per combined partition number, while the graphic has anywhere between 1 and 8 rows per combined partition number.

This graphic appears again later in this chapter in support of a more detailed explanation of multilevel partitioning (see “[Detailed Multilevel Partitioning Example](#)” on page 393).



1094A078

Multilevel partitioning allows each partition at a given level to be further partitioned into subpartitions. Each partition for a level is subpartitioned the same per a partitioning expression or by column partitioning defined for the next lower level. Rows are grouped by the combined partition number, and within a group are ordered first by hash value and then, if for a partitioned primary index, by uniqueness value. The combined partition numbers are mapped 1-to-1 with internal partition numbers on which the rows on an AMP are ordered. The file system orders rows by the internal partition number, rowhash value, and then uniqueness value.

For multilevel partitioning, Teradata Database hash orders the rows within the lowest partition levels. A multilevel partitioning undertakes efficient searches by using row partition elimination at the various levels or combinations of levels. See *SQL Request and Transaction Processing* for a description of row partition elimination and its various forms.

The following list describes the various access methods that are available when multilevel partitioning is defined for a table.

- If there is an equality constraint on the primary index and there are constraints on the partitioning columns such that access is limited to a single partition at each level, access is as efficient as with a nonpartitioned table.

This is a single-AMP, single-hash access in a single subpartition at the lowest level of the partition hierarchy.

- With constraints defined on the partitioning columns, performance of a primary index access can approach the performance of a nonpartitioned primary index depending on the extent of row partition elimination that can be achieved.

This is a single-AMP, single-hash access in multiple (but not all) subpartitions at the lowest level of the partition hierarchy.

- Access by means of equality constraints on the primary index columns that does not also include all the partitioning columns, and without constraints defined on the partitioning columns, might not be as efficient as access with a nonpartitioned primary index. The efficiency of the access depends on the number of populated subpartitions at the lowest level of the row partition hierarchy.

This is a single-AMP, single-hash access in all subpartitions at the lowest level of the partition hierarchy.

- With constraints on the partitioning columns of a partitioning expression such that access is limited to a subset of, say  $n$  percent, of the partitions for that level, the scan of the data is reduced to about  $n$  percent of the time required by a full-table scan.

This is an all-AMP scan of only the non-eliminated row partitions for that level. This allows multiple access paths to a subset of the data: one for each partitioning expression.

If constraints are defined on partitioning columns for more than one of the partitioning expressions in a multilevel partitioning definition, row partition elimination can lead to even less of the data needing to be scanned.

## Partitioning CHECK Constraint for Multilevel Partitioning

A multilevel partitioned table has the following partitioning CHECK constraint, which the system stores in *DBC.TableConstraints*. Call this partitioning constraint form 3.

```
— CHECK — ( /* nn */ — partitioning_expression_1 — IS NOT NULL —①
① — AND —————— partitioning_expression_2 — IS NOT NULL —②
② —————— ) ——————
      |————— 13 —————— |
      |————— AND —————— partitioning_expression_n — IS NOT NULL —————— |
```

1094A079

where:

Syntax element ...	Specifies the ...
<i>nn</i>	number of levels, or number of partitioning expressions, in the multilevel partitioning. <i>nn</i> can range between 02 and 62, inclusive.
<i>partitioning_expression_1</i>	the first multilevel partitioning level.
<i>partitioning_expression_2</i>	the second multilevel partitioning level.
<i>partitioning_expression_n</i>	the <i>n</i> <sup>th</sup> multilevel partitioning level. Note that if the multilevel partitioning has 3 levels, there are 3 NOT NULL partitioning expressions in the implied constraint, if the multilevel partitioning has 10 levels, there are 10 NOT NULL partitioning expressions in the implied constraint, and so on.

Single-level partitioned tables have a different implied table-level partitioning CHECK constraint (see “[Single-Level Partitioning](#)” on page 362).

You can use the following query to retrieve a list of tables and join indexes that have PPIs and their partitioning constraint text.

```
SELECT DatabaseName, TableName (TITLE 'Table/Join Index Name'),  
       ConstraintText  
  FROM DBC.IndexConstraintsV  
 WHERE ConstraintType = 'Q'  
 ORDER BY DatabaseName, TableName;
```

You can use a query like the following to retrieve partitioning constraint information for each of the multilevel partitioned objects.

```
SELECT *  
  FROM DBC.TableConstraints  
 WHERE ConstraintType = 'Q'  
   AND SUBSTRING(TableCheck FROM 1 FOR 13) >= 'CHECK /*02*/'  
   AND SUBSTRING(TableCheck FROM 1 FOR 13) <= 'CHECK /*15*/';
```

The *TableCheck* column of *DBC.TableConstraints* contains the unresolved condition text for a table-level constraint check or implicit table-level constraint such as a partitioning constraint. The *ConstraintType* code for such an implicit table-level constraint is Q for an object with a partitioned primary index, where in the case of a multilevel partitioning, each of the partitioning levels for the index appears once in the order defined for the table in the text contained in *TableCheck*. See *Data Dictionary* for details.

Rows that violate this implied index CHECK constraint, including those whose partitioning expression evaluates to null, are not allowed in the table.

Assume you create the following table:

```
CREATE TABLE orders (  
    o_orderkey      INTEGER NOT NULL,  
    o_custkey       INTEGER,  
    o_orderstatus   CHARACTER(1) CASESPECIFIC,  
    o_totalprice    DECIMAL(13,2) NOT NULL,  
    o_orderdate     DATE FORMAT 'yyyy-mm-dd' NOT NULL,
```

```

o_orderpriority CHARACTER(21),
o_clerk           CHARACTER(16),
o_shipppriority   INTEGER,
o_comment          VARCHAR(79))
PRIMARY INDEX (o_orderkey)
PARTITION BY (RANGE_N(o_custkey)    BETWEEN 0
              AND 49999
              EACH 100),
              RANGE_N(o_orderdate) BETWEEN DATE '2000-01-01'
                                         AND     DATE '2006-12-31'
                                         EACH INTERVAL '1' MONTH))
UNIQUE INDEX (o_orderkey);

```

The partitioning CHECK constraint SQL text that would be stored in *DBC.TableConstraints* for this multilevel partitioned primary index is as follows.

```

CHECK /*02*/ RANGE_N(o_custkey)    BETWEEN 0
      AND 49999
      EACH 100)
      IS NOT NULL
      AND RANGE_N(o_orderdate) BETWEEN DATE '2000-01-01'
                                    AND     DATE '2006-12-31'
                                    EACH INTERVAL '1' MONTH)
      IS NOT NULL )

```

The maximum size of this partitioning CHECK constraint is 16,000 characters.

## Partitioning CHECK Constraints

A partitioned table or join index has the following partitioning CHECK constraint.

```

CHECK /*nn bb cc*/ partitioning_constraint_1
      [AND partitioning_constraint_2] ... )

```

where:

Syntax element ...	Specifies ...
<i>nn</i>	<p>the number of partitioning levels.</p> <ul style="list-style-type: none"> <li>For 2-byte partitioning, <i>nn</i> ranges between 01 and 15, inclusive.</li> <li>For 8-byte partitioning, <i>nn</i> ranges between 01 and 62, inclusive.</li> </ul>
<i>bb</i>	<p>the type of partitioning.</p> <ul style="list-style-type: none"> <li>For 2-byte partitioning, <i>bb</i> is 02.</li> <li>For 8-byte partitioning, <i>bb</i> is 08.</li> </ul>
<i>cc</i>	<p>the column partitioning level.</p> <ul style="list-style-type: none"> <li>For no column partitioning, <i>cc</i> is 00.</li> <li>Otherwise, <i>cc</i> ranges between 01 and <i>nn</i>, inclusive.</li> </ul>

Each of the partitioning constraints corresponds to a level of partitioning in the order defined for the table or join index.

The constraint specified by *partitioning\_constraint\_i* can be the partitioning constraint for any level and can represent either a row partitioning expression or a column partitioning expression.

Syntax element ...	Specifies ...
partitioning_constraint_i	When <i>partitioning_expression_i</i> is the <i>row</i> partitioning expression at level <i>i</i> , and partitioning constraint is the following.  $\text{partitioning\_expression}_i /*i \ d+a*/ \text{ IS NOT NULL}$
<i>i</i>	When <i>partitioning_expression_i</i> is the <i>column</i> partitioning expression at level <i>i</i> is the following.  $\text{PARTITION}\#L_i /*i \ d+a*/ =1$
<i>d</i>	an integer ranging between 1 and <i>nn</i> , inclusive. Leading zeros are not specified.
<i>a</i>	the number of currently defined partitions for the level. Leading zeros are not specified. For a column-partitioned level, this includes the two internal column partitions.
	the number of additional partitions that could be added (this can be 0 or <i>x</i> ). Leading zeros are not specified. <i>x</i> occurs for level 2 and higher if the partitioning constraint form is only being used because the cost profile constant PartitioningConstraintForm is set to 1 to force use of the new constraint form in all cases. If the constraint form would be used regardless of the setting of PartitioningConstraintForm or this is for level 1, <i>a</i> is an integer number.

Assume you create the following table:

```

CREATE TABLE orders (
    o_orderkey      INTEGER NOT NULL,
    o_custkey        INTEGER,
    o_orderstatus   CHARACTER(1) CASESPECIFIC,
    o_totalprice    DECIMAL(13,2) NOT NULL,
    o_orderdate     DATE FORMAT 'yyyy-mm-dd' NOT NULL,
    o_comment        VARCHAR(79))
NO PRIMARY INDEX
PARTITION BY (RANGE_N(o_custkey BETWEEN 0
                      AND 100000
                      EACH      1),
              COLUMN)
UNIQUE INDEX (o_orderkey);

```

The table-level partitioning CHECK constraint SQL text for this 8-byte partitioning is as follows:

```

CHECK /*02 08 02*/ RANGE_N(o_custkey BETWEEN 0
                           AND 100000
                           EACH      1
/*1 100001+485440633518572409*/
IS NOT NULL AND PARTITION#L2 /*2 9+10*/ =1)

```

The maximum size of this table-level CHECK constraint is 16,000 characters.

You can use the following request to retrieve the level for the column partitioning for each of the objects that have column partitioning in the system.

```
SELECT DBaseId, TVMId, ColumnPartitioningLevel
      (TITLE 'Column Partitioning Level')
   FROM DBC.TableConstraints
 WHERE ConstraintType = 'Q'
   AND ColumnPartitioningLevel >= 1
 ORDER BY DBaseId, TVMId;
```

See “[Partitioning CHECK Constraints for Single-Level Partitioning](#)” on page 362 and “[Partitioning CHECK Constraint for Multilevel Partitioning](#)” on page 374 for information about the table-level CHECK constraints that Teradata Database creates for single-level and multilevel partitioned primary index tables and join indexes.

## Row Partition Elimination With Multilevel Partitioning

The following examples taken from various industries present examples of how multilevel partitioned primary indexes and row partition elimination can greatly enhance the performance of a query workload.

### Example From the Insurance Industry

There are many cases for which row partition elimination using multiple expressions for WHERE clause predicate filtering can enhance query performance (see *SQL Request and Transaction Processing* for details about the various forms of row partition elimination). For example, consider an insurance company that frequently performs an analysis for a specific state and within a date range that constitutes a relatively small percentage of the many years of claims history in its data warehouse.

If an analysis is being performed only for claims filed in Connecticut, only for claims filed in all states in June, 2005, or only for Connecticut claims filed during the month of June, 2005, a partitioning of the data that allows elimination of all but the desired claims should deliver a dramatic performance advantage.

The following example shows how a claims table could be partitioned by a range of claim dates and subpartitioned by a range of state identifiers using multilevel partitioning.

Consider the following multilevel PPI table definition:

```
CREATE TABLE claims (
    claim_id      INTEGER NOT NULL,
    claim_date    DATE NOT NULL,
    state_id      BYTEINT NOT NULL,
    claim_info    VARCHAR(20000) NOT NULL)
PRIMARY INDEX (claim_id)
PARTITION BY (RANGE_N(claim_date) BETWEEN DATE '1999-01-01'
              AND      DATE '2005-12-31'
              EACH INTERVAL '1' MONTH),
             RANGE_N(state_id) BETWEEN 1
              AND      75
              EACH 1))
UNIQUE INDEX (claim_id);
```

Eliminating all but one month out of their many years of claims history would facilitate scanning of less than 5% of the claims history (because of business growth, there are many more recent than past claims) for satisfying the following query:

```
SELECT *
FROM claims
WHERE claim_date BETWEEN DATE '2005-06-01' AND DATE '2005-06-30';
```

Similarly, eliminating all but the Connecticut claims from the many states in which this insurance company does business would make it possible to scan less than 5% of the claims history to satisfy the following query:

```
SELECT *
FROM claims, states
WHERE claims.state_id = states.state_id
AND states.state = 'Connecticut';
```

State selectivity varies by the density of their business book. The company has more business in Connecticut than, for example Oregon and, therefore, a correspondingly larger number of claims for Connecticut. Note that the partitioning in the example specifies up to 75 *state\_id* values to allow for any of the 50 US states plus US territories and protectorates such as Puerto Rico, Guam, American Samoa, the Marshall Islands, Federated States of Micronesia, Northern Mariana Islands, Palau, and the American Virgin Islands in the future.

Combining both of these predicates for row partition elimination makes it possible to scan less than 0.5% of the claims history to satisfy the following query.

```
SELECT *
FROM claims, states
WHERE claims.state_id = states.state_id
AND states.state = 'Connecticut'
AND claim_date BETWEEN DATE '2005-06-01' AND DATE '2005-06-30';
```

For evenly distributed data, you would expect scanning of less than 0.25% of the data; however, the data for this company is not evenly distributed among states and dates, leading to a higher percentage of data to scan for the query.

Clearly, combining both predicates for row partition elimination has a significant performance advantage. Row partition elimination by both of these columns, as described, provides higher performance, more space efficiency, and more maintenance efficiency than a composite NUSI or join index for most of the queries run at this insurance company.

In fact, the performance advantage described previously could theoretically be achieved with the current single-level partitioning by using a single partitioning expression that combines both state and monthly date ranges. There are issues with using such a complex, single partitioning expression in this scenario but it is possible to do so.

The bigger problem is that a significant portion of the workload at this insurance company does not specify equality conditions on both partitioning columns. Though theoretically possible, Teradata Database is currently unable to evaluate more complex conditions with such a partitioning expression. With single-level partitioning, you would have to choose between partitioning by state or partitioning by date ranges.

If you chose partitioning by state, and a user submits a query that specifies a narrow date range without a state filter in the WHERE clause, the system does not perform row partition

elimination. At this insurance company, both state-level (and, in some cases, also for a specific date range) analysis and enterprise-level (all states, but for a specific date range) analyses are common.

The users performing state-level analysis do not want to be penalized by having their data combined with all other states, but the users performing enterprise-level analyses also want their queries to be reasonably high-performing, at least by getting date range elimination, and be easy to construct.

### **Example From the Retail Industry**

An example for a large retailer is similar to the insurance example, substituting division for state. In this case, the differences in size of partitions are even more exaggerated. For this company, division 1 is all of the United States, and represents greater than 85% of the entire business. But there are a number of divisions in other countries. The retailer currently replicates the data model for each country in order to provide reasonably efficient access to the data for an individual country. However, this is difficult to maintain, and limits the ability to do analyses across the entire company.

With multilevel partitioning, all the data can be stored in the same table, and analyses can be performed efficiently across either all of the data, or with subsets of the data. This example is also applicable to manufacturers with multiple divisions and date associated data.

### **Example From the Telecommunications Industry**

Now consider the telecommunications industry. An example might be providing access to corporate billing information while also using the data for wider analysis. The difference is that there are many more customers than there are states or divisions. For example, suppose the company wants to start with 3,000 business customers, but they expect that number to grow considerably. This in turn results in more finely grained partitions. It also puts a great deal more pressure on the number of partitions and, therefore, limits choices in the granularity of date ranges or the next level partitioning.

You can use multilevel partitioning to improve query performance via row partition elimination, either at each of the partition levels or by combining all of them. Multilevel partitioning provides multiple access paths to the rows in the base table. As with other indexes, the Optimizer determines if the index is usable for a query and, if usable, whether its use provides the estimated least costly plan for executing the query.

You create multilevel partitioning by specifying two or more partitioning expressions, where each expression must be defined using either a RANGE\_N function or a CASE\_N function exclusively. The system combines the individual partitioning expressions internally into a single partitioning expression that defines how the data is partitioned on an AMP.

For example, assume there are three partitioning expressions,  $p_1$ ,  $p_2$ , and  $p_3$ , defined as follows:

The first partitioning expression is the highest level partitioning. Within each of those partitions, the second partitioning expression defines how each of the highest-level partitions is subpartitioned. Within each of those second-level partitions, the third-level partitioning expression defines how each of the second level partitions is subpartitioned.

PARTITION BY (p<sub>1</sub>, p<sub>2</sub>, p<sub>3</sub>)

Within each of these lowest level partitions, rows are ordered by the row hash value of their primary index and their assigned uniqueness value.

Assume that the number of partitions defined at each level is d<sub>1</sub>, d<sub>2</sub>, and d<sub>3</sub>, respectively. Multilevel partitioning organizes the rows of a table in a similar manner as the single-level partitioning expression (referred to as the combined partitioning expression) defined as follows:

$$\text{Combined partitioning expression} = ((p_1 - 1) \times dd_1) + ((p_2 - 1) \times dd_2) + p_3$$

where:

Equation element ...	Specifies ...
d <sub>1</sub>	the number of partitions defined at level 1.
d <sub>2</sub>	the number of partitions defined at level 2.
d <sub>3</sub>	the number of partitions defined at level 3.
p <sub>1</sub>	partitioning expression 1.
p <sub>2</sub>	partitioning expression 2.
p <sub>3</sub>	partitioning expression 3.
dd <sub>1</sub>	d <sub>2</sub> * d <sub>3</sub>
dd <sub>2</sub>	d <sub>3</sub>

The product of d<sub>1</sub> \* d<sub>2</sub> \* d<sub>3</sub> cannot exceed 65,535, nor can it be less than 8. This limit restricts the number of levels and the number of partitions that you can define at each level.

Advantages of multilevel partitioning include the following:

- Simplicity of the partitioning specification.
- Partitioning expressions can be validated when they are created. For example, d<sub>1</sub> \* d<sub>2</sub> \* d<sub>3</sub> can be validated to be less than or equal to 65,535, and d<sub>1</sub>, d<sub>2</sub>, and d<sub>3</sub> are each greater than or equal to two.

For the above single-level partitioning expression, Teradata Database cannot enforce that d<sub>1</sub>, d<sub>2</sub>, and d<sub>3</sub> are the same as the corresponding number of partitions defined for p<sub>1</sub>, p<sub>2</sub>, and p<sub>3</sub>, respectively, at creation time; it can only enforce that the final result of the expression for a row must be between 1 and 65,535, inclusively, when rows are inserted or updated.

- Multilevel partitioning allows for efficient altering of partitioning expressions in some common cases (see “ALTER TABLE” in *SQL Data Definition Language*).

Note that the number of levels of partitioning cannot exceed 15 for 2-byte partitioning or 62 for 8-byte partitioning because each level must define at least two partitions. This is because  $2^{16} = 65,536$  and  $2^{63} = 9,223,372,036,854,775,808$ , both of which exceed the maximum number of partitions that can be defined for a given partitioning by 1. The number of levels of partitioning might be further restricted by other limits such as the maximum size of the table header, data dictionary entry sizes, and so on.

## Importance of Partition Order for Specifying Partitioning Expressions

The specification order of partitioning expressions can be important for multilevel partitioning. The system maps multilevel partitioning expressions into a single-level combined partitioning expression. It then maps the resulting combined partition number 1-to-1 to an internal partition number. Rows are in logical RowID order, where a RowID consists of an internal partition number, a row hash value, and a row uniqueness value (RowIDs are identical for single-level and multilevel partitioning).

This implies a partial physical ordering based on how the file system manages the data blocks and cylinders, though this is not a strict relationship. This physical ordering maintains the ordering of partitions (except for a possible wraparound of internal numbers at one point in the internal number sequence for each level). Multilevel partitioning expressions are analogous with a single-level partitioning expression that is identical to the *combined* partitioning expression for multilevel partitioning, at least in terms of expressing how rows are logically ordered by the file system.

There are several implications of this ordering.

- Usage

You can alter a partition level to change the number of partitions at that level to within 1 and the maximum number of partitions defined for that level when the table is populated with rows.

- Query performance

Row partition elimination at the lowest levels can increase overhead because of the frequent need to skip to the next internal partition to be read. This is because a partition at a lower level is split among the partitions at higher levels in the partition hierarchy.

At higher levels in the partition hierarchy, there are more contiguous internal partitions to scan and skip.

- Bulk data load performance

If a load is order-based on one of the partitioning columns, having that partitioning column at the highest level in the partition hierarchy can improve load performance because those partitions are contiguous. For example, a date-based partition with daily data loads might benefit from having the date-based partitioning at the first level.

You define the ordering of the partitioning expressions in your CREATE TABLE SQL text, and that ordering implies the logically ordering by RowID. Because the partitions at each level are

distributed among the partitions of the next higher level in the hierarchy, scanning a partition at a certain level requires skipping some internal partitions.

If the number of rows in each internal partition is large, then skipping to the next internal partition to be read incurs relatively little overhead.

If the system reads or skips only a few rows for each internal partition, performance might be somewhat worse than a full-table scan (or even significantly worse if a full-table scan were able to get more benefit out of block read-ahead or cylinder reads). You can exploit this ordering for a locality of reference performance benefit (meaning that by increasing the probability that certain partitions fall in the same cylinder by having their partitioning expression at a higher level, you can greatly enhance the performance of the scan).

Partition expression order does *not* affect the ability to eliminate partitions, but *does* affect the efficiency of a partition scan. As a general rule, this should not be a concern if there are many rows, which implies multiple data blocks, in each of the partitions.

Consider the following 2 cases.

- For a table with 65,535 combined partitions, the maximum number of combined partitions per partitioned primary index is  $6.5535 \times 10^9$  (*billion*) rows per AMP, 100 byte rows, and 50 KB data blocks, and assuming an equal distribution of rows among the partitions, each combined partition spans 200 to 201 data blocks.

In this case, skipping over internal partitions should not incur significant overhead. Either all, or nearly all, of the rows in the data blocks read qualify, and the system skips at least 199 data blocks between each set of data blocks read.

- If the table has only  $6.5535 \times 10^6$  (*million*) rows per AMP, each combined partition has about 0.2 data blocks. In the worst case, where the system eliminates only every other partition at the lowest level, every data block is read, and a full-table scan would be more efficient.

If there are 5 partitions at the lowest level, and the system can eliminate 4 out of 5 partitions, it still must read every data block.

If there are 6 partitions at the lowest level, and the system can eliminate 5 out of the 6, it can skip some data blocks, but possibly not enough to overcome the overhead burden, so this might not be more efficient than a full-table scan.

With a large number of partitions at the lowest level, and a large number of eliminated partitions at the lowest level, the system can probably skip enough more data blocks that the overhead burden can be overcome, and the operation might be more efficient than a full-table scan.

This second case is somewhat artificial, and probably not a good use of multilevel partitioning. Instead of the demonstrated case, you should consider an alternative partitioning that results in multiple data blocks for each internal partition.

A partitioning scheme that defines fewer levels of partitioning and fewer partitions per level, where the lowest level has the greatest number of partitions, ensures that there are more rows per combined partition, and would be far more useful. For example, if one level was initially partitioned in intervals of one day, changing the interval to one week or one month might be better.

Row partition elimination at the lower partition levels of a row partition hierarchy requires more skipping, which can both cause more I/O operations and increase the CPU path length.

To achieve optimal performance, you should specify a partitioning expression that is more likely to evoke row partition elimination for queries at a higher level and specify those expressions that are not as likely to evoke row partition elimination either at a lower level, or not at all. You should also consider specifying the row partitioning expression with the greatest number of row partitions at the lowest level.

As previously noted, the order of the row partitioning expressions might not be a significant concern if there are many data blocks per combined partition.

You can use the following query to find the average number of rows per populated combined row partition.

```
SELECT AVG(pc)
  FROM (SELECT COUNT(*) AS pc
        FROM t
       GROUP BY PARTITION) AS pt;
```

Assuming the average block size is  $b$  and the row size is  $r$ , you can use the following query to find the average number of data blocks per populated combined row partition. The ideal number of data blocks per populated combined row partition is 10 or more.

```
USING (b FLOAT, r FLOAT)
SELECT (:r / :b) * AVG(pc)
  FROM (SELECT COUNT(*) AS pc
        FROM t
       GROUP BY PARTITION) AS pt;
```

## Different Multilevel Row Partitioning of the Same Table

The purpose of the following multipart example is to demonstrate the various properties of different multilevel row partitionings of the same data.

**Stage 1:** First multilevel row partitioning of the *orders* table.

```
CREATE TABLE orders (
    o_orderkey INTEGER NOT NULL,
    o_custkey1 INTEGER,
    o_custkey2 INTEGER)
PRIMARY INDEX (o_orderkey)
PARTITION BY (RANGE_N(o_custkey1) BETWEEN 0
              AND      50
              EACH     10), /* p1 */
              RANGE_N(o_custkey2) BETWEEN 0
              AND      100
              EACH     10)/* p2 */;
```

This definition implies the following information about the row partitioning of *orders*.

- Number of row partitions in the first level, or highest, level =  $d_1 = 6$
- Number of row partitions in second, or lowest, level =  $d_2 = 11$
- Total number of combined partitions =  $d_1 * d_2 = 6 * 11 = 66$
- Combined partitioning expression =  $(p_1 - 1) * d_2 + p_2 = (p_1 - 1) * 11 + p_2$

Now if  $o\_custkey1$  is 15 and  $o\_custkey2$  is 55, the following additional information is implied:

- Row partition number for level 1 = PARTITION#L1 =  $p1(15) = 2$ .
- Row partition number for level 2 = PARTITION#L2 =  $p2(55) = 6$ .
- PARTITION#L3 through PARTITION#L62 are all 0.
- Combined partition number = PARTITION =  $(2-1)*11 + 6 = 17$ .

The following table indicates the row partition numbers for the various defined ranges for  $o\_custkey1$  and  $o\_custkey2$ .

Value of $o\_custkey1$	Value of $o\_custkey2$	Value of $(p1-1)*d2+p2$ PARTITION	Value of $p1$ PARTITION#L1	Value of $p2$ PARTITION#L2
0 - 9	0 - 9	1	1	1
	10 - 19	2		2
	20 - 29	3		3
	30 - 39	4		4
	40 - 49	5		5
	50 - 59	6		6
	60 - 69	7		7
	70 - 79	8		8
	80 - 89	9		9
	90 - 99	10		10
	100	11		11
10 - 19	0 - 9	12	2	1
	10 - 19	13		2
	20 - 29	14		3
	30 - 39	15		4
	40 - 49	16		5
	50 - 59	17		6
	60 - 69	18		7
	70 - 79	19		8
	80 - 89	20		9
	90 - 99	21		10
	100	22		11

Chapter 9: Primary Indexes and NoPI Objects  
 Multilevel Partitioning

Value of o_custkey1	Value of o_custkey2	Value of $(p1-1)*d2+p2$ PARTITION	Value of p1 PARTITION#L1	Value of p2 PARTITION#L2
20 - 29	0 - 9	23	3	1
	10 - 19	24		2
	20 - 29	25		3
	30 - 39	26		4
	40 - 49	27		5
	50 - 59	28		6
	60 - 69	29		7
	70 - 79	30		8
	80 - 89	31		9
	90 - 99	32		10
	100	33		11
30 - 39	0 - 9	34	4	1
	10 - 19	35		2
	20 - 29	36		3
	30 - 39	37		4
	40 - 49	38		5
	50 - 59	39		6
	60 - 69	40		7
	70 - 79	41		8
	80 - 89	42		9
	90 - 99	43		10
	100	44		11

Value of o_custkey1	Value of o_custkey2	Value of (p1-1)*d2+p2 PARTITION	Value of p1 PARTITION#L1	Value of p2 PARTITION#L2
40 - 49	0 - 9	45	5	1
	10 - 19	46		2
	20 - 29	47		3
	30 - 39	48		4
	40 - 49	49		5
	50 - 59	50		6
	60 - 69	51		7
	70 - 79	52		8
	80 - 89	53		9
	90 - 99	54		10
	100	55		11
50	0 - 9	56	6	1
	10 - 19	57		2
	20 - 29	58		3
	30 - 39	59		4
	40 - 49	60		5
	50 - 59	61		6
	60 - 69	62		7
	70 - 79	63		8
	80 - 89	64		9
	90 - 99	65		10
	100	66		11

The following case examples demonstrate some of the properties of this multilevel partitioning scheme.

- ```
SELECT *
FROM orders
WHERE custkey1 = 15;
```

In this case, there might be qualifying rows in the row partitions where the combined row partition numbers =  $(2 - 1)*11 + (1 to 11) = 11 + (1 to 11) = 12$  to 22.

- ```
SELECT *
FROM orders
WHERE (o_custkey1 = 15
```

```
OR      o_custkey1 = 25)
AND      custkey2 BETWEEN 20 AND 50;
```

In this case, there might be qualifying rows in the row partitions where the combined partition numbers are in (14 to 17, 25 to 28).

- ```
SELECT *
FROM orders
WHERE o_custkey2 BETWEEN 42 AND 47;
```

In this case, there might be qualifying rows in the row partitions where the combined partition numbers are in (5, 16, 27, 38, 49, 60).

### Stage 2:

Suppose you submit the following ALTER TABLE request on *orders*.

```
ALTER TABLE orders
MODIFY PRIMARY INDEX
  DROP RANGE BETWEEN 0
    AND 9
    EACH 10
  ADD RANGE BETWEEN 51
    AND 70
    EACH 10,
  DROP RANGE BETWEEN 100
    AND 100
  ADD RANGE -100 TO -2;
```

This alters the row partitioning expressions to be the following expressions.

```
RANGE_N(o_custkey1) BETWEEN 10
  AND 50
  EACH 10, 51 AND 70
  EACH 10),
RANGE_N(o_custkey2) BETWEEN -100
  AND -2, 0
  AND 99
  EACH 10)
```

In other words, the table definition after you have performed the ALTER TABLE request is as follows.

```
CREATE TABLE orders (
  o_orderkey INTEGER NOT NULL,
  o_custkey1 INTEGER,
  o_custkey2 INTEGER)
PRIMARY INDEX (o_orderkey)
PARTITION BY (RANGE_N(o_custkey1) BETWEEN 10
  AND 50
  EACH 10, /* p1 */
  51
  AND 70
  EACH 10),
  RANGE_N(o_custkey2) BETWEEN -100
  AND -2, 0
  AND 99
  EACH 10) /* p2 */;
```

This changes the information implied by the initial table definition about the row partitioning of *orders* as follows.

- Number of partitions in the first, and highest, level =  $d1 = 7$
- Number of partitions in the second, and lowest, level =  $d2 = 11$
- Total number of combined partitions =  $d1 * d2 = 7 * 11 = 77$
- Combined partitioning expression =  $(p1 - 1) * d2 + p2 = (p1 - 1) * 11 + p2$

Now if  $o\_custkey1$  is 15 and  $o\_custkey2$  is 55, the following additional information is implied.

- Partition number for level 1 = PARTITION#L1 =  $p1(15) = 1$ .
- Partition number for level 2 = PARTITION#L2 =  $p2(55) = 7$ .
- PARTITION#L3 through PARTITION#L15 are all 0.
- Combined partition number = PARTITION =  $(1-1)*11 + 7 = 7$ .

The following table indicates the row partition numbers for the various defined ranges for  $o\_custkey1$  and  $o\_custkey2$ .

| Value of o_custkey1 | Value of o_custkey2 | Value of $(p1-1)*d2+p2$<br>PARTITION | Value of p1<br>PARTITION#L1 | Value of p2<br>PARTITION#L2 |
|---------------------|---------------------|--------------------------------------|-----------------------------|-----------------------------|
| 10 - 19             | -100 - -2           | 1                                    | 1                           | 1                           |
|                     | 0 - 9               | 2                                    |                             | 2                           |
|                     | 10 - 19             | 3                                    |                             | 3                           |
|                     | 20 - 29             | 4                                    |                             | 4                           |
|                     | 30 - 39             | 5                                    |                             | 5                           |
|                     | 40 - 49             | 6                                    |                             | 6                           |
|                     | 50 - 59             | 7                                    |                             | 7                           |
|                     | 60 - 69             | 8                                    |                             | 8                           |
|                     | 70 - 79             | 9                                    |                             | 9                           |
|                     | 80 - 89             | 10                                   |                             | 10                          |
|                     | 90 - 99             | 11                                   |                             | 11                          |

| Value of o_custkey1 | Value of o_custkey2 | Value of $(p1-1)*d2+p2$<br>PARTITION | Value of p1<br>PARTITION#L1 | Value of p2<br>PARTITION#L2 |
|---------------------|---------------------|--------------------------------------|-----------------------------|-----------------------------|
| 20 - 29             | -100 - -2           | 12                                   | 2                           | 1                           |
|                     | 0 - 9               | 13                                   |                             | 2                           |
|                     | 10 - 19             | 14                                   |                             | 3                           |
|                     | 20 - 29             | 15                                   |                             | 4                           |
|                     | 30 - 39             | 16                                   |                             | 5                           |
|                     | 40 - 49             | 17                                   |                             | 6                           |
|                     | 50 - 59             | 18                                   |                             | 7                           |
|                     | 60 - 69             | 19                                   |                             | 8                           |
|                     | 70 - 79             | 20                                   |                             | 9                           |
|                     | 80 - 89             | 21                                   |                             | 10                          |
|                     | 90 - 99             | 22                                   |                             | 11                          |
| 30 - 39             | -100 - -2           | 23                                   | 3                           | 1                           |
|                     | 0 - 9               | 24                                   |                             | 2                           |
|                     | 10 - 19             | 25                                   |                             | 3                           |
|                     | 20 - 29             | 26                                   |                             | 4                           |
|                     | 30 - 39             | 27                                   |                             | 5                           |
|                     | 40 - 49             | 28                                   |                             | 6                           |
|                     | 50 - 59             | 29                                   |                             | 7                           |
|                     | 60 - 69             | 30                                   |                             | 8                           |
|                     | 70 - 79             | 31                                   |                             | 9                           |
|                     | 80 - 89             | 32                                   |                             | 10                          |
|                     | 90 - 99             | 33                                   |                             | 11                          |

| Value of o_custkey1 | Value of o_custkey2 | Value of $(p1-1)*d2+p2$<br>PARTITION | Value of p1<br>PARTITION#L1 | Value of p2<br>PARTITION#L2 |
|---------------------|---------------------|--------------------------------------|-----------------------------|-----------------------------|
| 40 - 49             | -100 - -2           | 34                                   | 4                           | 1                           |
|                     | 0 - 9               | 35                                   |                             | 2                           |
|                     | 10 - 19             | 36                                   |                             | 3                           |
|                     | 20 - 29             | 37                                   |                             | 4                           |
|                     | 30 - 39             | 38                                   |                             | 5                           |
|                     | 40 - 49             | 39                                   |                             | 6                           |
|                     | 50 - 59             | 40                                   |                             | 7                           |
|                     | 60 - 69             | 41                                   |                             | 8                           |
|                     | 70 - 79             | 42                                   |                             | 9                           |
|                     | 80 - 89             | 43                                   |                             | 10                          |
|                     | 90 - 99             | 44                                   |                             | 11                          |
| 50                  | -100 - -2           | 45                                   | 5                           | 1                           |
|                     | 0 - 9               | 46                                   |                             | 2                           |
|                     | 10 - 19             | 47                                   |                             | 3                           |
|                     | 20 - 29             | 48                                   |                             | 4                           |
|                     | 30 - 39             | 49                                   |                             | 5                           |
|                     | 40 - 49             | 50                                   |                             | 6                           |
|                     | 50 - 59             | 51                                   |                             | 7                           |
|                     | 60 - 69             | 52                                   |                             | 8                           |
|                     | 70 - 79             | 53                                   |                             | 9                           |
|                     | 80 - 89             | 54                                   |                             | 10                          |
|                     | 90 - 99             | 55                                   |                             | 11                          |

| Value of o_custkey1 | Value of o_custkey2 | Value of $(p1-1)*d2+p2$<br>PARTITION | Value of p1<br>PARTITION#L1 | Value of p2<br>PARTITION#L2 |
|---------------------|---------------------|--------------------------------------|-----------------------------|-----------------------------|
| 51 - 60             | -100 - -2           | 56                                   | 6                           | 1                           |
|                     | 0 - 9               | 57                                   |                             | 2                           |
|                     | 10 - 19             | 58                                   |                             | 3                           |
|                     | 20 - 29             | 59                                   |                             | 4                           |
|                     | 30 - 39             | 60                                   |                             | 5                           |
|                     | 40 - 49             | 61                                   |                             | 6                           |
|                     | 50 - 59             | 62                                   |                             | 7                           |
|                     | 60 - 69             | 63                                   |                             | 8                           |
|                     | 70 - 79             | 64                                   |                             | 9                           |
|                     | 80 - 89             | 65                                   |                             | 10                          |
|                     | 90 - 99             | 66                                   |                             | 11                          |
| 61 - 70             | -100 - -2           | 67                                   | 7                           | 1                           |
|                     | 0 - 9               | 68                                   |                             | 2                           |
|                     | 10 - 19             | 69                                   |                             | 3                           |
|                     | 20 - 29             | 70                                   |                             | 4                           |
|                     | 30 - 39             | 71                                   |                             | 5                           |
|                     | 40 - 49             | 72                                   |                             | 6                           |
|                     | 50 - 59             | 73                                   |                             | 7                           |
|                     | 60 - 69             | 74                                   |                             | 8                           |
|                     | 70 - 79             | 75                                   |                             | 9                           |
|                     | 80 - 89             | 76                                   |                             | 10                          |
|                     | 90 - 99             | 77                                   |                             | 11                          |

The following cases provide examples of how multilevel row partitioning might be useful.

- ```
SELECT *
FROM orders
WHERE o_custkey1 = 15;
```

In this case, there might be qualifying rows in the row partitions where the combined partition numbers are in the range  $(1-1)*11 + (1 \text{ TO } 11) = 1 \text{ TO } 11$ .

- ```
SELECT *
FROM orders
WHERE (o_custkey1 = 15
```

```
OR o_custkey1 = 25)
AND o_custkey2 BETWEEN 20 AND 50;
```

In this case, there might be qualifying rows in the row partitions where the combined partition numbers are in the ranges 4 TO 7 and 15 TO 18.

- ```
SELECT *
FROM orders
WHERE o_custkey2 BETWEEN 42 AND 47;
```

In this case, there might be qualifying rows in the row partitions where the combined partition number is any of the following.

- 6
- 17
- 28
- 39
- 50
- 61
- 72

## Detailed Multilevel Partitioning Example

### Multilevel Partitioned Sales Table With 3 Levels of Row Partitioning

The following CREATE TABLE request defines a table with 3 levels of row partitioning.

```
CREATE TABLE sales (
    storeid      INTEGER NOT NULL,
    productid    INTEGER NOT NULL,
    salesdate    DATE FORMAT 'YYYY-MM-DD' NOT NULL,
    totalrevenue DECIMAL(13,2),
    totalsold    INTEGER,
    note         VARCHAR(256))
UNIQUE PRIMARY INDEX (storeid, productid, salesdate)
PARTITION BY (RANGE_N(salesdate) BETWEEN DATE '2003-01-01'
              AND DATE '2005-12-31'
              EACH INTERVAL '1' YEAR),
             RANGE_N(storeid) BETWEEN 1
              AND 300
              EACH 100),
             RANGE_N(productid) BETWEEN 1
              AND 400
              EACH 100));
```

### Combined Partitioning Expression for the Sales Table

The corresponding combined partitioning expression is the following.

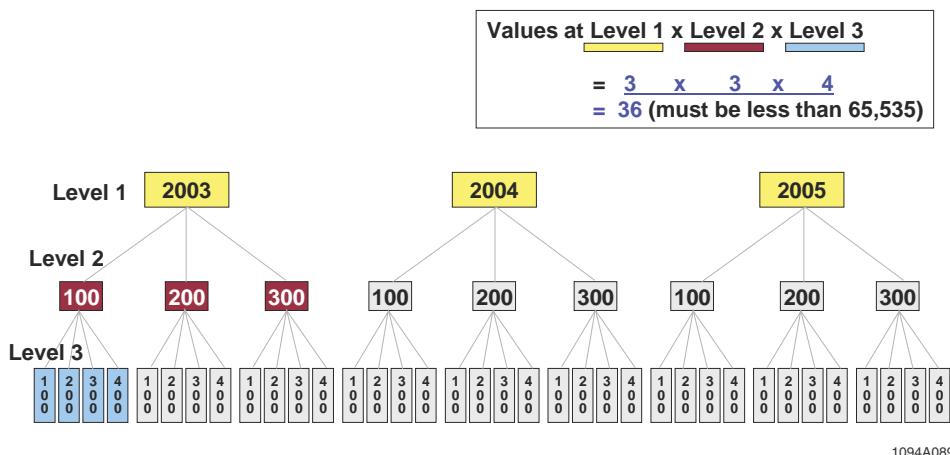
```
(RANGE_N(salesdate) BETWEEN DATE '2003-01-01'
              AND DATE '2005-12-31'
              EACH INTERVAL '1' YEAR)-1)*12+
(RANGE_N(storeid) BETWEEN 1
```

```

        AND      300
        EACH    100)-1)*4+
(RANGE_N(productid BETWEEN 1
        AND     400
        EACH    100)
    
```

There are 3 row partitions for both the first level and second levels, and 4 row partitions for the third level. Therefore, the total number of combined partitions is the product of  $3 \times 3 \times 4$ , or 36.

The following diagram indicates the logical hierarchy of the three row partitioning levels for sales (65,535 assumes 2-byte partitioning):



## How Rows for the Sales Table Are Grouped on an AMP

The table on the following page shows how rows would be grouped on an AMP and, for each row, the partition number for each level (the result of the row partitioning expression for that level) and combined partition number (the result of the combined partitioning expression).

Rows are grouped by the combined partition number, and within a group are ordered by hash value first, then by uniqueness value. The Note column is truncated in all of these examples.

Note that for this example, only one sample row is shown for each combined partition number.

You can see that all the rows for a particular year are grouped together in this table; therefore, accessing those rows can be accomplished by reading only a subset of the data on each of the AMPS.

Partition Number				Color/Pattern Representing This Combined Partition in the Graphic	Sales					
Combined Partition Number	L1	L2	L3		Storeid	Productid	Salesdate	Total Revenue	Total Sold	Note
1	1	1	1		96	10	2003-04-15	4158	42	Good day
2	1	1	2		71	184	2003-07-06	1972	68	Marginal Sa
3	1	1	3		80	241	2003-11-09	3055	47	Slow day
4	1	1	4		82	363	2003-12-24	1261	13	Promotion
5	1	2	1		186	1	2003-01-11	255	17	Shelf Life
6	1	2	2		122	163	2003-06-13	2405	65	Multiple
7	1	2	3		110	234	2003-05-01	2618	77	2 for 1
8	1	2	4		187	384	2003-08-03	684	9	Buy 3, 1 Fr
9	1	3	1		280	31	2003-03-10	493	29	Rain
10	1	3	2		202	116	2003-02-17	6732	68	Cash Sale
11	1	3	3		292	272	2003-01-30	539	11	Quarterly
12	1	3	4		213	385	2003-04-29	4233	83	Promotion
13	2	1	1		76	51	2004-10-05	442	34	2 for 1
14	2	1	2		26	149	2004-01-09	365	5	Good day
15	2	1	3		57	295	2004-11-04	2684	61	Marginal Sa
16	2	1	4		87	338	2004-08-02	696	58	Slow day
17	2	2	1		108	34	2004-04-27	4218	74	2 for 1
18	2	2	2		171	143	2004-10-30	5925	79	Shelf Life
19	2	2	3		114	228	2004-05-24	2064	48	Rain
20	2	2	4		135	347	2004-08-03	110	55	Promotion
21	2	3	1		257	75	2004-09-18	3417	51	Promotion
22	2	3	2		295	191	2004-11-11	376	8	Quarterly
23	2	3	3		208	204	2004-03-17	864	36	Multiple
24	2	3	4		221	330	2004-12-15	1456	52	Buy 3, 1 Fr
25	3	1	1		39	85	2005-09-15	192	48	Cash Sale
26	3	1	2		55	112	2005-07-12	232	29	Shelf Life
27	3	1	3		76	243	2005-03-13	6696	93	2 for 1
28	3	1	4		7	309	2005-08-20	116	58	Discounted
29	3	2	1		186	44	2005-05-30	4275	75	Credit
30	3	2	2		183	167	2005-06-14	2982	42	Promotion
31	3	2	3		171	218	2005-05-22	80	8	Rain
32	3	2	4		180	389	2005-06-09	4128	96	Good day
33	3	3	1		278	61	2005-09-22	1581	51	Buy 4, 1 Fr
34	3	3	2		256	110	2005-04-02	3280	80	Sale
35	3	3	3		246	233	2005-11-29	5133	59	Discounted
36	3	3	4		251	342	2005-02-11	968	44	50% off

For example:

- All the rows with a *storeid* in the range of 101 through 200, inclusively, can be found in the row partitions for combined partition numbers 5 to 8, 17 to 20, and 29 to 32.

As a result, if you were to specify a predicate of `WHERE storeid BETWEEN 101 AND 200`, Teradata Database could use row partition elimination to scan only combined partitions 5-8, 17-20, and 29-32 because only they contain rows that evaluate to TRUE for the condition.

The row partitions and rows that are not eliminated are highlighted in cyan.

Row Partition Number				Sales					
Combined Row Partition Number	L1	L2	L3	Storeid	Productid	Salesdate	Total Revenue	Total Sold	Note
1	1	1	1	96	10	2003-04-15	4158	42	Good day
2	1	1	2	71	184	2003-07-06	1972	68	Marginal Sa
3	1	1	3	80	241	2003-11-09	3055	47	Slow day
4	1	1	4	82	363	2003-12-24	1261	13	Promotion
5	1	2	1	186	1	2003-01-11	255	17	Shelf Life
6	1	2	2	122	163	2003-06-13	2405	65	Multiple
7	1	2	3	110	234	2003-05-01	2618	77	2 for 1
8	1	2	4	187	384	2003-08-03	684	9	Buy 3, 1 Fr
9	1	3	1	280	31	2003-03-10	493	29	Rain
10	1	3	2	202	116	2003-02-17	6732	68	Cash Sale
11	1	3	3	292	272	2003-01-30	539	11	Quarterly
12	1	3	4	213	385	2003-04-29	4233	83	Promotion
13	2	1	1	76	51	2004-10-05	442	34	2 for 1
14	2	1	2	26	149	2004-01-09	365	5	Good day
15	2	1	3	57	295	2004-11-04	2684	61	Marginal Sa
16	2	1	4	87	338	2004-08-02	696	58	Slow day
17	2	2	1	108	34	2004-04-27	4218	74	2 for 1
18	2	2	2	171	143	2004-10-30	5925	79	Shelf Life
19	2	2	3	114	228	2004-05-24	2064	48	Rain
20	2	2	4	135	347	2004-08-03	110	55	Promotion
21	2	3	1	257	75	2004-09-18	3417	51	Promotion
22	2	3	2	295	191	2004-11-11	376	8	Quarterly
23	2	3	3	208	204	2004-03-17	864	36	Multiple
24	2	3	4	221	330	2004-12-15	1456	52	Buy 3, 1 Fr
25	3	1	1	39	85	2005-09-15	192	48	Cash Sale
26	3	1	2	55	112	2005-07-12	232	29	Shelf Life
27	3	1	3	76	243	2005-03-13	6696	93	2 for 1
28	3	1	4	7	309	2005-08-20	116	58	Discounted
29	3	2	1	186	44	2005-05-30	4275	75	Credit
30	3	2	2	183	167	2005-06-14	2982	42	Promotion
31	3	2	3	171	218	2005-05-22	80	8	Rain
32	3	2	4	180	389	2005-06-09	4128	96	Good day
33	3	3	1	278	61	2005-09-22	1581	51	Buy 4, 1 Fr
34	3	3	2	256	110	2005-04-02	3280	80	Sale
35	3	3	3	246	233	2005-11-29	5133	59	Discounted
36	3	3	4	251	342	2005-02-11	968	44	50% off

- All the rows with a *productid* in the range of 201 through 300, inclusively, can be found in the combined partition numbers 3, 7, 11, 15, 19, 23, 27, 31, and 35.

As a result, if you were to specify a predicate of `WHERE productid BETWEEN 201 AND 300`, Teradata Database could use row partition elimination to scan only partitions 3, 7, 11, 15, 19, 23, 27, 31, and 35 because only they contain rows that evaluate to TRUE for the condition.

The row partitions and rows that are not eliminated are highlighted in red.

Partition Number				Sales					
Combined Partition Number	L1	L2	L3	Storeid	Productid	Salesdate	Total Revenue	Total Sold	Note
1	1	1	1	96	10	2003-04-15	4158	42	Good day
2	1	1	2	71	184	2003-07-06	1972	68	Marginal Sa
3	1	1	3	80	241	2003-11-09	3055	47	Slow day
4	1	1	4	82	363	2003-12-24	1261	13	Promotion
5	1	2	1	186	1	2003-01-11	255	17	Shelf Life
6	1	2	2	122	163	2003-06-13	2405	65	Multiple
7	1	2	3	110	234	2003-05-01	2618	77	2 for 1
8	1	2	4	187	384	2003-08-03	684	9	Buy 3, 1 Fr
9	1	3	1	280	31	2003-03-10	493	29	Rain
10	1	3	2	202	116	2003-02-17	6732	68	Cash Sale
11	1	3	3	292	272	2003-01-30	539	11	Quarterly
12	1	3	4	213	385	2003-04-29	4233	83	Promotion
13	2	1	1	76	51	2004-10-05	442	34	2 for 1
14	2	1	2	26	149	2004-01-09	365	5	Good day
15	2	1	3	57	295	2004-11-04	2684	61	Marginal Sa
16	2	1	4	87	338	2004-08-02	696	58	Slow day
17	2	2	1	108	34	2004-04-27	4218	74	2 for 1
18	2	2	2	171	143	2004-10-30	5925	79	Shelf Life
19	2	2	3	114	228	2004-05-24	2064	48	Rain
20	2	2	4	135	347	2004-08-03	110	55	Promotion
21	2	3	1	257	75	2004-09-18	3417	51	Promotion
22	2	3	2	295	191	2004-11-11	376	8	Quarterly
23	2	3	3	208	204	2004-03-17	864	36	Multiple
24	2	3	4	221	330	2004-12-15	1456	52	Buy 3, 1 Fr
25	3	1	1	39	85	2005-09-15	192	48	Cash Sale
26	3	1	2	55	112	2005-07-12	232	29	Shelf Life
27	3	1	3	76	243	2005-03-13	6696	93	2 for 1
28	3	1	4	7	309	2005-08-20	116	58	Discounted
29	3	2	1	186	44	2005-05-30	4275	75	Credit
30	3	2	2	183	167	2005-06-14	2982	42	Promotion
31	3	2	3	171	218	2005-05-22	80	8	Rain
32	3	2	4	180	389	2005-06-09	4128	96	Good day
33	3	3	1	278	61	2005-09-22	1581	51	Buy 4, 1 Fr
34	3	3	2	256	110	2005-04-02	3280	80	Sale
35	3	3	3	246	233	2005-11-29	5133	59	Discounted
36	3	3	4	251	342	2005-02-11	968	44	50% off

- All the rows with a *storeid* in the range of 1 and 100, inclusively, and a *productid* in the range 301 through 400, inclusively, can be found in the combined partitions for combined partition numbers 4, 16, and 28.

As a result, if you were to specify a predicate of `WHERE storeid BETWEEN 1 AND 100 AND productid BETWEEN 301 AND 400`, Teradata Database could use row partition elimination to scan only combined partitions 4, 16, and 28 because only they contain rows that evaluate to TRUE for the condition.

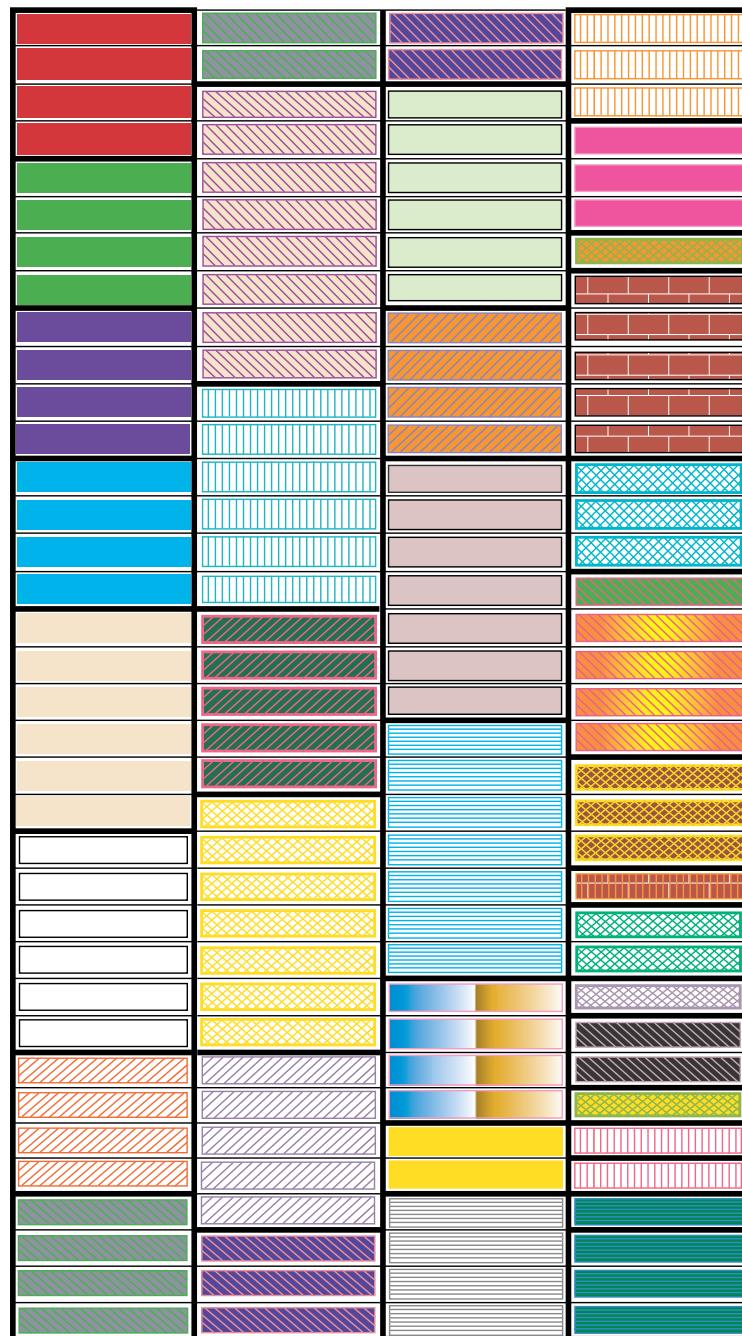
The combined partitions and rows that are not eliminated are highlighted in orange.

Partition Number				Sales					
Combined Partition Number	L1	L2	L3	Storeid	Productid	Salesdate	Total Revenue	Total Sold	Note
1	1	1	1	96	10	2003-04-15	4158	42	Good day
2	1	1	2	71	184	2003-07-06	1972	68	Marginal Sa
3	1	1	3	80	241	2003-11-09	3055	47	Slow day
4	1	1	4	82	363	2003-12-24	1261	13	Promotion
5	1	2	1	186	1	2003-01-11	255	17	Shelf Life
6	1	2	2	122	163	2003-06-13	2405	65	Multiple
7	1	2	3	110	234	2003-05-01	2618	77	2 for 1
8	1	2	4	187	384	2003-08-03	684	9	Buy 3, 1 Fr
9	1	3	1	280	31	2003-03-10	493	29	Rain
10	1	3	2	202	116	2003-02-17	6732	68	Cash Sale
11	1	3	3	292	272	2003-01-30	539	11	Quarterly
12	1	3	4	213	385	2003-04-29	4233	83	Promotion
13	2	1	1	76	51	2004-10-05	442	34	2 for 1
14	2	1	2	26	149	2004-01-09	365	5	Good day
15	2	1	3	57	295	2004-11-04	2684	61	Marginal Sa
16	2	1	4	87	338	2004-08-02	696	58	Slow day
17	2	2	1	108	34	2004-04-27	4218	74	2 for 1
18	2	2	2	171	143	2004-10-30	5925	79	Shelf Life
19	2	2	3	114	228	2004-05-24	2064	48	Rain
20	2	2	4	135	347	2004-08-03	110	55	Promotion
21	2	3	1	257	75	2004-09-18	3417	51	Promotion
22	2	3	2	295	191	2004-11-11	376	8	Quarterly
23	2	3	3	208	204	2004-03-17	864	36	Multiple
24	2	3	4	221	330	2004-12-15	1456	52	Buy 3, 1 Fr
25	3	1	1	39	85	2005-09-15	192	48	Cash Sale
26	3	1	2	55	112	2005-07-12	232	29	Shelf Life
27	3	1	3	76	243	2005-03-13	6696	93	2 for 1
28	3	1	4	7	309	2005-08-20	116	58	Discounted
29	3	2	1	186	44	2005-05-30	4275	75	Credit
30	3	2	2	183	167	2005-06-14	2982	42	Promotion
31	3	2	3	171	218	2005-05-22	80	8	Rain
32	3	2	4	180	389	2005-06-09	4128	96	Good day
33	3	3	1	278	61	2005-09-22	1581	51	Buy 4, 1 Fr
34	3	3	2	256	110	2005-04-02	3280	80	Sale
35	3	3	3	246	233	2005-11-29	5133	59	Discounted
36	3	3	4	251	342	2005-02-11	968	44	50% off

Similarly, Teradata Database can find rows by reading only a subset of the data for other combinations of two or more conditions on the partitioning columns. If you are looking for rows with a specific value of each of the partitioning columns, then only one combined partition for the specific combined row partition number needs to be read.

The following graphic, which is best viewed on a color monitor and best printed on a color printer, shows one possible representation of how the 36 combined partitions created for this table might be populated. The combined partitions have anywhere from 1 to 8 rows in this graphic, unlike the example table on which it is based, which provides only one example row per combined partition.

AMP with Multilevel PPI



1094A078

For this example, because it implies specifying values for all the columns in the primary index, only the group of rows with the same hash value determined from the primary index values need to be read, and only on a single AMP rather than the entire row partition for the specific combined row partition number on every AMP.

## Performance Implications of Multilevel Row Partitioning

- If a SELECT request specifies values for all the primary index columns, Teradata Database can determine the AMP on which the rows reside, and only a single AMP needs to be accessed.

If conditions are not specified on the partitioning columns, then Teradata Database can probe each combined partition to find the rows based on their hash value.

If conditions are also specified on the partitioning columns, row partition elimination might reduce the number of row partitions to be probed on that AMP.

- If a SELECT request does *not* specify the values for all the primary index columns or there is no primary index, then Teradata Database must do an all-AMP full table scan for a nonpartitioned table.

However, if row partitioning is defined on the table, and if you specify conditions on the partitioning columns, row partition elimination can reduce an all-AMP full file scan to an all-AMP scan of only the partitions of the combined partitioning expression that are not eliminated.

The degree of row partition elimination that can be achieved depends on the partitioning expressions, the conditions in the query, and the ability of the Optimizer to recognize such opportunities.

You need not specify values for all the partitioning columns in a query for row partition elimination to occur. Row partition elimination occurs at each level independently; the combination of the row partition elimination, if any, for each level determines which combined partitions need to be processed.

## Row Partition Elimination

Row partition elimination is a method for enhancing query performance against row partitioned tables by skipping row partitions that do not contain rows that meet the search conditions of a query. Row partition elimination is an automatic optimization in which the Optimizer (or, in the case of dynamic row partition elimination, the AMP software) determines, based on query conditions and a row partitioning expression, that some partitions for that partitioning expression cannot contain qualifying rows; therefore, those row partitions can be skipped during a file scan.

Note that the Optimizer cannot exert all of the optimizations that are possible for each of the individual types of row partition elimination.

Partitions that are skipped for a particular query are called eliminated row partitions.

When there are multiple partitioning expressions, Teradata Database combines row partition elimination at each of the levels to further reduce the number of data subsets that need to be scanned. For most applications, the greatest benefit of row partitioning is obtained from row partition elimination.

## Performance Gains Realized From Row Partition Elimination

The performance gain realized from row partitioning depends both on the number of row partitions defined and the specific query being measured. In the optimal case, the query conditions eliminate all but one row partition for each partitioning expression. When there are many thousands of row partitions, with a reasonably even distribution of rows among them, the elapsed time for such a query can be far less than 1% of the time that it would take to run the same query against a nonpartitioned table.

The following table lists the results of actual performance tests. The Baseline column is the performance for a nonpartitioned table, and the partitioning column is the performance for a counterpart table with a single-level partitioning. The tests are realistic, but the results obtained by running the same tests using your configuration and workloads might vary.

Test Description	Baseline Result (seconds)	Result With PPI (seconds)	Improvement
Select rows that have a specified value of the partitioning column from a table having 200 equal-sized partitions.	59	1	98.3% reduction in elapsed time
Select a month of activity from one row partition containing 6 months of data from a table having 11 years of data contained in 40 row partitions of unequal size.	58	2	96.5% reduction in elapsed time
Delete rows that have a specified value of the partitioning column from a table having 200 equal-sized row partitions.	239	1	99.996% reduction in elapsed time
Update one column in each row that has a specified value of the partitioning column from a table having 200 equal-sized row partitions.	237	3	98.7% reduction in elapsed time

Test Description	Baseline Result (rows/second/node)	Result With PPI (rows/second/node)	Improvement
MultiLoad INSERT a number of rows equal to 1% of the table size into one row partition out of 200.	1,394	14,742	10.58 times faster
MultiLoad INSERT a number of rows equal to 1% of the table size into one row partition out of 200 with one NUSI defined on the table.	841	5,666	6.74 times faster

For the column labelled Baseline Result, the larger the number of rows per second per node processed, the better.

Teradata Database supports several different types of row partition elimination:

- Static

When query conditions are such that they allow row partition elimination to be specified by the Optimizer during the early stages of query optimization, the form of row partition elimination used is referred to as *static row partition elimination*.

- Delayed

When query conditions are based on a comparison derived in part from USING request modifier variables or from the result of a built-in function, it is not possible for the Optimizer to reuse a cached query plan as it would otherwise do because a cached plan needs to be general enough to handle changes in search condition values in subsequent executions.

In this case, the Optimizer applies row partition elimination at a later point in the optimization process, at the time it builds the finalized query plan from a cached plan using the values for this specific execution of the plan. This form of row partition elimination is referred to as *delayed row partition elimination*.

- Dynamic

When query conditions reference values in other tables that would allow row partition elimination, row partition elimination is performed dynamically by the AMP database software after a query has already been optimized and while it is executing. This form of row partition elimination is referred to as *dynamic row partition elimination*.

Dynamic row partition elimination can also be used to simplify and enhance join performance by selecting the least costly method from a set of join methods especially designed to be used with row partition elimination.

Row partition elimination methods can be mixed within the same query. For example, static row partition elimination can be used for some partition levels, while dynamic row partition elimination can be used for other levels, and some levels might not have *any* row partition elimination. Some individual row partition levels might even benefit from a mix of multiple forms of row partition elimination.

Teradata Database can eliminate row partitions from search consideration at any number of levels or at a combination of levels.

Any single-table constraints on partitioning columns can be used for static row partition elimination, including those on the system-derived column PARTITION or any of the members of the system-derived PARTITION#Ln column set, where the value of n ranges from 1 through 62, inclusive. See “[PARTITION Columns](#)” on page 801 and *SQL Data Definition Language* for more information about the system-derived PARTITION columns.

See *SQL Request and Transaction Processing* for more information about the various forms of row partition elimination and their role in query optimization.

# Row-Partitioned and Nonpartitioned Primary Index Access for Typical Operations

The following table compares the access mechanisms for the two primary index types over a range of typical operations and conditions.

Operation	Nonpartitioned Table	Partitioning Where PI Includes All Row Partitioning Columns	Partitioning Where PI Does Not Include All Partitioning Columns
Selection with an equality (= or IN) constraint on the primary index columns	<p>Single-AMP and direct access to the first data block of rows with the same row hash value.</p> <p>Scan all rows with the same row hash value checking the primary index and residual conditions.</p> <p>Row-hash lock.</p>	<p>Single-AMP and direct access to the first data block of rows with the same row hash value for only the specified row partition.</p> <p>Scan all rows with the same row hash value in this row partition, checking primary index and residual conditions.</p> <p>Row-hash lock.</p>	<p>Equality constraint on the partitioning columns.</p> <ul style="list-style-type: none"> <li>Single-AMP and direct access to the first data block of rows with the same row hash value for only the specified row partition.</li> <li>Scan all rows with the same row hash value in the affected row partition, checking primary index and residual conditions.</li> <li>Row-hash lock.</li> </ul>
Selection with an equality (= or IN) constraint on the primary index columns (continued)	<p>Single-AMP and direct access to the first data block of rows with the same row hash value.</p> <p>Scan all rows with the same row hash value checking the primary index and residual conditions.</p> <p>Row-hash lock.</p> <p>(continued)</p>	<p>Single-AMP and direct access to the first data block of rows with the same row hash value for only the specified row partition.</p> <p>Scan all rows with the same row hash value in this row partition, checking primary index and residual conditions.</p> <p>Row-hash lock.</p> <p>(continued)</p>	<p>Constraint on the partitioning columns.</p> <ul style="list-style-type: none"> <li>Single-AMP and direct access to the first data block of rows with the same row hash value in each of the non-eliminated row partitions.</li> <li>Scan all rows with the same row hash value in the affected row partitions, checking primary index and residual conditions.</li> <li>Row-hash lock.</li> </ul> <p>Otherwise:</p> <ul style="list-style-type: none"> <li>Single-AMP and direct access to the first data block of rows with same row hash value in each populated row partition.</li> <li>Scan all rows with the same row hash value in each row partition, checking primary index and residual conditions.</li> <li>Row-hash lock.</li> </ul> <p>Teradata Database might use a secondary index to reduce the number of rows accessed.</p>

Operation	Nonpartitioned Table	Partitioning Where PI Includes All Row Partitioning Columns	Partitioning Where PI Does Not Include All Partitioning Columns
Selection without an equality constraint on the primary index columns and no applicable secondary, hash, or join index	All-AMP, full-file scan, checking each row for any residual conditions.  All-AMP table lock.	<p>Equality constraints on all the partitioning columns.</p> <ul style="list-style-type: none"> <li>• All-AMP, but only scan a single row partition on each AMP.</li> <li>• Scan all rows in the affected row partition checking for any residual conditions.</li> <li>• All-AMP table lock.</li> </ul>	<p>Constraints on the partitioning columns.</p> <ul style="list-style-type: none"> <li>• All-AMP, but only scan non-eliminated row partitions.</li> <li>• Scan all rows in the affected row partitions checking for any residual conditions.</li> <li>• All-AMP table lock.</li> </ul>
Joins	A wide variety of join possibilities exist for nonpartitioned primary-indexed tables.  In many cases, Teradata Database applies an all-AMPs table lock; however, in some cases it applies row-hash locks.	<p>A wide variety of join possibilities exist for PPI table joins.</p> <p>Joins on partitioning columns might be optimized in some cases using dynamic row partitioning elimination on the AMPs.</p> <p>In many cases, Teradata Database applies an all-AMP table lock; however, in some cases it applies row-hash locks.</p>	

Operation	Nonpartitioned Table	Partitioning Where PI Includes All Row Partitioning Columns	Partitioning Where PI Does Not Include All Partitioning Columns
Joins (continued)	<p>A wide variety of join possibilities exist for nonpartitioned primary-indexed tables.</p> <p>In many cases, Teradata Database applies an all-AMPs table lock; however, in some cases it applies row-hash locks.</p> <p>(continued)</p>	<p>Tables can be joined in the same manner as a table with a nonpartitioned primary index and can also take advantage of row partition elimination possibilities when possible.</p>	<p>For a non-direct join to a PPI table or a direct equality join on the primary index columns and the row partitioning columns, tables can be joined in the same manner as a table with a nonpartitioned primary index and can also take advantage of row partition elimination possibilities when possible.</p> <p>For a direct equality join on the primary index columns, but not on all the partitioning columns, several optimizations might be invoked; however, they might use more memory and processing time to handle the row partitions.</p> <p>In those cases where a direct join would be chosen for a nonpartitioned primary index table, the Optimizer might estimate that a non-direct join would have lower cost for a PPI table and use it instead.</p>
INSERT	<p>Single-AMP and direct access to the first data block of rows with the same row hash value.</p> <p>If a SET table and there are no unique indexes, verify that no duplicate rows within this same row hash value exist.</p> <p>If a unique primary index, verify the uniqueness within this same row hash value.</p> <p>Append to the end of the rows with the same row hash value.</p> <p>Update secondary indexes and validate uniqueness for USIs.</p> <p>Row-hash lock.</p>	<p>Single-AMP and direct access to the first data block of rows with the same row hash value in the specified row partition.</p> <p>If a SET table and there are no unique indexes, verify that no duplicate rows within this same row hash value exist in this row partition.</p> <p>If a unique primary index, verify the uniqueness within this same row hash value in this row partition.</p> <p>Append to the end of the rows with the same row hash value in this row partition.</p> <p>Update secondary indexes and validate uniqueness for USIs.</p> <p>Row-hash lock.</p>	<p>Single-AMP and direct access to the first data block of rows with the same row hash value in the specified row partition.</p> <p>If a SET table and there are no unique indexes, verify that no duplicate row within this same row hash value exist in this row partition.</p> <p>Append to the end of the rows with the same row hash value in this row partition.</p> <p>Update secondary indexes and validate uniqueness for USIs.</p> <p>Row-hash lock.</p>
INSERT ... SELECT	<p>Sort inserts as needed and merge into table.</p> <p>All-AMP table lock.</p>		

Operation	Nonpartitioned Table	Partitioning Where PI Includes All Row Partitioning Columns	Partitioning Where PI Does Not Include All Partitioning Columns
DELETE	Same as selection plus secondary, hash, and join index maintenance.	Same as selection.	Same as selection.
UPDATE (Non-Upsert form)	Same as selection unless index columns are updated, plus secondary, hash, and join index maintenance.  If index columns are updated, this is treated as an unreasonable update. That is, for an update of primary index columns, the updated rows must be deleted and reinserted based on the new values (most likely on a different AMP) using an all-AMP table lock.	Same as selection unless index columns are updated.  If index columns are updated, this is treated as an unreasonable update. That is, for an update of primary index columns, the updated rows must be deleted and reinserted based on the new values (most likely on a different AMP) using an all-AMP table lock.	Same as selection unless index or partitioning columns are updated.  If index or partitioning columns are updated, this is treated as an unreasonable update. That is, for an update of primary index or row partitioning columns, updated rows must be deleted and reinserted based on the new values using an all-AMP table lock.  If a primary index column is updated, the row will most likely be reinserted on a different AMP.  If partitioning columns are updated without updating any primary index columns, a row remains on the same AMP.
UPDATE (Upsert form)	This can be a more complex operation and is not described here. For a specific UPSERT operation, refer to the EXPLAIN report for the request.		
MERGE	This can be a more complex operation and is not described here. For a specific MERGE operation, refer to the EXPLAIN report for the request.		

See also “[Single-AMP Queries and Partitioned Tables](#)” on page 905 and “[All-AMP Tactical Queries and Partitioned Tables](#)” on page 909 for design issues specific to PPI support for tactical queries.

## Summary of Primary Index Selection Criteria

The following table summarizes the guidelines for selecting columns to be used as primary indexes.

Guideline	Comments
Select columns that are most frequently used to access rows.	Restrict selection to columns that are either unique or highly singular.
Select columns that are most frequently used in equality predicate conditions.	Equality conditions permit the system to hash directly to the row having the conditional value. When the primary index is unique, the response is never more than one row.  Inequality conditions require additional processing.

Guideline	Comments
Select columns that distribute rows evenly across the AMPs.	<p>Distinct values distribute evenly across all AMPs in the configuration. This maximizes parallel processing.</p> <p>Rows having duplicate NUPI values hash to the same AMP and often are stored in the same data block. This is good when rows are only moderately non-unique.</p> <p>Rows having NUPI columns that are highly nonunique distribute unevenly, use multiple data blocks, and incur multiple I/Os.</p> <p>Extremely nonunique primary index values can skew space usage so markedly that the system returns a message indicating that the database is full even when it is not. This occurs when an AMP exceeds the maximum bytes threshold for a user or database calculated by dividing the PERMANENT = n BYTES specification by the number of AMPs in the configuration, causing the system to incorrectly perceive the database to be “full.”</p>
Select columns that are not volatile.	Volatile columns force frequent row redistribution.
Select columns having very many more distinct values than the number of AMPs in the configuration.	<p>If this guideline is not followed, row distribution skews heavily, not only wasting disk space, but also devastating system performance.</p> <p>This rule is <i>particularly</i> important for large tables.</p>
Do not select columns defined with Period, ARRAY, VARRAY, Geospatial, JSON, XML, BLOB, CLOB, XML-based UDT, BLOB-based UDT, or CLOB-based UDT data types.	<p>You cannot specify columns that have BLOB, CLOB, BLOB-based UDT, CLOB-based UDT, XML-based UDT, Period, ARRAY, VARRAY, Geospatial, or JSON data types in a primary index definition. If you attempt to do so, the CREATE request aborts.</p> <p>You can, however, specify Period data type columns in the partitioning expression of a partitioned table.</p>
Do not select aggregated columns of a join index.	<p>When defining the primary index for a join index, you cannot specify any aggregated columns.</p> <p>If you attempt to do so, the CREATE JOIN INDEX request aborts.</p>

## Principal Criteria for Selecting a Primary Index

When assigning columns to be the primary index for a table, there are three essential factors to keep in mind: uniform distribution of rows, optimal access to the data, and the volatility of indexed column values.

You will sometimes encounter situations where the selection criteria conflict. For example, specifying a NUPI instead of a UPI, or specifying an alternate key as the UPI instead of the primary key.

There are additional criteria to evaluate when selecting the primary index for a queue table. See “[Selecting a Primary Index for a Queue Table](#)” on page 421 for a description of the primary index selection criteria you need to evaluate when choosing a primary index for a queue table.

Keep in mind that these criteria apply only to selecting a column set for the primary index. They do *not* apply to making a decision whether the primary index should be row-partitioned

or not. See “[Row-Partitioned and Nonpartitioned Primary Index Access for Typical Operations](#)” on page 403 and “[Considerations for Choosing a Primary Index](#)” on page 410 for guidelines on making that choice.

Be aware that with the exception of column-partitioned tables, Teradata Database assigns a default primary index to a table if you do not specify an explicit PRIMARY INDEX or NO PRIMARY INDEX in the CREATE TABLE request you use to create the definition for the table (see “[Primary Index Defaults](#)” on page 263).

## Uniform Data Distribution

With respect to uniform data distribution, you should always consider the following factors.

- The more distinct the primary index values, the better.
- Rows having the same primary index value are distributed to the same AMP.
- Parallel processing is more efficient when table rows are distributed evenly across the AMPs.

See “[Distribution Demographics](#)” on page 416 for details.

## Optimal Data Access

With respect to optimal data access using a primary index, you should consider the following factors.

- The primary index should be chosen on the most frequently used access path.  
For example, if rows are generally accessed by a range query, you should consider defining a partitioned primary index on the table or join index that creates a useful set of partitions.  
If the table is frequently joined with a specific set of tables, then you should consider defining the primary index on the column set that is typically used as the join condition.
- Primary index operations must provide the full primary index value.
- Primary index retrievals on a single value are always single-AMP operations.

See “[Access Demographics](#)” on page 417 for details.

## Index Column Volatility

The primary index column set should be rarely, and preferably never, updated.

See “[Volatility of Indexed and Partitioning Column Values](#)” on page 420 for details.

## Criteria for Selecting a Primary Index

The following guidelines and performance considerations apply to selecting a unique or a nonunique column set as the primary index for a table.

- Choose columns for the primary index based on the selection set most frequently used to retrieve rows from the table even when that set is not unique (if and only if the values of the selection set are fairly equally distributed across the AMPs).

- Choose columns for the primary index that do not have XML, BLOB, CLOB, BLOB-based UDT, CLOB-based UDT, XML-based UDT, Period, JSON, ARRAY, VARRAY, or VARIANT\_TYPE data types.

Distinct and structured UDT columns are valid components of a primary index, but UDT columns based on internal Teradata UDT types, such as the Period, Geospatial, ARRAY, and VARRAY types, are not.

- Choose columns for the primary index that distribute table rows evenly across the AMPs. The more singular the values for a column, the more optimal their distribution.
- Choose as few columns as possible for the primary index to optimize its generality. All the columns in a composite primary index must be specified in a WHERE clause predicate before the Optimizer can select it for use as the retrieval mechanism.
- If it is difficult to define a unique primary index for a table that must have one, you can generate arbitrary unique values for a single column if you define it as an identity column with the characteristics ALWAYS GENERATED and NO CYCLE (see “[Identity Columns](#)” on page 818).
- Base the column selection on an equality search (if the primary index is a PPI, then the search is done within each non-eliminated populated partition). For equality constraints *only*, the system hashes directly to the row set that satisfies the condition.

Tables with this kind of primary key ...	Tend to assign the primary index to ...
single-column	<p>the primary key.                      This is referred to as a Unique Primary Index (UPI).</p>
multicolumn	<p>one of the foreign key components of the primary key.                      This is referred to as a Non-Unique Primary Index (NUPI).</p>

- Primary and other alternate key column sets often can provide useful uniqueness constraints as well as a powerful access and join method when the logical design for a table is physically realized. If the primary or other alternate keys for a table are not selected to be its primary index, you should consider assigning a unique constraint, such as PRIMARY INDEX, UNIQUE, or a USI on those keys if the uniqueness constraint would facilitate table access and joins.

This recommendation is contingent on a number of complicated factors that must be considered fully before implementing unique constraints. See “[Using Unique Secondary Indexes to Enforce Row Uniqueness](#)” on page 457 for a list of the factors that should be considered when you consider implementing this recommendation.

A UPI ...	WHILE a NUPI ...
at most involves one row	can involve multiple rows.
does not require a spool	often creates a spool.

- Duplicate NUPI values are always stored on the same AMP and in the same data block if possible.
- NUPI retrieval only requires one I/O operation (or two I/Os if the cylinder index is not memory-resident) when the rows are stored in the same data block.

This type of value range ...	Seen when using this predicate in a WHERE clause ...	Results in this kind of retrieval action ...
implicit	BETWEEN	<p>full table scan, irrespective of any indexes defined for the table.</p> <p>The exceptions are the following:</p> <ul style="list-style-type: none"><li>• PPI tables and join indexes, where row partition elimination can be exploited.</li><li>• Hash and join index tables with a value-ordered NUPI, where value ordering can be exploited.</li></ul>
explicit	IN	individual row hashing.

## Considerations for Choosing a Primary Index

Selecting the optimum primary index for a table or uncompressed join index is often a complex task because some applications might favor one type of primary index, while other applications might perform more optimally using a different primary index. Tables can have only one primary index, however, so you must select one that best suits the majority of the applications that a table serves. Of course, if the overhead costs justify the expense, you can define multiple join indexes with different primary indexes.

You can always add additional indexes, such as secondary, hash, and join indexes, to facilitate particular applications. Be aware that these indexes all incur various overhead costs, including:

- Disk space required to store their subtables.
- System performance degrades whenever base table rows are updated because the index values for any indexed columns affected by that update must also be updated.

You should always consider these tradeoffs when planning your indexes, then be sure to test them to ensure that the assumptions that lead to your choices are correct. For example, if you design a primary index with even row distribution as your principal criterion, analyze the actual distribution of table rows to ensure that they *are* evenly distributed.

For many applications, particularly those that use range queries heavily, a partitioned primary index can provide a better solution to resolving these issues than a nonpartitioned primary index because it provides efficient access both via the primary index columns as well as via a constraint on the partitioning columns. As always, you should confirm that the partitioning actually improves query performance by carefully examining EXPLAIN reports and collecting the appropriate statistics.

You should always collect statistics on the PARTITION column *and* the partitioning columns.

The recommended practice for recollecting statistics is to set appropriate thresholds for recollection using the THRESHOLD options of the COLLECT STATISTICS statement. See “[COLLECT STATISTICS in SQL Data Definition Language](#)” for details on how to do this.

You should also weigh the costs of the index against the benefits it provides. This is particularly important if you have also defined a USI on the table because additional maintenance is required to enforce uniqueness, thus potentially neutralizing or even reducing the overall performance advantage of the index.

Creating a partitioned table does not guarantee that row-partition elimination plan. A partitioning might not be used for any of the following common reasons:

- It is not applicable to the actual queries in the workload.
- The Optimizer cost analysis for a query determines that another plan is less expensive.
- The query does not conform to any number of restrictions.

In some cases, a query plan with partitioning might not perform as well as one without partitioning (see “[Row-Partitioned and Nonpartitioned Primary Index Access for Typical Operations](#)” on page 403 for specific examples).

Various partitioning strategies can be followed.

- For some applications, defining the partition expressions such that each row partition has approximately the same number of rows might be an effective strategy.  
This task is far easier for single-level PPIs than for multilevel PPIs, though it can still be thought of as a goal to be approximated as best as possible.
- For other applications, having a varying number of rows per partition might be desirable. For example, more frequently accessed data (such as for the current year) might be divided into finer partitions (such as weeks) but other data (such as previous years) may have coarser partitions (such as months or multiples of months).  
Note that partitioning in this manner can make altering the partitions more difficult.
- Alternatively, defining each range with equal width, even if the number of rows per range varies, might be important.

The most important factors for row partitioning are accessibility and maximization of row partition elimination. In all cases, defining a primary index (or having no primary index) that distributes the rows of the table fairly evenly across the AMPs is critical for efficient parallel processing. See “[Evaluating the Relative Merits of Partitioning Versus Not Partitioning](#)” on page 413 for further information.

## Partitioning Guidelines

The following guidelines provide a high-level set of criteria for making an initial evaluation of whether row partitioning would provide more benefits to a query workload than a nonpartitioned table.

- Large tables and join indexes are usually better candidates for row partitioning than smaller tables and join indexes because there is not much benefit to partitioning a table or join index small enough that a full-table scan on the nonpartitioned table or join index takes only a few seconds.

The exception to this is a small table that is row-partitioned identically to a larger table with which it is frequently joined and with which it shares its primary index.

- When possible, you should row-partition on sets of columns that are frequently used as query conditions. For example, if half the queries against a table specify a date range that qualifies less than 25% of the rows, then that date column is a good candidate to be the partitioning column for the table.

If there is no column that is frequently used as a query condition, then there is probably little or no advantage to row-partitioning the table.

- All factors being equal, it is better to partition on a column set that is part of the primary index column set than to partition on a column that is not.

The exception to this is if the primary index is rarely, if ever, used for row access or join operations.

- Keep the number of row partitions relatively small. The key word in this guideline is *relatively*. The guideline also applies for multilevel partitioning situations, though it is more difficult to achieve for multilevel partitioning because the total number of partitions is a multiplicative factor of the number of partitioning levels defined for the table, so the number of partitions can grow very quickly even when there are few partitions defined for each level (see “[Usage Considerations and Rules for PARTITION and PARTITION#Ln Columns](#)” on page 808).

The exception to this guideline is if the primary index is rarely, if ever, used for row access or direct merge joins.

For example, if all the queries against the table access *at least* one month of activity, there is little or no benefit to partitioning by week or day instead of by month. An exception to this is if bulk data loading times are greatly reduced by a finer partition granularity.

See “[Scenario 4](#)” on page 434 for an example of evaluating these sorts of tradeoffs.

- Keep the number of partitions small even if you plan to expand predetermined table operations in the future. You can always increase the number of partitions later when they are needed.

**Note:** If you collect and maintain fresh statistics on the PARTITION columns of tables, this consideration is much less important.

You have greater flexibility with this guideline for single-level partitioned tables than you do for multilevel partitioned tables because it can be rather complicated to decrease the number of partitions for a multilevel partitioning because the number of combined partitions defined for such a table increases multiplicatively with each partition and with each level defined.

- The same criteria for selecting the column set for a nonpartitioned table also apply to partitioned tables (see “[Principal Criteria for Selecting a Primary Index](#)” on page 407 and “[Secondary Considerations for Selecting a Primary Index](#)” on page 416).

Choose a primary index column set that provides good row distribution, avoids skew, and is commonly used to access individual rows or do not use a primary index.

Optimal row distribution and frequent access are sometimes conflicting considerations, so you must evaluate their relative merits and come to some compromise if that is the case.

## Evaluating the Relative Merits of Partitioning Versus Not Partitioning

The following criteria provide a high-level means for evaluating the relative merits of partitioning or not partitioning for a table.

Potential advantages of row partitioning.

- The greatest potential gain in row-partitioning a primary index is the ability to read a subset of table or join index rows instead of scanning them all.

For example, a query that examines two months of sales data from a table that maintains two years of sales history can read about  $\frac{1}{12}$  of the table instead of having to scan it all.

The advantages of row partition elimination can be even greater for multilevel partitioned tables (see the examples in the topic “Static Partition Elimination” in *SQL Request and Transaction Processing* for some remarkable scan optimizations).

This provides the opportunity for a large performance boost to a wide range of queries. Importantly, the individuals who code those queries do not have to know the partitioning structure of the table and, as a result, there is no need to recode existing SQL applications.

- Appropriate row partitioning can also facilitate faster batch data loads.

For example, if a table is partitioned by transaction date, the loading of transactions for the current day can be dramatically enhanced, as can the deletion of rows from the table that are no longer necessary.

See “[Performance Gains Realized From Row Partition Elimination](#)” on page 401 for some examples of batch data loads running anywhere from six to ten times faster for a table having an appropriately defined partitioning versus the identical nonpartitioned table.

- Row partitioning can make one or more existing secondary, hash, or join indexes redundant, which permits them to be dropped from the database.

Potential disadvantages of row partitioning.

- Row partitioning can make single row (primary index) accesses to the table slower if a partitioning column is not a member of the primary index column set.

This disadvantage can be offset somewhat by using one of the following strategies:

- Choose a partitioning column that is a member of the primary index column set.
- Define a unique secondary index that can be used to make single row accesses to the table.
- Constrain the values of the partitioning column set to enable the Optimizer to eliminate some row partitions when the query search conditions permit.
- Row partitioning can make direct merge joins of tables slower unless both tables are partitioned identically.

This disadvantage can be offset when query search conditions allow some row partitions to be excluded from the join operation.

The Teradata Database query optimizer has several special product join and merge join methods available to it just for joining row-partitioned tables. See *SQL Request and Transaction Processing* for descriptions and examples of these join methods.

As with other physical database design choices, you must always evaluate the respective tradeoffs of the decisions that are available to you by prototyping and testing their relative merits.

## Primary Index Properties

- Teradata Database uses the primary index of a table to control its row distribution and retrieval using the Teradata Database hashing algorithm (see “[Teradata Database Hashing Algorithm](#)” on page 225).
- The primary index for an unpopulated table is defined using the CREATE TABLE data definition statement (see the documentation for the CREATE TABLE, CREATE HASH INDEX, and CREATE JOIN INDEX statements in *SQL Data Definition Language*).  
CREATE INDEX is used *only* to create secondary indexes.
- The primary index for an empty table can be modified, with restrictions, using the ALTER TABLE data definition statement (see the documentation for the ALTER TABLE statement in *SQL Data Definition Language*)
- If no explicit primary index is defined in a CREATE TABLE request, the system determines whether the table has no primary index or to assign one automatically according to the rules described in “[Primary Index Defaults](#)” on page 263.
- A primary index can be unique or nonunique (see “[Unique Primary Indexes](#)” and “[Nonunique Primary Indexes](#)” on page 265).
- If the primary index is not defined explicitly as unique, then its definition defaults to nonunique (see “[Primary Index Defaults](#)” on page 263).
- A primary index can be row-partitioned or nonpartitioned (see “[Row-partitioned Primary Indexes](#)” and “[Nonpartitioned Primary Indexes](#)” on page 267).

If row-partitioned, a primary index can be partitioned at a single level or at multiple levels (see “[Single-Level Partitioning](#)” on page 362 “[Multilevel Partitioning](#)” on page 372).

A primary index of a hash or join index can also be value-ordered, subject to constraints on the data type and field length of the ordering column. See the documentation for CREATE HASH INDEX and CREATE JOIN INDEX in *SQL Data Definition Language* for details.

- A primary index can be composed of as many as 64 columns (see “[Restrictions on Primary Indexes](#)” on page 262).
- The values for a primary index can be generated automatically if defined on an identity column with the characteristics ALWAYS GENERATED and NO CYCLE (see “[Identity Columns](#)” on page 818).
- A primary index cannot contain columns with a Period, Geospatial, JSON, ARRAY, VARRAY, VARIANT\_TYPE, XML, BLOB, CLOB, XML-based UDT, BLOB-based UDT, or CLOB-based UDT data type.

Distinct and structured UDT columns are valid components of a primary index. UDT columns based on internal Teradata UDT types (such as Period, ARRAY, VARRAY, and geospatial) are not valid components of a primary index.

- Zero or one primary index must be specified per base table or join index you create (see “[Restrictions on Primary Indexes](#)” on page 262).
- The following table types are the only exceptions to the rules that a table or join index must have a primary index.
- Global temporary trace tables.  
 You cannot define a primary index or any other kind of index on a global temporary trace table. See “[CREATE GLOBAL TEMPORARY TRACE TABLE](#)” in *SQL Data Definition Language Detailed Topics* for details.
  - Nonpartitioned NoPI tables.  
 See “[NoPI Tables, Column-Partitioned Tables, and Column-Partitioned Join Indexes](#)” on page 280 for details.
  - Column-partitioned tables and join indexes.  
 See “[Column-Partitioned Tables and Join Indexes](#)” on page 285 for details.
  - At most, one primary index can be defined per table or join index (see “[Restrictions on Primary Indexes](#)” on page 262).
  - A primary index improves performance when specified correctly in the WHERE clause of an SQL data manipulation request to perform the following actions (see “[Purposes of the Primary Index](#)” on page 262):
    - Single-AMP retrievals
    - Joins between tables with identical primary indexes, the optimal scenario.
    - Eliminate row partitions when selecting from and joining partitioned tables for various types of range queries.

## Related Topics

Topic	Reference
System-derived PARTITION and PARTITION#Ln columns	<a href="#">“PARTITION Columns” on page 801</a>
Primary indexes of temporal tables.	<i>ANSI Temporal Table Support and Temporal Table Support</i>
Row partition elimination	<i>SQL Request and Transaction Processing</i>
Primary indexes and their defaults, restrictions, and uses for: <ul style="list-style-type: none"> <li>ALTER TABLE</li> <li>CREATE HASH INDEX</li> <li>CREATE JOIN INDEX</li> <li>CREATE TABLE</li> </ul>	<i>SQL Data Definition Language</i>

## Secondary Considerations for Selecting a Primary Index

When you select primary indexes for your tables, there are several factors to keep in mind that are not orthogonal to the three principle selection criteria described in “[Principal Criteria for Selecting a Primary Index](#)” on page 407.

These considerations include the following factors.

- Performance (see “[Performance Considerations for Primary Indexes](#)” on page 441)
- Duplicate row checks on NUPIs for multiset tables (see “[Duplicate Row Checks for NUPIs](#)” on page 444)
- Space management

Primary indexes in Teradata Database are not stored in an index subtable, but are just a column set that is designated as the primary index at the time the table is created (or, rarely, when its primary index is altered), space management issues that are unique to primary indexes are minimal.

The only factors are the following two:

- The partition number field in the row header (see “[Base Table Row Format](#)” on page 740), where each row of a table having a partitioned primary index requires an additional 2 or 8 bytes (partitioned table rows are 4 bytes wider if multi-value compression is specified for the table) in the row header to indicate the partition number to which the row belongs.  
If a table does not have a partitioned primary index, then the partition number for each row is assumed to be 0, and nothing is stored in the row header to represent a partition number other than flag bits that occur every row (see “[Storage and Other Overhead Considerations for Partitioning](#)” on page 332).
- None of the columns in the primary index column set can be compressed (see “[Restrictions on Primary Indexes](#)” on page 262).

Each of these considerations is described in further detail in additional topics in this chapter.

## Distribution Demographics

The primary guideline for selecting a primary index column set is to achieve an even distribution of rows across the AMPs.

The more singular the values of a column chosen as the primary index, the more even the distribution of table rows across the AMPs of a system. A unique index is ideal for ensuring optimal distribution. When distribution is optimized, so is parallel processing.

While it is true that the ideal primary index for a table both optimizes retrieval and distribution, the reality is that you are often faced with trading one off against the other. It is not exceedingly rare for a particular primary index to provide maximal access, but poor distribution (or vice versa).

## Distribution Guidelines

The key to solving such a dilemma is to prototype several possibilities to see which is the best choice. You might have to make do with an optimal, rather than a maximal, solution.

The goal of this guideline is to optimize row distribution without worsening row accessibility.

With respect to partitioned primary indexes, the granularity of partitioning can be a significant factor in the general effectiveness of the index. The more row partitions, the finer the ability to eliminate them. At the same time, too fine a granularity can have a negative effect on primary index access as well as joins and aggregations on the primary index. See [“Row-Partitioned and Nonpartitioned Primary Index Access for Typical Operations” on page 403](#) for a detailed comparison of the various access mechanisms for the two primary index types over a range of typical operations and conditions.

## Access Demographics

Access demographics is composed of:

- Primary index value retrieval
- Join access

### Primary Index Value Retrieval Access: Definition

An *access* column is one that is commonly used as a valued predicate in a WHERE clause.

While it is true that the column set chosen to be the primary index for a table is often the same column set that defines the primary key, it is also true that primary indexes are often composed of fields that are neither unique nor components of the primary key for the table.

### Primary Index Value Retrieval Access: Guidelines for Nonpartitioned Selection

Keeping in mind that the principle goal for selecting the primary index for a table should always be achieving an even distribution of rows across the AMPs, the primary guideline for selecting a primary index to optimize retrieval should be based on the access demographics of the table. To facilitate optimal row access, choose a single column or, less preferably, a set of several columns, that is most frequently used to access the table. In other words, define the primary index for a table on a column set that is most frequently equated to discrete values in WHERE clause predicates in your application environment.

The reason for defining the primary index on the smallest possible column set is that you cannot hash or retrieve on a partial index value, so if a query condition specifies only a subset of the primary index column set, the Optimizer cannot build an access plan that uses the primary index. Note that this is also true for USIs.

Conversely, if you specify too few columns in the primary index definition, then each primary index value might correspond to a large number of rows, a situation that often not only causes data skew, but also degrades any data maintenance that must touch all rows in a row hash.

The goal of this guideline is to maximize the number of single-AMP (primary index only) operations.

## Primary Index Value Retrieval Access: Partitioning Guidelines for Selection

The same guideline applies to partitioned access. The principal reason to define a table or uncompressed join index with row partitioning is to facilitate row partition elimination. Row partition elimination is analogous to how the Optimizer uses column projection and row restriction in that it eliminates partitions that are not relevant at the earliest possible stage in query processing.

The degree of row partition elimination depends both on the partitioning expression specified for the index and on the conditions specified in the query. Increasing the number of populated row partitions can degrade the performance of primary index access, joins, and aggregations on the primary index, but it also permits finer row partition elimination; therefore, it is critical to understand the nature of the applications that are the predominant users of a row-partitioned table or uncompressed join index.

It is not always necessary for all values of the partitioning columns to be specified in a query for row partition elimination to occur (see [“Row-Partitioned and Nonpartitioned Primary Index Access for Typical Operations” on page 403](#) for details about the relative performance implications of various access methods on tables with partitioned or nonpartitioned primary indexes).

A PPI provides optimal access to base table or join index rows while also providing efficient join and aggregation strategies on the primary index in other situations. With a constraint on the partitioning columns of a table or uncompressed join index, partitioned access performance can approach the performance of a nonpartitioned table depending on the degree to which the Optimizer can eliminate partitions from consideration by the query.

Access via an equality constraint on a primary index that also includes all the partitioning columns is as efficient as with a nonpartitioned table. If there is an equality constraint on a primary index that does *not* include all the partitioning columns, but there *is* an equality or other constraint on the partitioning columns that limits access to a single partition, then access is also as efficient as with a nonpartitioned table.

Access via an equality constraint on the primary index that neither includes the partitioning columns nor constrains the partitioning columns, might not be quite as efficient as with a nonpartitioned table, depending on the number of populated partitions (a partition is said to be populated when it contains rows).

Although you can narrow access by specifying a particular partition set using the PARTITION keyword (see “[PARTITION Columns](#)” on page 801), access to particular partitions is generally performed internally and need not be specified explicitly in a request.

IF a request ...	THEN ...
specifies values for all primary index and partitioning columns	a row can be retrieved by single AMP access from a single partition.
specifies values for all primary index columns and also specifies search conditions on the partitioning columns	row partition elimination can reduce the number of combined partitions that must be probed on a particular AMP.
specifies values for all primary index columns but does not specify search conditions on the partitioning columns	each combined partition can be probed individually to locate rows based on their hash value only.
specifies search conditions on the partitioning columns	row partition elimination can reduce an all-AMPs full table scan to an all-AMPs scan of only the combined partitions relevant to the query.
does not specify the values for all primary index columns and there are no constraints on the partitioning columns	<p>the strategy the Optimizer elects to follow depends on whether a usable secondary, hash, or join index exists for the query, as explained by the following bullets.</p> <ul style="list-style-type: none"> <li>• If there is a cost effective secondary, hash, or join index, the Optimizer uses it.</li> <li>• If there is no cost effective secondary, hash, or join index, the Optimizer invokes an all-AMPs full-table scan.</li> </ul>

## About Join Access

Indexes are an extremely important component of any join. This does not apply to global temporary trace tables, which can neither have indexes nor be joined to other tables. See “[CREATE GLOBAL TEMPORARY TRACE TABLE](#)” in *SQL Data Definition Language Detailed Topics* for details.

If, after implementing primary indexes for tables that are frequently joined, you detect significant performance issues when those tables are joined, you might want to reconsider your index column choices, particularly for commonly joined column sets in large tables. You might also want to consider adding additional columns to an existing index, depending on what your EXPLAINs for the query look like.

## Join Access: Column Selection Guidelines

When designing for join access, you should first consider selecting common join columns for use as primary indexes. Joining tables on their primary indexes permits the Optimizer to specify hash or merge joins, two highly effective techniques, in the join plan to further optimize a query.

The effectiveness of a merge join relates directly to whether it is made on primary indexes or not. Possible scenarios, presented in order of optimum performance from best to worst, are as follows:

Join Predicate	Redistribution Action Required
Tables joined on their primary indexes (and partitioning columns, if there are any).	None.
Tables joined on their primary indexes, but not on all partitioning columns.	Direct join in some cases, otherwise redistribute and sort the rows to be joined.
One join column is a primary index; the other is not.	Qualified rows from one table must be redistributed and sorted.
Neither join column is a primary index.	Qualified rows from both tables must be redistributed and sorted.

For multicolumn joins, you should consider using all, or at least a subset, of the join columns as the primary index.

Analogously to designing for row access, the primary guideline for selecting a primary index for join access is to choose a column or, less preferably, a set of columns, that is most frequently used to make the join. In other words, define the primary indexes for the tables to be joined on a column set that is most frequently equated to discrete values in WHERE or ON clause predicates in your application environment.

If you define a primary index on too many columns, then those queries that do not specify all of the columns in the index are not assigned a hash or merge join by the Optimizer.

If you define a partitioned primary index with too large a number of populated partitions, join access performance can be degraded.

## Volatility of Indexed and Partitioning Column Values

Primary indexes and row partitioning should be defined on columns that rarely, if ever, change. Changing the value of a primary index column or partitioning column almost certainly means that affected rows must be redistributed to different AMPs, which results in excessive I/O traffic on the BYNET and disk subsystems, or moved to a different partition on the same AMP.

This will minimize system-wide performance degradations caused by frequently changed primary index or partitioning column values.

## Selecting a Primary Index for a Queue Table

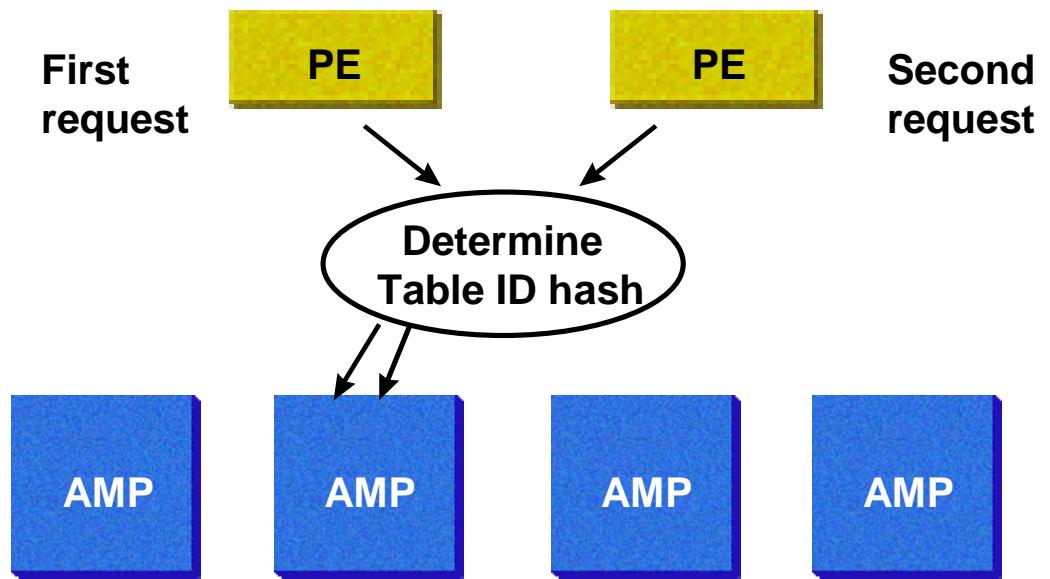
Selection of a primary index might or might not be important when defining a queue table (see *SQL Data Definition Language* for information about creating and using queue tables). If you do not define a primary index explicitly, the system uses the QITS (Queue Insertion Time Stamp) column, the first column in the queue table, as the default. The values of the QITS column reflect the time of insertion of their rows. If you need to perform a primary index update or delete one row from a queue table, you can, in most cases, easily browse the table and get the value of the QITS and other columns needed for the single-AMP update activity.

However, you might also choose a primary index column set based on the input data that is easily known, to support frequent updating. The queue table definition presented below has *orderkey* as its primary index, for example.

```
CREATE TABLE event02_QT, QUEUE (
    table_event02_QT_QITS TIMESTAMP(6) NOT NULL
                                DEFAULT CURRENT_TIMESTAMP(6),
    orderkey             DECIMAL(18,0) NOT NULL,
    productkey          DECIMAL(18,0) NOT NULL
)
PRIMARY INDEX (orderkey);
```

Using a business entity for the primary index makes sense if you have a requirement to reorder the queue on a regular basis (or otherwise manipulate the rows), and the number of rows it holds is not trivial, making browsing the entire queue table for each update less desirable.

When you perform INSERT ... SELECT processing from a staging table into a queue table, you must pay closer attention to the selection of the primary index for the queue table. As shown in the following figure, such an INSERT ... SELECT operation might perform a similar function as a trigger when doing row-at-a-time insert operations: select the few rows that compose events and insert them immediately into a queue table for further processing.



1101A313

If you are using a minibatch approach to loading data, then using set processing to identify events makes sense. In this case, the WHERE clause for the INSERT ... SELECT statement contains the event-identification criteria. In the case illustrated by the previous figure, having a queue table with the same primary index definition as the staging table improves the efficiency of the INSERT ... SELECT processing. The two inserts into the staging and queue tables then occur on the same AMP, eliminating the overhead of row redistribution.

Designing a queue table to share the same primary index as another base table could also be useful for tactical queries (see [Chapter 16: “Design Issues for Tactical Queries”](#)). A given tactical application might want either to peek into the queue or to seek out the presence of a specific row in the queue. If the primary index value of the other table is known, and it is also the primary index value of a possible queue table row, then the system enables single-AMP access.

A similar situation exists if Teradata Parallel Data Pump inserts are made into a target table that has a trigger into a queue table. If the queue table and the target table share the same primary index, and you want to serialize the input to avoid cross-session blocking, serializing on the base table primary index also causes inserts into the queue table to be serialized effectively.

## Column Distribution Demographics and Primary Index Selection

This topic examines guidelines for analyzing column distribution demographic data for use in selecting optimum columns to use for primary indexes.

### Distribution Demographics

Among the column demographics retrieved by the COLLECT STATISTICS (Optimizer Form) statement some figures are very useful for monitoring how evenly your primary index columns distribute rows across the AMPs.

The relevant demographics are the following.

- Number of distinct values within a primary index column
- Number of rows per distinct value
- Number of rows partly null
- Number of rows all null
- Typical number of rows per distinct value

### Number of Distinct Values

This number reports how many distinct values a particular primary index column contains.

With respect to NUPI behavior, the closer the ratio of distinct index column values to table cardinality is to 1.0, the better. For a UPI, this value is always 1.0 by definition.

FOR this type of primary index ...	The ratio of distinct index column values to table cardinality is ...
unique	always 1.0.
non-unique	best for distribution when it is close to 1.0 and usually, but not necessarily, worse the further away from 1.0 it becomes.  The evenness of the distribution of rows per value is also an important factor, with better distribution correlating with more even distributions.

With respect to space utilization, the number of distinct values should always be *very* much greater than the number of AMPs in the configuration to ensure that each table distributes its rows to all AMPs.

While this might not be possible to achieve for very small tables, it is an absolute necessity for anything larger.

If the number of rows per distinct value for a column is significantly larger than 1, then it might not be a good candidate for a primary index.

## Number of Rows Per Distinct Value

This value reports the largest number of rows per primary index column value.

FOR this type of primary index ...	The value for this measure is ...
unique	always 1.
non-unique	better as it approaches 1 and usually, but not necessarily, worse as it gets larger.

If the maximum number of rows per column value is much larger than 1, then the column is often not a good candidate for a primary index. Because the evenness of the distribution of rows per value is itself an important factor, with better distribution correlating with more even distributions, a large number of rows per distinct value is not necessarily an indicator that the column set is a poor choice for a primary index. The severity of the penalty paid for larger values is a function of several variables, including the cardinality of the table, the number of AMPs in the configuration, and so on.

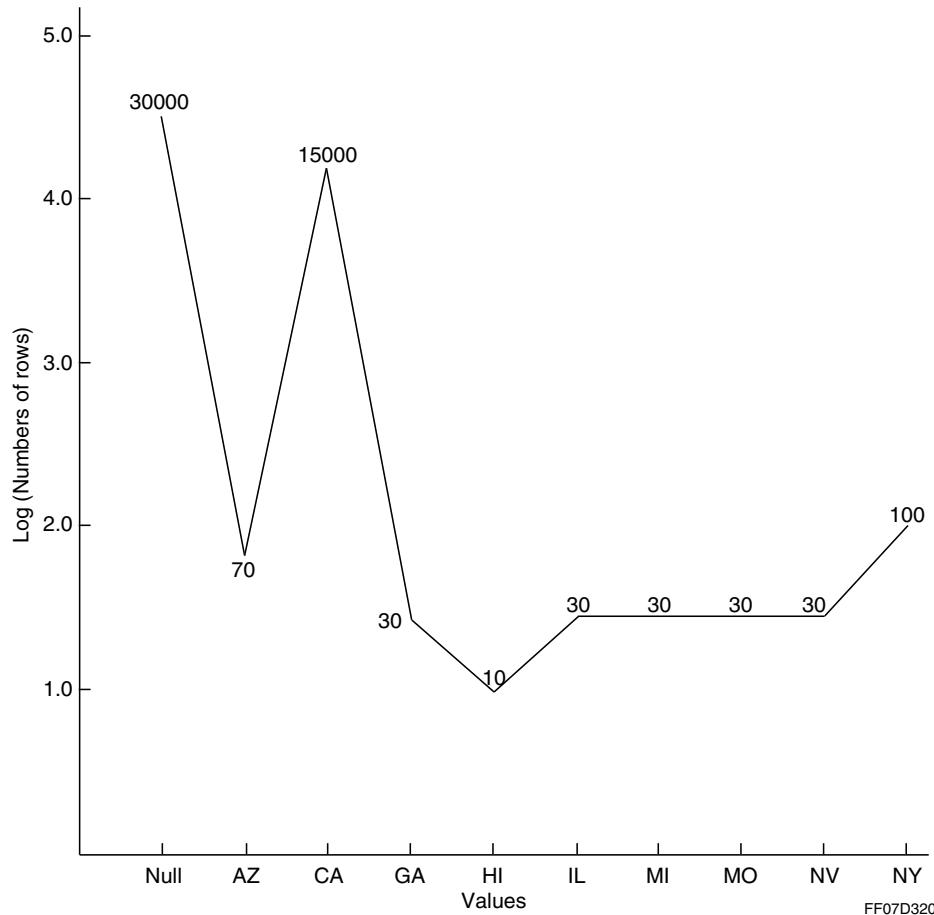
In the past, it was commonly stated that if the typical quantity of rows per column value does not fit into a single data block, then the column set is not a good candidate for a primary index. With the significant increase in data block size now used by most sites, the evaluation of this measure is less certain to provide strong guidance in picking a primary index column set.

You can graph these figures to provide an easy-to-comprehend graph of column value distributions. In the example provided here, a NUPI on an attribute called State was analyzed. Note the exceedingly skewed distribution, suggesting that State is not a good candidate for the primary index on this table.

The following table indicates the raw and logarithmic row cardinalities per state code value.

State Code	Number of Rows	$\log_{10}(\text{Number of Rows})$
Null	30,000	4.477
AZ	70	1.845
CA	15,000	4.176
GA	30	1.477
HI	10	1.000
IL	30	1.477
MI	30	1.477
MO	30	1.477
NV	30	1.477
NY	100	2.000

The following graph displays the row cardinalities on a logarithmic scale in the following graph. Notice how skewed the distribution is even when displayed on a logarithmic scale:



## Number of Null Rows

This value reports the number of rows having a null primary index.

FOR this type of primary index ...	The value for this measure is ...
unique	better as it approaches 0 and usually, but not necessarily, worse as it gets larger.
non-unique	

A very large number reflects a spike in the distribution, indicating the likelihood of serious problems with an uneven consumption of disk space because all rows with a null primary index hash to the same AMP (see “[Number of Distinct Values](#)” on page 422 for a description of the problem).

If the number of rows with null primary indexes columns is significantly larger than 0, then the column set is not a good candidate for a primary index. See [Chapter 13: “Designing for Missing Information,”](#) for a description of the many problems nulls cause for database management and suggestions about how to minimize or eliminate nulls from your tables.

## Effects of Skew on Query Processing Summarized

In the final analysis, a skewed rows per value measure does not necessarily indicate a problem. It is often possible to effect an even distribution of rows across AMPs and evenly distributed workloads when executing a query against skewed data. As long as the Optimizer has good statistics to work with, it is quite good at generating good query plans even when the distribution of table rows is skewed.

It is always better if the data is not skewed, but Teradata Database is better equipped to deal with skew than the database management systems of other vendors, and can often process skewed data successfully while other systems fail the task. Sometimes data is just naturally skewed, and the Optimizer has no choice but to deal with it.

## SQL Scripts For Detecting Skew

Notice that several different measures of skew can be made.

- Rows per value
- Rows per hash bucket
- Rows per AMP
- Rows per row partition
- Rows per row partition per AMP
- Rows per row partition per hash bucket
- Rows per row partition per value

Each type of skew can have a different effect on the query plan the Optimizer chooses, as does the concentration of the relevant rows within the data blocks.

If you are analyzing the demographics of an existing table, whether production or prototype, you can use the following set of useful scripts written by the Teradata technical support team to check for data skew. You can adjust the details of these statement to suit the needs of your site.

This is a good practice to undertake when new applications are being loaded on the system. It is also good practice to run these queries regularly if there are many data changes.

The following query identifies tables that are not evenly distributed. Ideally, variance should be less than 5%. In this query, variance is set to 1000%, which generally indicates that some or many AMPs have no rows from the table in question. You can use the BTEQ command RETLIMIT to limit the number of rows returned.

```

SELECT (MAX(CurrentPerm) - MIN(CurrentPerm)) * 100
      / (NULLIF(MIN(currentperm),0))(NAMED variance)
      (FORMAT 'zzzz9.99%'),MAX(CurrentPerm)(TITLE 'Max')
      (FORMAT 'zzz,zzz,zzz,999'),MIN(currentperm)
      (TITLE 'Min')(FORMAT 'zzz,zzz,zzz,999'),
      TRIM(DatabaseName)||'.'||TableName (NAMED Tables)
FROM DBC.TablesizeV
GROUP BY DatabaseName, TableName
HAVING SUM(CurrentPerm) > 1000000
AND variance > 1000
WHERE DatabaseName NOT IN('CrashDumps','DBC')
ORDER BY Tables;

```

Use the following query to display the detailed distribution of a table that has been identified as having a skewed distribution:

```

SELECT vproc, CurrentPerm
FROM DBC.TablesizeV
WHERE DatabaseName = '<dbname>'
AND TableName = '<tablename>'
ORDER BY 1;

```

The following query reports the row distribution of a table by AMP:

```

SELECT dt1.a (TITLE 'AMP'), dt1.b (TITLE 'Rows'),
      ((dt1.b/dt2.x (FLOAT)) - 1.0)*100 (FORMAT'+++9%',
      TITLE 'Deviation')
FROM (SELECT HASHAMP(HASHBUCKET(HASHROW(<index>))),COUNT(*)
      FROM <dbname>.<tablename>
      GROUP BY 1) AS dt1 (a,b),
      (SELECT (COUNT(*) / (HASHAMP() + 1))(FLOAT))
      FROM <dbname>.<tablename> AS dt2(x)
      ORDER BY 2 DESC,1;

```

The following query reports the distribution by AMP of the specified index or column.

```

SELECT HASHAMP(HASHBUCKET(HASHROW(<index or column>))),COUNT(*)
FROM <dbname>.<tablename>
GROUP BY 1
ORDER BY 2 DESC;

```

The following query reports the number of row hash collisions for the specified index or column.

```

SELECT HASHROW(index or column), COUNT(*)
FROM <dbname>.<tablename>

```

```
GROUP BY 1
ORDER BY 1
HAVING COUNT(*) > 10;
```

The following query reports the number of AMPs and the number of rows a given query accesses.

```
LOCKING TABLE <tablename> FOR ACCESS
SELECT COUNT(dt.ampNum) (TITLE '#AMPS'),
       SUM(dt.numRows) (TITLE '#ROWS')
  FROM (SELECT HASHAMP(HASHBUCKET(HASHROW(<index>))), COUNT(*)
         FROM <tablename>
        WHERE <selection criteria>
         GROUP BY 1)AS dt (ampNum, numRows);
```

## Scenario 1

### Table Structure and Update Schedule

Assume that a retail enterprise has a large, nonpartitioned primary index sales table containing the details of each transaction for the previous 24 full months plus the current month-to-date. Once a month, the transactions from the oldest month are deleted from the table. Current transactions are loaded into the table nightly using Teradata Parallel Transporter. Most transactions are added on the date they occur, but a small percentage of transactions might be reported a few days after they occur. The number of transactions per month is roughly the same for all months.

Each row contains, among other things, the product code for the item, the transaction date, an identifier for the sales agent, and the quantity sold. The rows are short, and the data blocks are large. The primary index is a composite of product code, transaction date, and the agent identifier.

### Query Workload

The query workload is a mix of tactical and strategic requests:

- There is a modest volume of short-running single-AMP (primary index) queries.
- There are many ad hoc queries follow a general pattern of comparing current-month-to-date sales to the same days of the previous month, or to the same days of the same month of the previous year for a few product code values.
- Some queries analyze agent performance, usually over an interval of a calendar quarter or less.
- Some queries examine sales trends over the previous 24 full months, usually for most or all product code values.

All the tables that support the workload have different primary indexes.

The sales table is frequently joined to relatively small tables containing information about each product code and each sales agent.

## Problem Statement

The current definition of *sales\_table* does not use row partitioning.

```
CREATE TABLE sale_stable (
    product_code      CHARACTER(8),
    sales_date        DATE,
    agent_id          CHARACTER(8),
    quantity_sold     INTEGER,
    product_description VARCHAR(50))
PRIMARY INDEX (product_code,sales_date,agent_id);
```

The DBA has been told to speed up the ad hoc queries and agent analysis queries. He considers two possible optimizations, neither of which uses row partitioning.

- Define either a UPI or a join index on the transaction date column.

The DBA then sets up tests for both scenarios. Unfortunately, the EXPLAIN reports show that the optimizer finds neither index to be selective enough to improve performance over a full-table scan, and does not use them.

- Split the table into 25 separate tables, each containing transactions for a calendar month, and then define a view that UNIONs all the tables. This view is intended to be used by the applications that analyze 24 months of sales history.

After some analysis, he concludes that this solution could indeed speed up the targeted queries, but that it also adds too much complexity for his end users, who would now have to understand the view structure and change the table names in their queries, code more complicated UNION statements, and select appropriate date and product code ranges.

The requirement to know the right table name also applies to the short-running single-AMP queries that specify primary index values. This proposed solution also complicates the nightly load jobs, especially in the first few days of a month when a small number of the transactions would be from the prior month, as well as complicating his long successful archive strategy. The DBA ultimately rejects this alternative as being too complicated and error-prone.

With these negative results in hand, the DBA next considers redefining the sales table with row partitioning.

## Analysis of Row Partitioning Benefits

Thinking that row partitioning might be an easy way to optimize his workloads for this situation, the DBA converts the sales table into a partitioned table partitioned by transaction month and finds that many of his critical queries run faster with no significant negative tradeoffs.

The DDL for the redefined sales table looks like this:

```
CREATE TABLE ppi_sales_table (
    product_code      CHARACTER(8),
    sales_date        DATE,
    agent_id          CHARACTER(8),
    quantity_sold     INTEGER,
    product_description VARCHAR(50))
PRIMARY INDEX (product_code,sales_date,agent_id)
PARTITION BY RANGE_N(sales_date BETWEEN DATE '2001-06-01'
```

```
AND      DATE '2003-06-31'
EACH INTERVAL '1' MONTH);
```

The following description examines each element of the workload as it relates to the newly redefined sales table, *ppi\_sales\_table*.

- The monthly deletes are faster because instead of a monthly Teradata Parallel Transporter delete job, the DBA can now perform a simple monthly ALTER TABLE statement to do the following things:

- a Drop the entire partition that contains the oldest data.
- b Create a few new partitions to hold data for the next few future months.

Deleting all the rows in a partition is optimized in much the same way that deleting all the rows in a nonpartitioned primary index table is optimized. For example,

- There is no need to record the individual rows in the transient journal as they are deleted.
- The rows for the month being deleted are stored contiguously on each AMP instead of being scattered more or less evenly among all the data blocks of the table, so there are fewer blocks to read.
- Most of the deletes are full-block deletes, so the block does not need to be rewritten.
- There is no need to touch any of the rows for the other months.

Dropping the oldest partition set is a nearly instantaneous operation, assuming there are no USIs or join indexes that must be updated.

The DDL for this ALTER TABLE statement looks like this for a selected monthly update:

```
ALTER TABLE sales_table
MODIFY PRIMARY INDEX (product_code, sales_date, agent_id)
DROP RANGE BETWEEN DATE '2001-06-01'
                  AND      DATE '2001-06-30'
ADD RANGE BETWEEN DATE '2003-07-01'
                  AND      DATE '2003-07-31'
WITH DELETE;
```

- The nightly Teradata Parallel Transporter insert job should run somewhat faster than before. Instead of the inserted rows distributing more or less evenly among all the data blocks of the table, they are concentrated in data blocks that correspond to the proper month. This increases the average hits per block count, which is a key measure of Teradata Parallel Transporter efficiency, as well as reducing the number of blocks that must be rewritten.
- The short-running, single-AMP queries are not affected by partitioning. Because the partitioning column is a component of the primary index, primary index access to the table is not changed.
- The largest gain is seen in the ad hoc queries comparing current month sales to a prior month. Only two of the 25 partitions need to be read, instead of the full-table scan required for a nonpartitioned table. This means that the number of disk reads will be reduced by roughly 92%, with a corresponding reduction in elapsed query response time. The 92% figure applies to the step that reads the sales table, not to the sum of all the steps used to implement the query.

Given the stated assumptions, the other steps should take roughly the same amount of time to complete as they did previously.

- The same considerations apply to the agent analysis queries. The number of partitions that must be read is determined by the time period specified in the query is known in most cases to be three or fewer. Even if the analysis is over twelve full months, there is roughly a 50% gain in reading twelve of 25 partitions for the step that reads the sales table.
- The decision support queries that analyze 24 months of sales data take roughly the same time and resources as they did for a nonpartitioned primary index table, although there is a marginal gain realized by reading 24 instead of 25 partitions. If the analysis is for 24 months plus the current month (the entire table), then the resource usage is the same as with the nonpartitioned primary index incarnation of the table because full-table scans are not affected by partitioning the primary index.
- The joins take the same amount of time. In the defined workload, the EXPLAIN reports indicate that there are no direct joins against the sales table. Instead, a spool file is always created from the sales table and the spool file is then joined to the smaller tables. In the partitioned table case, the spool file might be created more efficiently, but once created it is exactly the same as it is for the nonpartitioned table.

## Conclusions

The results show there are few disadvantages, and some significant advantages, to row partitioning this table given the workloads that access it. The partitioned table requires somewhat more disk space than its nonpartitioned counterpart. There is a 2-byte or 8-byte partition number recorded in each row that consumes additional storage space; however, the percentage increase seen for most row sizes does not exceed about 5%, and is often considerably less than that.

The following table summarizes the improvement opportunities for this case study:

Activity	Nonpartitioned Table	Partitioned Table	Improvement	Comments
Monthly delete of one month of data	Teradata Parallel Transporter job reads most blocks, updates most blocks	ALTER TABLE statement deletes partition	Much faster performance	Easier maintenance
Nightly inserts	Inserted rows scattered throughout table	Inserted rows concentrated in one partition	Faster performance	No changes to load script needed
Primary index access	1 block read	1 block read	No change	No SQL changes needed
Comparison of current month to prior month	All blocks read	2 partitions read	Step is 12 times faster (only 2 of 25 partitions read)	No SQL changes needed

Activity	Nonpartitioned Table	Partitioned Table	Improvement	Comments
Trend analysis over entire table	All blocks read	All blocks read	Little change	<ul style="list-style-type: none"> <li>Rows are two bytes longer for partitioning</li> <li>2% more blocks for 100 byte rows</li> </ul>
Joins	No direct Merge Joins	No direct Merge Joins	Little change	No direct merge joins in this example because of the choice of primary index.

## Scenario 2

### Table Structure and Update Schedule, Query Workload, and Problem Statement

This case study is a continuation of the scenario presented in “[Scenario 1](#)” on page 427. The table structure, update schedule, query workload, and problem statement are unchanged. The scenario evaluates whether the partitioning definition for this table can be improved further.

### Analysis of Row Partitioning Benefits

The partitioning definition used for “[Scenario 1](#)” on page 427 partitioned by month because many of the queries use months as their basic time unit. Another option is to attempt to optimize access further by changing the granularity of the partitioning scheme by partitioning at a finer level. Suppose the DBA considers partitioning the table by day instead of by month. The table now has about 760 partitions (25 months), with those corresponding to future dates in the current month being empty.

The results of telescoping the partition granularity for this table and query workload look like this:

- The effort to delete the oldest month of data is substantially the same as it was for the same table defined with 25 primary index partitions. The run time for that job is also substantially the same. Making the simplifying assumption that all months have 30 days, the DBA would have to delete 30 smaller partitions with the oldest month of data instead of deleting one larger partition, but the same number of rows are deleted, and the elapsed time to delete them is roughly the same.
- The nightly inserts benefit from the finer partitioning because instead of being concentrated in one or two partitions out of 25, the inserted rows are now directed to three, four, or at most five partitions out of 760; well under 1% of the partitions. Most of the inserts are directed to one partition, the one containing the activity for the day just completed.
- The short-running primary index access queries are not impacted either way by having 760 partitions instead of 25.

- The ad hoc queries that usually analyze two or three months of data are also little changed, now accessing roughly 60 out of 760 partitions rather than 2 out of 25, roughly the same percentage of the defined partitions for the primary index. Queries varying by the time of month, however, might realize some performance gain with the larger number of partitions. For example, a query submitted on the fourth day of the month might be likely to analyze the current day and the previous 33 days of data, while a query submitted later in the month might restrict the analysis to that calendar month. In this case, the 34 day query would involve 68 out of 760 partitions, instead of 4 out of 25, a significantly smaller percentage.
- The analysis queries examining 24 months of data run in about the same length of time because they touch most of the rows in the table in either case.
- The joins are not impacted by the number of partitions because there are no direct joins against this table.

## Conclusions

Increasing the number of partitions from 25 to 760 has only a modest effect on performance for this particular workload. The greatest gain is for queries that analyze only a few days of transactions.

# Scenario 3

## Table Structure and Update Schedule

While a transaction date or timestamp is frequently a good choice for the partitioning column, other choices present themselves for consideration for other categories of workloads and data. Consider a table with detailed information about telephone calls maintained by a telecommunications company. This table stores rows that contain the originating telephone number, a timestamp for the beginning of the call, and the duration of the call, among other things, for each outgoing call. Rows are retained for a variable length of time, but rarely for more than six weeks. Retention is based on the call date and the monthly bill preparation date. The primary index is a composite of the telephone number and the call-start timestamp. This column set implies that the index was chosen to provide good distribution, not to facilitate data access, and also that the likelihood of any direct primary index joins is remote.

## Query Workload

Some queries analyze all calls from a particular telephone number, while others analyze all calls for a particular period of time, perhaps for as long as a month, for customers meeting certain criteria.

## Problem Statement

The current definition of the call detail table does not use a partitioned primary index:

```
CREATE TABLE calldetail (
    phone_number      DECIMAL(10) NOT NULL,
```

```

call_start      TIMESTAMP,
call_duration   INTEGER,
call_description VARCHAR(30))
PRIMARY INDEX (phone_number, call_start);

```

Can the standard query workloads against this table be optimized by partitioning its primary index?

## Analysis of Partitioning Benefits

There are two candidate schemes for partitioning this table. The first partitioning scheme is to partition the rows by *call\_start*, probably defining one partition for each day. The results of this partitioning scheme for this table and query workload look like this.

- The timestamp partitioning scheme helps with inserting new transaction activity in the same way as the previous scenario (“[Scenario 1](#)” on page 427).
- Little performance gain is realized for row deletion because the delete operations are not performed strictly by call date. Because of this, the deleted rows are not clustered in a single partition.
- The analysis queries based on *call\_start* benefit from this partitioning scheme, with those queries that specify a range of a few days realizing the greatest gain.

The second possibility for partitioning this table is to use the telephone number. Telephone numbers contain too many digits to assign each number its own partition, but a subset of the digits can be used profitably to optimize telephone number-based row access. If, for example, the table is partitioned on the high order three digits of the telephone number, there are 1,000 partitions, some of which are always empty because of the way telephone companies assign numbers. The results of this partitioning scheme for this table and query workload look like this:

- This partitioning scheme does not improve the performance of Teradata Parallel Transporter bulk inserts or deletes because they are scattered across all partitions.
- The scheme does not facilitate date-based queries.
- The scheme allows queries that specify a telephone number to run much faster than they otherwise would because only one partition out of 500 or more partitions must be read to access the rows having that number.
- The scheme benefits geographic area analysis, at least in North America, because the first three digits of North American telephone numbers uniquely identify a narrowly defined geographic region.

If 1,000 partitions improve performance, then defining 10,000 partitions using the first four digits of the telephone number would probably improve performance even more. If 10,000 partitions are good, then 50,000 partitions might be better yet.

The DDL for the redefined *call\_detail* table looks like this:

```

CREATE TABLE ppi_call_detail (
    phone_number   DECIMAL(10) NOT NULL,
    call_start     TIMESTAMP,
    call_duration  INTEGER,
    call_description VARCHAR(30))

```

```
PRIMARY INDEX (phone_number, call_start)
PARTITION BY RANGE_N(phone_number/100000) (INTEGER) BETWEEN 0
AND 99999
EACH      2);
```

If mapping a geographic area to one or more partitions does not solve an application problem, then another potential solution is to maximize the number of partitions by using telephone number modulo 65,535 as the partitioning expression. Assuming the cardinality of the table is roughly 3.276 billion rows, then the average partition contains roughly 50,000 rows with this scheme. If the system has 100 AMPS, then each AMP contains roughly 500 rows per partition, a number that fits into one data block if the row width is fairly narrow. The decrease in response time of a single-partition scan for all activity for a particular telephone number is dramatic compared to the full-table scan that would be required for a nonpartitioned table.

A query to return activity for one telephone number from this table is a best case scenario for single-table response time improvement by using row partitioning. Disregarding the overhead cost of initiating the query and returning the answer set, the elapsed time could be reduced to  $\frac{1}{65535}$  of the time using a nonpartitioned table. Including the query initiation and termination overhead, the total query time improvement would be somewhat less than a factor of 65,535, but could be less than  $\frac{1}{10000}$  of the nonpartitioned primary index time. The DDL for the *ppi\_call\_detail* table using this scheme is as follows.

```
CREATE TABLE ppi_call_detail (
    phone_number DECIMAL(10) NOT NULL,
    call_start    TIMESTAMP,
    call_duration INTEGER,
    other_columns CHARACTER(30))
PRIMARY INDEX (phone_number, call_start)
PARTITION BY phone_number MOD 65535 +1;
```

The workload mix determines which, if any, of these proposed partitioning schemes is best. You can begin your analysis with the extended logical data model, but actual testing of the various proposed scenarios is often required.

## Conclusions

The partitioning scheme you use should be tied directly to the application workload accessing its data. For this scenario, the proposed partitioning schemes affect geographic localization applications differently than they do call date-based applications. To decide which scheme is better, you need to know all the various ways the table is currently accessed and have a solid concept of how it might be accessed in the future.

You need to examine the relative efficiencies of different partitioning schemes for the same table must from that perspective.

## Scenario 4

### Dealing With Ambiguous Scenarios

The previous scenarios illustrate situations where a row-partitioned table is the obvious choice to enhance the performance of a query workload. This scenario examines a more

ambiguous situation in which there are more tradeoff considerations and it is not possible to determine in advance one correct solution for all specific instances of the scenario.

## Table Structure and Update Schedule

An invoice table contains data about each invoice issued in the past four years. The unique primary index is invoice number. New rows are added nightly, using Teradata Parallel Transporter Update Operator, and the oldest month of data is deleted once each month.

## Query Workload

A moderately heavy volume of queries requests information about one specified invoice. There are also ad hoc analysis queries that examine all invoices for some period of time, which is usually less than one year. Other tables have invoice number as their primary index, but do not have an invoice date column, so there are frequent joins with those other tables.

## Problem Statement

The DBA is considering whether it would be advantageous to partition the invoice table on invoice date using one-month ranges.

The primary index is currently defined as unique, but would have to be redefined as nonunique if the table were row-partitioned. There is a business requirement to guarantee that invoice numbers are unique, so the DBA would need to define a uniqueness constraint on the invoice number column. If this uniqueness constraint is added, it creates an additional secondary index on the table (other than UPIs, all uniqueness constraints are implemented internally as USIs irrespective of whether they are specified explicitly as a UNIQUE constraint, a PRIMARY KEY constraint, or a USI constraint. See “[Using Unique Secondary Indexes to Enforce Row Uniqueness](#)” on page 457), which increases processing on insert, delete, and update operations, as well as requiring additional disk capacity to store the resulting secondary index subtable. The base table is also larger by two bytes per row, further increasing the required disk space.

## Analysis of Partitioning Benefits

The primary index access queries that were run against the nonpartitioned version of this table must be reformulated to use the USI to access the row. As a general rule, accessing a row takes roughly two to three times longer using a USI than it would using a UPI. However, UPI access is a very fast operation, so doubling or tripling the time might barely be noticeable to the users who issue those queries.

Without row partition elimination, direct Merge Joins require, at best, more memory and CPU utilization and might be measurably slower compared to a similar nonpartitioned table. The extent of performance degradation depends on the query conditions, how many partitions can be excluded, and the specific join plan chosen by the Optimizer. Actual measurement of representative queries is necessary to determine the overall difference in performance.

The nightly inserts benefit in the same way, and for the same reasons, as in “[Scenario 1” on page 427](#). However, the additional index on invoice number partially offsets that benefit. The same considerations apply to the monthly delete operations.

The ad hoc queries examining several months of invoices benefit in the same way as in “[Scenario 1” on page 427](#). The benefit is greatest when fewer months are examined.

Would it be worthwhile to convert the invoice table to a partitioned table? The DBA must measure the degree of improvement as well as the extent of degradation in the various types of query, and use that analysis to determine how much each query type contributes to the overall workload involving this table. This exercise produces a good estimate of the comparative workload performance against the table with and without partitioning.

If the measured performance difference between the otherwise equivalent partitioned and nonpartitioned tables is substantial, in either direction, then the choice might appear to be obvious. However, you must also weigh the relative importance to the enterprise of the various activities in the workload.

For example, consider the following contingencies:

- If the time required to perform the nightly volume of bulk inserts is beginning to exceed the time allotted for inserting new rows, then even a small improvement in load time might be considered sufficiently important to offset larger degradations in other aspects of the query workload.
- Similarly, if the response time of the PI-access queries is critical, even a small performance degradation might be considered unacceptable, whether net workload performance is improved or not.

## Conclusions

The decision whether to implement a table with or without partitioning is not always cut and dried, and the ultimate decision, like many others in physical database design, can often be more of an optimization than a maximization. In this scenario, careful and considered measurement, analysis, and evaluation are all required to make an optimal decision.

# Locating a Row Using Its Primary Index

Assume the following abstract SQL query.

```
SELECT column_1, column_2
  FROM table_1
 WHERE unique_primary_index = column_value;
```

This query instructs Teradata Database to retrieve the single row having the unique primary index value *column\_value* and then to display the values for *column\_1* and *column\_2* of that row.

## Process for Locating a Row Using Its Unique Primary Index

The following process describes how Teradata Database locates the row having the unique primary index value *column\_value*. Teradata Database follows the same process if the primary index is non-unique. The only difference is that the query might retrieve more than a single row in that case because multiple rows could have the same value for *column\_value*.

- 1 The Resolver does a data dictionary lookup to determine the table ID for *table\_1*, then adds it to the parse tree for later use by the Generator.
- 2 The Generator hashes the primary index value *column\_value*, computing its 32-bit row hash value.
- 3 The Generator builds a 3-part message from the following information.
  - Table ID for *table\_1*
  - The number of bits for the row hash value is always 32, but the number of bits used to represent the hash bucket number and remainder vary depending on the number of hash buckets defined for the system.

IF the system has this many hash buckets ...	THEN the row hash for <i>column_value</i> is 32 bits wide divided between the hash bucket number and remainder as follows ...
65,536	<ul style="list-style-type: none"> <li>• 16-bit hash bucket number</li> <li>• 16-bit remainder</li> </ul>
1,048,576	<ul style="list-style-type: none"> <li>• 20-bit hash bucket number</li> <li>• 12-bit remainder</li> </ul>

See “[Teradata Database Hashing Algorithm](#)” on page 225 for details.

- Data value for *column\_value*
- 4 The Generator scans the hash map to determine which AMP owns the hash bucket the row belongs to.
- 5 The message is inserted into an AMP step, the Dispatcher places it on the BYNET, and sends it point-to-point to the AMP identified by the hash map.
- 6 The file system on the receiving AMP uses the table ID and row hash value as a key to scan its master index for the cylinder number that contains the data block in which the row is stored.
- 7 The file system first determines if the cylinder index is cached.

WHEN the cylinder index is ...	THEN the file system ...
cached	scans it for the data block.
not cached	retrieves it from disk and scans for the data block (see stage 8).

- 8 The file system uses the table ID, row hash value, and cylinder number as a key to scan the cylinder index for the data block address known to contain the row being retrieved.

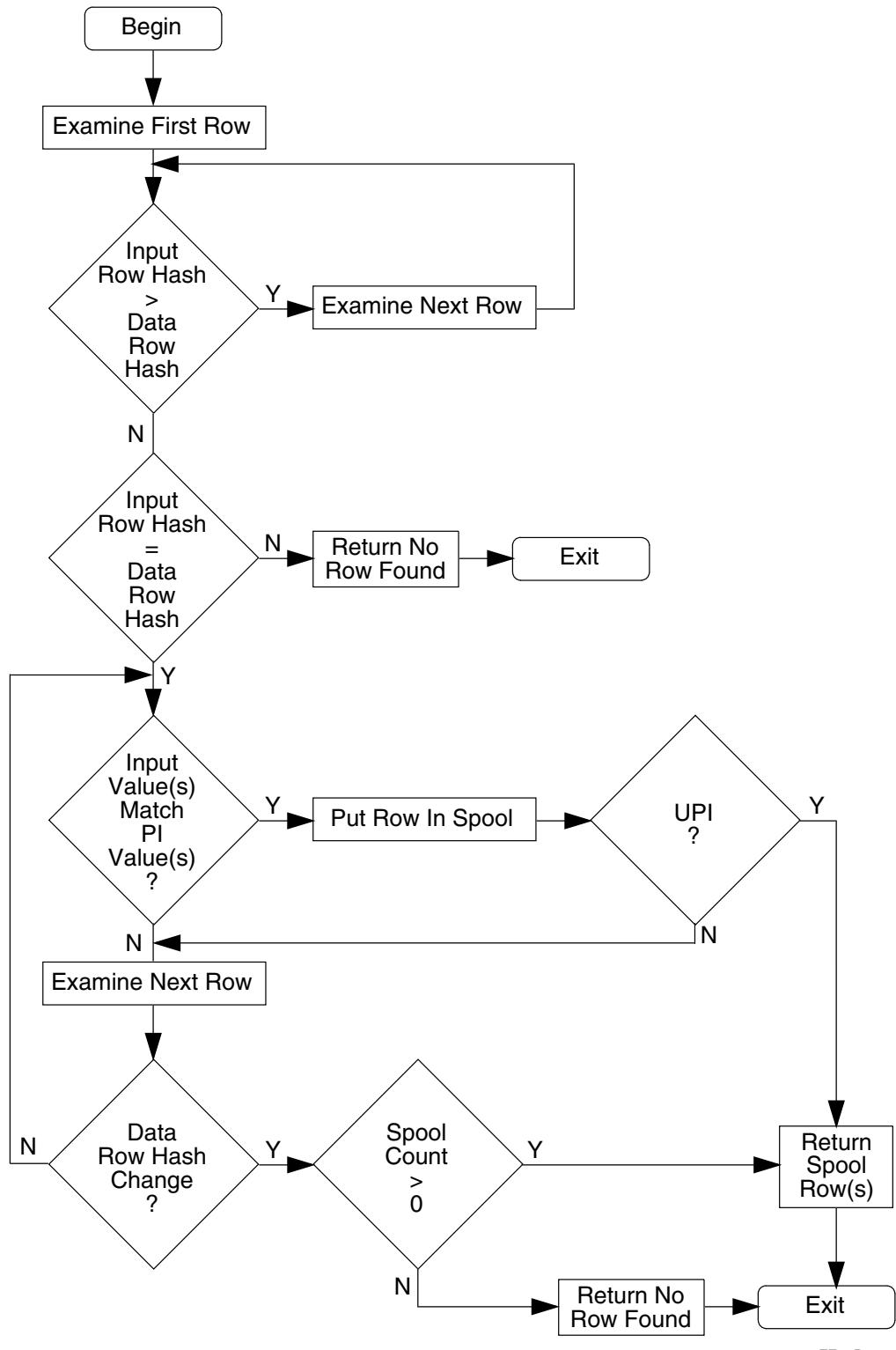
- 9 The file system determines if the data block is cached.

WHEN the data block is ...	THEN the file system ...
cached	scans it for the row.
not cached	retrieves it from disk and scans for the row (see stage 10).

- 10 The AMP uses the row hash value and primary index value as a key to scan the data block for the row being retrieved. The AMP checks to see if the row contains the desired values, and if it does, returns the values for *column\_1* and *column\_2* to the requestor.

## Primary Index Reads

The following flowchart illustrates the flow of a generic primary index read. The flowchart begins at the point that the data block containing the searched row has been located.



FF07D316

## Example: Primary Index Read

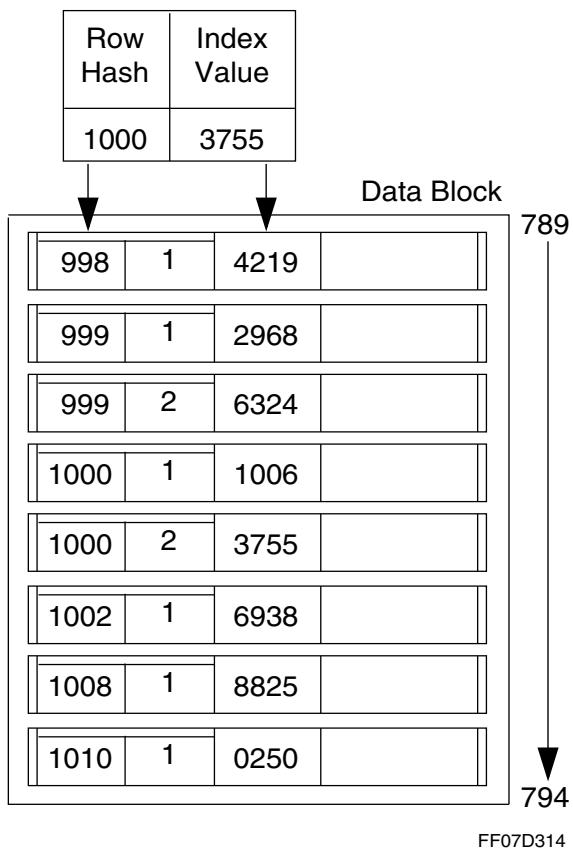
This example illustrates how the Teradata file system uses the primary index to eliminate synonyms from the returned result by performing a row hash search.

The original query is the following:

```
SELECT *
FROM employee
WHERE employee_number = 3755;
```

At this point in the process, the master index lookup showed that the row representing employee number 3755 is stored on cylinder 169. The cylinder index lookup showed that the row is in the six sector block beginning with sector number 789.

The following graphic illustrates the logical representation of that data block.



The file system must locate the row having a row hash value of 1000 and a primary index data value of 3755. The following process describes how the file system undertakes this final task.

- 1 The file system begins scanning the data block beginning at sector 789.
- 2 The first row found with a row hash value of 1000 has an index data value of 1006. This is a hash synonym for the searched row.

Because this is not the row specified by the WHERE clause predicate, the scan continues.

- 3 The next row found with a row hash value of 1000 has an index data value of 3755.

Because this is the requested row and it was accessed using a UPI, the data block scan terminates and the row is returned to the requestor. Had the index been a NUPI, the file system would have had to read another row to know that no further rows matched.

# Performance Considerations for Primary Indexes

This topic describes general performance considerations for both unique and nonunique primary indexes, paying particular attention to how the range of primary index singularity affects performance.

## Guideline for All Primary Indexes

Define the primary index for a table on as few columns as possible. Retrievals on columns that do not match the *entire* primary index do not use the primary index to retrieve the matching rows.

Hashing efficiency increases as the number of primary index columns decreases.

## Guidelines for Row Partitioning

The key guideline for determining the optimum granularity for the row partitions of a partitioned table is the nature of the workloads that most commonly access the PPI table or join index. The higher the number of row partitions you define for a partitioned table, the more likely an appropriate range query against the table will perform more quickly, given that the partition granularity is such that the Optimizer can eliminate all but one partition.

On the other hand, it is generally best to avoid specifying too fine a partition granularity. For example if query workloads never access data at a granularity of less than one month, there is no benefit to be gained by defining partitions with a granularity of less than one month. Furthermore, unnecessarily fine partition granularity is likely to increase the maintenance load for a partitioned table, which can lead to overall system performance degradation. So even though too fine a partition granularity itself does not introduce performance degradations, the underlying maintenance on such a table *can* indirectly degrade performance.

You should also consider the following items if a table you are designing is planned to support tactical queries. By knowing the specifics of your workloads, you can optimize your partitioning to best suit the queries in those workloads.

- For all-AMP tactical queries against partitioned tables, you should specify a constraint on the partitioning column set in the WHERE clause.
- If a query joins partitioned tables that are partitioned identically, using their common partitioning column set as a constraint enhances join performance still more if you also include an equality constraint between the partitioning columns of the joined tables.

## General Considerations

The following factors apply equally to Teradata Database UPIs and NUPIs.

- Each table has 0 or 1, and no more than 1, primary index.

- The primary index for a table need not be isomorphic with the primary key for the table. This is obviously true for NUPIs, because primary key values are unique by definition.
- A primary index can contain nulls, while a primary key cannot.  
Primary index nulls are not a good thing. If you anticipate that the primary index for a table might frequently be null, you should consider using a different column.
- All single-value primary index accesses are confined to a single AMP. In most cases, a single-value primary index access requires only 1 I/O.  
If the cylinder index is not in cache, a primary index access requires 2 I/Os.

## Unique Primary Index Considerations

The following factors describe the general performance considerations associated with UPIs.

- UPIs guarantee an even distribution of table rows across the AMPs when there are many more unique values than AMPs on the system.
- UPIs always guarantee the best retrieval and update performance for single table row access.
- UPI values are never associated more than one base table row.
- UPIs retrievals and updates never require a spool file.
- Uniqueness is enforced by the system using the UPI (thereby avoiding the need for duplicate row checks for multiset tables).

## Nonunique Primary Index Considerations

The following factors describe the general performance considerations associated with NUPIs.

- At best, NUPIs distribute table rows evenly across the AMPs and hash values.  
At worst, NUPI table row distribution can be skewed in various ways (see “[SQL Scripts For Detecting Skew](#)” on page 425) and, if so, decrement both retrieval and update performance.
- NUPIs at best can effect good retrieval and update performance.
- NUPI values can involve more than one base table row.
- NUPI retrievals can require spool files.
- Duplicate NUPI values hash to the same AMP and often are stored in the same data block.
- If all the rows for a duplicate NUPI hash value fit into a single data block, then only 1 or 2 I/Os are required to store or access the entire set.  
2 I/Os are only required when the cylinder index is not cached.
- If duplicate rows are excluded because the table is defined as SET and has no uniqueness constraint, then the system must make a duplicate row check for every table row inserted or updated.

See “[Duplicate Row Checks for NUPIs](#)” on page 444 and “[Using Unique Secondary Indexes to Enforce Row Uniqueness](#)” on page 457 for further details.

# Normalization, Denormalization, and Primary Index Usage

The following example shows how a well-designed normalized table set can be as high-performing as the equivalent denormalized form while avoiding all the redundancy and update anomaly difficulties of denormalization. Note that denormalization is a *logical*, not a physical, concept (see [Chapter 7: “Denormalizing the Physical Schema”](#)). The term is used here only because it is most commonly understood in such a context, not because it is correct.

The most frequently performed query against these tables determines the current price for a given part.

## Denormalized Table Set

The first set of tables is denormalized by including *current\_effective\_date* and *current\_price\_amount* columns in the *part* table. Because the price of the part is also carried in the *part\_price\_history* table, this is a violation of normalization due to redundant data.

part 7,000,000 rows					part_price_history 70,000,000 rows		
part_num	quantity	description	current_effective_date	current_price_amount	part_num	former_effective_date	former_price_amount
			DD				
PK					PK		
1234	100	Widget	2007/01/01	14.25	FK		
1235	97	Thingie	2007/01/01	28.99	1234	2005/01/01	13.95
					1234	2005/01/01	15.45
					1235	2005/01/01	27.89
					1235	2005/01/01	30.99

## Normalized Table Set

The second set of tables is normalized. By defining a NUPI on *part\_num*, you can determine the current price of any part by reading only one additional block on the same AMP that holds the *part* table row.

With a composite UPI defined over *part\_num* and *effective\_date*, the lookup cost for determining the current price of a part would be substantially higher.

part 7,000,000 rows			part_price_history 70 000 000 rows		
part_num	quantity	description	part_num	effect_date	price_amount
PK			PK		
UPI			FK		
1234	100	Widget	NUPI		
1235	97	Thingie	1234	2000/01/01	13.95
			1234	2003/01/01	14.25
			1234	2007/01/01	15.45
			1235	2000/01/01	27.89
			1235	2005/01/01	28.99
			1235	2007/01/01	30.99

## Duplicate Row Checks for NUPIs

Teradata Database tables defined with the SET attribute in the CREATE TABLE kind clause do not permit duplicate rows. Any time a row is inserted into a SET table having a NUPI and no uniqueness constraints or indexes, the system performs a duplicate row check.

When Teradata Database checks for duplicate rows, it considers nulls to be equal, while in SQL conditions, null comparisons evaluate to UNKNOWN. See “[Inconsistencies in How SQL Treats Nulls](#)” on page 674 for information about how nulls are interpreted by commercially available relational database management systems.

### Performance Effects of Duplicate NUPI Row Checks

Duplicate row checking can be a very I/O-intensive operation. The table in the topic “[Duplicate NUPI Row Read I/O as a Function of Number of Rows Inserted](#)” on page 445 illustrates the number of logical reads performed for the duplicate row checks required by a given number of duplicate NUPI row insertions under the following conditions:

- The table is a SET table (no duplicate rows permitted).
- There are no previous hash synonyms for the NUPI value being inserted.

The center column in the table indicates the number of logical reads required before the number of rows specified in the left column can be inserted into the table.

The right column indicates the cumulative number of logical reads required before the number of rows listed in the left column can be inserted. The cumulants are determined from the following equation.

More formally, the number of NUPI rows that must be read can be expressed as follows.

$$(\text{column 3, row } n) = (\text{column 2, row } n) + (\text{column 3, row } n-1)$$

$$\text{Number of rows to read} = n \times \frac{(n-1)}{2}$$

where  $n$  = number of NUPI rows.

### Duplicate NUPI Row Read I/O as a Function of Number of Rows Inserted

This table illustrates the absolute and cumulative number of read I/Os required as a function of the number of duplicate NUPI rows inserted into a table. These figures are not a function of the number of AMPs in the configuration because duplicate NUPI values all hash to the same AMP. Therefore, the magnitude of the effect is independent of the number of AMPs on the system.

As you can see, the read I/O burden is tractable up to about 100 duplicate NUPI rows inserted. Beyond that point, the read I/O requirements are increasingly *intractable*.

The performance of any system is noticeably worsened when any additional duplicate NUPI rows are added.

Number of Rows Inserted	Number of Prior Logical Reads Required to Process This Number of Inserted Rows	Cumulative Number of Logical Reads
1	0	0
2	1	1
3	2	3
4	3	6
5	4	10
6	5	15
7	6	21
8	7	28
9	8	36
10	9	45

Number of Rows Inserted	Number of Prior Logical Reads Required to Process This Number of Inserted Rows	Cumulative Number of Logical Reads
20	19	190
30	29	435
40	39	780
50	49	1,225
60	59	1,770
70	69	2,415
80	79	3,160
90	89	4,005
100	99	4,950
200	199	19,900
300	299	44,850
400	399	79,800
500	499	124,750
600	599	179,700
700	699	244,650
800	799	319,600
900	899	404,550
1,000	999	499,500

### Guidelines for Keeping NUPI Duplicates Below 100 Rows Per Value

As you can see in the previous table, beyond more than a very few rows, the number of logical reads required to insert duplicate NUPI value rows is prohibitive if duplicate row checks are done.

The file system must read all the rows on an AMP that have the same row hash in order to determine their uniqueness values before it can insert a new NUSI duplicate row. Beyond 100 rows per value, the performance decrement created by that activity becomes significant, so you should always limit NUPI duplicates to fewer than 100 rows per value for any table. This figure is dependent on data block and row sizes: the larger the row size, the fewer rows fit into a single data block. Similarly, the larger the data block, the more rows, on average, fit into it. Keep this in mind as you read this section and realize that the 100 rows per value measure is a relative, not an absolute, criterion.

The figure of 100 duplicates per NUPI value is based on the overwhelming likelihood that duplicate NUPI value rows will spill over into more than five data blocks. When the number

of duplicate NUPI values in a table exceeds 100, performance *always* degrades significantly. The specific operations affected are described in the following table.

Operation	Cause of Performance Degradation
Updates	Increased I/O
<ul style="list-style-type: none"> <li>• Inserts</li> <li>• FastLoads</li> </ul>	Increased comparisons, resulting in greater I/O activity and CPU usage

Note that performance for the following operations is also degraded significantly when the number of duplicate NUSI row values exceeds 100:

- Restore operation of the Archive/Recovery utility
- Table rebuilds using the Table Rebuild utility

These performance problems persist even for the unlikely case where every NUPI value in a table has the same number of duplicates, thus eliminating skew as a factor in the decrement.

## Number of Reads for Inserting 10, 100, and 1,000 NUPI Duplicate Rows

Note that a single order of magnitude increase in the number of duplicate NUPI row inserts into a table, from 10 to 100, results in a two orders of magnitude increase in the cumulative number of block reads that must be performed.

When duplicate NUPI inserts increase by another order of magnitude, from 100 to 1,000, there is an additional increase of *four* orders of magnitude over the cumulative number of reads required to insert 10 duplicate NUSI rows.

The count of cumulative read operations in the following table is a measure of data block reads, not individual row reads. For example, if there are 100 rows in a data block, then only one block read is required to retrieve all 100 rows.

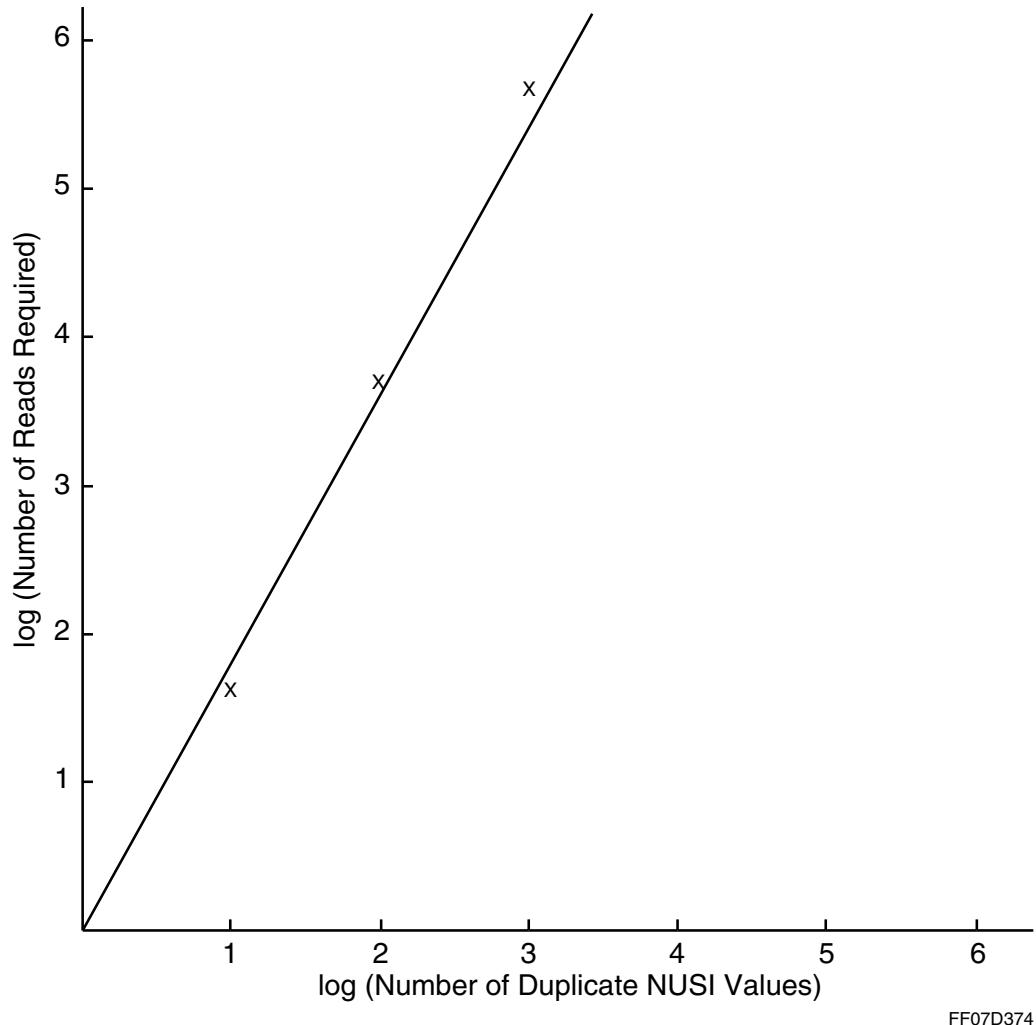
Table 1:

To insert this many duplicate NUSI rows ...		The system must perform this many cumulative data block read operations ...	
N	$\log_{10}N$	N	$\log_{10}N$
10	1	45	1.6532
100	2	4,950	3.6946
1,000	3	499,500	5.6985

## Cumulative Number of Data Block Read I/Os as a Function of Duplicate NUPI Inserts

A log-log plot of the data from the preceding table indicates extremely exponential growth in the cumulative number of data block read I/Os required to process duplicate NUPIs as a

function of the number of duplicate NUPI rows on the AMP. Notice that these are data block reads, *not* individual row reads. For example, if there are 100 rows in a given data block, only one data block read is required to retrieve all 100 rows.



## Minimizing Duplicate NUPI Row Checks

There are several ways to minimize or even eliminate duplicate NUPI row checks while still preventing duplicate row from being inserted. This topic describes some of the ways to achieve that goal.

- Use a non-primary index column set to define a UNIQUE constraint or USI.
- Use multiple columns to define NUPIs in order to make the index as close to being unique as possible.
- Keep the number of NUPI duplicate rows for each value below 100.

## Eliminating NUPI Duplicate Row Checks

You can eliminate duplicate row checking entirely by defining a uniqueness constraint such as PRIMARY KEY or UNIQUE NOT NULL or a USI on the primary key (or some other alternate key that can be constrained uniquely) of the table. See “[Using Unique Secondary Indexes to Enforce Row Uniqueness](#)” on page 457 for the factors that need to be considered prior to making this design decision.

## Optimizing Performance by Using Multicolumn NUPIs

You can minimize duplicate row check performance issues by making your NUPIs as close to being unique as possible. The more singular the NUPI value (that is, the closer to being unique it is), the more likely all rows having that NUPI can be stored within a single data block.

A powerful method for achieving the goal of maximal singularity is to define the NUPI on multiple columns. If you decide to use this approach to enhance the uniqueness of a NUPI, keep in mind that you should also define primary indexes on the *smallest* possible column set (see “[Primary Index Value Retrieval Access: Definition](#)” on page 417). The goal is to optimize the tradeoff between enhancing the generality of the index for row retrieval and reducing the number of duplicate row checks that must be performed.

Drawbacks of this method are described in “[Advantages and Disadvantages of Multicolumn NUPIs](#)” on page 449.

Consider the following example. Suppose you have a table with 3 name columns: *last\_name*, *first\_name*, and *middle\_name*. You determine that you must use one or more of the name columns as the NUPI for the table.

- 1 Start with *last\_name*.

Depending on your demographics, this might be a usable choice. If your demographics indicate that your population has mostly English names, you will probably find numerous Johnsons, Smiths, and Jones among your last name pool.

This would provide a fairly skewed distribution of rows across your AMPs.

- 2 Now add *first\_name* to *last\_name*.

Given the same population, you are still likely to find multiple names like Robert Smith or Jennifer Johnson, but the singularity of this NUPI is still greatly enhanced over the single column *last\_name* NUPI.

- 3 Now add *middle\_name* to *last\_name* and *first\_name*.

You might still find duplicate names like Robert David Smith, but the probability is great that there are fewer Robert David Smiths in your population than there are Robert Smiths.

The distribution of rows should be fairly even using this NUPI.

## Advantages and Disadvantages of Multicolumn NUPIs

The following table lists the advantages and disadvantages of multicolumn NUPIs.

Advantages	Disadvantages
Singularity is enhanced.	Primary index retrievals are possible only when you specify all of the NUPI columns in the WHERE clause of your SELECT requests.
Number of rows per value is lessened.	Partial primary index values cannot be hashed.
Selectivity is enhanced.	Usability is decreased.
<b>Summary Statement</b>	
The more columns defined for a NUPI, the closer it is to being unique.	The more columns defined for a NUPI, the less generalized its use.

## Keep the Number of Duplicate NUSI Value Rows Per Table Below 100

See “[Duplicate Row Checks for NUPIs](#)” on page 444 for a detailed explanation of why it is essential to follow this guideline.

# Primary Index Uniqueness and Row Distribution

The singularity of the primary index for a table is directly related to how evenly table rows are distributed across the AMPs. The more singular the index, the more optimal the use of disk space, with primary index uniqueness producing the most efficient space utilization. Because primary indexes are a component of base table rows, they do not require subtables or any other disk space beyond that required to store the primary index column data in the base table row.

The following mini-case studies illustrate how the degree of singularity of the values in the column selected to be the primary index for a table affects the distribution of rows across the AMPs.

The examples use the following *customer* table as their basis. Assignment of the primary index varies from case to case. Only a small sample of table rows is shown. Assume a 6-AMP system.

customer				
cust_num	cust_name	cust_status	parent_cust_num	sales_emp_num
PK			FK	NN
SA				
00001	J&R Sales	A	183	1252
00002	Nota Bene Ltd.	I	520	0492

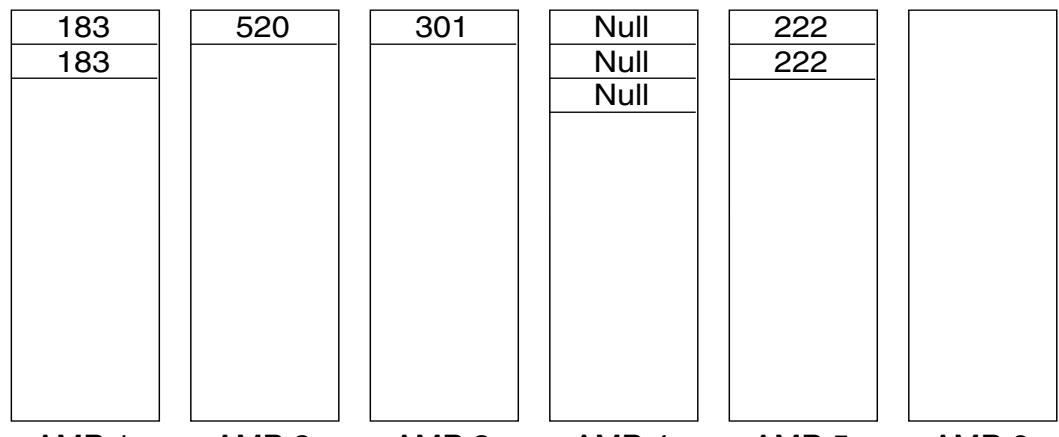
customer				
cust_num	cust_name	cust_status	parent_cust_num	sales_emp_num
PK			FK	NN
SA				
00003	PSI, Inc.	A	301	0655
00004	Vodun and Sons	A	Null	1973
00005	JellyJamUp	A	222	0034
00006	Kimble Korp.	A	Null	0005
00007	Kraus GmbH	I	Null	0708
00008	Piccolo SpA	I	183	0708
00009	The Wiltshire Company	A	222	1439
...	...	...	...	...

### Case Study 1: Fewer Distinct Primary Index Values Than AMPs

This study defines a NUPI on *parent\_cust\_number*, the foreign key. There are only five distinct values for this attribute in the entire *customer* table.

Never define a primary index on a column that has fewer distinct values than the number of AMPs in the system unless no other columns are available. Because only five of the 65,536 available hash buckets are used for this table, the very best distribution you can hope for is that each of those five buckets is assigned to a different AMP.

The illustration shows an example of such a best case distribution of rows.

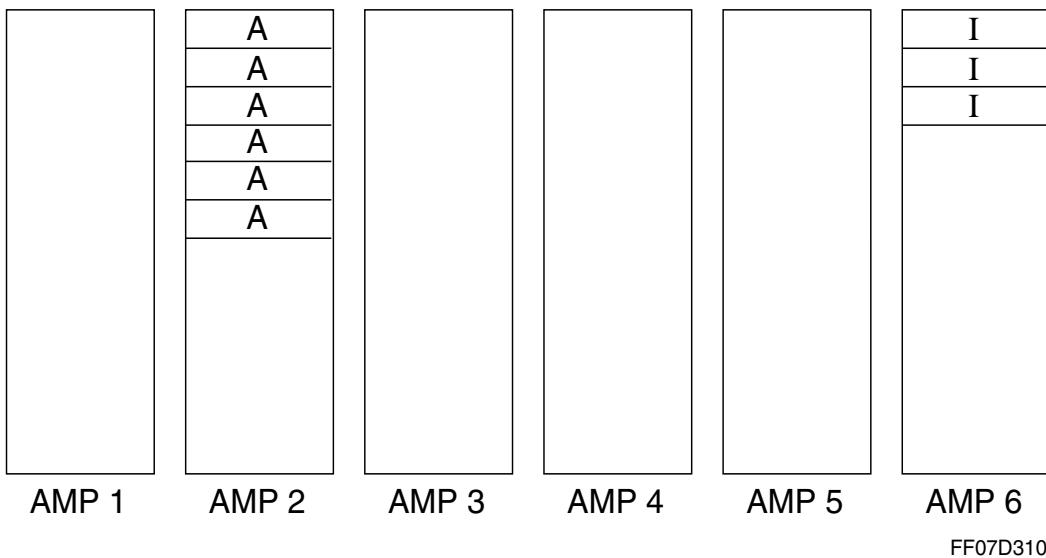


## Case Study 2: Highly Nonunique Primary Index Values

This study defines a NUPI on *customer\_status*. Because there are only two distinct values for this attribute (Active and Inactive), all rows hash to two AMPs, leaving the other four AMPs in the system unaffected by the *customer* table.

Distributions that are skewed like this cause major performance problems because they fail to harness the powerful parallel processing capabilities that make Teradata unique among commercially available relational database management systems.

The illustration shows the *customer* table rows stored on only two AMPs.



## Case Study 3: Highly Singular Primary Index Values

A carefully selected NUPI distributes the rows of a table nearly as evenly as a UPI. If you are fortunate, the NUPI column might never contain duplicates.

The illustration shows how evenly *customer* table rows are hashed when a NUPI is defined on the *sales\_employee\_number* attribute.

<table border="1"> <tr><td>1252</td></tr> <tr><td>0034</td></tr> <tr><td>0937</td></tr> <tr><td></td></tr> </table>	1252	0034	0937		<table border="1"> <tr><td>0708</td></tr> <tr><td>0708</td></tr> <tr><td>2321</td></tr> <tr><td>1865</td></tr> <tr><td></td></tr> </table>	0708	0708	2321	1865		<table border="1"> <tr><td>1439</td></tr> <tr><td>0434</td></tr> <tr><td>3515</td></tr> <tr><td></td></tr> </table>	1439	0434	3515		<table border="1"> <tr><td>0492</td></tr> <tr><td>0005</td></tr> <tr><td>1131</td></tr> <tr><td></td></tr> </table>	0492	0005	1131		<table border="1"> <tr><td>1973</td></tr> <tr><td>7316</td></tr> <tr><td>6310</td></tr> <tr><td>6310</td></tr> <tr><td></td></tr> </table>	1973	7316	6310	6310		<table border="1"> <tr><td>0655</td></tr> <tr><td>5994</td></tr> <tr><td>4216</td></tr> <tr><td></td></tr> </table>	0655	5994	4216	
1252																															
0034																															
0937																															
0708																															
0708																															
2321																															
1865																															
1439																															
0434																															
3515																															
0492																															
0005																															
1131																															
1973																															
7316																															
6310																															
6310																															
0655																															
5994																															
4216																															
AMP 1	AMP 2	AMP 3	AMP 4	AMP 5	AMP 6																										

FF07D311

#### Case Study 4: Unique Primary Index Values

If a table has many more unique values than the system has AMPS, a UPI always provides an even distribution of rows across the AMPS. To ensure efficient space utilization, you should define the primary index for a table on a unique column unless you plan to hash the rows of a minor entity table to the same AMPs as matching rows from a major entity table in order to enhance join processing. Be aware that join indexes might be a better solution, depending on the particular application. See [Chapter 11: “Join and Hash Indexes,”](#) for more information.

The illustration shows an ideal distribution of *customer* table rows when a UPI is defined on the system-assigned primary key, the *customer\_number* attribute.

<table border="1"> <tr><td>00001</td></tr> <tr><td>00007</td></tr> <tr><td>00013</td></tr> <tr><td>00019</td></tr> <tr><td></td></tr> </table>	00001	00007	00013	00019		<table border="1"> <tr><td>00002</td></tr> <tr><td>00008</td></tr> <tr><td>00014</td></tr> <tr><td>00020</td></tr> <tr><td></td></tr> </table>	00002	00008	00014	00020		<table border="1"> <tr><td>00003</td></tr> <tr><td>00009</td></tr> <tr><td>00015</td></tr> <tr><td>00021</td></tr> <tr><td></td></tr> </table>	00003	00009	00015	00021		<table border="1"> <tr><td>00004</td></tr> <tr><td>00010</td></tr> <tr><td>00016</td></tr> <tr><td>00022</td></tr> <tr><td></td></tr> </table>	00004	00010	00016	00022		<table border="1"> <tr><td>00005</td></tr> <tr><td>00011</td></tr> <tr><td>00017</td></tr> <tr><td>00023</td></tr> <tr><td></td></tr> </table>	00005	00011	00017	00023		<table border="1"> <tr><td>00006</td></tr> <tr><td>00012</td></tr> <tr><td>00018</td></tr> <tr><td>00024</td></tr> <tr><td></td></tr> </table>	00006	00012	00018	00024	
00001																																			
00007																																			
00013																																			
00019																																			
00002																																			
00008																																			
00014																																			
00020																																			
00003																																			
00009																																			
00015																																			
00021																																			
00004																																			
00010																																			
00016																																			
00022																																			
00005																																			
00011																																			
00017																																			
00023																																			
00006																																			
00012																																			
00018																																			
00024																																			
AMP 1	AMP 2	AMP 3	AMP 4	AMP 5	AMP 6																														

FF07D309



# CHAPTER 10 Secondary Indexes

---

This chapter describes Teradata Database Unique Secondary Indexes (USIs) and Nonunique Secondary Indexes (NUSIs),.

When column demographics suggest their usefulness, the Optimizer selects secondary indexes to provide faster single row or set selection, depending on whether the index is a USI or a NUSI, respectively.

A secondary index is never required for Teradata Database tables, but they can often improve system performance, particularly in decision support environments.

While secondary indexes are useful for optimizing repetitive and standardized queries, Teradata Database is also highly optimized to perform full-table scans in parallel. Because of the strength of full-table scan optimization in Teradata Database, there is little reason to be heavy-handed about assigning multiple secondary indexes to a table.

You can create secondary indexes when you create the table (see “CREATE TABLE” in *SQL Data Definition Language*), or you can add them later using the CREATE INDEX statement (see “CREATE INDEX” in *SQL Data Definition Language*). Unlike primary indexes, secondary indexes can be dropped and created at will without altering the table they index.

You can define a secondary index on a row-level security-protected column.

Note that Teradata does *not* use the term secondary index in the same way it is commonly used by other relational DBMS vendors to describe a non-clustered index in an indexing system based on B+ trees.

Data access using a secondary index varies depending on whether the index is unique or nonunique.

For more information about how Teradata Database accesses data using a unique secondary index, see “[USI Access](#)” on page 458.

For more information about how Teradata Database accesses data using a non-unique secondary index, see “[NUSI Access and Performance](#)” on page 471.

For design issues related to secondary index support for tactical queries, see “[Two-AMP Operations: USI Access and Tactical Queries](#)” on page 898, “[NUSI Access and Tactical Queries](#)” on page 899.

As is true for other types of indexes, you should keep the statistics on your secondary indexes as current as possible.

The recommended practice for recollecting statistics is to set appropriate thresholds for recollection using the THRESHOLD options of the COLLECT STATISTICS statement. See “[COLLECT STATISTICS in SQL Data Definition Language](#)” for details on how to do this.

## Purpose

Secondary indexes enhance selection by specifying access paths other than the primary index path. Secondary indexes are also used to facilitate aggregate operations.

If a secondary index covers a query (see “[Criteria](#)” on page 483 for a definition of covering), then the Optimizer determines that it would be less costly to access its rows directly rather than using it to access the base table rows it points to.

Sometimes multiple secondary indexes with low individual selectivity can be overlapped and bit mapped to provide enhanced retrieval when that would decrease the cost of a query (see “[NUSI Bit Mapping](#)” on page 479).

Teradata Database uses the standard hashing algorithm (see “[Teradata Database Hashing Algorithm](#)” on page 225)) to distribute unique secondary index rows to their subtables.

Nonunique secondary indexes are stored AMP locally to the rows they index and are not hash-distributed. This is also true for geospatial NUSIs.

## Restrictions

The following restrictions apply to secondary indexes.

- Teradata Database tables can have up to 32 secondary, hash, and join indexes.  
This includes the system-defined secondary indexes used to implement PRIMARY KEY and UNIQUE constraints.  
Each multicolumn NUSI that specifies an ORDER BY clause counts as two consecutive indexes in this calculation (see “[Importance of Consecutive Indexes for Value-Ordered NUSIs](#)” on page 485).  
You cannot define secondary indexes for a global temporary trace table. See “[CREATE GLOBAL TEMPORARY TRACE TABLE](#)” in *SQL Data Definition Language*.
  - No more than 64 columns can be specified for a secondary index definition.
  - You can include UDT columns in a secondary index definition.
  - You cannot include columns having XML, BLOB, CLOB, BLOB-based UDT, CLOB-based UDT, XML-based UDT, Period, JSON, ARRAY, or VARRAY data types in any secondary index definition.You can define a simple NUSI on a geospatial column, but you cannot include a column having a geospatial data type in a composite NUSI definition or in a USI definition.
  - You can include row-level security columns in a secondary index definition.
  - You cannot include the system-derived PARTITION or PARTITION#Ln columns in any secondary index definition.

## Space Considerations

Creating a secondary index causes the system to build a subtable to contain its index rows, thus adding another set of rows that requires updating each time a table row is inserted, deleted, or updated.

Secondary index subtables are also duplicated whenever a table is defined with FALBACK, so the maintenance overhead is effectively doubled.

When compression at the data block level is enabled for their primary table, secondary index subtables are *not* compressed. See “[Data Block](#)” on page 248 and “[Compression Types Supported by Teradata Database](#)” on page 695 for more information about data block compression.

## Unique Secondary Indexes

USIs are always preferable to NUSIs for row access using a single value, but might not be as efficient as NUSIs for range query access. The usual criterion for choosing between a USI and a NUSI is the data to which it is applied. Data to be indexed tends to be either inherently unique or inherently *nonunique*.

USIs are useful both for base table access (because USI access is, at worst, a two-AMP operation) and for enforcing data integrity by applying a uniqueness constraint on a column set. Like a unique primary index, a unique secondary index can be used to guarantee row uniqueness.

USIs can be assigned to any column you want to constrain to contain unique values, including row-level security constraint columns.

### Using Unique Secondary Indexes to Enforce Row Uniqueness

When a non-primary index uniqueness constraint is created, whether it is a PRIMARY KEY or UNIQUE constraint, Teradata Database implements it as a USI.

As a general guideline for decision support applications, whenever you define a primary index for a multiset table to be a NUPI, particularly if the table is created in ANSI/ISO session mode (where the default for tables is multiset), you should consider defining one of the following uniqueness constraints on its primary key or other alternate key to facilitate row access and joins.

- Unique secondary index
- UNIQUE NOT NULL constraint
- PRIMARY KEY NOT NULL constraint

PRIMARY KEY and UNIQUE constraints are both mapped internally as USIs unless they are used to define the default UPI for a table. See “[Primary Index Defaults](#)” on page 263.

Of course, you should always consider adding such constraints, including unique join indexes (see “[Functions of Single-Table Join Indexes](#)” on page 546), to your tables when they facilitate

row access or joins. This is particularly true for NoPI tables, because specifying a USI in a request is the only way to access a single row in a NoPI table (see “[NoPI Tables, Column-Partitioned Tables, and Column-Partitioned Join Indexes](#)” on page 280). This is true unless a NUSI happens to be defined that indexes only a single row.

All manner of database constraints are often useful for query optimization, and the richer the constraint set specified for a database, the more opportunities there are to enhance query optimization.

The likely benefits of adding uniqueness constraints are not restricted to multiset NUPI tables. It is also true that if you create a SET NUPI table, the system performs duplicate row checks by default unless you place a uniqueness constraint on the table. Unique constraint enforcement is often a less costly method of enforcing row uniqueness than system duplicate row checks.

Avoid defining a uniqueness constraint on the primary or alternate key of a multiset NUPI table solely to enforce row uniqueness because MultiLoad and FastLoad do not support target tables with non-primary index uniqueness constraints. You can avoid MultiLoad and FastLoad problems associated with indexes by loading data into a table that is otherwise identical with the target table, but has no constraints or non-primary indexes defined on it. After loading the data into that table, you can then INSERT ... SELECT it into the target table that has the desired constraints and indexes defined on it.

Furthermore, USIs impose a performance cost because their index subtables must be maintained by the system as rows are inserted, deleted, and each time the column set in the base table they reference is updated.

## USI Access

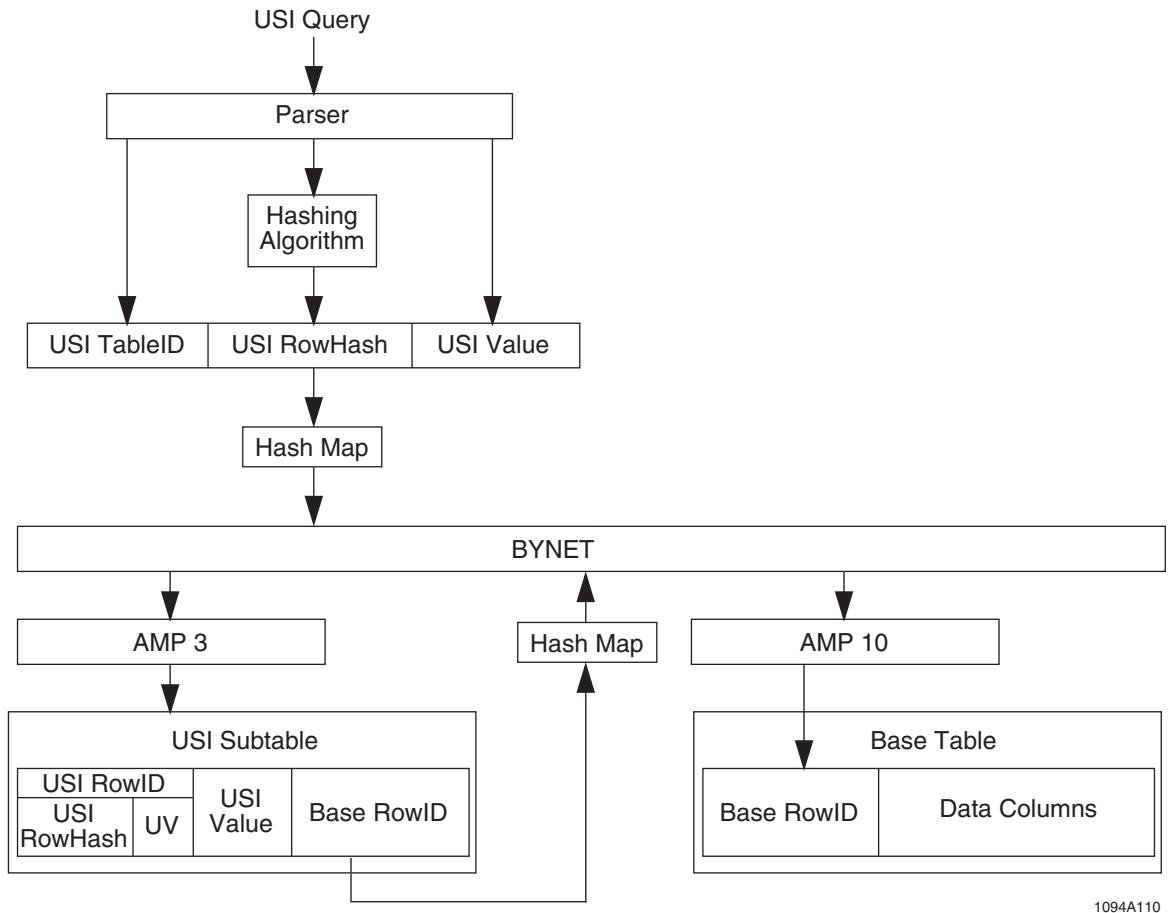
USI access is usually a two-AMP operation because Teradata Database typically distributes a USI row to a different AMP than the base table row the index points to. If the system distributes the USI subtable row to the same AMP as the base table row it points to, then only one AMP is accessed (but it is still a two-step operation).

The following stages are involved in a USI base table row access.

- The requested USI value is accessed by hashing to its subtable.
- The pointer to the base table row is read and used to access the stored row directly.

The flow diagram illustrates the following query. Note that *table\_name* is an NPPI table.

```
SELECT *
FROM table_name
WHERE USI_column = value;
```



1094A110

The process for locating a row using a USI is as follows.

- 1 After checking the syntax and lexicon of the query, the Parser looks up the Table ID for the USI subtable that contains the specified USI value.
- 2 The hashing algorithm hashes the USI value.
- 3 The Generator creates an AMP step message containing the USI Table ID, USI row hash value, and USI data value.
- 4 The Dispatcher uses the USI row hash to send the message across the BYNET to AMP 3, which contains the appropriate USI subtable row.
- 5 The file system on AMP 3 locates the appropriate USI subtable using the USI Table ID.
- 6 The file system on AMP 3 uses the USI rowID to locate the appropriate index row in the subtable.  
This operation might require a search through a number of rows with the same row hash value before the row with the desired value is located.
- 7 AMP 3 reads the base table rowID from the USI row and distributes a message containing the base table ID and the rowID for the requested row across the BYNET to AMP 10, which contains the requested base table row.

The distribution is based on the hash bucket value in the rowID of the base table row.

- 8 The file system uses the rowID to locate the base table row.

## Example

The following example performs a single-row lookup. The column named *cust\_num* is a USI for the *customer* table, which is an NPPI table.

```
SELECT name, phone
FROM customer
WHERE cust = 3;
```

The example *customer* table and USI subtable are as follows.

USI Subtable			Customer			
RowID	CustNum	BaseTable_RowID	RowID	CustNum	CustName	CustPhone
		USI		PK		
DQ + 1	1	H6 + 1	B4 + 1	3	Brown	444-3333
VP + 1	5	J5 + 1	A2 + 1	7	Black	333-4444
R9 + 2	3	B4 + 1	N6 + 1	13	Rice	888-9999
R9 + 1	13	N6 + 1	E3 + 1	2	James	555-4444
3J + 1	2	E3 + 1	E3 + 3	14	Brown	555-4444
22 + 1	14	E3 + 3	L2 + 1	10	Smith	222-9999
3S + 1	9	D7 + 2	J5 + 1	5	Smith	444-6666

The secondary index value (*cust\_num*) is hashed to generate rowID R9 (note that +2 represents the uniqueness number). The AMP retrieves row R9+2 from the secondary index subtable. The subtable row contains the rowID of the base table row, which can then be accessed.

## Unique Secondary Indexes and Performance

USIs provide alternate access paths.

Statistics play an important part in optimizing access when USIs define conditions for the following operations.

- Joining tables
- Satisfying WHERE predicates that specify comparisons, string matching, or complex conditionals
- Satisfying LIKE expressions
- Processing aggregates

Because of the additional overhead for index maintenance, USI values should not change frequently. When you change the value of a secondary index, Teradata Database must undertake the following maintenance operations.

- 1 Delete secondary index references to the current value.
- 2 Generate secondary index references to the new value.

The same considerations apply to NUSIs.

## **Creating a Unique Secondary Index as a Composite of a Row-Level Security Constraint Column and a NUPI Column Set**

You can create a USI for a row-level security-protected table as a composite of a row-level security constraint column and the columns of a NUPI for the table. This property can be used to implement polyinstantiation.

Polyinstantiation is a property that allows a relation to contain multiple rows with the same primary key value, where the multiple instances are distinguished by their security levels, where a security level is defined by a row-level security constraint column.

For this property not to violate the relational model, the security level instances would need to be defined as components of a composite primary key.

## **Restrictions on Load Utilities**

You cannot use FastLoad, MultiLoad, or the Teradata Parallel Transporter operators LOAD and UPDATE to load data into base tables that have unique secondary indexes. If you attempt to load data into base tables with USIs using these utilities, the load operation aborts and returns an error message to the requestor.

Before you can load data into a USI-indexed base table, you must first drop all defined USIs before you can run FastLoad, MultiLoad, or the Teradata Parallel Transporter operators LOAD and UPDATE.

Load utilities like Teradata Parallel Data Pump, BTEQ, and the Teradata Parallel Transporter operators INSERT and STREAM, which perform standard SQL row inserts and updates, *are supported* for USI-indexed tables.

You cannot drop a USI to enable batch data loading by utilities such as MultiLoad and FastLoad as long as requests are running that use that index. Each such query places an ACCESS or READ lock on the index subtable while it is running, so it blocks the completion of any DROP INDEX transactions until the ACCESS or READ lock is removed. Furthermore, as long as a DROP INDEX transaction is waiting to get an EXCLUSIVE lock or is running, requests and batch data loading jobs against the underlying table of the index cannot begin processing.

ALWAYS GENERATED ... NO CYCLE identity columns can be a better choice than USIs for the task of enforcing row uniqueness in multiset NUPI tables because they are supported by both MultiLoad and FastLoad. However, identity columns cannot be used to facilitate row access or joins.

If you define any non-primary index uniqueness constraints on a table, you must drop them all before you use MultiLoad or FastLoad to load rows into that table, then recreate them after the load operation completes (see [“Using Unique Secondary Indexes to Enforce Row Uniqueness” on page 457](#) for a workaround).

If the MultiLoad or FastLoad operation loads any duplicate rows into the table, then you cannot recreate a uniqueness constraint on the table. You must first detect the duplicates and then remove them from the table.

Application-based methods of duplicate row management make several assumptions that are difficult to enforce. These assumptions are.

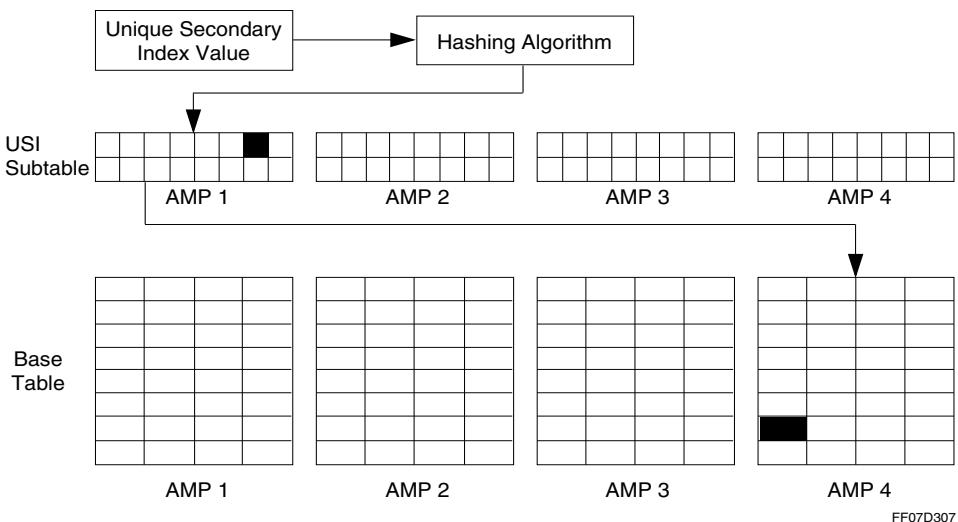
- The application code used to detect and reject duplicate rows is implemented identically across all applications in the enterprise.
- The application code used to detect and reject duplicate rows, even if universally implemented, is correct for all situations.
- All updates to the database are made by means of those applications. This assumption neglects the possibility of ad hoc interactive SQL updates that bypass the application-based duplicate row detection and rejection code.

See [Chapter 12: “Designing for Database Integrity”](#) for more information about the advantages of using system-based declarative constraints to enforce database integrity.

See [Chapter 16: “Design Issues for Tactical Queries”](#) for a description of the special design considerations that must be evaluated for using USIs to support tactical queries.

## USI Hashing

USIs are hash-partitioned on their index columns, as indicated by the following graphic.



## USI Subtable Row Structure

See “[Row Structure for Packed64 Systems](#)” on page 753 and “[Row Structure for Aligned Row Format Systems](#)” on page 755 for additional information about the row structures of secondary indexes.

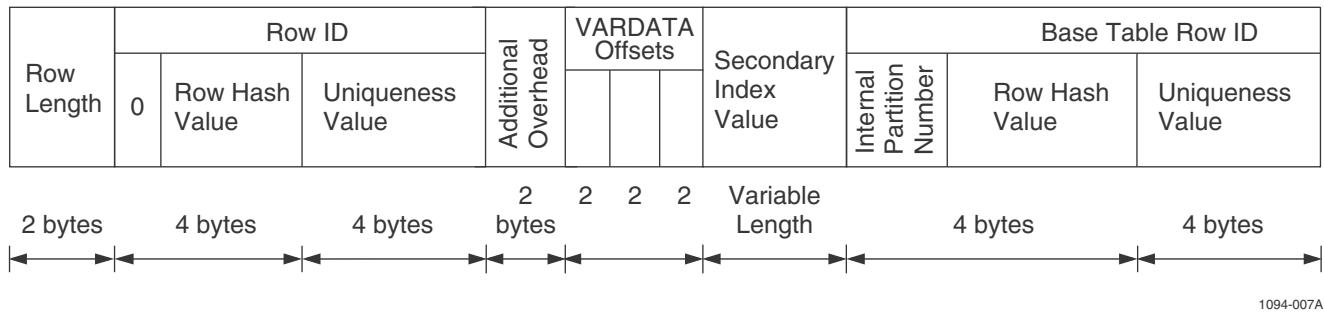
The internal partition number for a secondary index row is always 0, which is indicated by 2 flag bit settings.

The table header for the base table indicates whether it is partitioned or not and, if it is partitioned, whether it has 2-byte or 8-byte partitioning.

Teradata Database creates additional USI index subtable rows as they are needed.

### Packed64 Format USI Subtable Row Structure for a Nonpartitioned Base Table

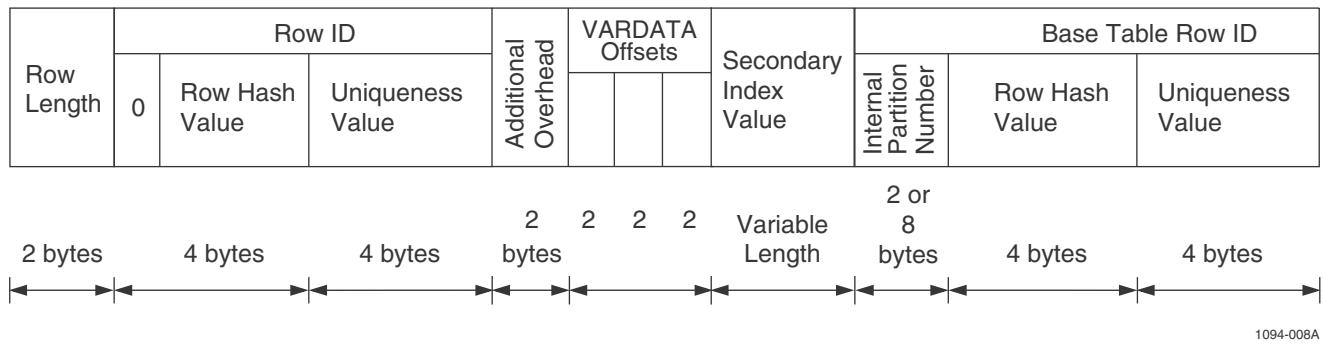
The USI subtable row structure for an index on an nonpartitioned base table is described by the following graphic.



The internal partition number for the base table row is implicitly 0 and is not stored.

### Packed64 Format USI Subtable Row Structure for a Partitioned Base Table

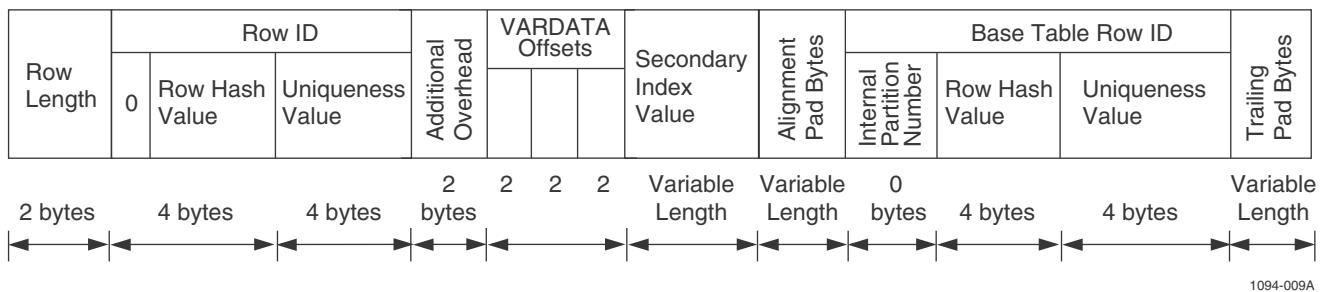
The USI subtable row structure for an index on a partitioned base table is described by the following graphic.



Because the base table is a partitioned table, the Internal Partition Number field stores the internal partition number for the corresponding base table row in a physical 2-byte or 8-byte field.

### Aligned Row Format USI Subtable Row Structure for an Nonpartitioned Base Table

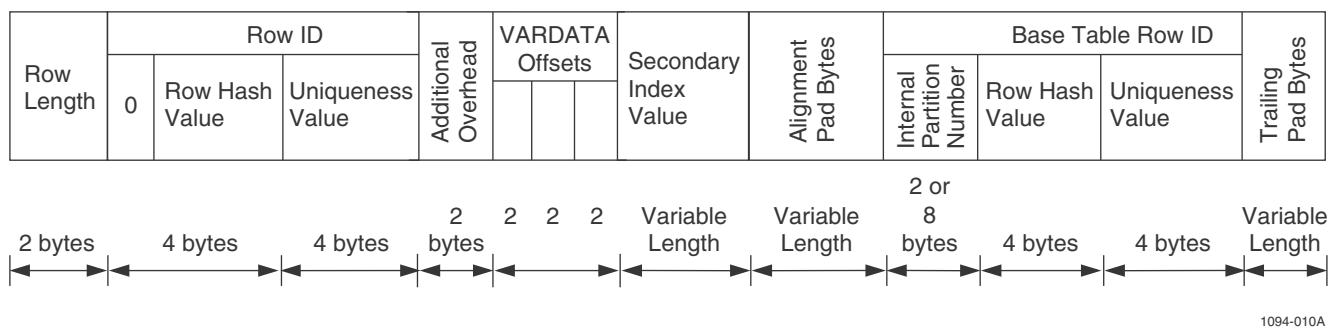
The USI subtable row structure for an index on an nonpartitioned base table is described by the following graphic.



Because the internal partition number for nonpartitioned base table rows is always 0, the Internal Partition Number field pictured here is logical only, which is why it does not consume 2 bytes or 8-bytes of overhead for the row like USI rows do for partitioned base tables.

### Aligned Row Format USI Subtable Row Structure for a Partitioned Base Table

The USI subtable row structure for an index on a partitioned base table is described by the following graphic.



Because the base table is a partitioned table, the Partition Number field stores the internal partition number for the corresponding base table row in a physical 2-byte or 8-byte field at the beginning of the base table RowID field.

### USI Row Structure Field Definitions

Stored Data	Length (bytes)	Function
Row length	2	Defines the number of bytes in the row.  If the row is aligned, this field includes any required pad bytes necessary to make the row length a multiple of 8 bytes. If the row is packed or packed 64, the row length does not include any extra pad bytes. Note that in this case, when space is allocated for the row, value is rounded up to a multiple of 2 bytes.

Stored Data	Length (bytes)	Function
RowID	8	Defines the USI row uniquely for its subtable by combining its row hash value with a uniqueness value.
Row hash	4	Defines the output of the hashing algorithm, which is a unique (or nearly unique) value based on a mathematical transformation of the unique secondary index value.
Uniqueness value	4	Defines a system-generated integer that ensures that the rowID is unique within a table.
Overhead (flag byte and 1st presence byte)	2	2 single-bit flag fields (set to zero) indicate the internal partition number is 0 for the USI row.
Secondary index value	up to 65,524	Defines the column values for the unique secondary index.
Alignment pad bytes	between 0 and 7 bytes	Ensures that the Base Table Row ID field begins on a modulo(8) boundary. If the Base Table Row IDs field naturally aligns on a modulo 8 boundary, there is no field of alignment pad bytes at the end of the Secondary Index Value field.
Base table rowID	<ul style="list-style-type: none"> <li>• 8 bytes for a nonpartitioned base table row.</li> <li>• 10-16 bytes for a partitioned base table row.</li> </ul>	<p>Defines the rowID of the base table row this secondary index row identifies. This row is usually on a different AMP from its unique secondary index rowID.</p> <p>Because the rowID of a partitioned table row also contains the internal partition number for the row, it is 2 or 8 bytes longer than the rowID of a nonpartitioned table row.</p>
Internal partition number	<ul style="list-style-type: none"> <li>• 0 for a nonpartitioned base table row.</li> <li>• 2 for a partitioned base table row with 2-byte partitioning.</li> <li>• 8 for a partitioned base table row with 8-byte partitioning.</li> </ul>	<p>Defines the partition number for a partitioned base table row. If the base table row is from a nonpartitioned base table, the internal partition number is implicitly 0 and is not stored.</p> <p>The table header for the base table indicates whether it is partitioned or not and, if it is partitioned, whether it has 2-byte or 8-byte partitioning.</p>
Row hash	4	Defines the output of the hashing algorithm on the primary index value.
Uniqueness value	4	Defines a system-generated integer that ensures that the rowID is unique within a table.
Trailing pad bytes	0-7 pad bytes	The trailing bytes are used to round up the row length of an aligned row to an 8 byte multiple. The trailing pad bytes are included in the row-length field (the 1st two bytes of the row). If the entire row naturally aligns on a modulo(8) boundary, there is no field of trailing alignment pad bytes

For a NoPI or column-partitioned table, the 8 bytes of row hash and uniqueness value are treated as a hash bucket, which has either 16 or 20 bits, and as a 44-bit uniqueness value. If the hash bucket is a 16-bit bucket, the 4 bits between the 16-bit hash bucket and the 44-bit uniqueness value are not used.

## Using Unique Secondary Index Maintenance and Rollback to Optimize Query Design

Teradata Database processes USI maintenance operations (INSERT ... SELECT, full-file DELETE, join DELETE, and UPDATE) block-at-a-time rather than row-at-a-time, whenever possible.

When the original index maintenance is processed block-at-a-time, the USI change rows are transient journaled block-at-a-time. As a result, the rollback of the USI change rows are block-at-a-time, that is, block optimized.

USI change rows are redistributed to the owner AMP, sorted, and applied block-at-a-time to the USI subtable, such that the index data blocks are updated once rather than multiple times.

## Nonunique Secondary Indexes

Nonunique secondary indexes are typically assigned to nonunique column sets that frequently appear in WHERE clause selection conditions, join conditions, ORDER BY and GROUP BY clauses, foreign keys, and miscellaneous other conditions such as UNION, DISTINCT, and any attribute that is frequently sorted.

**Note:** You can define a NUSI on a row-level security constraint column.

You can also define a simple NUSI, but not a composite NUSI, on a geospatial column.

Highly selective NUSIs are useful for reducing the cost of frequently made selections and joins on nonunique columns, and provide extremely fast access for equality conditions. This is particularly true for NoPI tables (see “[NoPI Tables, Column-Partitioned Tables, and Column-Partitioned Join Indexes](#)” on page 280), where the only other access method might be a full-table scan. Note that NUSIs with low selectivity can be less efficient than a full-table scan.

NUSIs are also useful for range access and in-list conditions and for geospatial indexes.

Also note the following about NUSIs:

- NUSI access is always an all-AMPs operation unless the index is defined on the same columns as the primary index. This is allowed when the NUSI is value-ordered or when the table or join index is partitioned and not all the partitioning columns are included in the primary index.
- The subtables must be scanned in order to locate the relevant pointers to base table rows. This is a fast lookup process when a NUSI is specified in an equality or range condition because the NUSI rows are either hash-ordered or value-ordered on each AMP.
- NUSI subtables are not covered by the active read fallback feature (see “[Physical Database Integrity](#)” on page 664 for details).

## Relationship Between a NUSI Subtable Row and Base Table Rows

A particular NUSI subtable row points to one or many base table rows on that same AMP. The relationship between a NUSI value and any individual AMP in a configuration is either 0:1, 1:1, or 1:M.

This relationship ...	Reflects the fact that ...
0:1	an AMP contains no NUSI subtable index rows for a particular NUSI value.
1:1	an AMPs contains 1 NUSI subtable index row for a particular NUSI value.
1:M	an AMP contains more than 1 NUSI subtable index row for a particular NUSI value.

## NUSI Row Structure

The subtable structure of a non-geospatial NUSI row is described by the following graphics. See “[Row Structure for Packed64 Systems](#)” on page 753 and “[Row Structure for Aligned Row Format Systems](#)” on page 755 for additional information.

The number of base table rows that can be referenced in a NUSI row is limited by the maximum row size for the system and the length of the secondary index value. See “[NUSI Sizing Equation](#)” on page 864 for more information about sizing NUSI subtables.

Teradata Database creates additional NUSI index subtable rows as they are needed.

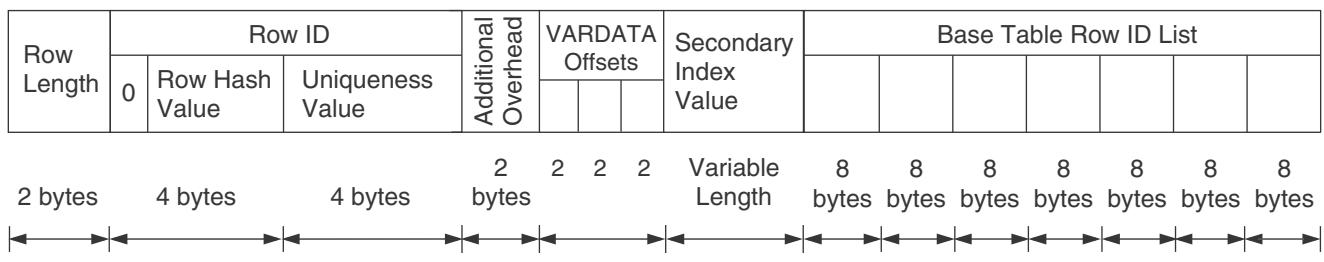
For geospatial indexes, the Secondary Index Value field contains the Minimum Bounding Rectangle (MBR) for the object being indexed. Unlike other Teradata Database indexes, geospatial NUSIs are maintained in a Hilbert R-tree structure that sits on top of the Teradata file system rather than as rows in a relational index subtable.

Scalar NUSIs contain index key values that are equal to their base table counterparts, while geospatial NUSIs contain index keys that are only an approximation of their base table counterparts (other than case of characters if the ALL option is not specified). Leaf rows in the Teradata implementation for geospatial R-trees have the form MBR (index key), base table row ID list.

For more information about Hilbert R-trees and geospatial NUSIs, see *SQL Request and Transaction Processing*.

### Packed64 Format NUSI Subtable Row Structure for a Nonpartitioned Base Table

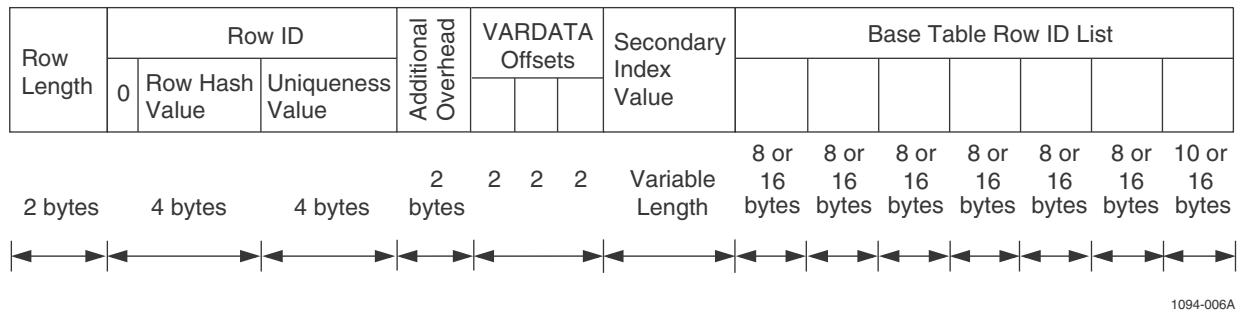
The NUSI subtable row structure for an index on an nonpartitioned base table is described by the following graphic.



1094-004A

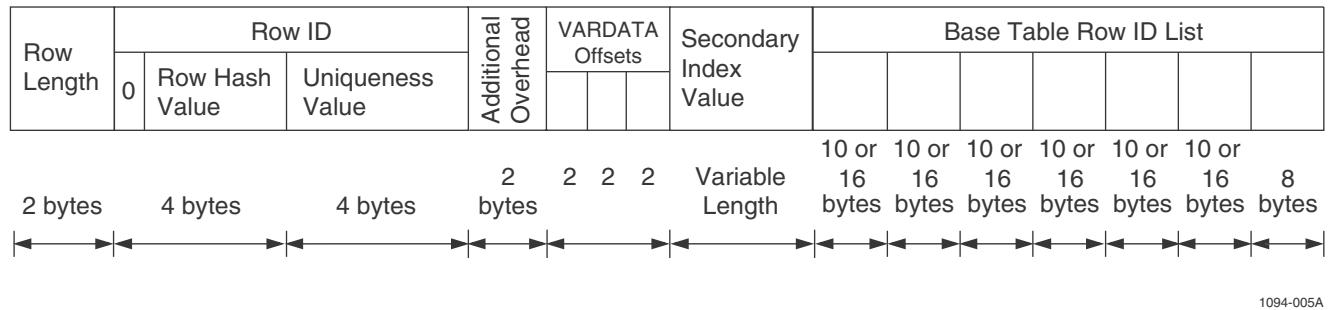
### Packed64 Format NUSI Subtable Row Structure for a Partitioned Base Table

The NUSI subtable row structure for an index on a partitioned base table is described by the following graphic.



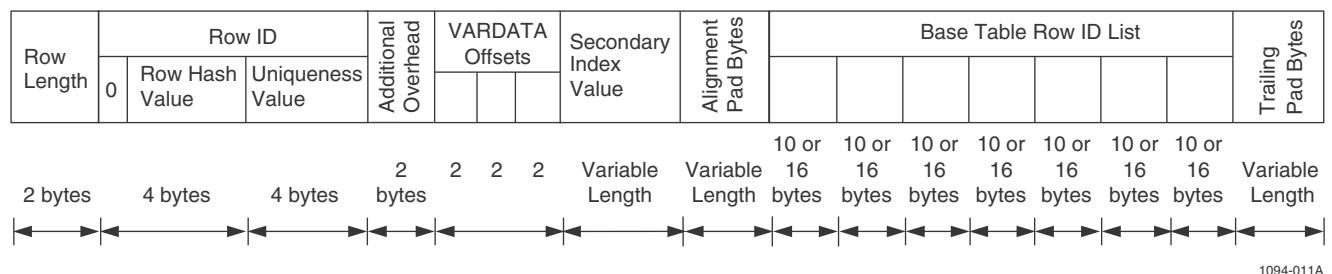
### Aligned Row Format NUSI Subtable Row Structure for a Nonpartitioned Base Table

The NUSI subtable row structure for an index on an nonpartitioned base table is described by the following graphic.



### Aligned Row Format NUSI Subtable Row Structure for a Partitioned Base Table

The NUSI subtable row structure for an index on a partitioned base table is described by the following graphic.



Each rowID in the base table rowID list consumes an additional 2 or 8 bytes over rowIDs in NUSI subtable rows for a nonpartitioned base table or join index. This is because either 2 or 8 additional bytes are required to store the internal partition number of the base table rowID.

### Definitions for NUSI Row Structure Fields

Stored Data	Length (bytes)	Function
Row length	2	Defines the number of bytes in the row.  If the row is aligned, this field includes any required pad bytes necessary to make the row length a multiple of 8 bytes. If the row is packed or packed 64, the row length does not include any extra pad bytes. Note that in this case, when space is allocated for the row, value is rounded up to a multiple of 2 bytes.
RowID	8	Defines the NUSI row uniquely for its subtable by combining its row hash value with a uniqueness value.
Row hash	4	Defines the output of the hashing algorithm, which is a unique (or nearly unique) value based on a mathematical transformation of the non-unique secondary index value.
Uniqueness value	4	Defines a unique system-generated integer that ensures that the rowID is unique within a table.
Overhead	2	2 single-bit flag fields indicate whether the internal partition number is 0.
Secondary index value	up to (65,524) - (table rowID list bytes)	Defines the column values for the nonunique secondary index.
Alignment pad bytes	between 0 and 7 bytes	Ensures that the Base Table Row ID field begins on a modulo(8) boundary. If the Base Table Row IDs field naturally aligns on a modulo 8 boundary, there is no field of alignment pad bytes at the end of the Secondary Index Value field.
Base table rowID list	<ul style="list-style-type: none"> <li>• 8 per nonpartitioned base table row.</li> <li>• 10 per 2-byte partitioned base table row.</li> <li>• 16 per 8-byte partitioned base table row.</li> </ul>	Defines the row IDs for the rows on the same AMP to which this non-unique index points.
Trailing pad bytes	0-7 pad bytes	The trailing bytes are used to round up the row length of an aligned row to an 8 byte multiple. The trailing pad bytes are included in the row-length field (the 1st two bytes of the row). If the entire row naturally aligns on a modulo(8) boundary, there is no field of trailing alignment pad bytes.

When the system adds new base table Row IDs to a NUSI Row ID list, it continues to write from the end of the row. This means that it overwrites the previously existing trailing

alignment pad bytes with the newly added entries to the list of Row IDs. Note that because the two alignment pads serve independent requirements, there might be pad bytes at the beginning of the row ID list as well as at the end of row, but there are never any pad bytes in the middle of the Row ID list.

## NUSI Access and Performance

NUSI access specifies a three-part BYNET message that is identical to the three-part message used for primary index access except that the subtable ID in the message references the NUSI subtable rather than the base table.

NUSI requests are all-AMP requests unless the NUSI is defined on the same columns as primary index.

The usefulness of a NUSI is correlated with the number of rows per value: the higher number of rows per value, the less useful the index. If the number of rows for a NUSI value exceeds the number of data blocks in the table, the usefulness of the NUSI might be questionable. On the other hand, as NUSI values approach uniqueness (meaning that the number of rows per value is either close to 1 or is significantly less than the number of AMPs in the system), an all-AMPs table access is wasteful and you should consider defining a join index (see [Chapter 11: “Join and Hash Indexes”](#)) to support DML requests against the table instead of a NUSI.

Because NUSI access is usually an all-AMPs operation, NUSIs may seem to have limited value. If you have to access all AMPs in the configuration to locate the requested rows, why bother with an index?

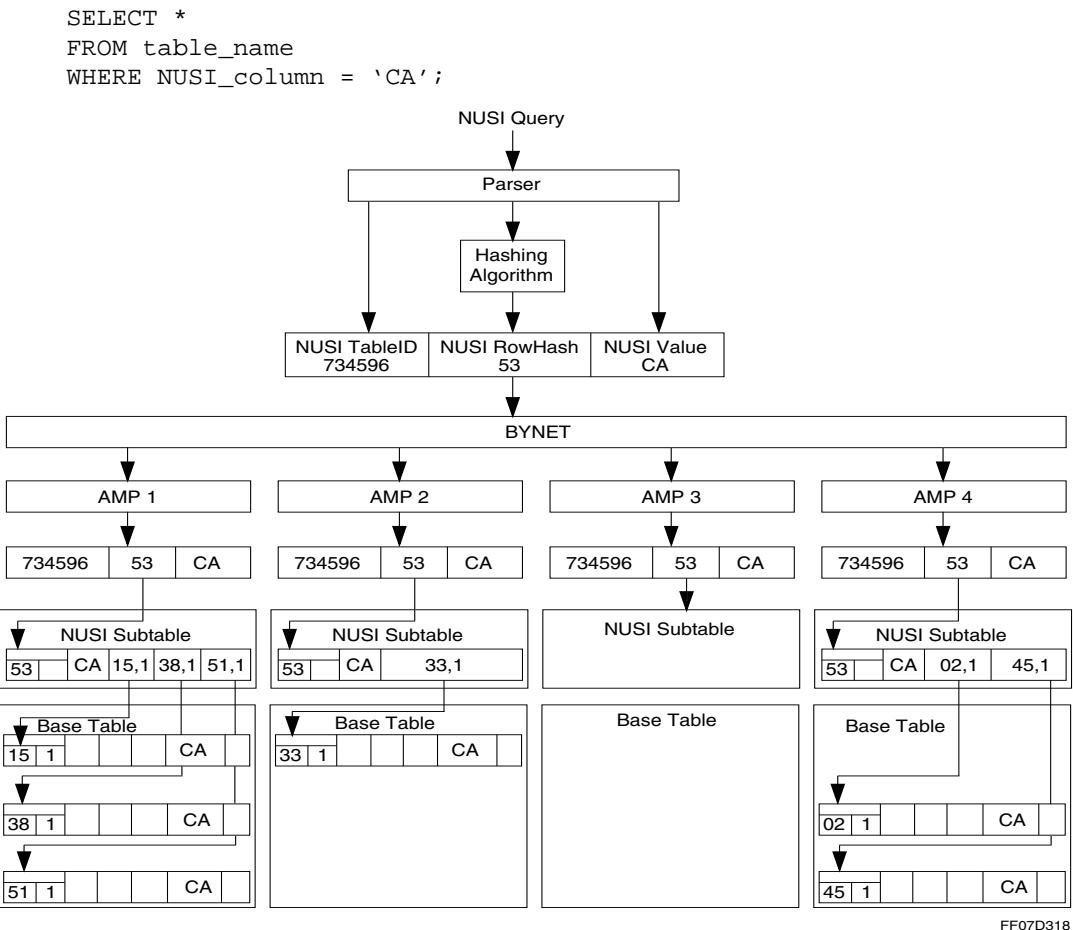
The answer to that question is provided by the following list of ways that NUSIs can improve the performance of your decision support queries:

- NUSI access is often faster than a full-table scan, particularly for extremely large tables. A full-table scan is also an all-AMP operation.
- A NUSI that covers (see [“NUSIs and Query Covering” on page 482](#) for a definition of covering) the columns requested by a query is often included in Optimizer access plans.
- A NUSI that covers a LIKE expression or any selective inequality conditions is often included in Optimizer access plans.
- A NUSI on the same columns as the primary index (this is only allowed when the primary index does not include all the columns of all the partitioning columns) may be more efficient than accessing using the primary index when there is no or limited partition elimination for a query.

While NUSI access is usually an all-AMPs operation, keep in mind that the AMPs work in parallel. If all the AMPs have qualified rows, then this is a very efficient operation. If some or many of the AMPs do not have qualified rows, then those AMPs are doing work just to determine that they have no qualified rows. Note that if there are more rows per NUSI value than AMPs, it is likely that every AMP will have one or more qualified rows.

At the same time, as the number of rows pointed to per NUSI value increases, the efficiency of a NUSI read decreases proportionately. Depending on demographics and environmental cost variables, the Optimizer will specify a full-table scan instead of a NUSI access when it determines that the scan would be a more efficient access method.

The flow diagram illustrates the following query.



The process used by this example for locating a row using the NUSI value CA is as follows.

- 1 After checking the syntax and lexicon of the query, the Parser looks up the TableID for the NUSI subtable that contains the NUSI value CA.
- 2 The hashing algorithm hashes the NUSI value.
- 3 The Generator creates an AMP steps message containing the NUSI TableID (734596), NUSI row hash value (53), and NUSI data value (CA) and then the Dispatcher distributes it across the BYNET to all AMPs.
- 4 The file system on a receiving AMP locates the appropriate NUSI subtable using its TableID.
- 5 The file system on a receiving AMP uses the NUSI row hash value to locate the appropriate index row in the subtable.
- 6 If there is a NUSI row, its table rowID list is scanned for base table row IDs.
- 7 The file system uses the row IDs to locate the base table rows containing the NUSI value CA.

For more information about secondary indexes and performance, see “[Unique Secondary Indexes and Performance](#)” on page 460.

## Using NUSIs

When using NUSIs there should be fewer rows that satisfy the NUSI qualification condition than there are data blocks in the table. Whether the Optimizer uses a NUSI depends on the percent of rows that satisfy the NUSI qualification condition and the overhead of reading the NUSI table, as follows.

Qualifying Rows	Result
< 1 per block	<p>NUSI access is generally faster than full-table scan. As the number of qualified rows approaches the number of data blocks, the additional cost of reading the NUSI subtable can make NUSI access more costly.</p> <p>For example, if there are 100 rows per block and 1 in 1000 rows qualify, the Optimizer reads 1 in every 10 blocks. NUSI access is faster.</p>
$\geq 1$ per block	<p>A full-table scan is faster than NUSI access.</p> <p>For example, if there are 100 rows per block and 1% of the data qualifies, the Optimizer reads almost every block. A full-table scan might be faster.</p>

In some cases, the values of a NUSI are distributed unevenly throughout a table. At other times, some values might represent a large percent of the table, while other values have few instances. When values are distributed unevenly and statistics are available on the NUSI that allow the Optimizer to see the values distribution, Teradata Database can use a NUSI as follows.

- Perform a full-table scan for queries on values that represent a large percentage of table.
- Use a NUSI for queries on the remaining values.

You can use a request like the following to report secondary index non-uniqueness.

```
.SET RETLIMIT 20

SELECT index_column_set, COUNT(*)
FROM table_name
GROUP BY 1
ORDER BY 2 desc;
```

## NUSIs and Block Size

Effective use of a NUSI requires that fewer rows qualify than there are data blocks in the base table.

Consider, for example, 100-byte rows. With a average block size of 31.5 KB, each multirow data block contains approximately 315 rows. This means fewer than one in 315 rows (that is, less than 0.32%) can qualify for a given NUSI value if the index is to be effective.

When the average block size is 63.5 KB, fewer than one in 635 rows (that is, less than 0.16%) can qualify for the NUSI to be effective.

When the average block size is 127.5 KB, fewer than one in 1275 rows (that is, less than 0.08%) can qualify for the NUSI to be effective. When the average block size is 1KB, fewer than one in

10 rows (that is, less than 10%) can qualify for the NUSI to be effective. This becomes even more significant if your system uses 1MB data blocks.

To reset the global absolute-maximum size for data blocks, see “PermDBSize” in *Utilities: Volume 1 (A-K)*.

## Selectivity Considerations

Selectivity refers to the percentage of rows in a table containing the same non-unique secondary index value. An index that has high selectivity retrieves few rows. A unique primary index retrieval, for example, is highly selective because it never returns more than one row. An index that has low selectivity retrieves many rows.

### Low Selectivity Indexes

When an index is said to have low selectivity, that means that many rows have the same NUSI value and there are relatively few distinct values.

A column with those characteristics is usually a poor choice for a NUSI because the cost of using it may be as high or higher than a full-table scan.

For example, assume that *employee* table contains 10,000 rows of about 100 bytes each, and there are only 10 different departments. If an average *employee* data block is 2,560 bytes and can store about 25 rows, then the entire table requires about 400 data blocks.

If *dept\_no* is defined as a non-unique secondary index on the *employee* table, and the *dept\_no* values are evenly distributed, then the following query accesses about 1,000 row selections.

```
SELECT *
  FROM employee
 WHERE dept_no = 300;
```

Each AMP reads its own rows of the *dept\_no* secondary index subtable. If any rows contain the index value 300, the AMP uses the associated rowIDs to select the data rows from its portion of the *employee* table.

Regardless of the number of AMPS involved, this retrieval requires 1,000 row selections from the *employee* table. To satisfy this number of select operations, it is likely that all 400 *employee* data blocks would have to be read.

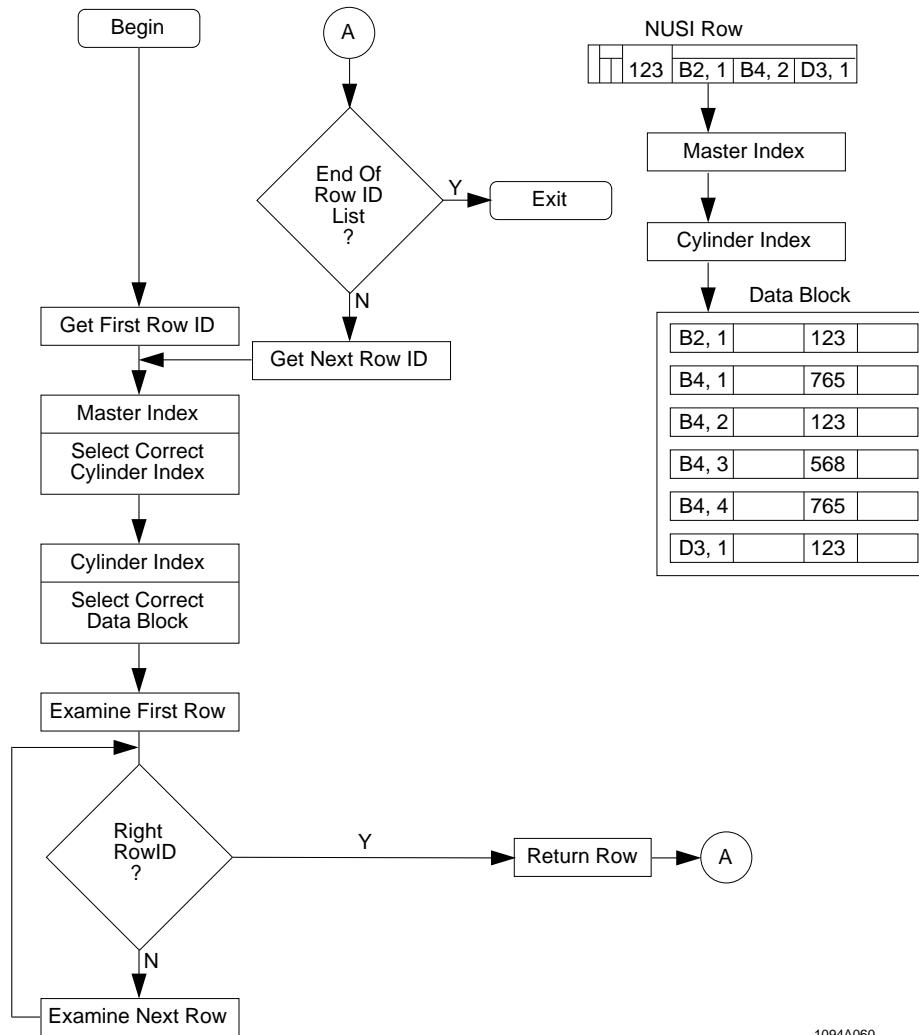
If that were the case, then the number of I/O operations undertaken by the retrieval could easily exceed the number required for a full-table scan. In such instances, a table scan would actually be a much more efficient solution than a NUSI-based retrieval.

### High Selectivity Index

If *deptno* is a high selectivity index, where few *employee* rows share the same *deptno* value, then using *deptno* for retrieval provides better performance than a full-table scan. Because of these selective conditions, the Optimizer specifies NUSI access for request processing when it is less costly than a full-table scan.

## Generic NUSI Read Process Flow

The following flowchart illustrates the logic of a NUSI read. The graphic also illustrates a somewhat more concrete NUSI read example where the rowID value B2,1 read (where B2 is the row hash value and 1 is the uniqueness value for the row and assuming that the table being accessed is an nonpartitioned base table) is followed in turn through the master index, the cylinder index, and the data block to access the base table row set matching that value.



1094A060

The following process describes the process flow for locating a row using a NUSI.

- 1 Read the first rowID from the subtable row.  
In the graphic example, the value read is B2, 1.
- 2 Scan the Master Index for the required Cylinder Index.
- 3 Scan the Cylinder Index for the required data block.
- 4 Scan the rows in the data block until the row having the requested rowID is located.
- 5 Return the retrieved row set to the requestor.

# Multiple NUSI Access

Database designers frequently define multiple NUSIs on a table. Whether the Optimizer chooses to include one, all, or none of them in its query plan depends entirely on their individual and composite selectivity.

## Multiple NUSI Access and Composite NUSI Access

Multiple secondary indexes are a good choice when NUSIs are not highly selective by themselves, but in combination they are highly selective: the number of rows returned is a small fraction of the total number of rows in the table.

For example, consider the following query.

```
SELECT last_name, first_name, salary_amount
FROM employee
WHERE department_number = 500
AND job_code = 2147;
```

The conditions have NUSIs defined on them.

```
CREATE INDEX (department_number) ON EMPLOYEE;
CREATE INDEX (job_code) ON EMPLOYEE;
```

If both indexes have low selectivity, then the Optimizer uses bit mapping (see “[NUSI Bit Mapping](#)” on page 479 for details).

IF ...	THEN the Optimizer ...
both columns are indexed	most often specifies a hash lookup in their NUSI subtables, calling for a non-duplicate spool file of the rowIDs, and then uses the result to access the base table.
one column is indexed	always specifies a full-table scan.

A composite set selection is one for which multiple logically related conditions apply.

An ORed composite set selection is one for which *any* predicate condition in the WHERE clause can evaluate to TRUE in order to retrieve rows that match the condition specified by the predicate.

An ANDed composite set selection is one for which *all* predicate conditions in the WHERE clause must evaluate to TRUE. If any condition evaluates to FALSE, then no row is retrieved for that composite set of conditions.

A composite condition set can reference any number of indexes, including none. The Optimizer selects an index for the query plan only when values for all of its components are supplied in the condition.

When a condition references an indexed column set, then the Optimizer selects that index for the query plan and uses it to access the table. Any remaining, or residual, conditions are applied to the intermediate result set to accomplish additional selectivity.

The following table explains how the Optimizer uses secondary indexes in the same scenario.

The Optimizer uses ...	WHEN ...
a single secondary index	<p>the cost of accessing the secondary index for the qualified rowids plus the cost of reading the data blocks in the table that contain the qualified rowids is lower than the cost of scanning the table to find the qualified rows. Typically this is the case when the index selectivity is high.</p> <p>The number of data blocks that need to be accessed depends on row size and data block size. Optimizer estimates the number of data blocks to read based on the number of qualified rowids computed from the index selectivity, the table row size and data block size.</p>
multiple secondary indexes	their combined selectivity is high.

## Example NUSI Definitions

Consider the following two NUSIs created on an *employee* table.

```
CREATE INDEX (department_number) ON employee;
CREATE INDEX (job_code) ON employee;
```

The examples “[AND Equality Condition](#)” on page 477 and “[OR Equality Condition](#)” on page 478 examine how these dual NUSIs on the *employee* table might be used by the Optimizer when they are used as ANDed and ORed conditions in a WHERE clause.

## AND Equality Condition

Consider the following simple query with an ANDed predicate set.

```
SELECT last_name, first_name, salary_amount
FROM employee
WHERE department_number = 500
AND job_code = 2147;
```

Whether the Optimizer includes these NUSIs in its query plan varies depending on their relative selectivity.

IF ...	THEN the Optimizer ...
only one of the NUSIs is strongly selective	uses that NUSI alone in its query plan.
both NUSIs are weakly selective individually, but strongly selective when combined	creates a bit-mapped intersection of their common rowIDs and uses them in its query plan (see “ <a href="#">NUSI Bit Mapping</a> ” on page 479).
both NUSIs are weakly selective both when examined individually and when combined	selects neither for its query plan and instead specifies a full-table scan when no other index option provides a less costly estimate.

## OR Equality Condition

Consider the following simple query with an ORed predicate set.

```
SELECT last_name, first_name, salary_amount
FROM employee
WHERE department_number = 500
OR job_code = 2147;
```

If both columns are indexed, as they are in this particular case, then a possible Optimizer plan would include the following stages.

- 1 Use their hash codes to find the indexed rows in both NUSI tables.
- 2 Retrieve the rowIDs of any qualifying base table rows and place them into single spool file.
- 3 Sort the rowIDs to eliminate duplicates.
- 4 Use the resulting spool to access the qualified base table rows.

This processing could be done on one NUSI if the set of ORed predicates is specified on the same indexed column, or extended to more than two NUSIs, of course, but the likelihood of that being a profitable exercise for the Optimizer to undertake vanishes as the number of rows that qualify the ORed predicate set increases. A full-table scan could take less time to perform than looking up the NUSI or multiple NUSIs for each value in the ORed predicate set, spool file production, and duplicate elimination required by this method.

## Queries Using BETWEEN, LESS THAN, GREATER THAN, or LIKE Operators

Assume that a query involves BETWEEN, LESS THAN, GREATER THAN, or LIKE operations on a single NUSI as in the following examples.

```
CREATE INDEX (hire_date) ON employee;

SELECT last_name, first_name, hire_date
FROM employee
WHERE hire_date BETWEEN 880101 AND 881231;

SELECT last_name, first_name, hire_date
FROM employee
WHERE hire_date < 880101;

SELECT last_name, first_name, hire_date
FROM employee
WHERE hire_date > 881231;
```

The following process outlines the approach taken to respond to these requests. This technique can be very efficient for a selective ANDed predicate when there is a composite NUSI that contains all the predicate columns. Optimizer chooses to use the NUSI if the savings in the number of data blocks to read is greater than the overhead of scanning the NUSI and writing and reading a rowID spool.

- 1 The Optimizer determines whether it is more efficient to do a full-table scan of the base table rows or to scan the secondary index subtable to get the rowIDs of the qualifying base table rows. Note that ordering the NUSI values can reduce the index scan time.

For purposes of this example, assume that the NUSI access method is the more efficient technique to pursue.

- 2 The access plan specifies to place those rowIDs into a spool file.
- 3 The access plan specifies to use the resulting rowIDs to access the base table rows.

## NUSI Bit Mapping

Bit mapping is a technique used by the Optimizer to effectively link several weakly selective indexes in a way that creates a result that drastically reduces the number of base rows that must be accessed to retrieve the desired data. The process determines common rowIDs among multiple NUSI values by means of the logical intersection operation. The method is explained in more detail in “[Computing NUSI Bit Maps](#)” on page 481.

### Performance Advantages of NUSI Bit Mapping

Bit mapping is significantly faster than the three-part process of copying, sorting, and comparing rowID lists. Additionally, the technique dramatically reduces the number of base table I/Os required to retrieve the requested rows.

### Restrictions for When Bit Mapping Is Used

Teradata Database only performs NUSI bit mapping when weakly selective indexed conditions are ANDed. When WHERE conditions are connected by a logical OR, the Optimizer typically performs a full-table scan and applies the ORed conditions to every row without considering an index.

Note that this is not always the case. If the weakly selective ORed conditions are ANDed with other weakly selective conditions, but the composite selectivity is high, the Optimizer may choose to use a composite NUSI or a single-column NUSI.

IF all conditions are ANDed together and ...	AND their composite selectivity is ...	THEN the Optimizer...
one index is strongly selective		selects it alone and applies the remaining constraints to the selected rows as residual constraints.
all of the indexes are weakly selective	also weakly selective	performs a full-table scan and applies the conditions to every row.
all of the indexes are weakly selective	strongly selective	can instruct each AMP to construct bit maps to determine which rowIDs their local NUSI rows have in common and then access just those rows, applying the conditions to them exclusively.

For example, consider the following SELECT statement with three WHERE conditions, each of which has a weakly selective NUSI defined on its column.

```
SELECT *
FROM employee_table
WHERE salary_amount > 20000
```

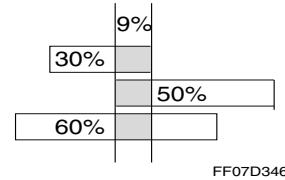
```
AND    sex_code = 'M'
AND    full_time = 'FT';
```

Suppose that the index on *salary\_amount* has 30% selectivity, the index on *sex\_code* 50% selectivity, and the index on *full\_time* 70% selectivity. The individual selectivity of these indexes is very weak.

If the overlap among their selectivities is such that the effect of combining their relative weaknesses results in a significant strength, then the Optimizer instructs the AMPs to use bit mapping to do just that, and the end result is vastly improved performance of the retrieval in comparison with a full-table scan.

In this example, the overlap selectivity is 9%, which is both significantly better than any of the individual selectivities of the NUSIs in the query and typically better than a full-table scan. This assumes fewer than 10 rows per data block. The figure must be adjusted for larger data blocks that can contain many more rows.

```
SELECT *
  FROM employee_table
 WHERE salary_amount > 20000
   AND sex_code = 'M'
   AND full_time = 'FT';
```



## Using NUSIs For Complex Conditional Expressions

Complex conditional expressions can be based on non-unique secondary indexes. Bit mapping is often used to solve such expressions when they are applied to a very large table. NUSI bit mapping can be used if the following statements are both true for the conditions under consideration.

- There are at least two NUSI equality conditions.
- All the NUSI conditions are ANDed.

Take the following request as an example.

```
SELECT COUNT(*)
  FROM LargeTable
 WHERE NUSI_1 = 'condition_1'
   AND NUSI_2 = 'condition_2'
   AND NUSI_3 = 'condition_3'
   AND NUSI_4 = 'condition_4';
```

To resolve this request, a bit map is built for  $n-1$  referenced indexes (see stage 2 in the process documented on the following page). For this example,  $n=4$ , so the system builds three bit maps. In each bit map, every qualifying data row bit is turned ON.

The bit maps are ANDed to form one large bit map, which is held in a spool file. The ON bits are used to access the result data rows directly.

The most selective index of the four accesses rows based on whether the bit is ON for the rowID in the ANDed bit map for the rowID from the previous NUSI.

## Computing NUSI Bit Maps

We use the following previously described query to example how NUSI bit maps are computed.

```
SELECT *
FROM employee_table
WHERE salary_amount > 20000
AND sex_code = 'M'
AND full_time = 'FT';
```

Recall the truth table for the AND operator.

0	+	0	=	0
0	+	1	=	0
1	+	0	=	0
1	+	1	=	1

The following process illustrates the method for computing NUSI bit maps. The actions described occur concurrently on all AMPs.

- 1 The literals 20,000, M, and FT are used to access the corresponding rows from the index subtables.
- 2 For  $n-1$  indexes, each AMP creates separate bit maps of all 0 values.  
In this case,  $n-1$  is 2 because there are three NUPIs defined, one for each condition in the WHERE clause.
- 3 Using a proprietary algorithm, the AMPs equate each base table rowID in each qualifying NUPI row with a single bit in the map, and each such map bit is turned ON.
- 4 The resulting bit maps are ANDed together to form the final bit map.
- 5 Each base table rowID in the remaining NUSI is equated to a single bit in the final bit map, and the state of that map is checked.

IF the map bit is ...	THEN ...
OFF (0)	no action is taken and the next map bit is checked.
ON (1)	the rowID is used to access the table row and all remaining, or residual, conditions are applied to that row.

## Determining If Bit Mapping Is Being Used

To determine whether bit mapping is being used for your NUSIs, use the EXPLAIN request modifier and examine the reports it produces for your queries.

### Example

The following example illustrates the bit mapping process.

The table queried contains the following set of rows.

Base Table

RowID	column_1	column_2	column_3
12,07	A	B	C
22,03	A	C	C
48,12	A	B	D
63,15	B	B	C
...	...	...	...

The query ANDs three equality conditions on three NUSI columns in the WHERE clause.

```
SELECT *
FROM Base_Table
WHERE column_1 = 'A'
AND   column_2 = 'B'
AND   column_3 = 'C';
```

The Optimizer evaluates the query and determines that bit mapping over the three columns is the least costly access plan.

The truth table looks like this.

NUSI Column Number	Value Required To Satisfy Condition	Base Table RowID List				
		12,07	22,03	48,12	63,15	...
1	A	1	1	1	0	...
2	B	1	0	1	1	...
3	C	1	0	0	0	...
ANDed Sum:		1	0	0	0	...

Of the four rows examined in the example, only the row identified by RowID 12,07 qualifies and is returned to the requesting application.

## Related Topics

Also see the topic “EXPLAIN Request Modifier” in *SQL Data Manipulation Language*, the chapter “Interpreting the Output of the EXPLAIN Request Modifier” in *SQL Request and Transaction Processing*, and *Teradata Visual Explain User Guide*.

# NUSIs and Query Covering

The Optimizer aggressively pursues NUSIs when they can cover a query. The expression *covering* means that all of the columns requested in a query or data necessary for satisfying the

query are also available from an existing index subtable, making it unnecessary to access the base table rows themselves to complete the query. Some vendors refer to this as index-only access.

For column-partitioned tables, other access methods are available, and may be chosen by the Optimizer based on cost comparisons.

Covering of a query can also be partial, or an index can fail to cover any aspect of a query.

The Optimizer also selects a secondary index to process an aggregation on indexed values if the index covers the necessary values.

In the case of a partial covering index for single-table access, the system can get the row IDs for those base table rows that possibly qualify for a query by preliminary examination from the index, but then must also access the base table itself to retrieve the definitively qualified rows. Even a partial covering index can accelerate the processing of a query, especially when the base table being accessed is very large and the query constraints on the partially covering NUSI are highly selective. The optimizer determines if a partial covering index should be used based on the cost comparisons.

## Criteria

A NUSI covers a query if any of the following criteria are true:

- The query does not reference any columns of the base table.  
For example, `SELECT COUNT(*) from table;`
- The NUSI includes all base table columns referenced in the query and any of the following criteria are true:
  - The query does not reference any *changeable* character columns.  
Changeable character columns are character columns that are not defined as CASESPECIFIC or UPPERCASE. Teradata Database converts changeable character column data to uppercase when it stores it in a NUSI defined without the ALL option. The data stored in the index subtable might be different from the original lowercase data stored in the base table.
  - This NUSI is defined with the ALL option.
  - This NUSI is *not* defined with the ALL option, *and* it contains a changeable character column set, *but* the changeable character columns are only specified in a COUNT function or UPPERCASE operator in the query.
  - This NUSI is *not* defined with the ALL option *and* it contains a changeable character column set, *and* the changeable character column set is only specified in a COUNT function or UPPERCASE operator in the select list, *and* there is no CASESPECIFIC condition on the changeable character column set in the query conditions.

A partially-covering NUSI is one that does not fully cover a query, but *does* satisfy both of the following criteria.

- Some single-table constraints of the query contain the NUSI column set.
- If the NUSI contains a changeable character column set, the query does not specify an inequality CASESPECIFIC condition on the changeable character column set, *and* no

CASE expression is specified in a query condition that also specifies the changeable character column set.

## Column Candidacy for Covered Access

The Optimizer attempts to use an index, if available, as a cover index, even when classical indexing techniques do not apply, because scanning an index is almost always faster than scanning the base table it references. The enhanced efficiency can result in significantly faster retrieval.

### Example 1: Index Used to Cover Simple Query

The following example demonstrates a situation in which the Optimizer can use the index in place of the base table to fully satisfy the query.

```
CREATE INDEX idxord (o_orderkey, o_date, o_totalprice)
ON orders;

SELECT o_date, AVG(o_totalprice)
FROM orders
WHERE o_orderkey >1000
GROUP BY o_date;
```

### Example 2: Index used to Cover Aggregate Query

Assume an index is defined on column *deptno*. The Optimizer can use that index and instruct the AMPs to aggregate on the index values, rather than using the more costly path of accessing the underlying base table.

```
SELECT deptno, COUNT(*)
FROM employee
GROUP BY deptno;
```

### Example 3: Index Values Can Be Manipulated Arithmetically

This example shows how index values can be manipulated arithmetically as well as being counted. In this case, an index is defined on *salary\_amount*. The index alone can be used to satisfy this query.

```
SELECT SUM(salary_amount)
FROM employee
GROUP BY salary_amount;
```

## Value-Ordered NUSIs and Range Conditions

Value-ordered NUSIs are very efficient for range conditions. Because the NUSI rows are sorted by data value, it is possible to search only a portion of the index subtable for a given range of key values.

Note that for most applications, a partitioned primary index on a join index is a better choice to handle range conditions than a value-ordered NUSI (see “[Designing for Range Queries: Guidelines for Choosing Between a PPI and a Value-Ordered NUSI](#)” on page 506).

## Limitations

- The sort key is limited to a single numeric or DATE column.
- The sort key column cannot exceed four bytes in length.
- If defined over multiple columns and with an ORDER BY clause, they count as two consecutive indexes against the total of 32 non-primary indexes you can define on a base or join index table. This includes the system-defined secondary indexes used to implement PRIMARY KEY and UNIQUE constraints.  
One index represents the column list and the other index represents the ordering column.

## Importance of Consecutive Indexes for Value-Ordered NUSIs

The system automatically assigns incrementally increasing numbers to indexes when they are created on a table. This is not important externally except for the case of composite value-ordered NUSIs, because these indexes not only consume two of the allotted 32 index numbers from the pool, but those two index numbers are *consecutive*. One index of the consecutively numbered pair represents the column list, while the other index in the pair represents the ordering column.

To understand why this is important, consider the following scenario.

- 1 You create 32 indexes on a table, none of which is value-ordered.  
This includes the system-defined secondary indexes used to implement PRIMARY KEY and UNIQUE constraints.
- 2 You drop every other index on the table, meaning that you drop either all the odd-numbered indexes or all the even-numbered indexes.  
For example, if you had dropped all the even-numbered indexes, there would now be 16 odd-numbered index numbers available to be assigned to indexes created in the future.
- 3 You attempt to create a value-ordered multicolumn NUSI.

The request aborts and the system returns an error message to you.

The reason the request aborts is that two *consecutive* index numbers were not available for assignment to the composite value-ordered NUSI.

You are still able to create 16 additional non-value-ordered NUSIs, single-column value-ordered NUSIs, USIs, hash indexes, or join indexes, but you cannot create any composite value-ordered NUSIs.

To work around this problem, perform the following procedure.

- 1 Drop any index on the table.  
This action frees 2 consecutive index numbers.
- 2 Create the value-ordered multicolumn NUSI that failed previously.
- 3 Recreate the index you dropped to free the consecutive index numbers.

Here is a simple example of why value-ordered composite NUSIs consume two consecutive index numbers from the total available number of 32.

First define a table on which a NUSI is to be defined.

```
CREATE MULTISET TABLE transsupplier (
    suppkey INTEGER NOT NULL,
    name     CHAR(25) CHARACTER SET LATIN CASESPECIFIC NOT NULL,
    seccode INTEGER NOT NULL,
    groupID INTEGER NOT NULL)
UNIQUE PRIMARY INDEX ( suppkey );
```

Then you create an ordered composite NUSI on *seccode* and *groupID*, ordering on *groupID*.

```
CREATE INDEX (seccode, groupID)
ORDER BY VALUES (groupID) ON transsupplier;
```

You find two rows in *DBC.Indexes* with an *IndexType* code of V (meaning a value-ordered secondary index) and an *IndexNumber* of 4 for this composite NUSI. The third row, with an *IndexType* of I and an *IndexNumber* of 8, represents the fact that the *groupID* column is the ordering column of the two columns making up the composite NUSI, so a single ordered composite NUSI uses up *two* of the available 32 secondary, hash, or join indexes that can be defined for *TransSupplier*.

```
SELECT s.CreateTimestamp, s.IndexType, s.IndexNumber, t.TVMName,
       s.FieldName
  FROM DBC.Indexes AS s, DBC.TVM AS t
 WHERE s.tableID = t.TVMID
   AND s.databaseID = t.databaseID
   AND s.fieldID = s.fieldID
   AND s.tableID = s.tableID
   AND s.databaseID = s.databaseID
   AND s.indextype <> 'P'
   AND t.TVMName = 'Transsupplier'
 ORDER BY s.createtimestamp DESC;
```

CreateTimeStamp	IndexType	IndexNumber	TVMName	FieldName
2007-04-27 14:51:55	I	8	TransSupplier	GROUPID
2007-04-27 14:51:55	V	4	TransSupplier	GROUPID
2007-04-27 14:51:55	V	4	TransSupplier	SECCODE

Note that Index Number 1 is reserved for the primary index of a table (which is *suppkey* in the example, but is not included in the report to simplify its meaning), and subsequent indexes are numbered beginning with 4 in increments of 4, so indexes 4 and 8 are consecutively numbered.

Suppose you had instead created the following ordered single-column NUSI on *TransSupplier*, this time on *groupID* only, ordering on *groupID*.

```
CREATE INDEX (groupID)
ORDER BY values (groupID) ON transsupplier;
```

You then run the same query against *DBC.Indexes*.

```
SELECT i.CreateTimestamp, i.IndexType, i.IndexNumber, t.TVMName,
       f.FieldName
```

```
FROM DBC.Indexes AS i, DBC.TVM AS t, DBC.TVFields AS f
WHERE i.tableID = t.TVMID
AND i.databaseID = t.databaseID
AND i.fieldID = f.fieldID
AND i.tableID = f.tableID
AND i.databaseID = f.databaseID
AND i.indextype <> 'P'
AND t.TVMName = 'Transsupplier'
ORDER BY i.createtimestamp DESC;
```

The query produces a report something like this, where you find only one row in *DBC.Indexes* with an *IndexType* code of V for this single-column NUSI, meaning that only *one* of the possible 32 secondary, hash, and join indexes has been consumed by the creation of this NUSI.

CreateTimeStamp	IndexType	IndexNumber	TVMName	FieldName
2007-04-27 14:56:24	V	4	TransSupplier	GROUPID

Index number 1, for primary index column *suppkey*, is not displayed in the report in order to simplify the point of the example.

## Typical Uses of Value-Ordered and Hash-Ordered NUSIs

The typical use of a hash-ordered NUSI is with an equality condition on the secondary index column set. The specified secondary key value is hashed, and then each NUSI subtable is probed for rows having the same row hash. For every matching NUSI entry, the corresponding rowIDs are used to access the base rows on the same AMP. Because the NUSI rows are stored in row hash order, searching the NUSI subtable for a particular row hash value is very efficient.

Value-ordered NUSIs, on the other hand, are useful for processing range conditions and conditions with either an equality or inequality on the secondary index column set.

Although hash-ordered NUSIs can be selected by the Optimizer to access rows for range conditions, a far more common response is to specify a full-table scan of the NUSI subtable to find the matching secondary key values. Therefore, depending on the size of the NUSI subtable and the number of qualified rows, this might not be very efficient.

By sorting the NUSI rows by data value, it is possible to search only a portion of the index subtable for a given range of key values. The Optimizer must still estimate the selectivity of a NUSI to be high for it to cost less than a full-table scan. The major advantage of a value-ordered NUSI is in the performance of range queries.

## Example

The following example illustrates a value-ordered NUSI (defined by an ORDER BY clause that specifies the VALUES keyword option on *o\_orderdate*) and a query that would probably be solved more efficiently if the specified value-ordered NUSI were selected by the Optimizer to access the requested rows.

```
CREATE INDEX Idx_Date (o_orderdate)
ORDER BY VALUES (o_orderdate)
```

```
ON Orders;

SELECT *
FROM Orders
WHERE o_orderdate
BETWEEN DATE '2005-10-01'
AND     DATE '2005-10-07';
```

## Selecting a Secondary Index

When assigning columns to be a secondary index for a table, there are numerous factors to consider, the most important of all being the selectivity of the index.

While USI retrievals are always very efficient, the efficiency of NUSI retrievals varies greatly depending on their selectivity.

You can use the Teradata Index Wizard client utility to assist your selection of secondary and single-table join indexes for a table. The utility recommends a set of indexes based on specific query workloads you supply. The Teradata Index Wizard can also validate its recommendations to ensure they actually provide the benefits expected. See *Teradata Index Wizard User Guide* for further information about how to use the utility and *SQL Request and Transaction Processing* for information about how the utility performs its component tasks.

### Optimal Data Access

Selectivity is a relative term that refers to the number of rows returned by an index. Most retrievals aim to return only a select few rows: very specific answers in response to a very specific request.

An index that returns a small number of rows is said to be highly selective. This is a positive attribute.

Indexes that return a large number of rows are said to have low selectivity. This is generally a negative attribute; so negative that, as often as not, the Optimizer selects a full-table scan over a NUSI with low selectivity because the full-table scan can be less costly.

All UPIs and USIs are highly selective by definition, as are most well-chosen NUPIs. High selectivity is favored not only because of its precision, but also because of its low cost, involving a very small number of disk I/Os, which is always a performance-enhancing attribute.

### Criteria for Selecting a Secondary Index

The following rules of thumb and performance considerations apply to selecting a unique or non-unique column set as a secondary index for a table.

- Consider naming secondary indexes whenever possible using a standard naming convention.
- Avoid assigning secondary indexes to frequently updated column sets.

- Avoid assigning secondary indexes to columns with lumpy distributions because there is a slight chance the Optimizer might mistake their usefulness.
- Avoid creating excessive secondary indexes on a table, particularly for a table used heavily, or even moderately, for OLTP processing. The less frequently the table is updated, the more desirable a multiple index solution.
- Consider building secondary indexes on column sets frequently involved in the following clauses, predicates, and other logical operations:
  - Selection criteria
  - Join criteria
  - ORDER BY clauses
  - GROUP BY clauses
  - Foreign keys (because of join and subquery processing)
  - UNION, DISTINCT, and other sort operations

When these operations act on well-indexed column sets, the number of scans and sorts that must be performed on the data by the database manager can be greatly reduced.

- Consider creating USIs for NoPI tables that require frequent single-row access because the only alternative is a full-table scan.
- Consider creating NUSIs for NoPI tables that require frequent set selection access because the only alternative is a full-table scan.
- Consider creating a simple NUSI on geospatial columns that are frequently queried. This is especially true for requests that contain geospatial predicate terms, geospatial join terms, or both.

Note the following about geospatial secondary indexes:

- You cannot create a USI on a geospatial column.
- You cannot create a composite geospatial NUSI. Geospatial indexes can only be defined on a single geospatial column.
- Consider creating covering indexes when possible and cost effective (including considering the cost of maintaining the index). The Optimizer frequently selects covering indexes to substitute for a base table access whenever the overall cost of the query plan is reduced. Such index-only access promotes faster retrievals.

Many applications are well served by join indexes, which can be used profitably in many covering situations where multiple columns are frequently joined. See [Chapter 11: “Join and Hash Indexes”](#) for further information about join indexes.

- Consider creating secondary indexes on columns frequently operated on by built-in functions such as aggregates.
- Consider assigning a uniqueness constraint such as PRIMARY KEY, UNIQUE, or USI, as appropriate, to the primary or other alternate key of any table built with a NUPI. This both enforces uniqueness, eliminating the burden of making row uniqueness checks, and enhances retrieval for applications where the primary or other alternate key is frequently used as a selection or join criterion.

This guideline is situational and is contingent on a number of factors. The various factors involved in the recommendation are described in “[Using Unique Secondary Indexes to Enforce Row Uniqueness](#)” on page 457.

A primary or alternate key USI might *not* be a good decision for a table that is frequently updated by OLTP applications.

- Plan to dynamically drop and recreate secondary indexes to accommodate specific processing and performance requirements such as bulk data loading utilities, database archives, and so on.

Create appropriate macros to perform these drop and create index operations if you need to undertake such specific processing tasks regularly.

- Ensure that your indexes are being used as planned by submitting EXPLAIN request modifiers to audit index selection for those queries they are designed to facilitate.

Indexes that are never selected by the Optimizer are a burden to the system for the following reasons.

- They consume disk resources that could profitably be used to store data or indexes that *are* used.
- They degrade update processing performance unnecessarily.
- You can include UDT columns in a secondary index definition.
- Never attempt to include columns defined with an XML, BLOB, CLOB, BLOB-based UDT, CLOB-based UDT, XML-based UDT, Period, JSON, ARRAY, or VARRAY data type in a secondary index definition.

You can define a simple NUSI, but not a composite NUSI, on a geospatial column.

You cannot include a column defined with a geospatial data type in a USI definition.

- Never attempt to define a secondary index on a global temporary trace table. See *SQL Data Definition Language*.

## Secondary Index Access Summarized by Example

This example indicates how a series of queries against a table can use various secondary indexes to access the rows in that table.

### Configuration

The system for this example has four AMPs.

### Table Definition

The table used in this demonstration is a simple three-column *customer* table, defined as follows.

Column Name	Attribute Described	Type of Index Defined on the Column
Cust	Customer Number	USI
Name	Customer Last Name	NUSI
Phone	Customer Phone Number	NUPI

The following is a snapshot instance of this table.

**Customer**

Cust	Name	Phone
USI	NUSI	NUPI
37	White	555-4444
98	Brown	333-9999
74	Smith	555-6666
95	Peters	555-7777
27	Jones	222-8888
56	Smith	555-7777
45	Adams	444-6666
31	Adams	111-2222
40	Smith	222-3333
72	Adams	666-7777
84	Rice	666-5555
49	Smith	111-6666
12	Young	777-7777
62	Black	444-5555
77	Jones	777-6666
51	Marsh	888-2222

Base table and secondary index subtable rows are distributed as illustrated by the following graphic.

<b>USI Subtable</b>	<b>USI Subtable</b>	<b>USI Subtable</b>	<b>USI Subtable</b>																																																																																																
<table border="1"> <thead> <tr> <th>RowID</th><th>Cust</th><th>RowID</th></tr> </thead> <tbody> <tr><td>244, 1</td><td>74</td><td>884, 1</td></tr> <tr><td>505, 1</td><td>77</td><td>639, 1</td></tr> <tr><td>744, 4</td><td>51</td><td>915, 9</td></tr> <tr><td>757, 1</td><td>27</td><td>388, 1</td></tr> </tbody> </table>	RowID	Cust	RowID	244, 1	74	884, 1	505, 1	77	639, 1	744, 4	51	915, 9	757, 1	27	388, 1	<table border="1"> <thead> <tr> <th>RowID</th><th>Cust</th><th>RowID</th></tr> </thead> <tbody> <tr><td>135, 1</td><td>98</td><td>555, 6</td></tr> <tr><td>296, 1</td><td>84</td><td>536, 5</td></tr> <tr><td>602, 1</td><td>56</td><td>778, 7</td></tr> <tr><td>969, 1</td><td>49</td><td>147, 1</td></tr> </tbody> </table>	RowID	Cust	RowID	135, 1	98	555, 6	296, 1	84	536, 5	602, 1	56	778, 7	969, 1	49	147, 1	<table border="1"> <thead> <tr> <th>RowID</th><th>Cust</th><th>RowID</th></tr> </thead> <tbody> <tr><td>288, 1</td><td>31</td><td>638, 1</td></tr> <tr><td>339, 1</td><td>40</td><td>640, 1</td></tr> <tr><td>372, 2</td><td>45</td><td>471, 1</td></tr> <tr><td>588, 1</td><td>95</td><td>778, 3</td></tr> </tbody> </table>	RowID	Cust	RowID	288, 1	31	638, 1	339, 1	40	640, 1	372, 2	45	471, 1	588, 1	95	778, 3	<table border="1"> <thead> <tr> <th>RowID</th><th>Cust</th><th>RowID</th></tr> </thead> <tbody> <tr><td>175, 1</td><td>37</td><td>107, 1</td></tr> <tr><td>489, 1</td><td>72</td><td>717, 2</td></tr> <tr><td>838, 1</td><td>12</td><td>147, 2</td></tr> <tr><td>919, 1</td><td>62</td><td>822, 1</td></tr> </tbody> </table>	RowID	Cust	RowID	175, 1	37	107, 1	489, 1	72	717, 2	838, 1	12	147, 2	919, 1	62	822, 1																																				
RowID	Cust	RowID																																																																																																	
244, 1	74	884, 1																																																																																																	
505, 1	77	639, 1																																																																																																	
744, 4	51	915, 9																																																																																																	
757, 1	27	388, 1																																																																																																	
RowID	Cust	RowID																																																																																																	
135, 1	98	555, 6																																																																																																	
296, 1	84	536, 5																																																																																																	
602, 1	56	778, 7																																																																																																	
969, 1	49	147, 1																																																																																																	
RowID	Cust	RowID																																																																																																	
288, 1	31	638, 1																																																																																																	
339, 1	40	640, 1																																																																																																	
372, 2	45	471, 1																																																																																																	
588, 1	95	778, 3																																																																																																	
RowID	Cust	RowID																																																																																																	
175, 1	37	107, 1																																																																																																	
489, 1	72	717, 2																																																																																																	
838, 1	12	147, 2																																																																																																	
919, 1	62	822, 1																																																																																																	
<b>NUSI Subtable</b>	<b>NUSI Subtable</b>	<b>NUSI Subtable</b>	<b>NUSI Subtable</b>																																																																																																
<table border="1"> <thead> <tr> <th>RowID</th><th>Name</th><th>RowID</th></tr> </thead> <tbody> <tr><td>448, 1</td><td>White</td><td>107, 1</td></tr> <tr><td>656, 1</td><td>Rice</td><td>536, 5</td></tr> <tr><td>567, 3</td><td>Adams</td><td>638, 1</td></tr> <tr><td>432, 8</td><td>Smith</td><td>640, 1</td></tr> </tbody> </table>	RowID	Name	RowID	448, 1	White	107, 1	656, 1	Rice	536, 5	567, 3	Adams	638, 1	432, 8	Smith	640, 1	<table border="1"> <thead> <tr> <th>RowID</th><th>Name</th><th>RowID</th></tr> </thead> <tbody> <tr><td>567, 2</td><td>Adams</td><td>717, 2</td></tr> <tr><td>372, 2</td><td>Adams</td><td>471, 1</td></tr> <tr><td>852, 1</td><td>Brown</td><td>555, 6</td></tr> <tr><td>432, 3</td><td>Smith</td><td>884, 1</td></tr> </tbody> </table>	RowID	Name	RowID	567, 2	Adams	717, 2	372, 2	Adams	471, 1	852, 1	Brown	555, 6	432, 3	Smith	884, 1	<table border="1"> <thead> <tr> <th>RowID</th><th>Name</th><th>RowID</th></tr> </thead> <tbody> <tr><td>432, 1</td><td>Smith</td><td>147, 1</td></tr> <tr><td>770, 1</td><td>Young</td><td>147, 2</td></tr> <tr><td>567, 6</td><td>Jones</td><td>338, 1</td></tr> <tr><td>448, 4</td><td>Black</td><td>822, 1</td></tr> </tbody> </table>	RowID	Name	RowID	432, 1	Smith	147, 1	770, 1	Young	147, 2	567, 6	Jones	338, 1	448, 4	Black	822, 1	<table border="1"> <thead> <tr> <th>RowID</th><th>Name</th><th>RowID</th></tr> </thead> <tbody> <tr><td>262, 1</td><td>Jones</td><td>639, 1</td></tr> <tr><td>396, 1</td><td>Peters</td><td>778, 3</td></tr> <tr><td>432, 5</td><td>Smith</td><td>778, 7</td></tr> <tr><td>155, 1</td><td>Marsh</td><td>915, 9</td></tr> </tbody> </table>	RowID	Name	RowID	262, 1	Jones	639, 1	396, 1	Peters	778, 3	432, 5	Smith	778, 7	155, 1	Marsh	915, 9																																				
RowID	Name	RowID																																																																																																	
448, 1	White	107, 1																																																																																																	
656, 1	Rice	536, 5																																																																																																	
567, 3	Adams	638, 1																																																																																																	
432, 8	Smith	640, 1																																																																																																	
RowID	Name	RowID																																																																																																	
567, 2	Adams	717, 2																																																																																																	
372, 2	Adams	471, 1																																																																																																	
852, 1	Brown	555, 6																																																																																																	
432, 3	Smith	884, 1																																																																																																	
RowID	Name	RowID																																																																																																	
432, 1	Smith	147, 1																																																																																																	
770, 1	Young	147, 2																																																																																																	
567, 6	Jones	338, 1																																																																																																	
448, 4	Black	822, 1																																																																																																	
RowID	Name	RowID																																																																																																	
262, 1	Jones	639, 1																																																																																																	
396, 1	Peters	778, 3																																																																																																	
432, 5	Smith	778, 7																																																																																																	
155, 1	Marsh	915, 9																																																																																																	
<b>Base Table</b>	<b>Base Table</b>	<b>Base Table</b>	<b>Base Table</b>																																																																																																
<table border="1"> <thead> <tr> <th>RowID</th><th>Cust</th><th>Name</th><th>Phone</th></tr> </thead> <tbody> <tr><td></td><td>USI</td><td>NUSI</td><td>NUPI</td></tr> <tr><td>107, 1</td><td>37</td><td>White</td><td>555-4444</td></tr> <tr><td>536, 5</td><td>84</td><td>Rice</td><td>666-5555</td></tr> <tr><td>638, 1</td><td>31</td><td>Adams</td><td>111-2222</td></tr> <tr><td>640, 1</td><td>40</td><td>Smith</td><td>222-3333</td></tr> </tbody> </table>	RowID	Cust	Name	Phone		USI	NUSI	NUPI	107, 1	37	White	555-4444	536, 5	84	Rice	666-5555	638, 1	31	Adams	111-2222	640, 1	40	Smith	222-3333	<table border="1"> <thead> <tr> <th>RowID</th><th>Cust</th><th>Name</th><th>Phone</th></tr> </thead> <tbody> <tr><td></td><td>USI</td><td>NUSI</td><td>NUPI</td></tr> <tr><td>471, 1</td><td>45</td><td>Adams</td><td>444-6666</td></tr> <tr><td>555, 6</td><td>98</td><td>Brown</td><td>333-9999</td></tr> <tr><td>717, 2</td><td>72</td><td>Adams</td><td>666-7777</td></tr> <tr><td>884, 1</td><td>74</td><td>Smith</td><td>555-6666</td></tr> </tbody> </table>	RowID	Cust	Name	Phone		USI	NUSI	NUPI	471, 1	45	Adams	444-6666	555, 6	98	Brown	333-9999	717, 2	72	Adams	666-7777	884, 1	74	Smith	555-6666	<table border="1"> <thead> <tr> <th>RowID</th><th>Cust</th><th>Name</th><th>Phone</th></tr> </thead> <tbody> <tr><td></td><td>USI</td><td>NUSI</td><td>NUPI</td></tr> <tr><td>147, 1</td><td>49</td><td>Smith</td><td>111-6666</td></tr> <tr><td>147, 2</td><td>12</td><td>Young</td><td>777-4444</td></tr> <tr><td>388, 1</td><td>27</td><td>Jones</td><td>222-8888</td></tr> <tr><td>822, 1</td><td>62</td><td>Black</td><td>444-5555</td></tr> </tbody> </table>	RowID	Cust	Name	Phone		USI	NUSI	NUPI	147, 1	49	Smith	111-6666	147, 2	12	Young	777-4444	388, 1	27	Jones	222-8888	822, 1	62	Black	444-5555	<table border="1"> <thead> <tr> <th>RowID</th><th>Cust</th><th>Name</th><th>Phone</th></tr> </thead> <tbody> <tr><td></td><td>USI</td><td>NUSI</td><td>NUPI</td></tr> <tr><td>639, 1</td><td>77</td><td>Jones</td><td>777-6666</td></tr> <tr><td>778, 3</td><td>95</td><td>Peters</td><td>555-7777</td></tr> <tr><td>778, 7</td><td>56</td><td>Smith</td><td>555-7777</td></tr> <tr><td>915, 9</td><td>51</td><td>Marsh</td><td>888-2222</td></tr> </tbody> </table>	RowID	Cust	Name	Phone		USI	NUSI	NUPI	639, 1	77	Jones	777-6666	778, 3	95	Peters	555-7777	778, 7	56	Smith	555-7777	915, 9	51	Marsh	888-2222
RowID	Cust	Name	Phone																																																																																																
	USI	NUSI	NUPI																																																																																																
107, 1	37	White	555-4444																																																																																																
536, 5	84	Rice	666-5555																																																																																																
638, 1	31	Adams	111-2222																																																																																																
640, 1	40	Smith	222-3333																																																																																																
RowID	Cust	Name	Phone																																																																																																
	USI	NUSI	NUPI																																																																																																
471, 1	45	Adams	444-6666																																																																																																
555, 6	98	Brown	333-9999																																																																																																
717, 2	72	Adams	666-7777																																																																																																
884, 1	74	Smith	555-6666																																																																																																
RowID	Cust	Name	Phone																																																																																																
	USI	NUSI	NUPI																																																																																																
147, 1	49	Smith	111-6666																																																																																																
147, 2	12	Young	777-4444																																																																																																
388, 1	27	Jones	222-8888																																																																																																
822, 1	62	Black	444-5555																																																																																																
RowID	Cust	Name	Phone																																																																																																
	USI	NUSI	NUPI																																																																																																
639, 1	77	Jones	777-6666																																																																																																
778, 3	95	Peters	555-7777																																																																																																
778, 7	56	Smith	555-7777																																																																																																
915, 9	51	Marsh	888-2222																																																																																																
AMP 1	AMP 2	AMP 3	AMP 4																																																																																																

FF07D315

## Sample Queries

The following queries can all be answered without having to do a full-table scan. Note that *uniqueness value* is abbreviated UV throughout.

### Query 1

The first query uses the NUPI column *phone* as the WHERE clause attribute, with the requested value being 555-7777.

```
SELECT *
FROM Customer
WHERE Phone = '555-7777';
```

- 1 The hashing algorithm generates the row hash for this primary index access and finds that rows having the NUPI value 555-7777 hash to AMP 4.
- 2 The Dispatcher sends an AMP retrieval step directly to AMP 4, where the file system retrieves two matching rows.

The first matching row has row hash=778 and UV=3, while the second matching row has row hash=778 and UV=7. The row hash values are identical because this primary index is non-unique.

- 3 The file system reads the requested rows and returns them to the requestor.

Cust=95, Name=Peters, Phone=555-7777

Cust=56, Name=Smith, Phone=555-7777

## Query 2

The second query uses the USI column *Cust* as the WHERE clause attribute, with the requested value being 95.

```
SELECT *
FROM Customer
WHERE Cust = 95;
```

- 1 The hashing algorithm generates the row hash for this USI access.  
The hash map indicates that an index row having the value 95 hashes to the *customer* USI subtable on AMP 3 with a row hash=588.
- 2 The Dispatcher sends an AMP retrieval step directly to AMP 3, where the file system reads the subtable row having the row hash=588 to determine which AMP owns the base table row.
- 3 The file system retrieves the base table row hash 778 and uniqueness value 3 from the USI subtable row and determines that the requested base table row is stored on AMP 4.
- 4 The retrieval directive is passed to AMP 4 where the row for customer number 95, having rowID value 778,3 is located.
- 5 The file system reads the requested row and returns it to the requestor.

Cust=95, Name=Peters, Phone=555-7777

## Query 3

The third query uses the NUSI column *name* as the WHERE clause attribute, with the requested value being the *name* column value Smith.

```
SELECT *
FROM Customer
WHERE Name = 'Smith';
```

- 1 The Dispatcher broadcasts an AMP retrieval step containing the NUSI value Smith to all AMPS.
- 2 The file system scans the NUSI subtable with row hash 432 and selects index rows for Smith in *customer* on each AMP, retrieving all the base table rowIDs and uniqueness values associated with the *name* column value of Smith and row hash value 432.

The steps are processed in parallel and one row is located on each AMP.

- AMP1 (NUSI row hash=432, UV=8; Base table row hash=640, UV=1)
  - AMP2 (NUSI row hash=432, UV=3; Base table row hash=884, UV=1)
  - AMP3 (NUSI row hash=432, UV=1; Base table row hash=147, UV=1)
  - AMP4 (NUSI row hash=432, UV=5; Base table row hash=778, UV=7)
- 3 The file system directly retrieves the base table rows on each AMP in parallel and returns them to the requestor.

Cust=40, Name=Smith, Phone=222-3333

Cust=74, Name=Smith, Phone=555-6666

Cust=49, Name=Smith, Phone=111-6666

Cust=56, Name=Smith, Phone=555-7777

## Secondary Index Build and Access Operations Summarized

The following table summarizes how the system processes USIs and NUSIs, based on the following simple SQL requests.

- Supporting USI build request.

```
CREATE UNIQUE INDEX (customer_number)
ON customer_table;
```

- Supporting NUSI build request.

```
CREATE INDEX NUSI_name (customer_name)
ON customer_table;
```

- Supporting USI access request.

```
SELECT *
FROM customer_table
WHERE customer_number=12;
```

- Supporting NUSI access request.

```
SELECT *
FROM customer_table
WHERE customer_name='SMITH';
```

## Secondary Index Build Process Table

Stage	Process			
	Build		Access	
	USI	NUSI	USI	NUSI
1	Each AMP accesses its subset of the base table rows.		The supplied index value hashes to the corresponding secondary index row and the base table rowID is located.	A message containing the secondary index value and its row hash for an equality condition, or a condition on an index value, is broadcast to every AMP.
2	Each AMP copies the secondary index value and appends the rowID for the base table row.	Each AMP builds a spool file containing each secondary index value found followed by the rowID for the row it came from.	The retrieved base table rowID is used to access the specific data row.	<p>For an equality condition, the system does a hash lookup to find the rows that satisfy the predicate.</p> <p>For an inequality condition, there are two possibilities.</p> <ul style="list-style-type: none"> <li>• A full subtable scan. If this is too costly, the Optimizer does not use it, but it is a legitimate possibility.</li> <li>• A hash lookup. This is a good choice when a predicate specifies a small range or if the statistics indicate that few rows satisfy the predicate.</li> </ul> <p>It is also a good choice if the base table is very wide, but the NUSI is very narrow.</p> <p>Otherwise, the system attempts to combine several NUSIs using bit mapping. Failing that, no index is used, and the system does a full-table scan.</p> <p>For a hash-ordered NUSI, each AMP uses the row hash and uses it to access its portion of the index subtable to see if a corresponding index row exists.</p> <p>Value-ordered NUSI index subtable values are scanned only for the range of values specified by the query.</p>

Stage	Process			
	Build		Access	
	USI	NUSI	USI	NUSI
2 (continued)				<p>On a range predicate, the system can enter the index at its lowest value and scan to its last value. This cannot be done with ORed ranges, so a full subtable scan must be used.</p> <p>If not a range predicate, then the system must scan the entire index.</p>
3	<p>Each AMP creates a Row Hash on the secondary index value and puts the value, the hash, and the row ID values onto the BYNET.</p> <p>The receiving AMP puts these values into the index subtable, sorts on the row hash, and adds a uniqueness value.</p>	<p>For hash-ordered NUSIs, each AMP sorts the row hash and row values for each secondary index value into ascending order.</p> <p>For value-ordered NUSIs, the rows are sorted in NUSI value order.</p>	<p>The process is complete. This is typically a two-AMP operation.</p>	<p>If an index row is found, the AMP uses the base table rowID set to access the corresponding base table rows.</p>
4	The process is complete.	Rows having the same row value are collapsed into one index row or multiple rows if the row length limit is reached.		The process is complete.
5		The process is complete.		

## Secondary Index Usage Summary

All Teradata Database secondary indexes have the following properties:

- Can enhance the speed of data retrieval.  
Because of this, secondary indexes are most useful in decision support applications.
- Do not affect base table data distribution.
- Maximum of 32 secondary, hash, and join indexes defined per table. Each composite NUSI that specifies an ORDER BY clause counts as 2 consecutive indexes in this calculation (see [“Importance of Consecutive Indexes for Value-Ordered NUSIs” on page 485](#)).

The limit of 32 indexes applies to any combination of secondary, hash, and join indexes defined on a table, ranging from 0 secondary indexes and 32 join indexes, 11 hash indexes, 11 join indexes, and 10 secondary indexes to 32 secondary indexes and 0 join indexes.

This includes the system-defined secondary indexes used to implement PRIMARY KEY and UNIQUE constraints.

- Can be composed of as many as 64 concatenated columns.
- Can include columns defined with a UDT data type.
- Cannot contain columns defined with XML, BLOB, CLOB, BLOB-based UDT, CLOB-based UDT, XML-based UDT, Period, or JSON data types.

You can define a NUSI on a single column with a geospatial data type, but you cannot define a USI on a geospatial column.

You cannot define a composite NUSI that contains a geospatial column.

- Cannot be defined on global temporary trace tables.
- Can be created or dropped dynamically as data usage changes or if they are found not to be useful for optimizing data retrieval performance.
- Require additional disk space to store subtables.
- Require additional I/Os on INSERTs, DELETEs, and possibly on UPDATEs and MERGEs.

Because of this, secondary indexes are not nearly as useful in OLTP applications as they are in DSS applications.

- Should not be defined on columns whose values change frequently.
- Should not include columns that do not enhance selectivity.
- Should not use composite secondary indexes when multiple single column indexes and bit mapping might be used instead.
- Composite secondary index is useful if it reduces the number of rows that must be accessed.
- The Optimizer does not use composite secondary indexes unless a WHERE clause condition specifies explicit values for each column in the index.
- Most efficient for selecting a small number of rows.
- Can be unique or nonunique.
- NUSIs can be hash-ordered or value-ordered.

- Ordering for NUSIs defined with an ORDER BY clause is restricted to a single numeric or DATE column of 4 or fewer bytes.
- If they cover, or partially cover, a query, then they further improve their usefulness.

## USI Summary

- Can be used to enforce row uniqueness for multiset NUPI and NoPI tables.
- Guarantee that each complete index value is unique.
- Any access is, at most, a two-AMP operation.

## NUSI Summary

- Useful for locating rows having a specific value in the index.
- Can be hash-ordered or value-ordered.  
Value-ordered NUSIs are particularly useful for enhancing the performance of range queries.
- Any access is an all-AMPs operation with the exception of the case where a NUSI is defined on the same column set as the primary index for the table.
- If an index is defined with an ORDER BY clause, it counts as 2 consecutive indexes against the table limit of 32 secondary, hash, and join indexes (see “[Importance of Consecutive Indexes for Value-Ordered NUSIs](#)” on page 485).

## Related Topics

For more information about secondary indexes., see *SQL Data Definition Language* under the topics CREATE INDEX and CREATE TABLE.

# CHAPTER 11 Join and Hash Indexes

---

This chapter describes join and hash indexes. These indexes permit you to undertake a wide variety of physical denormalizations of the database without affecting the normalization of the logical and physical models.

You should always define an equivalent single-table join index rather than a hash index so as not to depend on the default choices that are made for a hash index to be the correct choices. Also, hash indexes have some limitations such as multivalue compression is not carried over to a hash index from the base table.

Other topics include information on how to estimate the overhead of various join indexes, Optimizer criteria for selecting a join index, special storage options offered by join indexes, and numerous examples.

See [Chapter 16: “Design Issues for Tactical Queries”](#) for a description of the special design considerations that must be evaluated for using hash and join indexes to support tactical queries.

For information about design issues related to join indexes defined on temporal tables and system-defined join indexes, see *ANSI Temporal Table Support* and *Temporal Table Support*.

## Join Indexes

Join indexes are designed to permit queries (join queries in the case of multitable join indexes) to be resolved by accessing the index instead of accessing, and possibly joining, their underlying base tables.

Multitable join indexes are useful for queries where the index contains all the columns referenced by one or more joins, thereby allowing the index to cover that part of the query, making it possible to retrieve the requested data from the index rather than accessing its underlying base tables. For obvious reasons, an index with this property is often referred to as a covering index (some vendors refer to this as index-only access). Note that a join index that is defined with an expression in its select list provides less coverage than a join index that is defined using a base column (see [“Restrictions on Partial Covering by Join Indexes” on page 575](#)).

Even if a join index does not completely cover a query, the Optimizer can use it to join to its underlying base tables in a way that provides better query optimization than scanning the base tables for all the columns specified in the request (see [“Partial Query Coverage” on page 505](#)).

From the point of view of a database designer, the multitable join index permits great adaptability because it provides the flexibility of normalization while at the same time offering

the opportunity to create alternative, denormalized virtual data models, providing what might be called materialized views of the database.

Depending on how the index is defined, single-table join indexes can also be useful for queries where the index contains only *some* of the columns referenced in the statement. This situation is referred to as a partial covering of the query. Multitable join indexes can also be used to partially cover a query for one or more of the tables defined in the join index.

Join indexes are also useful for queries that aggregate columns from tables with large cardinalities. These indexes play the role of prejoin and summary tables without denormalizing the logical design of the database and without incurring the update anomalies and performance problems presented by denormalized tables. While it is true that denormalization often enhances the performance of a particular query or family of queries, it can and often does make other queries perform more poorly.

Unlike traditional indexes, join indexes are not required to store pointers to their associated base table rows. Instead, they are generally used as a fast path *final* access point that eliminates the need to access and join the base tables they represent. They substitute *for* rather than point *to* base table rows. The only exception to this is the case where an index partially covers a query. If the index is defined using either the ROWID keyword, the UPI of its base table, or a USI on the base table as one of its columns, then it can be used to join with the base table to cover the query. A join index defined in this way is sometimes called a global index or global join index. Note that you can only specify ROWID in the outermost SELECT of the CREATE JOIN INDEX statement. See “CREATE JOIN INDEX” in *SQL Data Definition Language Detailed Topics*.

In this case, you might create a join index to contain only the join column and the ROWID or the UPI column set of the table. The process is as follows.

- 1 The join index is joined with the other table and any selection conditions that can be evaluated during the join to eliminate disqualified rows are applied.
- 2 The result of the join is stored in a temporary table and redistributed based on the ROWID, UPI, or USI of the base table to perform a join with the base table.

This join can happen at different points in the query plan, depending on estimated costs. For example, the Optimizer might determine that it is cheaper to perform a join with another table involved in the query before joining to the base table.

Depending on the workloads they are designed to support, you can create join indexes that have either partitioned or nonpartitioned primary indexes or are column-partitioned. You can also create column-partitioned join indexes that have *no* primary index. See “CREATE JOIN INDEX” in *SQL Data Definition Language Detailed Topics* for the usage rules for column-partitioned join indexes.

## Rules for Using the ROWID Keyword in a Join Index Definition

- The ROWID keyword can only be used in a join index definition.
- You can optionally specify ROWID for a base table in the select list of a join index definition. For a column-partitioned join index, the ROWID for the base table in the select list of the join index definition is required.

If you reference multiple tables in the join index definition, then you must fully qualify each ROWID specification.

- You can reference an alias name for ROWID, or the keyword ROWID itself if no correlation name has been specified for it, in the primary index definition or in a secondary index defined for the join index in its index clause.

This does *not* mean that you can reference a ROWID or its alias in a secondary index defined separately with a CREATE INDEX statement (see CREATE INDEX in *SQL Data Definition Language*) after the join index has been created.

- An alias is required for ROWID if the join index is column partitioned.
- If you reference a ROWID alias in the select list of a join index definition, then you can also reference that alias in a CREATE INDEX statement that creates a secondary index on the join index.
- Aliases are required to resolve any column name or ROWID ambiguities in the select list of a join index definition.

An example is the situation where you specify ROWID for more than one base table in the index definition.

Also see the description of the CREATE JOIN INDEX statement in *SQL Data Definition Language*.

## Rules for Using the System-Derived PARTITION Column in a Hash or Join Index Definition

You *cannot* specify the system-derived PARTITION column in any hash or join index definition.

You *can* specify a user-named column named *partition* in an index definition.

## Default Column Multivalue Compression for Join Index Columns When the Referenced Base Table Column Is Compressed

When you create a join index, Teradata Database automatically transfers any column multivalue compression defined on the base table, with a few exceptions, to the join index definition. In contrast, hash indexes do *not* inherit the multivalue compression defined for the columns of their parent base table.

The following rules and restrictions apply to automatically transferring the column compression characteristics of a base table to its underlying join index columns. All of these rules and restrictions must be met for base table column multi-value compression to transfer to the join index definition:

- Base table column multivalue compression transfers to a join index definition even if there is an alias name specified for the columns in the join index definition.
- Base table column compression transfers to a multivalue join index definition only up to the point that the maximum table header length of the join index is exceeded.

The CREATE JOIN INDEX request does not abort at that point, but the transfer of column multi-value compression from the base table to the join index definition stops.

- Base table column multivalue compression does *not* transfer to a join index definition if the column is a component of the primary index for the join index.
- Base table column multivalue compression does *not* transfer to a join index definition for any of the columns that are components of a partitioned primary index, a partitioning expression for a PPI join index, or a partitioning expression for a column-partitioned join index.
- Base table column multivalue compression does *not* transfer to a join index column if the column is specified as the argument for any of the following functions.
  - COUNT
  - EXTRACT
  - MIN
  - MAX
  - SUM
- Base table column multivalue compression does *not* transfer to a column in a compressed join index that has indexes defined on its *column\_1* and *column\_2* sets.
- Base table column multivalue compression does *not* transfer to a join index definition if the column is a component of an ORDER BY clause in the join index definition.

## Compression of Join Indexes at the Block Level

Because join indexes are primary tables, they can be compressed at the block level. See “[Data Block](#)” on page 248, “[Reserved Query Bands for Managing the Block-Level Compression and Storage Temperature of Newly Loaded Data](#)” on page 694, and “[Compression Types Supported by Teradata Database](#)” on page 695 for more information about data block compression.

## Summary of Join Index Functions

A join index always has at least one of the following functions.

- Replicates all, or a vertical subset, of a single base table and partitions its rows using a different primary index (or a different column partitioning if the base table is a column-partitioned table) than the base table, such as a foreign key column to facilitate joins of very large tables by hashing them to the same AMP.

**Note:** A partitioning expression for a PPI join index or column-partitioned join index cannot contain a row-level security constraint column.

- Joins multiple tables (optionally with aggregation) in a prejoin table.
- Aggregates one or more columns of a single table as a summary table.

Join indexes are updated automatically, so the only administrative task a DBA must perform is to keep the statistics on multitable join index columns and their indexes current. For non-sparse single-table join indexes, the best policy is to use base table statistics rather than to collect statistics directly on the columns of the index.

The recommended practice for recollecting statistics is to set appropriate thresholds for recollection using the THRESHOLD options of the COLLECT STATISTICS statement. See “COLLECT STATISTICS in *SQL Data Definition Language*” for details on how to do this.

You cannot collect statistics on complex expressions specified in a base table definition. However, if you frequently submit queries that specify complex base table predicate expressions, you can create a single-table join index or hash index that specifies those frequently used predicate expressions in its select list or column list, respectively, and then collect statistics on the expression defined as a simple column in your index.

There are several specific cases where join index statistics can provide more accurate cardinality estimates than are otherwise available for base table predicates written using complex date expressions.

- The case where an EXTRACT expression specified in a query predicate can be *matched* with a join index predicate.
- The case where an EXTRACT/DATE expression specified in a query predicate condition can be *mapped* to an expression specified in the select list of a join index.

The Optimizer uses expression mapping when it detects an identical query expression or a matching query expression subset within a non-matching predicate. When this occurs, the Optimizer maps the predicate to the identical column of the join index, which enables it to use the statistics collected on the join index column to estimate the cardinality of the expression result.

When you create a single-table join index that specifies a complex expression, Teradata Database transforms the expression into a simple join index column. This enables the Optimizer to map the statistics collected on those complex expressions to the base table to facilitate single-table cardinality estimates or to match the predicates (see *SQL Request and Transaction Processing* for details).

Note that the derived statistics framework supports bidirectional inheritance of statistics between a non-sparse single-table join index and the base table it supports (see *SQL Request and Transaction Processing* for details), so the entity on which statistics are collected is no longer as important as it once was.

## Similarities of Join Indexes to User Data Tables

In many respects, hash and join indexes are identical to base user data tables.

For example, you can do the following things with a join index.

- Create a unique or non-unique primary index, either a PPI or an nonpartitioned primary index, on its columns.  
UPIs are supported only for uncompressed single-table join indexes without an ORDER BY clause. See *SQL Data Definition Language* for details.
- Create a column-partitioned, single-table uncompressed, non-aggregate join index. Such a join index can be sparse join index.
- Create non-unique secondary indexes on its columns.
- Perform any of the following statements against it.

- COLLECT STATISTICS (Optimizer Form)
- DROP JOIN INDEX
- DROP STATISTICS (Optimizer Form)
- HELP JOIN INDEX
- SHOW JOIN INDEX
- Specify UDT and BEGIN and END bound functions on Period or derived Period columns in its definition.
- Specify row-level security constraint columns in its definition.

You can do this only if both of the following criteria are true:

- The index references a maximum of one row-level security-protected base table.
- All of the row-level security constraint columns defined in the base table are included in the join index definition.

**Note:** You cannot specify row-level security constraint columns in a partitioning expression for a PPI or column-partitioned join index.

- Specify any valid expressions in the select list and WHERE clause when the expressions reference at least one column.

The following expression types are not valid in a join index definition:

- OLAP expressions
- UDF expressions
- Built-in functions that are explicitly not valid such as DEFAULT and PARTITION.

Note that a join index defined with an expression in its select list provides less coverage than a join index that is defined using a base column (see “[Restrictions on Partial Covering by Join Indexes](#)” on page 575).

Unlike base tables, you *cannot* do the following things with join indexes:

- Create a join index on a join index.
- Query or update join index rows.

For example, if ordCustIdx is a join index, then the following query is not legal:

```
SELECT o_status, o_date, o_comment
FROM ordCustIdx;
```

- Create a USI on its columns.
- Define multi-value or algorithmic compression on its columns.

If multi-value or algorithmic compression are defined on any columns of its parent base table set, however, a join index *does* inherit that compression under most circumstances (see “[Default Column Multivalue Compression for Join Index Columns When the Referenced Base Table Column Is Compressed](#)” on page 501).

## Join Index Applications

Join indexes are useful for queries where the index table contains all the columns referenced by one or more joins, thereby allowing the Optimizer to cover all or part of the query by planning to access the index rather than its underlying base tables. An index that supplies all the

columns requested by a query is said to cover that query and, for obvious reasons, is often referred to as a covering index. Some vendors refer to this as *index-only access*. A join index can be particularly useful for queries that access both nonpartitioned and column-partitioned tables (see “[NoPI Tables, Column-Partitioned Tables, and Column-Partitioned Join Indexes](#)” on page 280). If either an nonpartitioned NoPI table or a column-partitioned table has no secondary indexes, a covering join index is the only way to access its rows without using a full-table scan. Be aware that join indexes can slow the loading of rows into a table using Teradata Parallel Data Load array INSERT operations.

The Optimizer can also use join indexes that only cover a query partially if the index is defined properly (see “[Partial Query Coverage](#)” on page 505). Note that query covering is not restricted to join indexes: other indexes can also cover queries either in whole or in part.

Join indexes are also useful for queries that aggregate columns from tables with large cardinalities. For these applications, join indexes play the role of prejoin and summary tables without denormalizing the logical design of the database and without incurring the update anomalies and ad hoc query performance issues frequently presented by denormalized tables.

You can create join indexes that limit the number of rows in the index to only those that are accessed when a frequently run query references a small, well-known subset of the rows of a large base table. You create this type of join index by specifying a constant expression as the RHS of the WHERE clause, which narrowly filters the rows included in the join index. This is known as a sparse join index.

You can also create join indexes that have a partitioned primary index (you can only define a PPI for a join index if the index is not row-compressed) or that are column-partitioned join indexes. Note that you cannot create an *nonpartitioned* NoPI join index. PPI join indexes are useful for covering range queries (see “[Designing for Range Queries: Guidelines for Choosing Between a PPI and a Value-Ordered NUSI](#)” on page 506), providing excellent performance by means of row partition elimination (see *SQL Request and Transaction Processing*).

See “[Sparse Join Indexes and Tactical Queries](#)” on page 907 for specific design issues related to join index support for tactical queries.

## Partial Query Coverage

Partial query coverage allows join indexes whose columns do not match an entire query to be used to cover a subset of it. For example, one or two tables specified by the query might be covered by the join index, but the entire request is not. In some situations, there might be a number of commonly performed queries that join several tables where each of the queries joins two tables, say  $t_1$  and  $t_2$ , on the same columns. For this situation, you can create a join index to join  $t_1$  and  $t_2$ , and the Optimizer can use that join index for any queries that need to perform that join.

Partial query coverage also allows join indexes that contain only a subset of the columns of a base table referenced in the query to cover the query if that join index can be joined to the base table to retrieve additional referenced columns (this form of partial coverage is also used to implement hash indexes: see “[Hash Indexes](#)” on page 602).

For example, suppose there is a large table that needs to be joined frequently with another table on a column that is not the distributing column of the table. You can define a join index that redistributes the base table by the join column. However, because of the large number of rows and columns that need to be projected into the join index, the extra disk storage required does not allow the creation of such a join index.

You can also define a join index in such a way that its partial coverage of a query can be extended further by joining with a parent base table to pick up any columns requested by the query but not referenced in the join index definition.

Such a join index, sometimes called a global index or global join index, is defined with one of the following elements, which the Optimizer can use to join it with a parent base table to extend its coverage:

- One of the columns in the index definition is the keyword ROWID. You can only specify ROWID in the outermost SELECT of the CREATE JOIN INDEX statement. See “CREATE JOIN INDEX” in *SQL Data Definition Language Detailed Topics*.
- The column set defining the UPI of the underlying base table.
- The column set defining a USI on the underlying base table.

See “[Restrictions on Partial Covering by Join Indexes](#)” on page 575 for an example of a global join index.

## Designing for Range Queries: Guidelines for Choosing Between a PPI and a Value-Ordered NUSI

Row-partitioned primary indexes and value-ordered NUSIs are both designed to optimize the performance of range queries. Because you cannot define both a PPI and a value-ordered primary index on the same join index (nor can you define a PPI for a row-compressed join index), you must determine which is more likely to enhance the performance of a given query workload for those situations that exclude using a join index with a value-ordered primary index.

In general, a value-ordered primary index is the preferred choice if it meets the restriction of a 4-byte maximum column size. Note that this cannot be used for an MLPPI because MLPPIs do not support value-ordered NUSIs by definition.

You might want to consider creating a multilevel partitioning on the base table as an alternative if the usage would be roughly equivalent to a value-ordered NUSI on a join index plus a non-value-ordered NUSI.

In nearly all cases, a join index with a value-ordered primary index is the preferred choice over a value-ordered NUSI.

You should consider the following guidelines when determining whether to design a join index for a particular workload using row partitioning on the join index or using an nonpartitioned primary index and adding a value-ordered NUSI to create a value-ordered access path to the rows:

- It is better to use row partitioning on the join index than to use a value-ordered NUSI and an nonpartitioned primary index on the join index.

- If row compression is defined on the join index, then you cannot define partitioning for it. For this scenario, a value-ordered primary index or value-ordered NUSI are your only join index design choice possibilities for optimizing range queries.
- Each time an underlying base table for a join index is updated, the join index also must be updated.

If there is a value-ordered NUSI defined on a join index, then it, too, must be updated each time the base table (and join index) rows are updated.

You can avoid this additional maintenance when you define the join index with row partitioning. Row partition elimination makes updating a join index row even faster than the equivalent update operation against an nonpartitioned primary index join index. The word *update* is used in a generic sense to include the delete and update operations performed by the DELETE, MERGE, and UPDATE SQL statements, but excluding insert operations performed by the SQL INSERT and MERGE statements, where row partition elimination is not a factor.

Row partitioning can also improve insert operations if the inserted rows are clustered in the data blocks corresponding to the partitions. In this case, the number of data blocks read and written is reduced compared with the nonpartitioned primary index join index case where the inserted rows are scattered among all the data blocks. Because of the way that the rows of a column-partitioned join index are distributed to the AMPs (see “[Row Allocation for Teradata Parallel Data Pump](#)” on page 237), you should not expect to see the positive effects of data block clustering for singleton INSERT requests. However, you should see very positive effects for large INSERT ... SELECT loads into column-partitioned join indexes when the base table is loaded with rows using an INSERT ... SELECT request.

- Row-partitioned join indexes offer the benefit of providing direct access to join index rows, while a value-ordered NUSI does not.

Using a NUSI to access rows is always an indirect operation, touching the NUSI subtable before being able to go back to the join index to access a row set.

Besides offering a direct path for row access, row partitioning provides a means for attaining better join and aggregation strategies on the primary index of the join index. This benefit does not extend to column-partitioned join indexes because they do not have a primary index.

- If you specify the primary index column set in the query, row partitioning and join indexes offer the additional benefit of enhanced direct join index row access using row partition elimination.

The comparative row access times ultimately depend on the selectivity of a particular value-ordered NUSI, the join index row size, and whether a value-ordered NUSI covers a given query or not.

- If a join index partitioning column has more than 65,535 unique values, and the query workload you are designing the index for probes for a specific value, a value-ordered NUSI is likely to be more selective than a join index partitioned on that column.

Note that because NUSI access is indirect, the system might read entire data blocks to retrieve one or a few rows using a NUSI, while row-partitioned index access is direct, with

like rows being clustered together, so row-partitioned join index read operations are usually far more efficient than value-ordered NUSI read operations.

## Collecting Statistics on a Join Index

Issues concerning collecting statistics on the indexes of a join index are documented in “CREATE JOIN INDEX” in *SQL Data Definition Language Detailed Topics*. Also see *SQL Request and Transaction Processing* for information about using single-table join index statistics to make cardinality estimates that cannot otherwise be made with the same level of confidence.

## Fallback With Join Indexes

You can define fallback for join indexes. The criteria for deciding whether to define a join index with fallback are similar to those used for deciding whether to define fallback on base tables.

If you do not define fallback for a join index and an AMP is down, then the following additional criteria become critical:

- The join index cannot be used by the Optimizer to solve any queries that it covers.
- The base tables on which the join index is defined cannot be updated.

## Limits for Hash and Join Indexes

- Teradata tables can have up to 32 secondary, hash, and join indexes.

These 32 indexes can be any combination of secondary, hash, and join indexes, *including* the system-defined secondary indexes used to implement PRIMARY KEY and UNIQUE constraints.

Each multicolumn NUSI that specifies an ORDER BY clause counts as two consecutive indexes in this calculation.

- Hash index columns cannot have XML, BLOB, CLOB, BLOB-based UDT, CLOB-based UDT, XML-based UDT, Period, JSON, ARRAY, VARRAY, or Geospatial data types.

This also means that a join index cannot have a geospatial NUSI.

Join index columns *can* have Period data types and can include expressions composed of system and user-defined functions and methods, including the use of the BEGIN, END, and P\_INTERSECT built-in functions on a Period data type column to compose an expression in the projection list and a single-table condition in its WHERE or ON clauses.

- Both hash and join indexes can be created on a row-level security-protected table only if all of the following criteria are met.
  - The hash or join index references a maximum of one row-level security-protected table.
  - All of the row-level security constraint columns in the indexed table are included in the hash or join index definition.

## Further Information

Consult the following manuals for more detailed information on creating and using join indexes to enhance the performance of your database applications:

- *SQL Data Definition Language*
- *Temporal Table Support*
- *Security Administration*

# Using Join Indexes

You can create a join index to perform any of the following operations:

- Join multiple tables, optionally with aggregation, in a prejoin table.
- Replicate all or a vertical subset of a single base table and distribute its rows by a primary index on a foreign key column to facilitate joins of very large tables by hashing them to the same AMP.
- Aggregate one or more columns of a single table or the join results of multiple tables in a summary table.
- Support querying only those rows that satisfy the conditions specified by its WHERE clause. This is known as a sparse join index.
- If the index has a unique primary index, and a request specifies an equality condition on the columns that define the primary index for the index, then the index can be used for the access path in two-AMP join plans similarly to how USIs are used.

The guidelines for creating a join index are the same as those for defining any regular join query that is frequently executed or whose performance is critical. The only difference is that for a join index the join result is stored as a subtable and automatically maintained by Teradata Database. For the syntax of

## Performance and Join Indexes

Requests that can use join indexes can run many times faster than queries that do not use them. Performance improves whenever the Optimizer can rewrite a request to use a join index instead of the base tables specified by the query.

A join index is most useful when its columns can cover most or all of the requirements in a request. For example, the Optimizer might consider using a covering index instead of performing a merge join.

Covering indexes improve the speed of join queries. The extent of improvement can be dramatic, especially for requests involving complex, large-table, and multiple-table joins. The extent of the improvement depends on how often an index can be used to rewrite a query.

In-place join indexes, where the columns of the covering index and the columns of the table to which it is to be joined both reside on the same AMP, outperform indexes that require row redistribution. An in-place, covering, aggregate join index that replaces 2 or more large tables

in requests with complex joins, aggregations, and redistributions can enable a request to run hundreds of times faster than it would otherwise.

## Partial Covering Multitable Join Indexes

Teradata Database optimizes queries to use a join index on a set of joined tables even if the index does not completely cover the columns referenced in the table if the following things are true.

- The index includes either the Row ID or the columns of a unique index on the table containing a non-covered column referenced by the query.
- The cost of such a plan is less than other competing query plans.

A partial covering multitable join index provides some of the query improvement benefits that covering join indexes offer without replicating all of the table columns required to cover requests in the join index, but an additional overhead from having to access base table rows to retrieve column values occurs when a non-covered column is specified in a request.

## Covering Bind Terms

A bind term is a condition that connects an outer query and a subquery. If the connecting condition of a subquery is IN and the column it is connecting to in the subquery is unique, you can define a join index on the bind term columns. This provides one more type of index for the Optimizer to consider using in place of multiple base tables.

## Using Single-Table Join Indexes

Single-table join indexes are useful in tactical applications because they can support alternative access paths to data. This is a good approach to consider when a tactical query carries a value in an equality condition for a column, such as a customer phone number, that is in the table but is not its primary index. This might be a customer key, for example. A single-table join index can be constructed using the available non-indexed column, the customer phone number, as its primary index, thereby enabling single-AMP access to the data and avoiding more costly all-AMP non-primary index access to the base table.

Single-table join indexes are also valuable when your applications often join the same large tables, but their join columns are such that some row redistribution is required. A single-table join index can be defined to contain the data required from one of the tables, but using a primary index based on the FK of the table, preferably the primary index of the table to which it is to be joined. A single-table join index can also be used as a virtual vertical partitioning of a base table, creating an index subtable that contains frequently accessed columns from a table with many columns that generally are not accessed.

Use of such an index greatly facilitates join processing of large tables, because the single-table index and the table with the matching primary index both hash to the same AMP.

The Optimizer evaluates whether a single-table join index can replace or partially cover its base table even when the base table is referenced in a subquery unless the index is compressed and the join is complex, such as an outer join or correlated subquery join.

## Using Outer Joins to Define Join Indexes

If there is a need for a non-aggregate multitable join index between several large tables, considering using an outer-join to define your join index. This approach offers the following benefits.

- The Optimizer will consider a join index defined using an outer-join for queries that reference only the outer tables of the defining outer join.
- The join index preserves the unmatched rows in the outer join within the join index structure.

## Defining Join Indexes with Inequality Conditions

You can define join indexes using inequality conditions.

To define inequality conditions between 2 columns of the same type, either from the same table or from two different tables, you must AND them with the other join conditions.

This enables the Optimizer to resolve a request using a join index more frequently than would otherwise be possible, avoiding the need to access base data tables to retrieve the required data.

## Defining Join Indexes on UDT Columns and Expressions

You can define join indexes on the following UDT columns and expressions in the select list and in single-table conditions in the WHERE and ON clauses.

- UDT columns, including using a method that is associated with a UDT column
- The following types of expressions.
  - Non-aggregate expressions
  - Non-OLAP expressions
  - Non-UDF expressions

This general support for UDT columns and expressions enables you to specify Period data type columns and BEGIN, END and P\_INTERSECT expressions on a Period data type in the select list, WHERE clause, and ON clause of a join index definition.

## Refreshing Join Indexes

The ALTER TABLE TO CURRENT statement enables you to refresh the content of a join index without having to drop it and recreate it.

The efficiency of the ALTER TABLE TO CURRENT alternative compared with dropping and recreating a join index depends on how often an ALTER TABLE TO CURRENT request is executed and the type of current date condition defined in the join index.

If the join index is refreshed infrequently and the current date condition requires a large volume of old rows to be removed and a large volume of new rows to be inserted, it might be more efficient to drop and recreate the join index.

## Using Aggregate Join Indexes

Aggregate join indexes offer an extremely efficient, cost-effective method of resolving requests that frequently specify the same aggregation operations on the same column or columns. When aggregate join indexes are available, the system does not have to repeat aggregation calculations for every request.

You can define an aggregate join index on two or more tables or on a single table. A single-table aggregate join index includes:

- A subset of the columns in a base table
- Additional columns for the aggregate summaries of the base-table columns

You can create an aggregate join index using the GROUP BY clause and the following built-in aggregate functions.

- SUM
- COUNT
- MAX
- MIN

The following restrictions apply to defining an aggregate join index.

- Only the COUNT, MAX, MIN, and SUM aggregate functions are valid in any combination.  
COUNT DISTINCT and SUM DISTINCT are not valid.
- To avoid overflow, always type the COUNT, MAX, MIN, and SUM columns in an aggregate join index definition as FLOAT.

Teradata Database enforces this restriction as follows.

IF you ...	THEN Teradata Database ...
do not define an explicit data type for a COUNT, MAX, MIN, or SUM column	assigns the FLOAT data type to it automatically.
define a COUNT, MAX, MIN, or SUM column as anything other than FLOAT	returns an error and does not create the aggregate join index.

Many aggregate functions are based on the SUM, MAX, MIN, and COUNT functions, so even though you cannot specify many individual aggregate functions in an aggregate join index, you can combine these 4 functions in a number of ways to create an aggregate join index to resolve requests that use more complicated aggregate functions.

A simple example is using the COUNT and SUM functions to compute an average.

$$\text{AVG}(x) = \frac{\text{SUM}(x)}{\text{COUNT}(x)}$$

## Join Indexes and the Optimizer

For each base table in a query, the Optimizer performs certain processing phases to decide how a database operation that uses a join index is to be processed.

In this phase...	The optimizer...
Qualification	<p>evaluates up to 10 join indexes to choose the one with the lowest cost.</p> <p>Qualification for the best plan includes one or more of the following benefits:</p> <ul style="list-style-type: none"> <li>Smallest size to process</li> <li>Most appropriate distribution</li> <li>Ability to take advantage of covered fields within the join index</li> </ul>
Analysis of results	determines if this plan will result in unique results, analyzing only those tables in the query that are used in the join index.

Subsequent action depends on analysis of the results.

IF the results are...	THEN the Optimizer...
unique	skips the sort-delete steps used to remove duplicates.
nonunique	<p>determines whether eliminating all duplicates can still produce a valid plan, recognizing any case where the following things are true.</p> <ul style="list-style-type: none"> <li>No <i>column_name</i> parenthetical clause exists</li> <li>All logical rows will be accessed</li> </ul>

## System Processing of Join Indexes

The Optimizer does the following when it rewrites requests using a join index.

- Selects cost-based query rewrites using the best available aggregate join index when several possible aggregate join indexes are available.
- Provides a larger number of opportunities to perform cost-based rewrites of requests using aggregate join indexes for queries with subqueries, spooled derived tables, outer joins, COUNT(DISTINCT), and extended grouping sets.

When you create multiple aggregate join indexes, the creation of the current aggregate join index makes use of an existing aggregate join index that is the most efficient for the calculation of the aggregate join index being created so that the CREATE JOIN INDEX request has better performance.

With the existence of multiple join indexes, including aggregate join indexes and non-aggregate join indexes, aggregate queries perform better with the cost-based rewrite and more chances to use an aggregate join index.

- Uses join indexes with Partial GROUP BY optimizations during join planning, making it possible to produce better join plans.

## Join Index Optimizations

The Optimizer uses join indexes in several ways, including the following.

- Selects cost-based query rewrites using the best available aggregate join index when several possible aggregate join indexes are available.
- Provides a larger number of opportunities to perform cost-based rewrites of requests using aggregate join indexes for queries with subqueries, spooled derived tables, outer joins, COUNT(DISTINCT) operations, and extended grouping sets.

When a user creates multiple aggregate join indexes, the creation of the current aggregate join index makes use of an existing aggregate join index that is the most efficient for the calculation of this aggregate join index so that the CREATE JOIN INDEX request will have better performance.

With the existence of multiple join indexes, including aggregate join indexes and non-aggregate join indexes, aggregate requests perform better with the cost-based rewrite and more chances to use an aggregate join index.

- Uses join indexes with Partial GROUP BY optimizations during join planning, making it possible to produce better join plans.

## Protecting a Join Index with Fallback

You can define fallback protection for a simple or aggregate join index.

With fallback, you can access a join index and the base table it references if an AMP fails, with little impact on performance.

Without fallback, an AMP failure has significant impact on both availability and performance as follows.

- You cannot update the base table referenced by a join index even if that base table is defined with fallback.
- The Optimizer cannot access a join index on a down AMP to create query plans. Performance can be degraded significantly when this occurs.

The cost of having fallback for a join index when executing a DML request that modifies a base table referenced by the join index is a slight increase in processing to maintain the fallback copy of the join index.

## Collecting Statistics for Join Indexes

Hash indexes and single-table join indexes that are not defined as sparse join indexes inherit all statistics from their base table, including dynamic AMP samples and collected statistics.

Only sparse join indexes and multitable join indexes require statistics collection. It is particularly important that statistics be collected on the sparse-defining column in the WHERE clause of a sparse join index or the Optimizer might not select the sparse join index for use.

Consider collecting statistics to improve performance during the following operations.

- Creation of a join index

- Update maintenance of a join index

You need to submit separate COLLECT STATISTICS requests for the columns in the join index and the source columns in the base tables. This does not have a very high cost because Teradata Database can collect statistics while queries are accessing the underlying base tables of a join index.

## Costing Considerations for Join Indexes

Join indexes, like secondary indexes, incur both space and maintenance costs. For example, INSERT, UPDATE, and DELETE operations must be performed twice: once for the base table and once for the join index.

### Space Costs for Join Indexes

The following formula estimates the space overhead required for a join index.

$$\text{Join Index Size} = U \times (F + O + (R \times A))$$

where:

Parameter	Description
F	Length of the fixed column <i>join_index_column_1</i>
R	Length of a single repeating column <i>join_index_column_2</i>
A	Average number of repeated fields for a given value in <i>join_index_column_1</i>
U	Number of unique values in the specified <i>join_index_column_1</i>
O	Row overhead (assume 14 bytes)

Updates to the base tables can cause a physical join index row to split into multiple rows. The newly formed rows each have the same fixed field value but contain a different list of repeated field values. This applies specifically when the compressed join index format is being used.

The system, however, does *not* automatically recombine logically related split rows. To re-compact such rows, you must drop and recreate the join index.

### Maintenance Costs for Join Indexes

The use of a join index entails the following.

- Initial time consumed to calculate and create the index
- Whenever a value in a join index column of the base table is updated, the join index must also be updated, including any required aggregation or pre-join effort.

However, if join indexes are suited to your applications, the improvements in request performance can far outweigh the costs.

Join indexes are maintained by generating additional AMP steps in the base table update execution plan. Those join indexes defined with outer joins usually require additional steps to maintain any unmatched rows.

Expect a single-table join index INSERT operation to have similar maintenance overhead as would an insert operation with an equivalent NUSI. UPDATE or DELETE operations, however, might incur greater overhead with a single-table join index, unless a value for the primary index of the join index is available at the time of the update.

Overhead for an in-place aggregate join index can be perhaps 3 times more expensive than maintaining the same table without that index. For an aggregate join index that redistributes rows, the maintenance overhead can be several times as expensive.

Maintenance overhead for multitable join indexes without aggregates can be small or very large, depending on the pre-join effort involved in constructing or changing a join index row. This could be up to 20 times or more expensive than maintaining the table without the index. The overhead is greater at higher hits per block, where *hits* means the number of rows in a block are touched.

Since Teradata Database writes a block only once regardless of the number of rows modified, as the number of hits per block increases:

- The CPU path per transaction decreases (faster for the case with no join index than for the case with a join index)
- Maintenance overhead for aggregate join indexes decreases significantly

If a DELETE or UPDATE request specifies a search condition on the primary index or secondary index of a join index, the join index may be directly searched for the qualifying rows and modified accordingly.

This direct-update approach is employed when the request adheres to these requirements:

- A primary index or secondary index access path to the join index
- If a `join_index_column_2` is defined, little or no modification to the `join_index_column_1` columns
- No modifications to the join condition columns in the join index definition
- No modifications to the primary index columns of the join index

It is not necessary to drop the join index before a backup. It is important, however, to drop join indexes before the underlying tables and databases are restored, should a restore ever be required. Otherwise an error is reported and the restore will not be done.

## Join Indexes Versus NUSIs

A join index offers the same benefits as a standard secondary index in that it, like the standard secondary index, has the following properties.

- Optional
- User defined
- Maintained by the system
- Transparent to end users

- Immediately available to the Optimizer
- If a covering index, considered by the Optimizer for a merge join

However, a join index offers the following performance benefits over a NUSI.

IF a join index is...	THEN performance improves by...
defined using joins on one or more columns from two or more base tables	eliminating the need to perform the join step every time a joining query is processed.
used for direct access in place of some or all of its base tables, if the Optimizer determines that it covers most or all of the query.	eliminating the I/Os and resource usage required to access the base tables.
limited to only certain data types of your choice, such as Date	allowing direct access to the join index rows within the specified value-order range.
a single-table join index with a FK primary index	<p>reducing I/Os and message traffic because row redistribution is not required, since the following are hashed to the same AMP:</p> <ul style="list-style-type: none"> <li>• A single-table join index having a primary index based on the base table foreign key.</li> <li>• The table with the column set making up the foreign key.</li> </ul>
defined with an outer join	<ul style="list-style-type: none"> <li>• Giving the same performance benefits as a single-table join index, for queries that reference only outer tables.</li> <li>• Preserving unmatched rows.</li> </ul>
created using aggregates	eliminating both the aggregate calculations and the join step for every query requiring the join and aggregate.

For more information on the syntax, applications, restrictions, and benefits of join indexes, see *SQL Data Definition Language*.

## Join Index Design Tips

This topic describes various tips that you can use to optimize the functionality of your join indexes.

### When You Should Consider Defining a Join Index

Join indexes are not suited for all applications and situations. The usefulness of a join index, like that of any other index, depends on the type of work it is designed to perform. Always prototype any join index and evaluate its usefulness to the applications it is designed to support before adding it to your production environment. The overhead of updating join index tables can outweigh their benefit in some situations.

The following situations all make a join index a likely performance enhancer:

- Frequent joins of large tables with other large or moderately-sized tables that result in a significant number of the rows from both tables being joined.
- Frequent joins of tables of high degree (having many columns) for which the same relatively small set of columns is repeatedly requested.
- An alternate partitioning sequence for a vertical subset of data in one of the base tables (a so-called *single-table join index*) would remove the necessity of redistributing rows for a frequently made join.
- The overhead in time and storage capacity for the creation and maintenance of a join index does not outweigh its retrieval benefits.
- The performance of frequent range queries requiring joins of large tables with other large or moderately-sized tables that result in a significant number of the rows from both tables being joined.
- A row-partitioned join index can enhance the performance of queries if you specify an equality or range constraint on the partitioning column set. For example, a single-table row-partitioned join index can take advantage of row partition elimination to improve both the performance of a query retrieving rows from itself.

Be aware that you cannot define row partitioning for a row-compressed join index.

- If a frequently run query specifies a complex expression in its predicate, consider creating a single-table join index or a hash index on the table that includes that expression in the select list or column list, respectively, of its definition. Although you cannot collect statistics on complex base table expressions, creating a single-table join using the expression transforms it into a simple column, and you can then collect statistics on that column. The Optimizer can then use those statistics to estimate the single-table cardinality of evaluations of the expression in a query predicate that specifies the expression using a base table column. See *SQL Request and Transaction Processing* for more information.
- Most queries against a column-partitioned table or join index are expected to be very selective on a variable subset of columns, and project a variable subset of the columns where the subset of accessed columns is less than 10% of the column partitions for any particular query.

Sometimes you just need to experiment.

For example, application of a row-partitioned join index might be for queries that involve row-partitioned base tables. However, if the base table is *not* a row-partitioned table, but is designed to handle efficient joins on the primary index, it is also conceivable that a row-partitioned join index might be defined to provide an alternative organization of the data for optimal access based on row partitions. This is only valid if the join index is not row-compressed. Partitioning is not valid for row-compressed join indexes.

See *SQL Data Definition Language* for further information about PPI join indexes.

## Using Outer Joins to Define Simple Join Indexes

Because join indexes generated from inner joins do not preserve unmatched rows, you should consider using outer joins to define simple join indexes, noting the following restrictions.

- Inequality conditions are not supported under any circumstances for ON clauses in join index definitions.
- Outer joins are not supported under any circumstances for aggregate join indexes.

Defining join indexes on outer joins enables them to satisfy queries with fewer join conditions than those used to generate the index. See “[Defining a Simple Join Index on a Binary Join Result](#)” on page 542 and “[Using Outer Joins to Define Join Indexes](#)” on page 559 for a demonstration of this property.

Also see “[Restriction on Coverage by Join Indexes When a Join Index Definition References More Tables Than a Query](#)” on page 577 and related examples.

## Collecting Statistics on the Columns and Indexes for Join Indexes

Keeping fresh statistics on join indexes is just as critical for join indexes as it is for base tables for optimizing the query plans generated by the Optimizer.

This is particularly true if you have created a single-table join index whose definition includes a complex expression in its select list and whose statistics are to be used to make more accurate single-table cardinality estimates when a query that specifies a matching expression in its predicate is run against the table on which the join index is defined. The Optimizer can also use expression mapping when it detects an expression in a query predicate within a non-matching predicate. In this case, the Optimizer maps to the join index column that is defined using the matched expression. See *SQL Request and Transaction Processing* for details.

You *cannot* collect statistics on the PARTITION column of a PPI join index.

## Basing a Join Index on Foreign Key–Primary Key Equality Conditions

To avoid storing redundant data, base join index definitions on `foreign key=primary key` predicates. Add either the primary key or the foreign key to the index definition, but not both, because the Optimizer has the intelligence to derive either from the other.

## Adding Join Constraints That Facilitate Joining to Other Tables

Include join constraint columns that support joining to tables not defined in the join index. It makes no difference whether you assign these columns to the fixed part of the join index or to the repeating part.

For example, you might carry the join columns `c_nationkey` and `l_partkey` in the join index to facilitate joining the `nation` and `parts` tables to the join index.

When the join index is defined using an outer join, use the outer table join column rather than the inner table join column to enhance performance.

## Specifying a Row-Partitioned or a Value-Ordered Sort Key for Range Queries

Sorting a join index by data values, as opposed to row hash values, is especially useful for range queries that involve the sort key. This means that defining a join index designed to

support range queries using row-partitioning can significantly facilitate the performance of range queries.

Similarly, in some circumstances, you might obtain better performance by designing the join index without row partitioning, but using a value-ordered NUSI.

Note the following limitations:

- Value ordering is limited to a single numeric or DATE column with a maximum length of four bytes.
- The column you specify in the ORDER BY clause must be drawn from the set of fixed columns. You cannot order by a column from the set of repeating columns.

In the following example, the join index rows are hash-distributed across the AMPs on the primary index, *c\_name*. Once assigned to an AMP, the join index rows are value-ordered using *c\_custkey* as the sort key.

```
CREATE JOIN INDEX ord_cust_idx AS
    SELECT (o_custkey, c_name), (o_status, o_date, o_comment)
    FROM Orders LEFT JOIN Customer ON o_custkey = c_custkey
    ORDER BY o_custkey
PRIMARY INDEX (c_name);
```

## Join Index Benefits and Costs

Before you create a join index to optimize one or more of your frequently repeated queries, you should consider the following issues carefully:

- Cost of disk resources required to store a join index
- Cost of creating a join index
- Cost of maintaining a join index
- Benefits of reduced performance times for the queries the join index is designed to facilitate.

### Cost

Cost is expressed as follows:

- Disk resources: Cost is expressed in bytes (space or size). Join indexes are space-intensive because they require separate tables, often of fairly high cardinality, for storage.
- Creation and maintenance: cost is expressed in time (elapsed time and CPU path per transaction).

Elapsed time is a fairly simple measure, but it is not comparable across different system configurations, nor is it independent of other workloads running on a system. It has the advantages of conceptual simplicity and ease of measurement. With respect to join index creation, it is a measure of how long it takes to create the index. With respect to maintenance, it is a measure of how long it takes to update a join index table beyond the time required to perform the update of the base table. In this sense, the term *update* refers generically to the

data manipulation operations insert, update, and delete performed by any of the following SQL statements:

- DELETE
- INSERT
- MERGE
- UPDATE

CPU path per transaction is a sophisticated measure that is comparable across different system configurations. It has the disadvantages of being both difficult to measure without the proper tools and difficult to understand conceptually.

Whether creation and maintenance cost is measured as elapsed time or as CPU path per transaction, the same proportional relationship holds: the smaller the number, the better.

The total cost for a join index is the sum of its creation and maintenance costs.

## Join Index Costs Summary

Creation and maintenance costs for join indexes can be a resource burden because of their processing overhead. In the case of join index maintenance, the burden is an ongoing process that lasts for the life of the index.

Costs vary considerably among the various types of joins used as well as between simple and aggregate types. The following bulleted list summarizes the conclusions to be drawn from the performance analyses performed.

- Maintenance overhead for in-place aggregate join indexes ranges from 1.0 to 2.9.
- Maintenance overhead for aggregate join indexes that redistribute rows ranges from 2.6 to 9.5.
- Maintenance overhead for simple join indexes ranges from 1.0 to 23.1, increasing as the number of hits per data block increases.
- Maintenance overhead for aggregate join indexes decreases as the number of hits per data block increases because of the efficiencies of block-at-a-time operations in the file system.
- Maintenance overhead for insert operations is significantly less than the maintenance overhead required for update and delete operations.

## Benefit and Benefit Percentage

Benefit is a measure of the difference in elapsed time for the same query performed with and without a join index.

Benefit percentage is a normalized expression of benefit. It is defined as the product of benefit and 100 divided by the elapsed query time without a join index.

The *larger* the number, the better.

## Payback

Payback is an expression of the number of queries that must be run to achieve a break even point for the total cost of the join index. Its value is derived from dividing total costs by benefit.

The *smaller* the number, the better.

## Cost of Disk Resources

A join index table has the same properties as a base table except that it cannot be queried directly.

Because it is essentially a base table, the cardinality of a simple join index is generally of the same order as its component join table with the highest cardinality, with adjustments being necessary for row compression and other miscellaneous issues. The cardinality of an aggregate join index is typically much smaller than that of any of its component join tables.

The degree of either type of join index table is entirely dependent on how it is defined.

Irrespective of any other factors, join index tables extract a cost in terms of increased disk space requirements. If a join index table is fallback-protected, then it exerts twice again the burden in disk space.

The disk resources required for a join index are described in “[Join Index Storage](#)” on page 598.

## Cost of Join Index Creation

The cost of creating a join index is described in terms of time: how long does it take to create the index? Equally important in many situations is the question of how many times a query for which a join index was designed must be performed to offset the creation time by recovering response time.

Different types of join index require a longer or shorter time to create.

If your processing environment requires that join indexes be created and dropped on a regular basis, the cost of creating the index becomes a critical factor that must be considered carefully in determining its worth.

Because of this creation overhead, you should evaluate the benefit of a join index created to enhance your standard queries vis-a-vis its creation cost if you must drop and create your join indexes frequently. If the gross effect of an index is negative with respect to overall system performance, you might not want to create it even if it enhances the performance of particular queries.

Unfortunately, it is not possible to predict join index creation times with any degree of accuracy because there are too many significant variables involved to be able to produce a useful predictive model. This multivariate complexity also prevents the derivation of a useful predictive model for maintenance overhead or even for query time reductions.

## Example

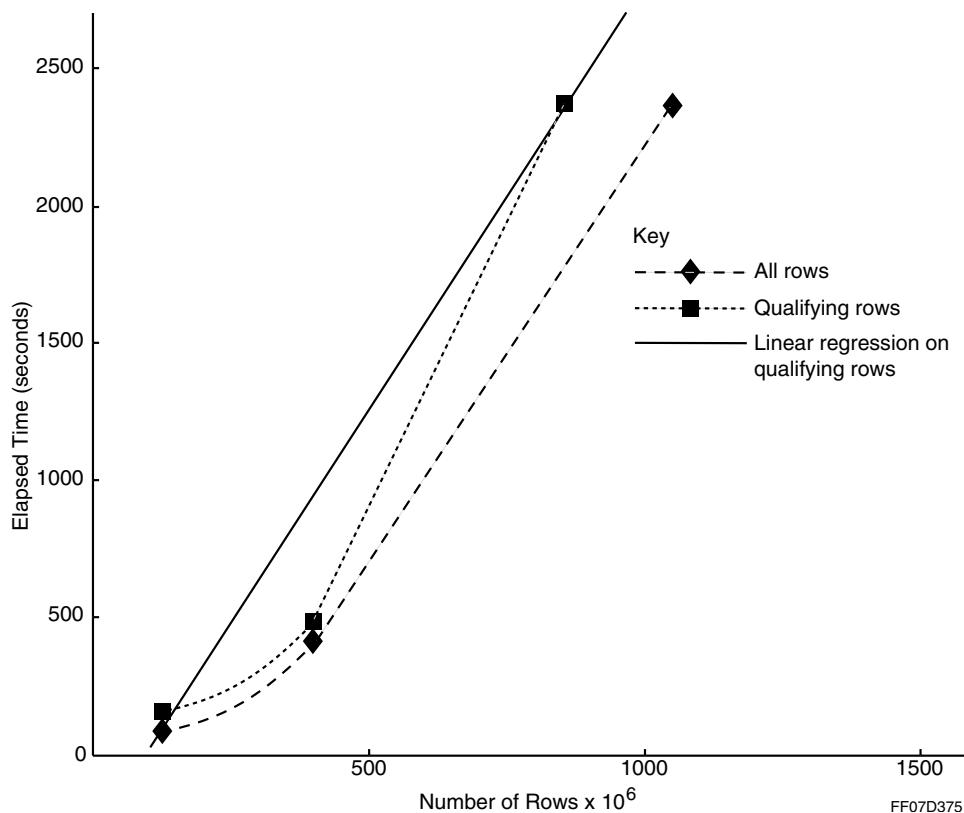
Consider the following example. Empirical testing indicates that one of the most simple relationships is that between the time required to create an in-place aggregate join index and

the number of rows in the base tables. Even this simple relationship is complicated by the fact that the resulting creation time is likely to be more closely related to the number of rows that qualify for the join rather than the total number of rows.

The following table indicates a measured relationship between the elapsed creation time for a join index and the number of qualified rows.

Elapsed Time (seconds)	Number of Qualified Rows $\times 10^6$
133	90
415	300
2,344	900

These numbers, along with the total number of rows in each test and a least squares linear regression of number of qualifying rows on elapsed time, are graphed in the following figure:



The equation for this regression is as follows:

$$y = 28,368x - 255.81$$

This translates into physical terms as follows:

where the value for Create time is expressed in elapsed seconds and the order of magnitude for Number of qualified rows is expressed in millions of rows.

$$\text{Create time} = 2.8 \times \left( \left( \sum \text{Number of qualified rows} \right) - 256 \right)$$

The computed coefficient of determination for this regression,  $R^2$ , is 0.9819. This means that 98% of the variance in the sample data is accounted for by this equation, for which the correlation coefficient is 0.991. In other words, the line is nearly a perfect fit to the data.

A sanity check of the data reveals that this regression might not always be as accurate as it seems to be upon initial scrutiny. For example, consider the 300 million row case. The measured elapsed time value is 415 seconds, but the predicted value is 595.23 seconds, a difference of just over 180 seconds, which is an error of 43%.

There are several possible explanations for this discrepancy, but the message to be taken away from this example is that while the model performs well in predicting the general direction of how much time it should take to create a join index with a given number of qualifying rows, the absolute accuracy of the prediction might vary considerably at some points on the curve.

### **Creation and Elapsed Query Times for Different Join Indexes**

The following tables list the creation and elapsed query times for several different kinds of join indexes.

Note that when the definition of a join index can be covered by an existing join index, the existing index can be used to create the join index.

In the following table, when calculating the values for the Query Ratio and Benefit columns, query times of less than 1 second were rounded up to 1.0 seconds.

Approximate Creation Costs and Query Benefits in Terms of Elapsed Query Time						
Join Index Type	Creation Cost (seconds)	Elapsed Query Time (seconds)		Query Ratio (seconds)	Benefit (seconds)	Benefit Percent
		Without Join Index	With Join Index			
Single-table aggregate	51	41	< 1	41.00	40	98
2-table in-place aggregate	135	107	< 1	107.00	106	99
4-table in-place aggregate	279	243	< 1	243.00	242	100
Single-table simple	199	80	63	1.27	17	21
2-table in-place	190	116	78	1.49	38	33
2-table in-place outer join	231	147	98	1.50	49	33
2-table foreign aggregate	232	230	< 1	230.00	229	100

Approximate Creation Costs and Query Benefits in Terms of Elapsed Query Time						
Join Index Type	Creation Cost (seconds)	Elapsed Query Time (seconds)		Query Ratio (seconds)	Benefit (seconds)	Benefit Percent
		Without Join Index	With Join Index			
2-table ad hoc aggregate	397	357	< 1	357.00	356	100

In the following table, the figure <<1 means the CPU path per row time was too short to measure. When calculating the values for the Benefit column, query times of less than 1 second were rounded up to 1.0 seconds. The Benefit value is based on elapsed query time rather than CPU path per row time.

Creation Costs and Query Benefits in Terms of CPU Path Per Row					
Join Index Type	Creation Cost (µseconds)	Query Time CPU Path Per Row (µseconds)		Benefit	Benefit Percentage
		Without Join Index	With Join Index		
Single-table aggregate	9	7	<< 1	40	98
2-table in-place aggregate	7	6	<< 1	106	99
4-table in-place aggregate	10	8	<< 1	242	100
Single-table simple	33	14	11	17	21
2-table in-place	14	10	7	38	33
2-table in-place outer join	16	11	7	49	33
2-table foreign aggregate	19	19	<< 1	229	100
2-table ad hoc aggregate	34	30	<< 1	356	100

You can use these figures, which are derived from tests on a two node system, to scale an approximately linear adjustment for your own table sizes and configuration.

## Cost of Join Index Maintenance

The largest resource burden for any join index is incurred by its maintenance: inserting, updating, and deleting rows in the join index table. Similar to creation costs, various types of join indexes incur very different maintenance costs.

The time resources required to maintain a join index are described in “[Maintenance Costs of Join Indexes](#)” on page 526.

## Maintenance Costs of Join Indexes

Join indexes can be expensive to maintain. For many customers, the most important factor in the decision to use a join index is likely to be how much it costs to maintain.

Each time a join-indexed base table column is updated, the corresponding join index table column must also be updated. Each time a new row is added to or an existing row is deleted from a join-indexed base table, the corresponding join index table rows must also be inserted or deleted.

Because of this maintenance overhead, you should always carefully evaluate the benefit of a join index created to enhance your standard queries vis-a-vis its cost to create and maintain (see “[Cost/Benefit Analysis for Join Indexes](#)” on page 534).

## Maintenance Cost Optimizations Based on Foreign Key-Primary Key Joins

You should consider the added cost of join index maintenance carefully when you are designing the indexes for your data warehouse to ensure that the minimum number of join indexes can be called upon by the Optimizer to cover the maximum number of queries. Designing with foreign key-primary key joins (see “[Restriction on Coverage by Join Indexes When a Join Index Definition References More Tables Than a Query](#)” on page 577) allows you make these optimizations.

Whenever a base table column set that is shared with a join index is updated or deleted, or when a new row is inserted into the base table, the system generates extra steps to maintain the base table and join index concurrently. If the base table is specified as part of an outer join in the join index definition, the steps can be more complex because maintenance might be needed for both matched and unmatched row sets.

However, when the join columns have a foreign key-primary key relationship, the system treats inner and outer joins alike (see “[Restriction on Coverage by Join Indexes When a Join Index Definition References More Tables Than a Query](#)” on page 577).

## Maintenance Cost Optimizations for DELETE ALL Operations

A *fastpath* optimization is one that can be performed faster if certain conditions are met. For example, in some circumstances DELETE and INSERT operations can be performed faster if they can avoid reading the data blocks and avoid transient journaling.

Teradata Database uses both fastpath and deferred fastpath row partition DELETE operations for the following cases:

- Deferred row partition deletion on a row-partitioned base table when a join index defined on the base table is not row-partitioned
- Deferred partition deletion on a row-partitioned join index that is defined on a table
- Deferred partition deletion on both a row-partitioned join index and its row-partitioned base table

In this case, Teradata Database performs the fastpath row partition deletion operations on the join index and the base table independently.

Teradata Database can perform fastpath DELETE ALL operations, but not deferred row partition deletion operations, for the following cases:

- If the deleted table has a conditional DELETE with predicates and it covers the entire join index, the join index is eligible for a fast path DELETE.
- All single-table join indexes.
- A multitable join index when the join between the tables is either an inner join or the table being deleted is the outer table in the join.
- An implicit transaction with a single-statement DELETE ALL *table\_name* when the table has a join index defined on it.
- An implicit transaction with a multistatement request.
- An ANSI/ISO session mode transaction with a multistatement request.
- A Teradata session mode transaction with a multistatement request.

### **Types of Join Index Examined for This Analysis**

Different types of join indexes incur different costs of maintenance. For the analyses provided here, several different types of join index were used. The types are far from exhaustive, but they provide a fairly representative range of data that you can use to extrapolate roughly how much benefit a join index that uses a particular type of join is likely to provide.

The following table provides a list of the various types of joins defined for the join indexes used in this study. When you think of types of joins, you probably think of join processing types like merge join, nested join, product join, and so on.

The types of joins defined here (in-place, foreign, and ad hoc) are orthogonal to those join processing types. The types of joins and the types of join indexes go hand in hand. In both cases, it is the number of tables that are being redistributed that determines the type of join or join index being described.

Each join type can be used with either a simple or an aggregate join index.

Join Type	Definition
In-place	<p>The joined tables have a common primary index and are joined on that column set.</p> <p>A common example is a logical entity-subentity relationship as seen in an Employee - Employee_Phone relationship, where both tables have Employee Number as a common primary index.</p> <p>As a result, rows to be joined are always on a common AMP and do not have to be redistributed.</p> <p>This is the least expensive join of the three types examined.</p>

Join Type	Definition
Foreign	<p>The tables are joined on a primary key - foreign key column set relationship, where the primary key column set is also the primary index column set. The primary index of the other join table is typically a foreign key in the first table.</p> <p>A common example is a join between an Employee table and a Department table, where the join is on the common Employee Number column, which is the primary index for the Employee table, but is only a foreign key in the Department table.</p> <p>As a result, the rows to be joined must be redistributed from the AMP having the Department table rows to the AMP having the Employee table rows.</p> <p>This join is more expensive than the homogeneous primary index join, but less expensive than the ad hoc join.</p>
Ad hoc	<p>The tables are joined on a column set that is not a primary index in either table.</p> <p>As a result, rows from both tables must be redistributed to make the join.</p> <p>This join is the most expensive of the three types examined.</p>

## Maintenance Cost as a Function of Number of Hits Per Data Block

The number of hits per block is a measure of how many rows in a data block are accessed (inserted, deleted, or updated) during an operation on a table. Generally speaking, the greater the number of hits per block, the better the performance provided the hits can be combined into one update of the data block, as would be the case, for example, with an `INSERT ... SELECT` involving a set of updates containing multiple rows.

If a large number of the data blocks for a table have become significantly smaller than half the maximum size for the defined maximum data block size, an appropriate specification for the `MERGEBLOCKRATIO` option of the `ALTER TABLE` and `CREATE TABLE` statements can enable the file system to merge up to 8 data blocks into a single larger data block. See *SQL Data Definition Language* for details.

On the other hand, if each hit of a data block involves a separate read operation, with hits occurring semi-randomly, performance is worse.

## Data Processing Scenarios for the Hit Rates Studied

The following table provides example data processing scenarios that correspond with the hit rates studied.

Number of Hits per Data Block	Percentage of Rows Touched	Scenario
1	0.2	Update a table that contains 500 days of data (roughly 1.5 years) with one typical day of data.
5	1.0	Insert into a table that contains 104 weeks of data (2 years) one typical week of data.
20	4.0	Delete from a table that contains 104 weeks of data (2 years) one typical month (4 weeks) of data.

## Standard Test Procedure

The standard procedure for these tests was to make any maintenance changes to the left table in the join.

## Maintenance Costs In Terms of Elapsed Time

Elapsed times increase as a function of increased hits per data block because more rows are touched. At the same time, the CPU path per row times *decrease* because the elapsed times increase at a lesser rate than the increase in the number of rows touched.

## Maintenance Costs In Terms of CPU Path Length Per Transaction

Suppose you report the same information in terms of CPU path length per transaction. For this data, the term *transaction* equates to *qualifying row*. This number is a measure of the amount of CPU time a transaction requires; roughly, the number of instructions performed per transaction.

CPU path per transaction is a better way to compare various manipulations than elapsed time for two reasons:

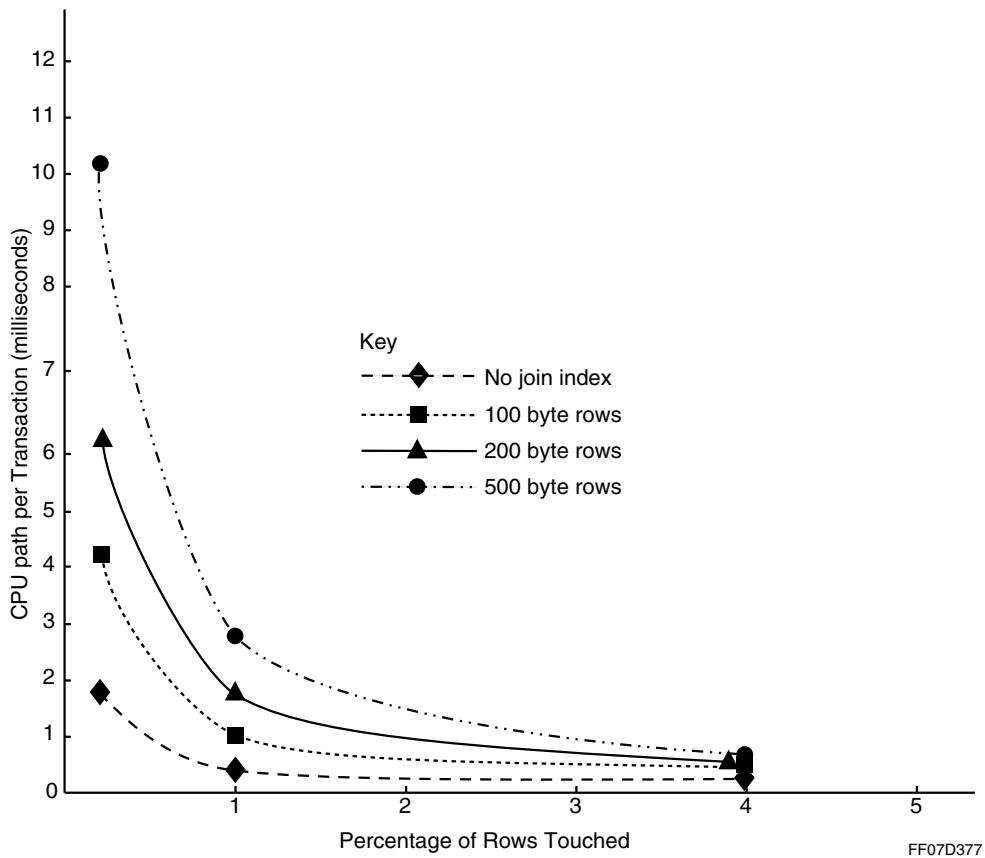
- CPU path per transaction figures include both CPU utilization information and elapsed time.
- Because the measure is normalized, it factors out configuration-specific variables such as number of CPUs per node and number of nodes, making the numbers comparable across all configurations.

## Maintenance Cost as a Function of Row Size

The size of join index rows is another important factor to consider when evaluating the maintenance overhead of a join index. Given a constant disk space, as the size of the rows increases, the number of rows decreases, while the number of data blocks remains the same. The result is that the number of hits per data block decreases.

## Maintenance Costs in Terms of CPU Path Per Transaction

CPU path per transaction times for updates increase steadily as a function of increased row size, while the number of transactions, or qualified rows, decreases drastically. This can be seen in the following graph, where CPU path per transaction is presented as a function of percentage of rows touched during an update operation.



### Maintenance Costs in Terms of Other Measures

The following table indicates the trends in performance cost of join index maintenance as a function of increasing row size in terms of various performance measures:

Performance Measure	Effect as a Function of Increasing Row Size
Elapsed time to insert, delete, or update join index rows.	Decreases
I/Os per insert, delete, or update transaction	Increases
Number of point-to-point BYNET insert, delete, or update transactions	Increases

### Summary

Summarizing these results, each join index maintenance transaction performs more work as a function of increased row size as measured by the following list of variables.

- CPU path per transaction
- I/Os per transaction
- BYNET transactions per transaction

## Join Index Maintenance Cost as a Function of Insert Method

For myriad reasons, some methods of inserting rows into tables are much more efficient than others. Four different methods of insertion are examined here for their efficiency in lowering the maintenance cost of join indexes. The following methods are reported.

- Case 1: FastLoad and INSERT ... SELECT the rows
- Case 2: Drop the join index, insert the rows into the base table, recreate the join index
- Case 3: Teradata Parallel Data Pump the rows into the base table
- Case 4: Insert the rows into the base table using SQL

### Case 1: INSERT ... SELECT

This method uses the following procedure.

- 1 FastLoad the rows into an empty table.
- 2 INSERT ... SELECT the rows from the freshly loaded table into the table that has a join index defined.

This procedure uses block-at-a-time optimization and is the fastest of the methods examined.

### Case 2: Dropping a Join Index and Recreating It After Inserting Rows Into Its Base Table

This method uses the following procedure:

- 1 Drop the join index.
- 2 Insert the rows into the base table using any method.
- 3 Recreate the join index.

The elapsed time for this method averages to about 1.3 times the elapsed time measured for the method provided in Case 1.

### Case 3: Teradata Parallel Data Pump

This method uses the Teradata Parallel Data Pump utility to insert rows into the base table with a join index. The Teradata Parallel Data Pump utility has several procedural advantages that make it an attractive option, including restartability and resource throttleability.

Because the Teradata Parallel Data Pump utility performs row-at-a-time operations, however, its performance for this application is not optimal. Measurements confirm that the Teradata Parallel Data Pump utility, when performed using a non-optimized run, is more than 200 times slower than the method of Case 1 with respect to elapsed time measurements.

Optimization of the run using techniques like sorting input data in different orders can reduce this figure by an order of magnitude in many circumstances, bringing the cost differential down to a more reasonable 20:1 ratio for a 1 hit per data block situation. As the number of hits per data block increases, the cost differential increases because methods that use block-at-a-time methods become increasingly efficient.

### Case 4: Single Row INSERT Request

This method uses the SQL INSERT statement to insert the rows into the base table. The data presented in “[Maintenance Cost as a Function of Number of Hits Per Data Block](#)” on page 528 confirms that this method is slightly better than the method presented in Case 2.

### Comparative Elapsed Times to Insert

The following table indicates the comparative elapsed times required to insert the same number of 100 byte rows into a base table and join index table.

Insertion Method	Operation Performed	Elapsed Time (seconds)
“Case 1: <a href="#">INSERT ... SELECT</a> ”	FastLoad into empty table	26
	INSERT ... SELECT into base table with join index	99
	Total	125
“Case 2: <a href="#">Dropping a Join Index and Recreating It After Inserting Rows Into Its Base Table</a> ”	Drop join index	0
	INSERT ... SELECT into base table without join index	59
	Recreate join index	104
	Total	163
“Case 3: <a href="#">Teradata Parallel Data Pump</a> ”	Use the Teradata Parallel Data Pump utility to load rows into a base table with a join index	25,896

### Summary Evaluation

The following table presents a summary evaluation of the insert methods examined in this topic.

Insertion Method	Evaluation
“Case 1: <a href="#">INSERT ... SELECT</a> ”	Greatest speed
“Case 2: Dropping a Join Index and Recreating It After Inserting Rows Into Its Base Table”	Least advantageous method
“Case 3: Teradata Parallel Data Pump”	<ul style="list-style-type: none"><li>Relatively slow</li><li>Offers other advantages</li></ul>
“Case 4: Single Row INSERT Request”	<ul style="list-style-type: none"><li>Simple to perform</li><li>Greatest efficiency for a small number of rows</li></ul>

## Generalizations Derived From These Tests

The following generalizations are made from these test results:

- Maintenance costs for aggregate join indexes are much lower than maintenance costs for comparable simple join indexes.
- Maintenance costs for no join index are lower than maintenance costs for foreign and ad hoc join indexes.

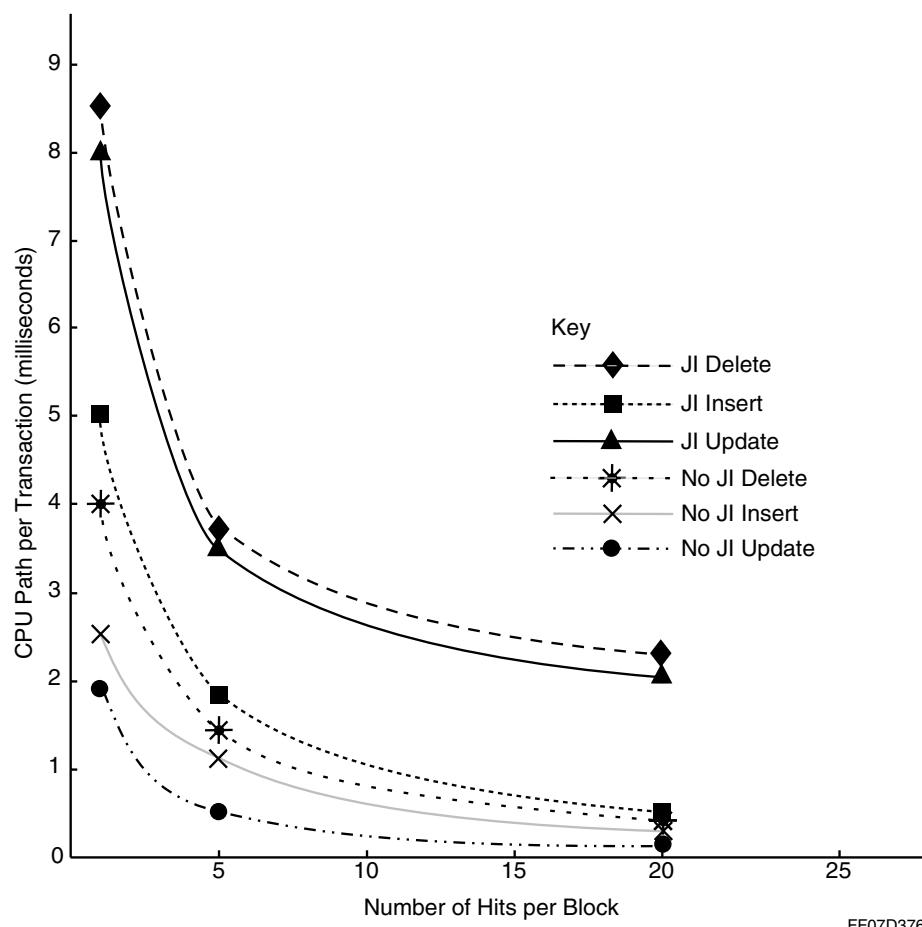
For example, maintenance costs for a 2-table in-place aggregate join index are 1.1 to 2.5 times greater than maintenance costs computed for just the base table without a join index.

- Maintenance costs vary with the type of join index defined.

For example, with an in-place aggregate join index, the higher the number of hits per data block, the less the overhead incurred. This effect is marginal for inserts, for which maintenance costs are already minimal.

On the other hand, for an in-place simple join index at 1 hit per data block, deletes cost 3.4 times more and updates cost 4.5 times more than the case where no join index is defined.

In contrast to the in-place aggregate join index, maintenance costs *worsen* as the number of hits per data block increase, as indicated by the following graph of CPU path per transaction as a function of number of hits per data block.



In the case of an in-place simple join index, inserts cost less than 1.4 times the maintenance required when no join index is defined.

In all cases, inserts are never more than four times more expensive than the maintenance cost when no join index is defined.

## Cost/Benefit Analysis of Join Indexes

Any query performs faster when it uses a join index than it does without a join index. The issue that must be examined is whether or not the benefit of the decreased response times provided by a join index offset the costs of its creation, maintenance, and storage.

The relative costs and benefits of defining a join index are described in “[Cost/Benefit Analysis for Join Indexes](#)” on page 534.

# Cost/Benefit Analysis for Join Indexes

This topic introduces the concepts of cost/benefit analyses for various types of join indexes. Several different computations are described to support these analyses including the following metrics.

- Cost
- Benefit
- Benefit percentage
- Payback
- Query ratio

## Join Index Benefits Summary

The benefits of join indexes vary considerably among the various types of joins used as well as between simple and aggregate types. The following bulleted list summarizes the conclusions to be drawn from the performance analyses performed.

- All queries that require join processing benefit from join indexes; often dramatically, and sometimes spectacularly.
- In all tests performed, aggregate join indexes strongly outperform simple join indexes, both with respect to their performance and with respect to their maintenance burden. Queries using an aggregate join index frequently run hundreds of times faster than the same queries against the same tables when no join index is defined.
- The benefits of simple join indexes are typically more modest. Queries using a simple join index typically run from 1.3 to 4.9 times faster than the same queries against the same tables when no join index is defined.
- In-place join indexes handily outperform any join indexes that redistribute rows.

## Computing the Benefits of Join Indexes

This topic describes a measure you can use to calculate the benefit of a join index.

Begin the calculation by computing some preliminary measures, as described in the following set of equations.

Define the benefit of a join index as follows:

$$\text{Benefit} = \text{ET}_{wo} - \text{ET}_w$$

where:

Syntax element ...	Specifies elapsed query time ...
$\text{ET}_{wo}$	without a join index defined for the query.
$\text{ET}_w$	with a join index defined for the query.

Benefit is a simple measure that measures the advantage of a query in terms of the difference between its elapsed completion time when a join index is not defined and its elapsed completion time when a join index *is* defined.

Define the benefit percentage as follows:

$$\text{Benefit Percentage} = (\text{ET}_{wo} - \text{ET}_w) \times \frac{100}{\text{ET}_{wo}}$$

The benefit percentage for a query is just a normalized form of the raw benefit. It provides an easily understood measure of the reduction in processing time gained by creating the join index.

Rearrange terms to determine the elapsed time to process a query with the join index defined. This measure is necessary to determine if the cost of creating the join index exceeds its usefulness in reducing response time.

$$\text{ET}_w = \text{ET}_{wo} - \text{Benefit Percentage} \times \frac{\text{ET}_{wo}}{100}$$

## Computing the Query Ratio

Define the query ratio as follows:

$$\text{Query Ratio} = \frac{\text{Elapsed query time without join index}}{\text{Elapsed query time with join index}}$$

The query ratio is a normalized measure of how much faster a query executes with a join index than without a join index. For example, consulting the table about “[Creation and Elapsed Query Times for Different Join Indexes](#)” on page 524, you find a column of calculated query ratios.

You interpret these ratios as follows. Consider the query ratio for the creation cost of a single-table aggregate join index. The reported query value for this join index is 41. This means that the test query runs 41 times faster with the single-table aggregate join index defined than it does without it.

## Computing the Payback Factor

This topic explains how to calculate the payback factor for a join index. Note that the term Costs is something you must measure. It is the time required to compute the join index being investigated.

$$\text{Payback Factor} = \frac{\text{Costs}}{\text{Benefit}}$$

where:

Syntax element ...	Specifies ...
Costs	the processing time required to create the join index.

Because the term *payback factor* represents a number of queries, it is always rounded up to represent a whole number. The smaller the payback factor, the sooner benefits accrue from using the join index.

## Example of Computing a Payback Factor for a Simple Join Index

Suppose you have a frequently performed, non-aggregate join query that you think might benefit from a join index. You decide to test the join index and collect the following data.

Parameter	Measured Value (seconds)
ET <sub>wo</sub>	416
ET <sub>w</sub>	226
Cost	1055

What is the payback factor for creating this join index?

$$\text{Benefit} = 416 - 226 = 190 \text{ seconds}$$

$$\text{Benefit Percentage} = 190 \times \frac{100}{416} = 46\%$$

The interpretation of the benefit percentage is that the time to complete the query is cut nearly in half.

$$\text{Payback Factor} = \frac{1055}{190} = 6$$

The interpretation of the payback factor is that the cost of creating the join index is recovered if the query is run six or more times.

## Example of Computing a Payback Factor for an Aggregate Join Index

In the scenario supporting this example, a 30 day period is used for the analysis. You do not regularly create and drop join indexes in your production environment, but you want to calculate the cost of creating the index.

The query to be supported by the join index joins two tables and uses aggregation. The join is made on the primary index of both tables. This is the definition of an in-place aggregate join index.

The 2-node production system used to generate these numbers had the following general configuration.

- Two 550 MHz 4-processor nodes with 1 MB cache and 2 GB memory per node
- RAID1 (mirrored) disk arrays with a segment size of 968
- BYNET 2
- 20 AMPs, each with 2 LUNs (4 physical disks)
- 4 network-attached PEs

The following table indicates the demographics of your data.

Data Characteristic	Table A	Table B
Row length (bytes)	100	
Cardinality	450 million rows	600 million rows
Number of rows for each value in the aggregating column	10,000	
Number of rows updated weekly	900,000	Not applicable
Data block size	64 KB	

The join indexes were created on base tables, A and B with the following characteristics.

- Row size of 100 bytes (except where row size is manipulated as a variable)
- Initial block size of 65,024 bytes
- Base and join index table cardinalities

Type of Join Index	Cardinality of Join Index Table (Rows)	Cardinality of Base Table 100A (Rows)	Cardinality of Base Table 100B (Rows)
Simple	$45 \times 10^6$	$45 \times 10^6$	$60 \times 10^6$
Aggregate	$45 \times 10^2$		

The following table lists the tunable parameter settings used on the test system throughout the test cycle.

Parameter	Setting	Comment
ReadAhead flag setting	TRUE	Disk I/O
DBS Don't Cache flag setting	TRUE	Use DBC Cache Threshold setting
DBC Cache Threshold	10	Default setting
FSG Cache Percent	80	Default setting
RSS logging settings	<ul style="list-style-type: none"> <li>• SPMA</li> <li>• IPMA</li> <li>• SVPR</li> </ul>	15 second logging interval
Datablock size	64 KB	
Free Space Percent	0	
Mini Cyl Pack	10 cylinders	Low Cylinder

How many times must the query be performed to recover the cost in the first month? Here is the procedure.

1 Estimate the benefit.

The benefit should be approximately one order of magnitude greater than the figure generated by this study because there are that many more rows in the base tables, while all other factors are identical.

Examine the table in the topic “[Creation and Elapsed Query Times for Different Join Indexes](#)” on page 524. The measured benefit is 106 seconds. Multiply this by 10 to produce the estimated benefit for your configuration, which is 1060 seconds.

2 Estimate the cost of creation.

Examine the table in the topic “[Creation and Elapsed Query Times for Different Join Indexes](#)” on page 524. The cost of creation for an in-place aggregate join index is seen to be 135 seconds. Scale this value up by an order of magnitude for the same reason as Step 1.

The estimated cost of creation is 1350 seconds.

3 Estimate the approximate number of hits per block.

You update 900,000,100-byte rows in a 450 million row table weekly.

Calculate the number of blocks these 450 million 100-byte rows occupy. The equation is as follows.

$$\text{Number of blocks} = \frac{450\,000\,000 \times 100}{65\,024} \approx 700\,000$$

Determine the approximate hit rate. With approximately 700,000 blocks and 900,000 rows to be updated weekly, the hit rate is determined as follows.

$$\text{Number of hits per block} = \frac{900\,000}{700\,000} \approx 1$$

4 Determine the update cost.

In a 30 day period, the table will be updated four times.

Suppose that the cost of this update is 151 seconds for the test system.

Scale this number up by an order of magnitude and multiply it by four weeks to obtain the approximate update cost.

$$\text{Update cost} = 4 \times 1510 = 6040 \text{ seconds}$$

- 5 Calculate the payback factor.

$$\text{Payback} = \frac{1350 + 6040}{1060} = 7 \text{ queries}$$

If the query for which this join index was designed is run twice weekly, it recovers the time spent creating and maintaining it.

## Two More Brief Examples

The following examples work through the following join indexes.

- In-place simple join index
- Ad hoc aggregate join index

Follow the identical procedure to that used in the detailed example.

For the in-place simple join index, the calculation is as follows.

$$\text{Payback} = \frac{1900 + (2064 \times 4)}{380} = 27 \text{ queries}$$

To recover its cost in a 30 day period, this query must be run slightly more than once per business day.

For the ad hoc aggregate join index, the calculation is as follows.

$$\text{Payback} = \frac{3970 + (2510 \times 4)}{3560} = 4 \text{ queries}$$

To recover its cost in a 30 day period, this query must be run once per week.

## Join Index Types

You can define several types of join index, each having its unique role in enhancing the performance of your database queries without denormalizing the base tables that support your ongoing ad hoc data warehouse activities. Because join index tables cannot be accessed directly by users and are not part of the logical design for a database, they can be used to create persistent prejoin and summary tables without removing or otherwise lessening the ability of your databases to support a wide range of ad hoc queries.

You can also create views whose SELECT definitions are identical to those of a join index, then whenever a user accesses data using that view, the similarly-defined join index should cover the request, and thereby be selected by the Optimizer for the query plan it develops (unless it discovers and chooses a plan that is less costly).

Note that any join index type, except for the column-partitioned variety, can be defined with a row-partitioned primary index as long as it is not also row-compressed. A column-partitioned join index must be defined without a primary index, but may also have row partitioning.

Following is a summary of the join index types and their common uses.

Join Index Type	Description
Simple	A join index table defined without aggregation.
Single table	<p>Several potential applications exist.</p> <ul style="list-style-type: none"><li>• A column subset of a very large base table defined with a NUPI defined on a join key that causes its rows to be hashed to the same AMP as another very large table with a primary index defined on that same join key and to which it is frequently joined. Useful for resolving joins on large tables without having to redistribute the joined rows across the AMPS. Subentities are typically small tables, while major entity tables are typically quite large.</li><li>• The index can emulate vertical partitioning of the base table by selecting a small subset of the most frequently accessed columns of a very wide table. Several possibilities exist for the vertical partitioning solution:<ul style="list-style-type: none"><li>• The base table and the join index can have the same primary index.</li><li>• The base table and the join index can have different primary indexes.</li></ul>This option is particularly useful when frequent requests against the base table specify predicates on non-primary index columns.</li><li>• The index can be defined to contain all the columns defined for its base table, but with a different primary index.</li><li>• Another variable to consider is alternate orderings of the index:<ul style="list-style-type: none"><li>• Hash ordering.</li><li>• Value ordering.</li></ul></li><li>• A final valid, though pointless, option is to define the single-table join index over all the columns of its base table and also to define its primary index on the same column set as the base table. All this does is create a mirror image of the base table that not only will never be used by the Optimizer for its query plans, but which also adds a great deal of useless maintenance overhead to the system.</li></ul> <p>See “<a href="#">Single-Table Join Indexes</a>” on page 546 for additional information.</p>
Multitable	A column subset of two or more major tables that are frequently joined defined as a prejoin of those tables. The defined prejoin, or join index, permits the Optimizer to select it to cover frequently made join queries rather than specifying that its underlying base tables be searched and joined dynamically.

Join Index Type	Description
Aggregate	A join index table defined with aggregation on one or more of its columns.
Single-table aggregate	<p>A column subset of a base table defined with additional columns that are aggregate summaries of base table columns.</p> <ul style="list-style-type: none"> <li>If the summary table is to be used simply to maintain aggregates for an overlying base table without denormalizing the database, then its NUPI need not be defined on a join key column set.</li> <li>If the summary table is to be used to maintain aggregates for an overlying base table and to be joined frequently with another very large table, then its NUPI should be defined on a join key column set that hashes its rows to the same AMPs as the base table primary index.</li> </ul>
Multitable aggregate	<p>A column subset, including aggregates defined for one or more columns, of two or more major tables that are frequently joined defined as a prejoin of those tables.</p> <p>The defined prejoin, or join index, permits the Optimizer to select it to cover frequently made join queries that also compute aggregates rather than specifying that its underlying base tables be searched, aggregated, and joined dynamically.</p>
Sparse	<p>Any join index that limits its rows to those satisfying a constant condition in the WHERE clause of its definition.</p> <p>A sparse join index permits the database designer to limit the rows in the index to a tightly filtered subset of the rows in the component base table set. A common use of this feature might be to restrict the population of a join index to only those rows that are frequently accessed by a commonly performed query or set of queries.</p>
Column-partitioned	<p>Useful in support of DML requests that access a variable, selective, small subset of the columns (as specified in predicates or projection lists) and rows of a column-partitioned table. Also useful in support of RowID joins between a column-partitioned table and another table.</p> <p>Column-partitioned join indexes have the following restrictions:</p> <ul style="list-style-type: none"> <li>Must be a single-table join index</li> <li>Cannot have a primary index</li> <li>Cannot compute aggregates</li> <li>Cannot be row-compressed</li> <li>Cannot have value-ordering (but can have row partitioning).</li> </ul>

## Simple Join Indexes

The primary function of a simple join index is to provide the Optimizer with a high-performing, cost-effective means for satisfying any query that specifies a frequently performed join operation. The simple join index permits you to define a permanent prejoin table without violating the normalization of the database schema. Simple join indexes are also referred to as multitable join indexes.

## Simple Join Index Example

For example, suppose that a common task is to look up customer orders by customer number and date. You might create a join index like the following, linking the customer table, the order table and the order detail table:

```
CREATE JOIN INDEX cust_ord2
AS SELECT cust.customerid,cust.loc,ord.ordid,item,qty,odate
FROM cust, ord, orditm
WHERE cust.customerid = ord.customerid
AND ord.ordid = orditm.ordid;
```

While you might never issue a query that completely joined these three tables, the key benefit of this join index is its versatility. For example, a query that only looks at the customers for a single state, like the following, can still use the cust\_ord2 join index rather than accessing its underlying base tables.

```
SELECT cust.customerid, ord.ordid, item, qty
FROM cust, ord, orditm
WHERE cust.customerid = ord.customerid
AND ord.ordid = orditm.ordid
AND cust.loc = 'WI';
```

## Simple Join Indexes and Partial Query Covering

It is also possible for a join index to be used to partially cover a query to improve query performance. For example, if you wanted to count the number of orders made by customers in the European region during October, you might use the following query:

```
SELECT cust.customerid,COUNT(ord.ordid)
FROM cust, ord, orditm, location
WHERE ord.ordid = orditm.ordid
AND cust.customerid = ord.customerid
AND cust.loc = location.loc
AND location.region = 'EUROPE'
AND EXTRACT(MONTH, ord.orddate) = 10
GROUP BY cust.customerid;
```

In this example, the query includes the location table which is not included in the join index. Teradata Database can still use the join index to partially cover the query by joining the contents of the join index with the location table.

See “[Partial Query Coverage](#)” on page 505 for more information about partial query coverage.

## Defining a Simple Join Index on a Binary Join Result

### Table Definitions

Suppose you define the following *customer* and *orders* tables.

```
CREATE TABLE customer (
  c_custkey      INTEGER NOT NULL,
  c_name         CHARACTER(26) NOT NULL,
```

```

c_address      VARCHAR(41),
c_nationkey   INTEGER,
c_phone        CHARACTER(16),
c_acctbal     DECIMAL(13,2),
c_mktsegment  CHARACTER(21),
c_comment      VARCHAR(127))
PRIMARY INDEX(c_custkey);

CREATE TABLE orders (
    o_orderkey      INTEGER NOT NULL,
    o_date          DATE FORMAT 'yyyy-mm-dd',
    o_status        CHARACTER(1),
    o_custkey       INTEGER,
    o_totalprice   DECIMAL(13,2),
    o_orderpriority CHARACTER(21),
    o_clerk         CHARACTER(16),
    o_shipppriority INTEGER,
    o_comment       VARCHAR(79))
UNIQUE PRIMARY INDEX(o_orderkey);

```

## Example Query Request

Consider the following SELECT request against these tables.

```

SELECT o_custkey, c_name, o_status, o_date, o_comment
FROM orders, customer
WHERE o_custkey=c_custkey;

```

The next few topics examine the query plan for this SELECT request: first without and then with a join index.

### Query Plan: No Join Index Defined

Without a defined join index, the execution plan for this query would typically redistribute the *orders* table into a spool file, sort the spool on *o\_custkey*, and then perform a merge join between the spool and the *customer* table.

### Query Plan: Join Index Defined

Now consider the execution plan for this same query when the following join index has been defined:

```

CREATE JOIN INDEX OrdCustIdx AS
    SELECT (o_custkey, c_name), (o_status, o_date, o_comment)
    FROM orders, customer
    WHERE o_custkey=c_custkey;

```

With this join index defined, the execution plan for the query specifies a simple scan of the join index without accessing any of the underlying base tables and without having to join them on the predicate WHERE *o\_custkey* = *c\_custkey*.

In the join index defined for this example, (*o\_custkey*, *c\_name*) is the specified fixed part of the index and (*o\_status*, *o\_date*, *o\_comment*) is the repeated portion. Therefore, assume the following specimen base table entries (where the ? character indicates a null).

Customer

CustKey	Name	Address
100	Robert	San Diego
101	Ann	Palo Alto
102	Don	El Segundo

Orders

OrderKey	Date	Status	CustKey	Comment
5000	2004-10-01	S	102	rush order
5001	2004-10-01	S	100	big order
5002	2004-10-03	D	102	delayed
5003	2004-10-05	U	?	unknown customer
5004	2004-10-05	S	100	credit

You cannot collect statistics on a complex expression from a base table. If your applications frequently run queries that specify complex expressions in their predicates, you should consider creating a single-table join index that specifies a matching complex expression in its select list or column list, respectively. When Teradata Database creates the index, it transforms the complex expression into a simple index column on which you can collect statistics.

If the complex expression specified by the index is a term that matches a predicate condition for a query made against the base table the index is defined on, statistics collected on the index expression can be mapped to the base table so the Optimizer can use them to make more accurate single-table cardinality estimates.

## Materialized Join Index

The materialized logical join index rows are the following:

OrdCustIdx

Fixed Part		Repeated Part		
CustKey	Name	Status	Date	Comment
100	Robert	S	2004-10-01	big order
		S	2004-10-05	credit
101	Ann	P	2004-10-08	discount
102	Don	S	2004-10-01	rush order
		D	2004-10-03	delayed

Note that the information for the null customer is not included in this join index because it was defined using an inner join.

**Note:** Join indexes are not limited to binary joins: like any other join, they can be defined on joins involving as many as 128 tables.

## Defining and Using a Simple Join Index With an n-way Join Result

The following example shows the creation of a join index defined with an *n*-way join result and then shows how the Optimizer uses the join index to process a query on the base tables for which it is defined.

### Join Index Definition

The following statement defines a join index with a three-table join using both natural and outer joins.

```
CREATE JOIN INDEX cust_order_join_line AS
  SELECT (l_orderkey, o_orderdate, c_nationkey, o_totalprice),
         (l_partkey, l_quantity, l_extendedprice, l_shipdate)
    FROM (lineitem LEFT JOIN orders ON l_orderkey = o_orderkey)
      INNER JOIN customer ON o_custkey = c_custkey
 PRIMARY INDEX (l_orderkey);

*** Index has been created.
*** Total elapsed time was 20 seconds.
```

### EXPLAIN for Query With Complicated Predicates

The following EXPLAIN shows how the Optimizer might use the join index for a query that accesses all three of the base tables defined in the index.

```
EXPLAIN SELECT l_orderkey, o_orderdate, o_totalprice, l_partkey,
              l_quantity, l_extendedprice, l_shipdate
        FROM lineitem, orders, customer
       WHERE l_orderkey = o_orderkey
         AND o_custkey = c_custkey
         AND c_nationkey = 10;

*** Help information returned. 16 rows.
*** Total elapsed time was 1 second.
```

#### Explanation

---

- 1) First, we lock a distinct LOUISB."pseudo table" for read on a Row Hash to prevent global deadlock for LOUISB.cust\_order\_join\_line.
- 2) Next, we lock LOUISB.cust\_order\_join\_line for read.
- 3) We do an all-AMPS RETRIEVE step from join index table LOUISB.cust\_order\_join\_line by way of an all-rows scan with a condition of ("LOUISB.cust\_order\_join\_line.c\_nationkey = 10") into Spool 1, which is built locally on the AMPS. The input table will not be cached in memory, but it is eligible for synchronized scanning. The result spool file will not be cached in memory. The size of Spool 1 is estimated to be 200 rows. The estimated time for this step is 3 minutes and 57 seconds.

- 4) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.

## Single-Table Join Indexes

You can define a simple join index on a single table. A single-table join index is a database object created using the CREATE JOIN INDEX statement, but specifying only one table in its FROM clause. This permits you to hash some or all of the columns of a large replicated base table on a foreign key that hashes rows to the same AMP as another large table. In some situations, this is more high-performing than building a multitable join index on the same columns. In effect, you are redistributing an entire base table or a frequently accessed subset of base table columns using a join index when you do this. The main advantage comes from less under-the-covers update maintenance on the single-table form of the index.

Single-table join indexes are the only type of join index that can be defined with a unique primary index.

The term *single-table join index* might seem to be a contradiction because there are no joins in a single-table join index. However, the Optimizer can use single-table join indexes to facilitate joins. The single-table join index came about because an observant software architect had the insight that it was possible to use the join index mechanism with a single table to horizontally partition all or a subset of a very large base table as a join index on a different primary index than that used by the original base table in order to hash its rows to the same AMPs as another very large base table that with which it was frequently joined. In this respect, a single-table join index is essentially a hashed NUSI.

Because of the way the rows of a column-partitioned join index are distributed to the AMPs, this advantage does not generalize to them (see “[Row Allocation for Teradata Parallel Data Pump](#)” on page 237) or by the Fast Load utility (see “[Row Allocation for FastLoad Operations Into Nonpartitioned NoPI Tables](#)” on page 238). However, column-partitioned join indexes are useful as an alternative method to partition a base table in an entirely different way when such a option provides an appropriate choice for the Optimizer to consider for some queries.

This application is analogous to how NUPIs are often used in database design to hash the base table rows of a minor entity to the same AMP as rows from another table they are likely to be joined with in a well known query workload (see “[Nonunique Primary Indexes](#)” on page 265), though you cannot explicitly specify a join to a join index in a DML request. Instead, the Optimizer must determine if joining base table rows with join index rows would be less costly than other methods.

### Functions of Single-Table Join Indexes

Even though each single-table join index you create partly or entirely replicates its base table, you cannot query or update them directly just as you cannot directly query or update any other join index.

When you have an application for which join queries against a base table would benefit from replicating some or all of its columns in a different table hashed on the join key (usually the primary index of the table to which it is to be joined) rather than the primary index of the

original base table, then you should consider creating one or more single-table join indexes on that table.

For example, you might want to create a single-table join index to avoid redistributing a large base table or to avoid the sometimes prohibitive storage requirements of a multitable join index. A single-table join index might be useful for commonly made joins having low predicate selectivity but high join selectivity, for example.

This strategy substitutes the join index for the underlying base table and defines a primary index that ensures that rows containing only the columns to be joined are hashed to the same AMPs, eliminating the need to redistribute rows when the database manager joins the tables.

As another example, suppose you have a primary index defined on a major entity column that joins with many foreign key subtentity columns. The cost of the maintenance required to update a multitable join index defined on this table is many times greater than the cost of maintaining the underlying base table.

The Optimizer can use unique single-table join indexes to access base table rows.

When you have a table with a large number of columns that is queried frequently, but only on a small subset of those columns, you can create either a hash index or a single-table join index to effectively partition the table vertically. Partitioning the rows of a table, as Teradata Database does to distribute rows to the AMPs, is often called *horizontal partitioning*. This is *not* what a single-table join index or hash index does. Instead, those indexes effectively partition tables on their columns, a method referred to as *vertical partitioning*. For example, for a table with 1,500 columns, only 25 of which are frequently queried, you could create a hash or single-table join index on those 25 frequently queried columns, which has the same effect as vertically partitioning the base table itself into two sets of columns: one set of 25 frequently queried columns and another set of 1,475 *infrequently* queried columns. Note that neither horizontal partitioning nor vertical partitioning is related in any way to how Teradata Database partitions the rows of a table having a partitioned primary index on an AMP, and that is why the terms horizontal partitioning and vertical partitioning are generally avoided in this book.

With a hash or single-table join index available that contains all of the frequently queried columns from the base table (and in the case of a join index, either the ROWID key word, the unique primary index of the base table, or a USI from the base table), the Optimizer can use that index to cover queries on that column subset, and then join to the base table to pick up any additional columns from the table that a query might specify in its select list.

You can also use single-table join indexes as a mechanism to collect statistics on complex expressions that are defined in their select list. The Optimizer can then either use mapping to exploit a matched expression that it finds in a non-matching predicate by mapping to the join index column statistics, or it can use matching when it detects identical predicates in both the join index definition and in a query made against the base table on which the join index is defined. See *SQL Request and Transaction Processing* for details.

## Column-Partitioned Single-Table Join Indexes

You can create column-partitioned single-table join indexes with the following restrictions.

- The index cannot have a primary index.
- The index cannot compute aggregates.
- The index cannot be row-compressed.
- The index cannot have value ordering (but they can be row-partitioned).

Column-partitioned join indexes are designed to support requests that very selectively access a variable small subset of the columns and rows, either in predicates or as column projections.

The Optimizer can also use column-partitioned join indexes to support direct access to a column-partitioned table using a RowID join.

See “CREATE JOIN INDEX” in *SQL Data Definition Language* for more information about column partitioning and single-table join indexes.

## Maintenance Costs of Single-Table Join Indexes

For a single-table join index, the maintenance cost is roughly double the cost of maintaining the base table.

When you design a schema, there are often some tables that are queried in such a way that for some frequently run workloads, the table is joined on one column, but for another important query, the table is joined on another column. The usual design solution is to distribute the rows of this table on the column that is most frequently used in a join. If there is more than one column, then a join index might be a good design choice. A join index can be used to redistribute the table on the secondary join attribute so that joins can be done without a redistribution step.

Join indexes can also be used to evaluate parameterized queries. For the Optimizer to use a join index in this situation, the query must also contain a non-parameterized condition in its WHERE clause that the join index covers.

For example, suppose you create the following base table and single-table join index:

```
CREATE TABLE tpl (
    pid      INTEGER,
    name     VARCHAR(32),
    address  VARCHAR(32),
    zipcode  INTEGER);

CREATE JOIN INDEX tpl_ji AS
    SELECT pid, name, zipcode
    FROM tpl
    WHERE zipcode > 50000
        AND zipcode < 55000;
```

## Parameterized Queries and Single-Table Join Indexes

The following parameterized query can use this join index because the Optimizer knows that the matching rows are contained in the index because the WHERE clause predicate in the query is a conjunction between the covered term zipcode and the parameterized term :N.

```
USING (N VARCHAR(32))
SELECT pid, name
FROM tpl
```

```
WHERE zipcode IN (54455, 53066)
AND name = :N;
```

The explanation for this query looks like the following report:

Explanation

- ```
-----  
1) First, we lock a distinct CURT."pseudo table" for read on a RowHash to prevent global deadlock for CURT.TP1_JI.  
2) Next, we lock CURT.TP1_JI for read.  
3) We do an all-AMPS RETRIEVE step from CURT.TP1_JI by way of an all-rows scan with a condition of ("((CURT.TP1_JI.zipcode = 53066 ) OR (CURT.TP1_JI.zipcode = 54455 )) AND (CURT.TP1_JI.name = :N)") into Spool 1 (group_amps), which is built locally on the AMPS. The size of Spool 1 is estimated with no confidence to be 1 row ( 64 bytes). The estimated time for this step is 0.03 seconds.  
4) Finally, we send out an END TRANSACTION step to all AMPS involved in processing the request.  
-> The contents of Spool 1 are sent back to the user as the result of statement 1. The total estimated time is 0.03 seconds.
```

Only prototyping can determine which is the better design for a given set of tables, applications, and hardware configuration.

## Related Strategies

Other functionally similar strategies for solving this problem can also be used. In general, only prototyping can determine which among the possible choices is best for a particular application environment and hardware configuration.

The following list describes some of the alternative strategies to creating single-table join indexes:

- You can create a hash index.

For some applications, a hash index is a better choice than a single-table join index if only because of its simpler syntax; however, it might be unclear what defaults Teradata Database used to create the index. In nearly all cases you can, and should, create single-table join indexes that have identical effects on query workloads as the equivalent hash index. Also, multi-value compression can be carried over to join index but not for a hash index.

See "[Hash Indexes](#)" on page 602 for more information.

- The design technique of assigning a NUPI to a subentity table that hashes related rows to the same AMPS as a related major entity is superficially similar to a single-table join index. The differences are as follows.
  - Cardinalities

The cardinalities of tables for which a single-table join index is defined are typically very similar to the base tables they are designed to be joined with, while those for major entity-subentity joins are typically very different, with the major entity typically having many more rows than the subentity.

The entity PI-subentity NUPI strategy is typically used when the subentity is a relatively small table in terms of its degree as well as its cardinality.

The single-table join index strategy is typically used when only a small subset of the columns from the base table from which the single-table join index is derived are frequently joined with the base table in question.

- Specialization  
When you create a single-table join index, the parent base table from which it is derived might have a different primary index, in which case its rows hash to different AMPs. The single-table join index is a denormalized, specialized database object defined for a specific purpose, while the parent base table is a normalized, more general database object. Both tables in an entity-subentity relationship remain normalized and generalized database objects.
- You can create a multitable join index that prejoins the entity attributes most likely to be joined in a query.  
Updating a multitable join index can have a varying cost depending on which table in the multitable join is update, the indexes on this join index and base tables, and so on. In some cases, the update can be about the same as single-table, sometimes it can be very expensive if it requires an expensive join to be able to do the maintenance.  
The upside of a standard multitable join index strategy is that, at least for the queries for which they are designed, Teradata Database does not have to perform any join processing because the required rows are already prejoined. The single-table join index can avoid a costly redistribution of table rows, but join processing is still required to respond to the query.
- You can create a denormalized prejoin base table.  
Denormalization reduces the generality of the database for ad hoc queries and data mining operations as well as introducing various problematic update anomalies. Nevertheless, a relatively mild degree of denormalization is standard in physically implemented databases, and for some applications might be the only high-performing solution.

See “[Single-Table Join Index](#)” on page 550 for an example of using a single-table join index.

## Single-Table Join Index

This example shows how a single-table join index can be used as a substitute for a standard join index to minimize update maintenance while at the same time making join processing more high-performing than it would otherwise be.

### Table Definitions

Suppose you have the following tables that you query frequently using join expressions, and both are very large.

| Table Name | Primary Index | Primary Index Type |
|------------|---------------|--------------------|
| LineItem   | OrderKey      | NUPI               |
| Part       | PartKey       | UPI                |

The table definitions are as follows.

```

CREATE TABLE LineItem (
    l_OrderKey      INTEGER NOT NULL,
    l_PartKey       INTEGER NOT NULL,
    l_SupplierKey   INTEGER,
    l_LineNumber    INTEGER,
    l_Quantity      INTEGER NOT NULL,
    l_ExtendedPrice DECIMAL(13,2) NOT NULL,
    l_Discount      DECIMAL(13,2),
    l_Tax           DECIMAL(13,2),
    l_ReturnFlag    CHARACTER(1),
    l_LineStatus    CHARACTER(1),
    l_ShipDate      DATE FORMAT 'yyyy-mm-dd',
    l_CommitDate    DATE FORMAT 'yyyy-mm-dd',
    l_ReceiptDate   DATE FORMAT 'yyyy-mm-dd',
    l_ShipInstruct   VARCHAR(25),
    l_ShipMode       VARCHAR(10),
    l_Comment        VARCHAR(44))
PRIMARY INDEX (l_OrderKey);

CREATE TABLE part (
    p_PartKey        INTEGER NOT NULL,
    p_PartDescription CHARACTER(26),
    p_SupplierNumber INTEGER)
UNIQUE PRIMARY INDEX (p_PartKey);

```

## Example Query Request

A frequently performed query on these tables might be the following:

```

SELECT l_PartKey, p_PartDescription, l_Quantity, l_SupplierKey
FROM LineItem, Part
WHERE l_PartKey=p_PartKey;

```

## Decision: Ordinary Join Index Versus Single-Table Join Index

You could create an ordinary join index on the *LineItem* and *Part* tables, but there is a high cost to keeping this join index updated because each update requires a costly minijoin operation, so the ordinary join index would have to be updated each time either a line item or a new part was inserted, deleted, or updated in the respective primary base tables.

A better solution might be to create a single-table join index on the columns of *LineItem* that need to be joined frequently with the *Part* table and then make the primary index for the join index *l\_PartKey*. Single-table join indexes are not cost-free, but the cost of performing the single row updates for a single-table index is far less expensive than the minijoins required by a multitable join index.

## Single-Table Join Index Definition

The definition for the join index might look something like this.

```

CREATE JOIN INDEX PartKeyLineItem AS
    SELECT l_PartKey, l_Quantity, l_SupplierKey
    FROM LineItem
PRIMARY INDEX (l_PartKey);

```

The intent of defining this join index is to permit the Optimizer to select it in place of the base table *LineItem* in cases like the equality condition `l_PartKey = p_PartKey`, eliminating the need to redistribute the *LineItem* table (because its proxy, the join index table *PartKeyLineItem*, has the same primary index as that of the *Part* table, so the rows are stored on the same AMP). This avoids the large redistribution of *LineItem*, but not the join processing.

Not only can the Optimizer use single-table join indexes for rewriting queries, it can also use statistics collected on complex expressions in the index definition to better estimate single-table cardinalities. See *SQL Request and Transaction Processing* for more information about using hash and single-table join indexes to estimate single-table cardinalities.

## General Procedure for Defining a Single-Table Join Index

- 1 Define a `column_1_name` for each `column_name` in the primary base table to be included in the single-table join index.
- 2 To enhance join selectivity, define the primary index on a different column set than the primary base table, or define column partitioning with no primary index.
- 3 If the physical database design warrants, use:
  - `CREATE INDEX` to create one or more NUSIs on the join index
  - A `WHERE` clause to define a sparse join index
  - A unique primary index for the join index

## Aggregate Join Indexes

An aggregate join index is a database object created using the `CREATE JOIN INDEX` statement, but specifying one or more columns that are derived from an aggregate expression. An aggregate join index is a join index that specifies MIN, MAX, SUM, COUNT, or that extracts a DATE value aggregate operations. No other aggregate functions are permitted in the definition of a join index; however, most of the other simple aggregate functions can be derived from these using column expressions. You can create aggregate join indexes as either single-table or as multitable join indexes. Aggregate join indexes can also be sparse (see “[Sparse Join Indexes](#)” on page 556).

### Functions of Aggregate Join Indexes

The primary function of an aggregate join index is to provide the Optimizer with a high-performing, cost-effective means for satisfying any query that specifies a frequently made aggregation operation on one or more columns.

In other words, aggregate join indexes permit you to define a persistent summary table without violating the normalization of the database schema. This allows a join index to precompute an aggregate value that would otherwise potentially require a table scan and sort operation.

Aggregate join indexes can be especially helpful for queries that roll up values for dimensions other than the primary key dimension, which would otherwise require redistribution.

An aggregate join index can be used to cover aggregate queries that only consider a subset of groups contained in the join index or have more join tables than the join index. In order to allow the aggregate join index to be used in this way, its definition must satisfy the following conditions:

- The grouping clause must include all columns that are specified in the grouping clause of the query.
- All columns in the query WHERE clause that join to tables not in the aggregate join index must be part of the join index definition.
- If you define row partitioning for an aggregate join index, its partitioning columns must be members of the column set specified in the GROUP BY clause of the index definition. In other words, you cannot specify an *aggregated* column as a partitioning column.
- An aggregate join index cannot be column partitioned.

An aggregate join index can also be used to cover the following:

- Requests that specify COUNT(DISTINCT) and extended grouping such as CUBE, ROLLUP, and GROUPING SETS.
- Requests that specify subqueries or spooled derived tables.
- Both the outer and the inner tables in a request that specifies an outer join.

## Related Strategies

Other functionally similar strategies for solving this problem can also be used. In general, only prototyping can determine which among the possible choices is best for a particular application environment and hardware configuration.

- You can create a global temporary table definition and then populate a materialized instance of it with aggregated result sets. This is not so much an alternate strategy as it is an entirely *different* strategy designed for an entirely different application.

Global temporary tables are private to the session that materializes them and their data does not persist beyond the end of that session. Unless you specify the ON COMMIT PRESERVE option when the definition for the temporary table is created, its contents do not persist even across individual database *transactions*. Unless provisions are made to write their contents to a persistent base table when a session ends, their data is not saved. Without numerous safeguards built into the process, this is not a method that provides any assurances about the integrity of the data it produces because while the contents of any global temporary table are private to the session in which it is created, the definition of the table is global within a database and any number of different sessions and users could materialize a different version of the table, populate it, and write the results to the same base table as any other session.

Note that the containing database or user for a global temporary table must have a minimum of 512 bytes of available PERM space to contain the table header for the GTT.

- You can create a volatile table with aggregate expressions defined on some or all its columns. The drawbacks of global temporary tables for this application apply equally to volatile tables.
- You can create a denormalized base table and populate it with an aggregated result set. Denormalization always reduces the generality of the database for any ad hoc queries or data mining operations you might want to undertake as well as introducing various problematic update anomalies. While a relatively mild degree of denormalization is standard in physically implemented databases, the sort of denormalization called for by this solution is probably beyond what most DBAs would find acceptable, the enthusiasm of dimensional modeling theorists notwithstanding.

Because there is no mechanism for keeping such a table synchronized with its base table, it can become quickly outdated.

Nonetheless, for some applications this approach might be the only high-performing solution.

## Example

This example shows how a simple aggregate join index can be used to create prejoins with aggregation on one or more of its columns while retaining full normalization of the *physical* database. Strictly speaking, the term *normalization* applies only to the *logical* schema for a database, not to its physical schema; however, the term has unfortunately come to be used equally for both logical and physical database schemas. See [Chapter 5: “The Normalization Process”](#) for details.

### Table Definitions

This example set uses the following table definitions.

```
CREATE TABLE customer (
    c_custkey      INTEGER NOT NULL,
    c_name         CHARACTER(26) CASESPECIFIC NOT NULL,
    c_address      VARCHAR(41),
    c_nationkey   INTEGER,
    c_phone        CHARACTER(16),
    c_acctbal     DECIMAL(13,2),
    c_mktsegment  CHARACTER(21),
    c_comment      VARCHAR(127)
    UNIQUE PRIMARY INDEX (c_custkey);

CREATE TABLE orders (
    o_orderkey     INTEGER NOT NULL,
    o_custkey      INTEGER,
    o_orderstatus  CHARACTER(1) CASESPECIFIC,
    o_totalprice   DECIMAL(13,2) NOT NULL,
    o_orderdate    DATE FORMAT 'YYYY-MM-DD' NOT NULL,
    o_orderpriority CHARACTER(21),
    o_clerk        CHARACTER(16),
    o_shipppriority INTEGER,
    o_comment      VARCHAR(79))
    UNIQUE PRIMARY INDEX (o_orderkey);
```

## Example Query Statement

Consider the following aggregate join query.

```
SELECT COUNT(*), SUM(o_totalprice)
FROM orders, customer
WHERE o_custkey = c_custkey
AND o_orderdate > DATE '2004-09-20'
AND o_orderdate < DATE '2004-10-15'
GROUP BY c_nationkey;
```

## Query Plan: No Aggregate Join Index Defined

Without an aggregate join index, a typical execution plan for this query might involve the following stages:

- 1 Redistribute orders into a spool file.
- 2 Sort the spool file on *o\_custkey*.
- 3 Merge join the sorted spool file and the *customer* file.
- 4 Aggregate the result of the merge join.

## Aggregate Join Index Definition

Suppose you define the following aggregate join index, which aggregates *o\_totalprice* over a join of *orders* and *customer*.

```
CREATE JOIN INDEX ord_cust_idx AS
  SELECT c_nationkey, SUM(o_totalprice(FLOAT))
        AS price, o_orderdate
   FROM orders, customer
  WHERE o_custkey = c_custkey
 GROUP BY c_nationkey, o_orderdate
 ORDER BY o_orderdate;
```

## Query Plan for Aggregate Join Index

The execution plan produced by the Optimizer for this query includes an aggregate step on the aggregate join index, which is much smaller than either one of the join tables. You can confirm this by performing an EXPLAIN on the query.

```
EXPLAIN SELECT COUNT(*), SUM(o_totalprice)
  FROM orders, customer
 WHERE o_custkey = c_custkey
 AND o_orderdate > DATE '2005-09-20'
 AND o_orderdate < DATE '2005-10-15'
 GROUP BY c_nationkey;

*** Help information returned. 18 rows.
*** Total elapsed time was 3 seconds.
```

### Explanation

- 
- 1) First, we lock a distinct TPCD."pseudo table" for read on a RowHash to prevent global deadlock for TPCD.ord\_cust\_idx.
  - 2) Next, we lock TPCD.ord\_cust\_idx for read.
  - 3) We do a SUM step to aggregate from join index table TPCD.ordcustidx by way of an all-rows scan with a condition of "(TPCD.ord\_cust\_idx.O\_ORDERDATE > DATE '2005-09-20') AND (TPCD.ord\_cust\_idx.O\_ORDERDATE < DATE '2005-10-15'))", and the grouping identifier in field 1. Aggregate Intermediate Results

are computed globally, then placed in Spool 2. The size of Spool 2 is estimated to be 1 row.

- 4) We do an all-AMPS RETRIEVE step from Spool 2 (Last Use) by way of an all-rows scan into Spool 1, which is built locally on the AMPS. The size of Spool 1 is estimated with no confidence to be 1 row. The estimated time for this step is 0.17 seconds.
- 5) Finally, we send out an END TRANSACTION step to all AMPS involved in processing the request.

-> The contents of Spool 1 are sent back to the user as the result of statement 1.

## Aggregate Join Index With EXTRACT Function

Join index definitions, both simple and aggregate, support the EXTRACT function. This example illustrates the use of the EXTRACT function in the definition of an aggregate join index.

### Aggregate Join Index Definition

The index is defined as follows.

```
CREATE JOIN INDEX ord_cust_idx_2 AS
    SELECT c_nationkey, SUM(o_totalprice(FLOAT)) AS price,
           EXTRACT(YEAR FROM o_orderdate) AS o_year
      FROM orders, customer
     WHERE o_custkey = c_custkey
   GROUP BY c_nationkey, o_year
  ORDER BY o_year;
```

The aggregation is based only on the year of o\_orderdate, which has fewer groups than the entire o\_orderdate, so *ord\_cust\_idx\_2* is much smaller than *ord\_cust\_idx*.

On the other hand, the use for *ord\_cust\_idx\_2* is more limited than *ord\_cust\_idx*. In particular, *ord\_cust\_idx\_2* can only be used to satisfy queries that select full years of orders.

### Example Query Statement

While the join index defined for this example, *ord\_cust\_idx\_2*, cannot be used for the query analyzed in “[Query Plan for Aggregate Join Index](#)” on page 555, the following query *does* profit from its use because dates are on year boundaries.

```
SELECT COUNT(*), SUM(o_totalprice)
  FROM orders, customer
 WHERE o_custkey = c_custkey
   AND o_orderdate > DATE '2004-01-01'
   AND o_orderdate < DATE '2004-12-31'
 GROUP BY c_nationkey;
```

## Sparse Join Indexes

Any join index, whether simple or aggregate, multitable or single-table, can be sparse. A sparse join index specifies a constant expression in the WHERE clause of its definition to narrowly filter its row population. For example, the following DDL creates an aggregate join index containing only the sales records from 2000:

```
CREATE JOIN INDEX j1 AS
    SELECT store_id, dept_id, SUM(sales_dollars) AS sum_sd
```

```
FROM sales
WHERE EXTRACT(year FROM sales_date) = 2000
GROUP BY store_id, dept_id;
```

This method limits the rows included in the join index to a subset of the rows in the table based on an SQL request result.

When base tables are large, you can use this feature to reduce the content of the join index to only the portion of the table that is frequently used if the typical query only references a portion of the rows.

It is important to collect statistics on the sparse-defining column of the join index, or the Optimizer may not use the join index.

## Sparse Join Indexes and Query Optimization

When a user query is entered, the Optimizer determines if accessing *j1* gives the correct answer and is more efficient than accessing the base tables. This sparse join index would be selected by the Optimizer only for queries that restricted themselves to data from the year 2000. For example, a query might require data from June of 2000. Because the join index *j1* contains all of the data for year 2000, it might be used to satisfy the following query:

```
SELECT store_id, dept_id, SUM(sales_dollars) AS sum_sd
FROM sales
WHERE sales_date >= 6/1/2000
AND   sales_date < 7/1/2000
GROUP BY storeid, dept_id;
```

As another example, the following DDL creates a join index containing only those customers living in four western states of the US:

```
CREATE JOIN INDEX westcust AS
SELECT cust.id, cust.address, donations.amount
FROM cust, donations
WHERE cust.cust_id = donations.cust_id
AND   donations.d_date > '2003/01/01'
AND   cust.state IN ('CA', 'OR', 'WA', 'NV');
```

The Optimizer can use this join index for the following query written to find all donors from California with donations of 1,000 USD or more made since January 1, 2003.

```
SELECT custid, cust.address
FROM cust, donations
WHERE donations.amount > 1000
AND   cust.state = 'CA'
AND   donations.d_date > '2003/01/01';
```

Because the customers and donations considered by the query are part of the subset included in the join index, the Optimizer uses it to answer the query. This can save a great deal of time especially in situations where the base tables are very large, but queries typically look only at subsets of the tables.

Maintenance of sparse join indexes can be much faster than maintenance for other join indexes because the sparse types have fewer values, and so are generally updated much less frequently.

See “[Sparse Join Indexes and Tactical Queries](#)” on page 907 for information about design issues specific to sparse join index support for tactical queries.

## Performance Impact of Sparse Join Indexes

A sparse index can focus on the portions of the tables that are most frequently used to do the following things.

- Make the costs for maintaining an index proportional to the percent of rows actually referenced in the index.
- Make request access faster where the resulting sparse index is smaller than a standard join index.
- Reduce the storage requirements for a join index where possible.

## Using Outer Joins in Join Index Definitions

Join indexes should be defined using outer joins if they are intended to cover both inner and outer join queries. To understand how this is possible, an outer join can be thought of as producing a result consisting of two sets of rows. The first set corresponds to the set of matched rows obtained when a row from the outer table matches one or more rows from the inner table (this set corresponds to the set of rows defined by the inner join with the same join condition). The second corresponds to the set of unmatched rows: those rows from the outer table that do not match any rows from the inner table.

Except for the presence of the unmatched row set, an outer join is the same as an inner join, and produces the same result. Therefore if an inner join query can be completely satisfied by the matched set of rows from an outer join index, the Optimizer uses it.

### Join Index With Outer Join in Its Definition

Consider the following join index defined on the three tables  $t1$ ,  $t2$ , and  $t3$ . Tables  $t1$  and  $t2$  are joined with an inner join, and the result is joined with table  $t3$  using an outer join. The outer table is the result of joining tables  $t1$  and  $t2$ .

```
CREATE JOIN INDEX jil AS
SELECT a1, a2, a3
FROM (t1 INNER JOIN t2 ON a1 = a2)
LEFT OUTER JOIN t3 ON a1 = a3;
```

Column  $a3$  is the unique primary index for table  $t3$ . Column  $a3$  might be specified in the table definition explicitly as a primary index, or simply as unique. This means that all of the rows from the join of  $t1$  with  $t2$  are in the join index exactly once, either in the matched set, or in the unmatched set. Therefore, the following query can be satisfied by the join index:

```
SELECT a1, a2
FROM t1, t2
WHERE a1 = a2;
```

## Extended Query Coverage With Outer Joins in Index Definition

A coverage algorithm determines that there is partial coverage, and the Optimizer uses the join index to join with the base tables to project the non-covered columns if the cost is lower than performing the query using the base tables alone.

The join index optimizations introduced by extending query coverage through defining extra foreign key-primary key joins does not negate the advantage of defining outer joins in your join index definitions because the normalization of outer joins to inner joins used by that optimization is specific to a particular class of queries. Queries that do not meet those specific criteria continue to benefit from the unnormalized outer join definitions in the join index.

Join indexes defined with outer joins can cover a query submitted in inner join format directly. Once the Optimizer converts the outer join in the query to an inner join by taking advantage of the equivalency of outer and inner joins for foreign key-primary key relationships (see “[Restriction on Coverage by Join Indexes When a Join Index Definition References More Tables Than a Query](#)” on page 577), the system can make a coverage test instead of identical matching. In other words, the predicates in the join index definition and in the query need not be identical.

## More On Outer Join Index Coverage of Queries

A join index can be used to cover a wide variety of queries as long as the rows required in these queries form a subset of the row set contained in the join index. For example, consider a join index defined with the following SELECT query, where x1, x2 is a foreign key-primary key pair:

```
CREATE JOIN INDEX loj_cover AS
SELECT x1, x2
FROM t1, t2
WHERE x1=x2;
```

Any query of the following form can use this join index, where *c* represents any set of constant conditions:

```
SELECT x1, x2
FROM t1 LEFT OUTER JOIN t2 ON x1=x2 AND c;
```

This property greatly increases the applicability of many join indexes.

Both the join index and the query are normalized to inner joins when the original form is defined with an outer join and there is a foreign key-primary key relationship between the join column set (see “[Restriction on Coverage by Join Indexes When a Join Index Definition References More Tables Than a Query](#)” on page 577). The result is that a less restrictive coverage test can be applied to both the query and to the join index.

## Using Outer Joins to Define Join Indexes

There are several benefits in defining non-aggregate join indexes with outer joins:

- Unmatched rows are preserved.

These rows allow the join index to satisfy queries with fewer join conditions than those used to generate the index.

- Outer table row scans can provide the same performance benefits as a single-table join index.

For example, the Optimizer can choose to scan the outer table rows of a join index to satisfy a query that only references the outer table provided that a join index scan would be more high-performing than scanning the base table or redistributing rows.

## Redefined Join Index

The following example changes the previous join index example (see “[Defining a Simple Join Index on a Binary Join Result](#)” on page 542) to use an Outer Join in the join index definition.

```
CREATE JOIN INDEX OrdCustIdx AS
  SELECT (o_custkey,c_name)
    ,
  (o_status,o_date,o_comment)
  FROM orders LEFT JOIN customer ON o_custkey=c_custkey;
```

## Materialized Join Index

The resulting join index rows would be the following (where the ? character indicates a null).

| OrdCustIdx |        |               |            |                  |
|------------|--------|---------------|------------|------------------|
| Fixed Part |        | Repeated Part |            |                  |
| CustKey    | Name   | Status        | Date       | Comment          |
| 100        | Robert | S             | 2004-10-01 | big order        |
|            |        | S             | 2004-10-05 | credit           |
| 101        | Ann    | P             | 2004-10-08 | discount         |
| 102        | Don    | S             | 2004-10-01 | rush order       |
|            |        | D             | 2004-10-03 | delayed          |
| ?          | ?      | U             | 2004-10-05 | unknown customer |

With the join index defined in this way, the following query could be resolved using just the join index, without having to scan the base tables.

```
SELECT o_status, o_date, o_comment
  FROM orders;
```

In this particular case, it is more efficient to access the join index than it is to access the *orders* base table. This is true whenever the cost of scanning the join index is less than the cost of scanning the *orders* table. The Optimizer evaluates both access methods, choosing the more efficient, less costly of the two for its query plan.

# Creating Join Indexes Using Outer Joins

This example examines the EXPLAIN reports for a different outer join-based join index in some detail.

## Table Definitions

```

CREATE TABLE customer (
    c_custkey      INTEGER NOT NULL,
    c_name         CHARACTER(26),
    c_address      VARCHAR(41),
    c_nationkey   INTEGER,
    c_phone        CHARACTER(16),
    c_acctbal     DECIMAL(13,2),
    c_mktsegment  CHARACTER(21),
    c_comment      VARCHAR(127))
UNIQUE PRIMARY INDEX(c_custkey);

CREATE TABLE orders (
    o_orderkey     INTEGER NOT NULL,
    o_custkey      INTEGER,
    o_orderstatus  CHARACTER(1),
    o_totalprice   DECIMAL(13,2) NOT NULL,
    o_orderdate    DATE FORMAT 'yyyy-mm-dd' NOT NULL,
    o_orderpriority CHARACTER(21),
    o_clerk        CHARACTER(16),
    o_shipppriority INTEGER,
    o_comment      VARCHAR(79))
UNIQUE PRIMARY INDEX (o_orderkey);

CREATE TABLE lineitem (
    l_orderkey     INTEGER NOT NULL,
    l_partkey      INTEGER NOT NULL,
    l_suppkey      INTEGER,
    l_linenumber   INTEGER,
    l_quantity      INTEGER NOT NULL,
    l_extendedprice DECIMAL(13,2) NOT NULL,
    l_discount      DECIMAL(13,2),
    l_tax           DECIMAL(13,2),
    l_returnflag   CHARACTER(1),
    l_linestatus    CHARACTER(1),
    l_shipdate      DATE FORMAT 'yyyy-mm-dd',
    l_commitdate    DATE FORMAT 'yyyy-mm-dd',
    l_receiptdate   DATE FORMAT 'yyyy-mm-dd',
    l_shipinstruct  VARCHAR(25),
    l_shipmode      VARCHAR(10),
    l_comment       VARCHAR(44))
PRIMARY INDEX (l_orderkey);

```

## Join Index Definition

The following statement defines a join index on these tables. Subsequent examples demonstrate the effect of this join index on how the Optimizer processes various queries.

```

CREATE JOIN INDEX order_join_line
AS SELECT (l_orderkey, o_orderdate, o_custkey,

```

```

        o_totalprice),
        (l_partkey, l_quantity, l_extendedprice, l_shipdate)
FROM lineitem LEFT JOIN orders ON l_orderkey = o_orderkey
ORDER BY o_orderdate
PRIMARY INDEX (l_orderkey);

*** Index has been created.
*** Total elapsed time was 15 seconds.

```

## EXPLAIN for Query With Simple Predicate

The following EXPLAIN report shows how the newly created join index, *order\_join\_line*, might be used by the Optimizer.

```

EXPLAIN SELECT o_orderdate, o_custkey, l_partkey, l_quantity,
           l_extendedprice
      FROM lineitem, orders
     WHERE l_orderkey = o_orderkey;

```

### Explanation

- 1) First, we lock a distinct LOUISB."pseudo table" for read on a Row Hash to prevent global deadlock for LOUISB.order\_join\_line.
- 2) Next, we lock LOUISB.order\_join\_line for read.
- 3) We do an all-AMPS RETRIEVE step from join index table LOUISB.order\_join\_line by way of an all-rows scan with a condition of ("NOT (LOUISB.order\_join\_line.o\_orderdate IS NULL)") into Spool 1, which is built locally on the AMPS. The input table will not be cached in memory, but it is eligible for synchronized scanning. The result spool file will not be cached in memory. The size of Spool 1 is estimated to be 1,000,000 rows. The estimated time for this step is 4 minutes and 27 seconds.
- 4) Finally, we send out an END TRANSACTION step to all AMPS involved in processing the request.

## EXPLAIN for Query With More Complicated Predicate

The following EXPLAIN report shows how the join index might be used in a query when an additional search condition is added on the join indexed rows.

```

EXPLAIN SELECT o_orderdate, o_custkey, l_partkey,
           l_quantity, l_extendedprice
      FROM lineitem, orders
     WHERE l_orderkey = o_orderkey
       AND o_orderdate > '2003-11-01';

```

### Explanation

- 1) First, we lock a distinct LOUISB."pseudo table" for read on a Row Hash to prevent global deadlock for LOUISB.order\_join\_line.
- 2) Next, we lock LOUISB.order\_join\_line for read.
- 3) We do an all-AMPS RETRIEVE step from join index table LOUISB.order\_join\_line with a range constraint of ( "LOUISB.order\_join\_line.Field\_1026 > 971101" ) with a residual condition of ("(NOT (LOUISB.order\_join\_line.o\_orderdate IS NULL )) AND (LOUISB.order\_join\_line.Field\_1026 > 971101)") into Spool 1, which is built locally on the AMPS. The input table will not be cached in memory, but it is eligible for synchronized scanning. The size of Spool 1 is estimated to be 1000 rows. The estimated time for this step is 0.32 seconds.
- 4) Finally, we send out an END TRANSACTION step to all AMPS involved in processing the request.

## EXPLAIN for Query With Aggregation

The following EXPLAIN shows how the join index might be used in a query when aggregation is performed on the join indexed rows.

```
EXPLAIN SELECT l_partkey, AVG(l_quantity),
             AVG(l_extendedprice)
      FROM lineitem , orders
     WHERE l_orderkey = o_orderkey
       AND o_orderdate > '2003-11-01'
  GROUP BY l_partkey;
```

### Explanation

- 1) First, we lock a distinct LOUISB."pseudo table" for read on a RowHash to prevent global deadlock for LOUISB.order\_join\_line.
- 2) Next, we lock LOUISB.order\_join\_line for read.
- 3) We do a SUM step to aggregate from join index table LOUISB.order\_join\_line with a range constraint of ("LOUISB.order\_join\_line.Field\_1026 > 971101") with a residual condition of ("(LOUISB.order\_join\_line.Field\_1026 > 971101) AND (NOT (LOUISB.order\_join\_line.o\_orderdate IS NULL))"), and the grouping identifier in field 1. Aggregate Intermediate Results are computed globally, then placed in Spool 3. The input table will not be cached in memory, but it is eligible for synchronized scanning.
- 4) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1, which is built locally on the AMPS. The size of Spool 1 is estimated to be 10 rows. The estimated time for this step is 0.32 seconds.
- 5) Finally, we send out an END TRANSACTION step to all AMPS involved in processing the request.

## EXPLAIN for Query With Base Table-Join Index Table Join

The following EXPLAIN shows how the join index might be used in a query when join indexed rows are used to join with another base table.

```
EXPLAIN SELECT o_orderdate, c_name, c_phone, l_partkey, l_quantity,
             l_extendedprice
      FROM lineitem, orders, customer
     WHERE l_orderkey = o_orderkey
       AND o_custkey = c_custkey;
```

### Explanation

- 1) First, we lock a distinct LOUISB."pseudo table" for read on a Row Hash to prevent global deadlock for LOUISB.order\_join\_line.
- 2) Next, we lock a distinct LOUISB."pseudo table" for read on a Row Hash to prevent global deadlock for LOUISB.customer.
- 3) We lock LOUISB.order\_join\_line for read, and we lock LOUISB.customer for read.
- 4) We do an all-AMPs RETRIEVE step from join index table LOUISB.order\_join\_line by way of an all-rows scan with a condition of ("NOT (LOUISB.order\_join\_line.o\_orderdate IS NULL)") into Spool 2, which is redistributed by hash code to all AMPS. Then we do a SORT to order Spool 2 by row hash. The size of Spool 2 is estimated to be 1,000,000 rows. The estimated time for this step is 1 minute and 53 seconds.
- 5) We do an all-AMPs JOIN step from LOUISB.customer by way of a Row Hash match scan with no residual conditions, which is joined to Spool 2 (Last Use). LOUISB.customer and Spool 2 are joined using a merge join, with a join condition of ("Spool\_2.o\_custkey = LOUISB.customer.c\_custkey"). The result goes into Spool 1, which is built locally on the AMPS. The size of Spool 1 is estimated to be 1,000,000 rows. The estimated time for this step is 32.14 seconds.
- 6) Finally, we send out an END TRANSACTION step to all AMPS involved in processing the request.

## EXPLAIN for Query Against Single Table

The following EXPLAIN report shows how the join index might be used in a query of a single table.

```
EXPLAIN SELECT l_orderkey, l_partkey, l_quantity, l_extendedprice
   FROM lineitem
 WHERE l_partkey = 1001;
```

### Explanation

- 1) First, we lock a distinct LOUISB."pseudo table" for read on a Row Hash to prevent global deadlock for LOUISB.order\_join\_line.
- 2) Next, we lock LOUISB.order\_join\_line for read.
- 3) We do an all-AMPs RETRIEVE step from join index table LOUISB.order\_join\_line by way of an all-rows scan with a condition of ("LOUISB.order\_join\_line.l\_partkey = 1001") into Spool 1, which is built locally on the AMPs. The input table will not be cached in memory, but it is eligible for synchronized scanning. The result spool file will not be cached in memory. The size of Spool 1 is estimated to be 100 rows. The estimated time for this step is 59.60 seconds.
- 4) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.

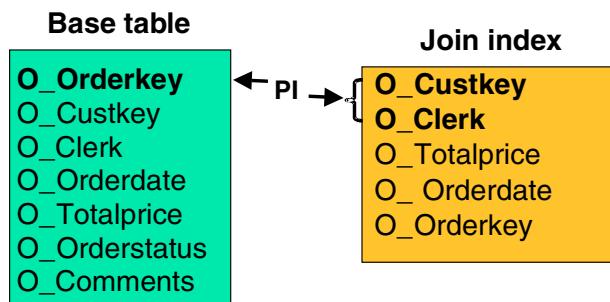
## Join Indexes and Tactical Queries

### Single-Table Join Indexes

One of the most useful constructs for tactical queries is the single-table join index. Because you can define a primary index for the join index composed of different columns than those used to define the base table primary index, you can create an alternative method of directly accessing data in the associated base table.

For example, you could create a join index on the *orders* table that includes only a subset of the columns that a particular tactical query application might require. In the example shown in the following graphic, assume that the application has available a value for *o\_custkey* and the clerk that placed the order, but does not have a value for *o\_orderkey*. Specifying the primary index defined for the join index *OrderJI* supports direct access to order base table using the single-table join index *OrderJI*.

```
CREATE JOIN INDEX OrderJI AS
  SELECT o_custkey, o_clerk, o_totalprice, o_orderdate, o_orderkey
    FROM orders
 PRIMARY INDEX(o_custkey,o_clerk);
```



1094A032

A test against an orders table with 75 million rows, both with and without the *OrderJI* join index, returned the following response times from the query and join index above:

| If the query is run ... | The response time is ... |
|-------------------------|--------------------------|
| without the join index  | 1:55.                    |
| with the join index     | subsecond.               |

The larger the base table is, the longer it takes to process via a table scan, and the greater the benefit a join index providing single-AMP access provides. In this example, the join index took 8:23 to create.

A single-table join index like this one is particularly useful when the tactical application does not have the primary index of the base table available, but has an alternative row identifier. This might be the case when a social security number is available, but a member ID, the primary index of the base table, is not. Single-AMP access would still be achievable using the join index if its primary index is defined on social security number.

## Aggregate Join Indexes

If your application supports repeated access to the same table using aggregation along the same set of dimensions, you should consider using aggregate join indexes to enhance the performance of those queries.

If you include one or more aggregating columns in the join index select list, then that index is an aggregate join index. Aggregate join indexes are dynamic summary tables, not snapshots. For example, the following aggregate join index computes a running sum on *o\_totalprice* from the orders table:

```
CREATE JOIN INDEX ordersum AS
  SELECT o_clerk, SUM(o_totalprice) AS sumprice
  FROM orders
  GROUP BY o_clerk
  PRIMARY INDEX(o_clerk);
```

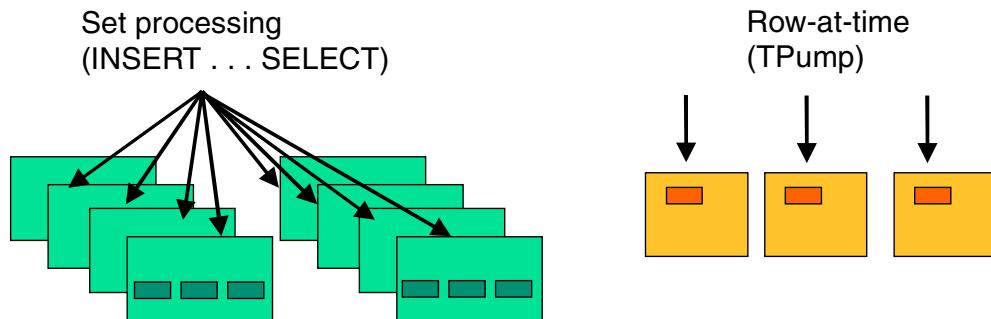
Each time the *o\_totalprice* column of orders is updated, a new sum is computed and stored in the *ordersum* aggregate join index. If there are frequent queries throughout the day that request summaries of the total prices for orders placed by a specific clerk, this join index would be able to deliver response times suitable for tactical queries like the following example:

```
SELECT o_clerk, SUM(o_totalprice)
FROM orders
GROUP BY o_clerk
WHERE o_clerk = 'Clerk#000046240';
```

Use the GROUP BY column set as the primary index of the join index if this is the column whose value is specified by the application. Each query that specifies a value for that column will then be processed as a single-AMP request.

## Join and Hash Index Maintenance Considerations

Each time a base table row is updated, its corresponding join or hash index data is modified within the same unit of work. Because MultiLoad and FastLoad are incompatible with join and hash indexes, maintenance of a base table that has a join or hash index must be performed using SQL, or the join or hash index must be dropped before the MultiLoad or FastLoad utility is run and the index then rebuilt afterward. As illustrated by the following figure, you can use SQL to take either a set processing approach or a row-at-a-time approach to join and hash index maintenance.



1094A033

If an `INSERT ... SELECT` request is used to load data into the base table, then the unit of work is the entire SQL request. In such a case the join or hash index maintenance is performed in batch mode. Inserts to the base table are spooled and applied to the join or hash index structure in a single step of the query plan. In particular, when an `INSERT ... SELECT` request specifies the primary index value of the target row set in its select list or in its WHERE clause, a single-AMP merge step is used to process the `INSERT` operation.

When row-at-a-time updates are performed, the join or hash index structure is updated concurrently, once per base table row.

Join and hash index maintenance is optimized to use localized rowhash-level locking whenever possible. Table-level locks are applied when the maintenance is performed using all-AMPs operations such as spool merges.

When the table being updated contains a join or hash index, an EXPLAIN of the UPDATE request illustrates whether an all-AMPs operation and table-level locking are used at the time of the UPDATE operation, or a single-AMP operation with rowhash locking.

The following is a list of the conditions that support single-AMP updating and rowhash-level locking on join or hash indexes when the base table is updated a row-at-a-time. Be aware that these optimizations might not be applied to complicated indexes when cardinality estimates are made with low confidence or when the index is defined on three or more tables with all its join conditions based on non-unique and non-indexed columns.

When a row-at-a-time INSERT on the base table is being performed, the following restrictions apply:

- The join index can have a different primary index from the base table.

- Inserts that specify an equality constraint on the primary index column set of a table with joins between some non-primary index columns of the table and the primary index columns of another table are optimized to use rowhash-level locks. For example,

Given the following join index definition:

```
CREATE JOIN INDEX j2 AS
    SELECT x1,x2,x3,y2,y3,z2
    FROM t1,t2,t3
    WHERE x1=y1
    AND y2=z1
    PRIMARY INDEX (y3);
```

When a row is inserted into *t1* with the following INSERT statement,

```
INSERT INTO t1
VALUES (1,1,1);
```

corresponding *j2* rows are materialized by a query like the following:

```
SELECT 1,1,1,y2,y3,z2
FROM t2,t3
WHERE y1=1
AND y2=z1;
```

As long as the number of rows that qualify *t2.y1=1* (see step 2-1) is within the 10% of the number of AMPs threshold and the number of rows resulting from the *t2* and *t3* join step 5 is also within this threshold, this INSERT statement does not incur any table-level locks.

For those INSERT operations that specify an equality constraint on a non-primary index or that involve joins with non-primary index columns, temporary hash indexes can be created by the system to process them in such a way that single-AMP retrieves are used and non-primary index joins are converted to primary index joins. See the following examples:

Given the following join index definition:

```
CREATE JOIN INDEX j1 AS
    SELECT x1,x2,x3,y2,y3
    FROM t1,t2
    WHERE x2= y1
    PRIMARY INDEX (x1);
```

When a row is inserted into *t2* with the following INSERT statement

```
INSERT INTO t2
VALUES (1,1,1);
```

corresponding *j1* rows are materialized by a query like the following:

```
SELECT x1, x2, x3 ,1 ,1
FROM t1
WHERE x2=1;
```

Given the following join index definition:

```
CREATE JOIN INDEX j1 AS
    SELECT x1,x2,x3,y2,y3
    FROM t1,t2
    WHERE x2=y1
    PRIMARY INDEX (x1);
```

When a row is deleted from  $t_2$ , `delete t2 where t2.y1=1`, corresponding  $j_1$  rows are materialized by a query like the following:

```
SELECT x1,x2,x3,y2,y3
  FROM t1,t2
 WHERE x2=y1
   AND y1=1;
```

Given the following join index definition:

```
CREATE JOIN INDEX j2 AS
  SELECT x1,x2,x3,y1,y3
    FROM t1, t2
   WHERE x2=y2;
```

When a row is deleted from  $t_2$ , `delete t2 where t2.y1=1`, corresponding  $j_2$  rows are materialized by a query which involves a non-PI-to-non-PI join:

```
SELECT x1,x2,x3,y1,y3
  FROM t1, t2
 WHERE x2=y2
   AND y1=1;
```

If a single-table non-covering join index is defined on  $t_1.x_2$ , calling it  $stji\_t1\_x2$ , the  $(t_1.x_2=t_2.y_2)$  join that is processed by duplicating the qualified  $t_2$  row in step 4 to join with  $t_1$  in step 5 can be processed as follows:

- 1 a single-AMP retrieve from  $t_2$  by way of PI  $t_2.y1=1$  into Spool 2. Spool 2 is hash redistributed by  $t_2.y2$  and qualifies as a group-AMPs spool.
- 2 a few-AMPs join from Spool 2 to  $stji\_t2\_x2$  on  $y2=x2$ , results going into Spool 3. Spool 3 is redistributed by  $t1.rowid$  and also qualifies as a group-AMPs spool.
- 3 a few-AMPs join back from Spool 3 to  $t_1$  on ROWID.

As long as the number of rows that qualify the join  $t_1.x_2=t_2.y_2$  `WHERE t1.y1=1` is within the 10% threshold, no table-level locks are incurred for the DELETE statement.

Given the following join index definition:

```
CREATE JOIN INDEX j3 AS
  SELECT x1,x2,x3,y2,y3,z2,z3
    FROM t1, t2, t3
   WHERE x2=y2
     AND y1=z1
 PRIMARY INDEX (y3);
```

When a row is inserted into  $t_1$ , `INSERT INTO t1 VALUES (1,1,1)`, corresponding  $j_3$  rows are materialized by a query of the kind:

```
SELECT 1,1,1,y2,y3,z2,z3
  FROM t2,t3
 WHERE y2=1
   AND y1=z1
```

The index must be a single-table join index or hash index with the following exceptions:

- The join index primary index is composed of a column set from the target table of the simple insert.

- The join index primary index is composed of a column set that is joined, either directly or indirectly through transitive closure, to the target table of the simple insert.

In these cases, multitable join indexes are also optimized for single-AMP merge steps with rowhash-level locking.

For example, given the following tables, the join index definitions that follow qualify for single-AMP merge optimization:

```

CREATE TABLE t1 (
    x1 INTEGER,
    x2 INTEGER,
    x3 INTEGER)
PRIMARY INDEX (x1);

CREATE TABLE t2 (
    y1 INTEGER,
    y2 INTEGER,
    y3 INTEGER)
UNIQUE PRIMARY INDEX (y1);

CREATE TABLE t3 (
    z1 INTEGER,
    z2 INTEGER,
    z3 INTEGER)
PRIMARY INDEX (z1);

CREATE JOIN INDEX j1 AS
    SELECT x1,x2,x3,y2,y3
    FROM t1,t2
    WHERE x2=y1
PRIMARY INDEX (x1);

CREATE JOIN INDEX j2 AS
    SELECT x1,x2,x3,y2,y3,z2,z3
    FROM t1,t2,t3
    WHERE x2=y1
    AND y1=z1
PRIMARY INDEX (x2);

```

The following table indicates several base table inserts, their corresponding single-AMP merge join index inserts, and the qualifying condition that permits the single-AMP optimization:

| Base Table Insert                         | Join Index Insert                                                                                                                              | Qualifying Condition                                                                                        |
|-------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------|
| <pre>INSERT INTO t1 VALUES (1,1,1);</pre> | <pre>INSERT INTO j1 SELECT 1,1,1,y1,y3 FROM t2, t3 WHERE y1=1;  INSERT INTO j2 SELECT 1,1,1,y1,y3,z1,z3 FROM t2,t3 WHERE y1=1 AND y1=z1;</pre> | The primary index of the join index is composed of a column set from the target table of the simple insert. |

| Base Table Insert                               | Join Index Insert                                                                                                   | Qualifying Condition                                                                                                                                                       |
|-------------------------------------------------|---------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>INSERT INTO t2<br/>VALUES (1,1,1);</code> | <code>INSERT INTO j2<br/>SELECT x1,x2,x3,1,1,z2,z3<br/>FROM t1,t3<br/>WHERE x2=1<br/>AND 1=z1<br/>AND x2=z1;</code> | The primary index of the join index is composed of a column set that is joined directly or indirectly through transitive closure to the target table of the simple insert. |
| <code>INSERT INTO t3<br/>VALUES (1,1,1);</code> | <code>INSERT INTO j2<br/>SELECT x1,x2,x3,y2,y3,1,1<br/>FROM t1,t2<br/>WHERE x2=y1<br/>AND y1=1;</code>              |                                                                                                                                                                            |

- When a row-at-a-time UPDATE on the base table is being performed, the following restrictions apply:
  - The value for the primary index of the join index must be specified in the WHERE clause predicate of the request.
  - The UPDATE cannot change the primary index of the join index.
  - When it is cost effective to access the affected join or hash index rows by means of a NUSI, it is done using rowhash locks and a direct update step. If only a few rows are updated (a few-AMPs operation), rowhash READ locks are placed on the NUSI subtable for the index rows that are read. Rowhash locks are also applied to the base table using the rowID values extracted from the index rows.
- When a row-at-a-time DELETE on the base table is performed, the following restrictions apply:
  - The value for the primary index of the join index must be specified in the WHERE clause predicate of the DELETE statement.
  - The deleted row must not be from the inner table of an outer join in the CREATE JOIN INDEX statement with the following exceptions:
    - The outer join condition in the join index is specified on a UPI column from the inner table.
    - The outer join condition in the join index is specified on a NUPI column from the inner table.
  - When it is cost effective to access the affected join or hash index rows by means of a NUSI, it is done using rowhash-level locks and a direct delete step. If only a few rows are deleted (a few-AMPs operation), rowhash-level READ locks are placed on the NUSI subtable for the index rows that are read. Rowhash-level locks are also applied to the base table using the rowID values extracted from the index rows.

Under all other conditions, a single-row update causes a table-level WRITE lock to be placed on the hash or join index.

If table-level locks are reported in the EXPLAIN text, then consider using set processing approaches with one or more secondary indexes as an alternative.

## Examples of Row-at-a-Time **INSERT** Maintenance Overhead

Using the guidelines provided in “[Join and Hash Index Maintenance Considerations](#)” on [page 566](#), you can create a single-table join index that performs well with continuous row-at-a-time inserts to its base table. The maintenance to that join index structure, if Teradata Parallel Data Pump were performing continuous inserts, would be one additional single-AMP operation accompanied by a single rowhash-level WRITE lock.

A further example of join index maintenance shows that row-at-a-time inserts into a base table that participates in a multitable join index exert a table-level lock on the join index except for the following cases:

- The primary index of the join index is composed of a column set from the target table of the simple insert.
- The primary index of the join index is composed of a column set that is joined either directly or indirectly through transitive closure to the target table of the simple insert.

In the following example, the primary index of a row-compressed multitable join index is a subset of the base table being updated, so a single-AMP merge can be used.

### CREATE JOIN INDEX

```
CREATE JOIN INDEX JImulti AS
SELECT (o_custkey, c_nationkey,
        c_acctbal, c_name),
       (a_orderdate, o_orderpriority)
  FROM orders, customer
 WHERE o_custkey = c_custkey
 PRIMARY INDEX (c_name);
```

### Base table

|              |
|--------------|
| C_Custkey    |
| C_Name       |
| C_Nationkey  |
| C_Acctbal    |
| C_Mktsegment |

← PI →

|                 |
|-----------------|
| C_Name          |
| C_Nationkey     |
| C_Acctbal       |
| O_Custkey       |
| O_Orderdate     |
| O_Orderpriority |

1094A035

If neither exception is true, then consider using set processing approaches to update the base table (see [“Set Processing Alternative” on page 572](#)).

An aggregate join index incurs the same maintenance overhead as a multitable join index for row-at-a-time base table inserts. The following example uses a single-table aggregate join index.

### CREATE JOIN INDEX

```
CREATE JOIN INDEX cust_JI AS
SELECT c_nationkey, c_mktsegment,
       SUM(c_acctbal) AS accl_bal_sum
  FROM customer
 GROUP BY c_nationkey, c_mktsegment
PRIMARY INDEX (c_nationkey,
                c_mktsegment);
```

### Base table

|              |
|--------------|
| C_Custkey    |
| C_Nationkey  |
| C_Mktsegment |
| C_Acctbal    |
| C_Name       |
| C_Address    |

### Join index

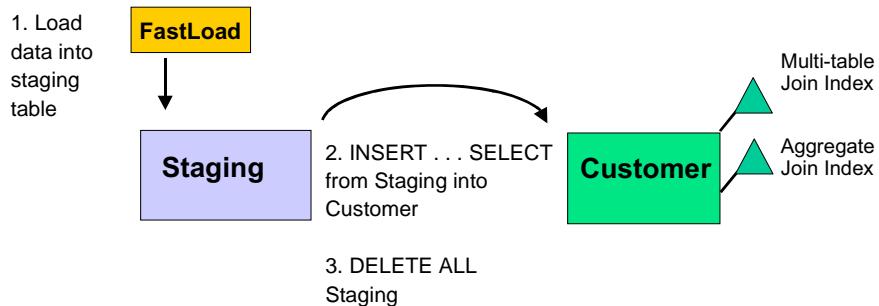
|                |
|----------------|
| C_Nationkey    |
| C_Mktsegment   |
| Sum(C_Acctbal) |

1094A036

For multitable join indexes, the Optimizer uses rowhash-level locks wherever possible, resulting in fewer table-level locks on the join index table. If there are just a few rows in the join index that are impacted, the Optimizer places several rowhash-level locks on just those AMPs and then uses a group-AMPs, rather than an all-AMPs operation.

## Set Processing Alternative

For situations where row-at-a-time maintenance imposes table-level locking, consider updating the base table with periodic set processing approaches. The following example shows a set processing approach to updating the customer table, using frequent, short INSERT ... SELECT operations into the base table that also update its associated multitable and aggregate join indexes.



## Join Index Definition Restrictions

This topic lists several restrictions on join index definitions. For complete details about join index syntax, see the documentation for CREATE JOIN INDEX in *SQL Data Definition Language*.

### Restrictions on Number of Join Indexes Defined Per Base Table

The maximum number of secondary, hash, and join indexes that can be defined for a table, in any combination, is 32. This includes the system-defined secondary indexes used to implement PRIMARY KEY and UNIQUE constraints. Each composite NUSI that specifies an ORDER BY clause counts as 2 consecutive indexes in this calculation (see “[Importance of Consecutive Indexes for Value-Ordered NUSIs](#)” on page 485). You cannot define join, or any other, indexes on global temporary trace tables. See “CREATE GLOBAL TEMPORARY TRACE TABLE” in *SQL Data Definition Language Detailed Topics*.

Suppose you have four tables, each with multiple secondary, hash, and join indexes defined on them:

- *Table\_1* has 32 secondary indexes and no hash or join indexes.
- *Table\_2* has 16 secondary indexes, no hash indexes, and 16 join indexes.
- *Table\_3* has 10 secondary indexes, 10 hash indexes, and 12 join indexes.
- *Table\_4* has no secondary or hash indexes, but has 32 join indexes.

Each of these combinations is valid, but they all operate at the boundaries of the defined limits.

Note that if any of the secondary indexes defined on tables 1, 2, or 3 is a composite NUSI defined with an ORDER BY clause, the defined limits are exceeded, and the last index you

attempt to create on the table will fail. Because each composite NUSI defined with an ORDER BY clause counts as 2 consecutive indexes in the count against the maximum of 32 per table, you could define only 8 of them on *table\_2*, for example, if you also defined 16 join indexes on the table.

## Restrictions on the Number of Columns Per Referenced Base Table

The following restrictions apply to the number of columns per table that can be specified in a join index definition.

- The maximum number of columns that can be specified in a join index per referenced base table is 64.
- The maximum number of columns that can be specified in a multitable join index definition is 2,048.
- When you specify join index row compression, then each of the *column\_1* and *column\_2* sets is limited to 64 common base table references.

## Restrictions on the Data Type of Join Index Columns

You cannot specify columns having any of the following data types in the definition of a join index:

- XML columns
- BLOB columns
- CLOB columns
- BLOB-based UDT columns
- CLOB-based UDT columns
- XML-based UDT columns
- ARRAY/VARRAY columns
- Geospatial columns

This also means that a join index cannot have a geospatial NUSI.

## Restrictions on the Use of the System-Derived PARTITION[#Ln] Column

You *cannot* use a system-derived PARTITION[#Ln] column in a join index definition.

## Restrictions on Outer Join Definitions

The following restrictions apply to outer joins when used to define a join index.

- FULL OUTER JOIN is not valid.
- When you specify a LEFT or RIGHT outer join, the following rules apply:
  - The outer table joining column for each condition must be contained in either *column\_1\_name* or *column\_2\_name*.
  - The inner table of each join condition must have at least one non-nullable column in either *column\_1\_name* or *column\_2\_name*.

## Restrictions on Secondary Index Definitions

You cannot define unique secondary indexes on a join index.

You can define the primary index of a join index to be row partitioned if and only if the join index is not also defined with row compression. See the documentation for CREATE JOIN INDEX in *SQL Data Definition Language* for more information about specifying partitioned primary indexes for join indexes.

## Restrictions on Built-In Functions and Join Index Definitions

When you create a join index that specifies a built-in, or system, function in its WHERE clause (see *SQL Functions, Operators, Expressions, and Predicates* for more information about built-in functions), the system resolves the function at the time the join index is created and then stores the result as part of the index definition rather than evaluating it dynamically at the time the Optimizer would use the index to build a query plan.

As a result, the Optimizer does not use a join index in the access plan for a query that qualifies its WHERE clause with the same built-in function used to define that join index because it cannot determine whether the index covers the query or not. The Optimizer *does* use the join index if the query specifies an explicit value in its WHERE clause that matches the resolved value stored in the index definition.

(The exception to this is the CURRENT\_TIME and CURRENT\_TIMESTAMP functions, which are resolved for a query. The resolved value is used to check if the query can be covered by a join index.)

For example, suppose you decide to define a join index using the CURRENT\_DATE built-in function on January 4, 2010 as follows:

```
CREATE JOIN INDEX curr_date AS
  SELECT *
  FROM orders
  WHERE order_date = CURRENT_DATE;
```

On January 7, 2010, you perform the following SELECT statement:

```
SELECT *
FROM orders
WHERE order_date = CURRENT_DATE;
```

When you EXPLAIN this query, you find that the Optimizer does not use join index *curr\_date* because the date stored in the index definition is the explicit value ‘2010-01-04’, not the current system date ‘2010-01-07’.

Note that if you had defined *curr\_date* with a predicate of *order\_date* >= CURRENT\_DATE instead of *order\_date* = CURRENT\_DATE, then the Optimizer could use *curr\_date* to cover the query if it were the least costly way to process the request.

On the other hand, if you were to perform the following SELECT statement on January 7, 2010, or any other date, retaining the original *curr\_date* predicate, the Optimizer *does* use join index *curr\_date* for the query plan because the statement explicitly specifies the same date that was stored with the join index definition when it was created:

```
SELECT *
  FROM orders
 WHERE order_date = DATE '2010-01-04';
```

## Restriction on Number of Join Indexes Selected Per Query

The Optimizer can use several join indexes for a single query, selecting one multitable join index as well as additional single-table join indexes for its join plan. The join indexes selected depend on the structure of the query, and the Optimizer might not choose all applicable join indexes for the plan. Always examine your EXPLAIN reports to determine which join indexes *are* used for the join plans generated for your queries. If a join index you think should have been used by a query was not included in the join plan, try restructuring the query and then EXPLAIN it once again.

The limit on the number of join indexes considered per query is enforced to limit the number of possible combinations and permutations of table joins in the Optimizer search space during its join planning phase. The rule helps to ensure that the optimization is worth the effort. In other words, that the time spent generating the query plan does not exceed the accrued performance enhancement.

## Restrictions on Partial Covering by Join Indexes

The Optimizer can use a join index that partially covers a query in the following cases:

- One of the columns in the index definition is the keyword ROWID.  
You can only specify ROWID in the outermost SELECT of the CREATE JOIN INDEX statement. See “CREATE JOIN INDEX” in *SQL Data Definition Language Detailed Topics*.
- The column set defining the UPI of the underlying base table is also carried in the definition.
- The column set defining the NUPI of the underlying base table *plus* either of the following is also carried in the definition:
  - One of the columns in the definition of that index is the keyword ROWID.  
You can only specify ROWID in the outermost SELECT of a CREATE JOIN INDEX request.
  - The column set defining a USI on the underlying base table.  
The ROWID option is the preferable choice.
- Partial covering is not supported for the inner table of an outer join.
- Partial covering is not supported for queries that contain a TOP *n* or TOP *m* PERCENT clause.

If statistics indicate that it would be cost-effective, the Optimizer can specify that the partially covering single-table join index be joined to one of its underlying base tables using either the ROWID or the UPI or USI to join to the column data not defined for the index itself.

Even though you do not explicitly specify this join when you write your query, it counts against the 128 table restriction on joins.

For example, suppose you define the tables  $t_1$ ,  $t_2$ , and  $t_3$  and the join indexes  $j_1$  and  $j_2$  as follows:

```
CREATE TABLE t1 (
    a1 INTEGER,
    b1 INTEGER,
    c1 INTEGER);

CREATE TABLE t2 (
    a2 INTEGER,
    b2 INTEGER,
    c2 INTEGER);

CREATE TABLE t3 (
    a3 INTEGER,
    b3 INTEGER,
    c3 INTEGER);

CREATE JOIN INDEX j1 AS
    SELECT b1, b2, t1.ROWID AS t1rowid
    FROM t1,t2
    WHERE a1=a2;

CREATE JOIN INDEX j2 AS
    SELECT b1, b2, t1.ROWID t1rowid, t2.rowid t2rowid
    FROM t1,t2
    WHERE a1=b2;
```

Join index  $j_1$  partially covers the following queries:

```
SELECT a1, b1, c1, b2
FROM t1,t2
WHERE t1.a1=t2.b2
AND t1.b1=10;

SELECT a1, b1, c1, b2, b3, c3
FROM t1,t2,t3
WHERE t1.a1=t2.b2
AND t1.b1=t3.a3
AND t3.b3 > 0;
```

The same join index does *not* partially cover the following queries:

```
SELECT *
FROM t1,t2
WHERE t1.a1=t2.b2
AND t1.b1=10;

SELECT *
FROM t1,t2,t3
WHERE t1.a1=t2.b2
AND t1.b1=t3.a3
AND t3.b3 > 0;
```

Join index  $j_2$ , on the other hand, partially covers all of these queries.

Even though you do not explicitly specify the join back to the base table when you write your query, it counts against the 64 tables per query block restriction on joins.

Be aware that a join index defined with an expression in its select list provides less coverage than a join index that is defined using base columns.

For example, the Optimizer can use join index *ji\_f1* to rewrite a query that specifies any character function on *f1*.

```
CREATE JOIN INDEX ji_f1 AS
  SELECT b1, f1
  FROM t1
  WHERE a1 > 0;
```

At the same time, because it is defined with a SUBSTR expression in its select list, the Optimizer can only use join index *ji\_substr\_f1* to rewrite a query that specifies the same SUBSTR function on *f1*.

```
CREATE JOIN INDEX ji_substr_f1 AS
  SELECT b1, SUBSTR(f1,1,10) AS s
  FROM t1
  WHERE a1 > 0;
```

## Restriction on Coverage by Join Indexes When a Join Index Definition References More Tables Than a Query

Whether the Optimizer decides to include a join index in its query plan is a more complicated choice than simply determining if the index contains all the table columns specified in the query. The columns on which the base tables in the index definition are joined and their respective referential constraints also play an important role. See “[Rules for Whether Join Indexes With Extra Tables Cover Queries](#)” on page 580.

In many cases, the Optimizer does not consider using a join index in its access plan if that index is defined on more tables than the query references. This is because the so-called extra inner joins involving the tables not referenced by the query can cause both spurious row loss and spurious duplicate row creation during the optimization process, and either outcome produces incorrect results.

This outcome can be avoided, and the join index used in the query access plan, if the extra inner joins are defined on primary key-foreign key relationships in the underlying base tables that ensure proper row preservation. Such a join index is referred to as a broad join index. The referential integrity relationship between the base table primary and foreign keys can be specified using any of the three available methods for establishing referential constraints between tables. See *SQL Data Definition Language* for further information.

A broad join index is a covering join index whose definition includes one or more tables that is not specified in the query it covers. A wide range of queries can make use of a broad join index, especially when there are foreign key-primary key relationships defined between the fact table and the dimension tables that enable the index to be used to cover queries over a subset of dimension tables.

In the case of join index outer joins, outer table rows are always preserved automatically, so there is no requirement for a referential integrity constraint to exist in order to preserve them.

In the case of foreign key-primary key inner joins, the same preservation of rows follows from a declarative referential integrity constraint. In this case, the Optimizer does consider a join

index with extra inner joins in its definition to cover a query. The following paragraphs explain why a referential constraint preserves logical integrity:

Assume that the base tables in a join index definition can be divided into two distinct sets,  $s1$  and  $s2$ .

$s1$  contains the base tables referenced in the query, while  $s2$  contains the extra base tables the query does *not* reference. The base tables in  $s1$  are joined to the base tables in  $s2$  on foreign key-primary key columns, with the tables in  $s2$  being the primary keys in the relationships. The foreign key values cannot be null because if the foreign key column set contains nulls, the losslessness of the foreign key table cannot be guaranteed.

The following assertions about these base tables are true:

- The extra joins do not eliminate valid rows from the join result among the base tables in  $s1$  because FOREIGN KEY and NOT NULL constraints ensure that every row in the foreign key table finds its match in the primary key table.
- The extra joins do not introduce duplicate rows in the join result among the base tables in  $s1$  because the primary key is, by definition, unique and not nullable.

These assertions are also true for extra joins made between base tables that are both in  $s2$ .

Therefore, the extra joins in a join index definition, if made on base tables that are defined in a way that observes these assertions, preserve all the rows resulting from the joins among the base tables in  $s1$  and do not add spurious rows. This result permits the Optimizer to use the join index to cover a query that references fewer tables than the index definition inner joins together.

The Optimizer can select a join index for a query plan if the index contains either the same set, or a subset, of the tables referenced by the query. If more tables are referenced in the join index definition than are referenced by the query, the Optimizer generally does not consider that index as a candidate for coverage because the extra joins can either eliminate rows or produce duplicate rows or both. Because a referential integrity relationship guarantees the losslessness of the foreign key table in the join with its primary key table, extra tables in a join index definition do not disqualify it from query plan consideration if the extra joins allow the join index to preserve all the rows for the join result of the subset of tables in the query.

For example, suppose you define a join index on a set of 5 tables,  $t1 - t5$ , respectively, with foreign key-primary key joins in the directions indicated (arrows point from a foreign key table to its parent primary key table) by the following diagram:

$$t1 \rightarrow t2 \rightarrow t3 \rightarrow t4 \rightarrow t5$$

Queries that reference the following table subsets can be covered by this join index because the extra joins, either between two tables where one is in the query and the other is not, or between two tables that are both not in the query, do not cause any loss of rows for the join result of the subset of tables in the query:

- $t1$
- $t1, t2$
- $t1, t2, t3$

- $t1, t2, t3, t4$

As a result of this property, the following conditions that reference extra joins can be exploited by the Optimizer when the number of tables referenced by the join index definition exceeds the number of tables referenced by the query. In each case,  $x1$  is a unique RI-related column in table  $t1$  and  $x2$  is a unique RI-related column in table  $t2$ :

| Join Index Extra Join Condition | Qualifications                                                                                                                                                                                                                                                    |
|---------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $x1 = x2$                       | <ul style="list-style-type: none"> <li>• <math>x1</math> is the foreign key in the relationship.</li> <li>• <math>t1</math> is referenced by the query.</li> <li>• <math>t2</math> is not referenced by the query.</li> </ul>                                     |
| $x1 = x2$                       | <ul style="list-style-type: none"> <li>• <math>x1</math> is the primary key in the relationship.</li> <li>• <math>t2</math> is referenced by the query.</li> <li>• <math>t1</math> is not referenced by the query.</li> </ul>                                     |
| $x1 = x2$                       | <ul style="list-style-type: none"> <li>• <math>x1</math> is the foreign key in the relationship.</li> <li>• <math>x2</math> is the primary key in the relationship.</li> <li>• Neither <math>t1</math> nor <math>t2</math> is referenced by the query.</li> </ul> |
| $x1 = x2$                       | <ul style="list-style-type: none"> <li>• <math>x1</math> is the primary key in the relationship.</li> <li>• <math>x2</math> is the foreign key in the relationship.</li> <li>• Neither <math>t1</math> nor <math>t2</math> is referenced by the query.</li> </ul> |

One restriction with two critical exceptions must be added to the above optimization to make the coverage safe: when one table referenced in the join index definition is the parent table of more than one FK table, the join index is generally disqualified from covering any query that references fewer tables than are referenced in the join index.

For example, suppose you define a join index on a set of 5 tables,  $t1 - t5$ , respectively, with foreign key-primary key joins in the directions indicated (arrows point from a foreign key table to its parent primary key table) by the following diagram:

$$t1 \rightarrow t2 \rightarrow t3 \rightarrow t4 \leftarrow t5$$

For these RI relationships, table  $t4$  is the parent table of both tables  $t3$  and  $t5$ . The losslessness of the foreign key table depends on the fact that the parent table has all the primary keys available. Joining the parent table with another child table can render this premise false. Therefore, the final join result cannot be viewed as lossless for any arbitrary subset of tables.

The following two points are exceptions to this restriction:

- When the foreign key tables are joined on the foreign key columns, there is no loss on any of the foreign key tables because they all reference the same primary key values in their common parent table.
- All foreign key tables reference the same primary key column in the primary key table. By transitive closure, all these foreign key tables are related by equijoins on the foreign key columns.

By specifying extra joins in the join index definition, you can greatly enhance its flexibility.

For example, suppose you have a star schema based on a Sales fact table and the following dimension tables:

- Customer
- Product
- Location
- Time

You decide it is desirable to define a join index that joins the fact table *Sales* to its various dimension tables in order to avoid the relatively expensive join processing between the fact table and its dimension tables whenever ad hoc join queries are made against them.

If there are foreign key-primary key relationships between the join columns, which is often the case, the join index can also be used to optimize queries that only reference a subset of the dimension tables.

Without taking advantage of this optimization, you must either create a different join index for each category of query, incurring the greater cost of maintaining multiple join indexes, or you lose the benefit of join indexes for optimizing the join queries on these tables altogether. By exploiting the foreign key-primary key join properties, the same join index can be selected by the Optimizer to generate access plans for a wide variety of queries.

As is always true, even if this optimization can be used for a given situation, the plan produced using it is compared with other generated plans, and the least costly plan from the generated plan set is the one that the system uses.

## Rules for Whether Join Indexes With Extra Tables Cover Queries

The following rules explain how to design a set of underlying base tables for a join index definition in such a way to ensure that the Optimizer selects the index for an access plan if it inner joins more tables than the query references.

|                                                                                                                                 |                                                                                                                                                                                                                                                  |
|---------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>IF there are more inner-joined tables in a join index definition than the number of tables referenced in a query and ...</p> | <p>THEN the Optimizer ...</p>                                                                                                                                                                                                                    |
| <p>the extra joins are on <i>not</i> made on foreign key-primary key columns in the underlying base tables</p>                  | <p>does <i>not</i> consider the join index for the query plan.<br/>This is because the presence of extra joins in the definition can either eliminate existing rows from the query evaluation or produce duplicate rows during optimization.</p> |

IF there are more inner-joined tables in a join index definition than the number of tables referenced in a query and ...

the extra joins *are* made on foreign key-primary key columns in the underlying base tables

both of the following conditions are true:

- The join column set of the inner table in the extra outer join is unique
- Either the inner table or both the inner and outer tables involved in the extra outer join are extra tables

THEN the Optimizer ...

considers the join index for use in for the query plan.

## Examples That Obey the General Covering Rules for Extra Tables in the Join Index Definition

The following set of base tables, join indexes, queries, and EXPLAIN reports demonstrate how the referential integrity relationships among the underlying base tables in a join index definition influence whether the Optimizer selects the index for queries that reference fewer base tables than are referenced by the join index.

```

CREATE SET TABLE t1, NO FALBACK, NO BEFORE JOURNAL,
           NO AFTER JOURNAL (
    x1 INTEGER NOT NULL,
    a1 INTEGER NOT NULL,
    b1 INTEGER NOT NULL,
    c1 INTEGER NOT NULL,
    d1 INTEGER NOT NULL,
    e1 INTEGER NOT NULL,
    f1 INTEGER NOT NULL,
    g1 INTEGER NOT NULL,
    h1 INTEGER NOT NULL,
    i1 INTEGER NOT NULL,
    j1 INTEGER NOT NULL,
    k1 INTEGER NOT NULL,
CONSTRAINT ri1 FOREIGN KEY (a1, b1, c1) REFERENCES t2 (a2,b2,c2),
CONSTRAINT ri2 FOREIGN KEY (d1) REFERENCES t3(d3),
CONSTRAINT ri3 FOREIGN KEY (e1,f1) REFERENCES t4 (e4,f4),
CONSTRAINT ri4 FOREIGN KEY (g1,h1,i1,j1) REFERENCES t5(g5,h5,i5,j5),
CONSTRAINT ri5 FOREIGN KEY (k1) REFERENCES t6(k6));

CREATE SET TABLE t2, NO FALBACK, NO BEFORE JOURNAL,
           NO AFTER JOURNAL (
    a2 INTEGER NOT NULL,
    b2 INTEGER NOT NULL,
    c2 INTEGER NOT NULL,
    x2 INTEGER)
UNIQUE PRIMARY INDEX(a2, b2, c2);

CREATE SET TABLE t3, NO FALBACK, NO BEFORE JOURNAL,
           NO AFTER JOURNAL (
    d3 INTEGER NOT NULL,
    x3 INTEGER)
UNIQUE PRIMARY INDEX(d3);

```

```
CREATE SET TABLE t4, NO FALBACK, NO BEFORE JOURNAL,
           NO AFTER JOURNAL (
   e4 INTEGER NOT NULL,
   f4 INTEGER NOT NULL,
   x4 INTEGER)
UNIQUE PRIMARY INDEX(e4, f4);

CREATE SET TABLE t5, NO FALBACK, NO BEFORE JOURNAL,
           NO AFTER JOURNAL (
   g5 INTEGER NOT NULL,
   h5 INTEGER NOT NULL,
   i5 INTEGER NOT NULL,
   j5 INTEGER NOT NULL,
   x5 INTEGER)
UNIQUE PRIMARY INDEX(g5, h5, i5, j5);

CREATE SET TABLE t6, NO FALBACK, NO BEFORE JOURNAL,
           NO AFTER JOURNAL (
   k6 INTEGER not null,
   x6 INTEGER)
UNIQUE PRIMARY INDEX(k6);
```

## Example 1: All Outer Joins in Join Index Definition

The following join index definition left outer joins *t1* to *t3* on *da=d3* and then left outer joins that result to *t6* on *k1=k6*.

```
CREATE JOIN INDEX jiout AS
  SELECT d1, d3, k1, k6, x1
    FROM t1 LEFT OUTER JOIN t3 ON d1=d3
      LEFT OUTER JOIN t6 ON k1=k6;
```

You would expect the Optimizer to use *jiout* with the following query, because all the outer joins in the join index are inner joined to the query tables on unique columns (*d1* and *k1* are declared foreign keys in *t1* and *d3* and *k6*, the primary keys for *t3* and *t6*, respectively, are declared as the unique primary index for those tables).

The bold EXPLAIN report text indicates that the Optimizer does use *jiout* in its query plan.

```
EXPLAIN SELECT d1, SUM(x1)
  FROM t1, t3
 WHERE d1=d3
 GROUP BY 1;

*** Help information returned. 17 rows.
*** Total elapsed time was 1 second.
```

### Explanation

- 1) First, we lock a distinct HONG\_JI."pseudo table" for read on a RowHash to prevent global deadlock for HONG\_JI.jiout.
- 2) Next, we lock **HONG\_JI.jiout** for read.
- 3) We do an all-AMPs SUM step to aggregate from **HONG\_JI.jiout** by way of an all-rows scan with no residual conditions, and the grouping identifier in field 1. Aggregate Intermediate Results are computed locally, then placed in Spool 3. The size of Spool 3 is estimated with low confidence to be 1 row. The estimated time for this step is 0.03 seconds.
- 4) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1 (group\_amps), which is built locally on the AMPS. The size of Spool 1 is estimated with low confidence

- to be 1 row. The estimated time for this step is 0.04 seconds.
- 5) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.  
-> The contents of Spool 1 are sent back to the user as the result of statement 1.

## Example 2: All Inner Joins Without Aggregation in Join Index Definition

The following simple join index definition specifies inner joins on tables *t1*, *t3* and *t6*.

```
CREATE JOIN INDEX jiin AS
  SELECT d1, d3, k1, k6, x1
  FROM t1, t3, t6
  WHERE d1=d3
  AND   k1=k6;
```

You would expect the Optimizer to use *jiin* with the following query, and the bold EXPLAIN report text indicates that it does.

```
EXPLAIN SELECT d1, SUM(x1)
  FROM t1, t3
  WHERE d1=d3
  GROUP BY 1;

*** Help information returned. 17 rows.
*** Total elapsed time was 1 second.
```

### Explanation

- 
- 1) First, we lock a distinct HONG\_JI."pseudo table" for read on a RowHash to prevent global deadlock for **HONG\_JI.jiin**.
  - 2) Next, we lock **HONG\_JI.jiin** for read.
  - 3) We do an all-AMPs SUM step to aggregate from **HONG\_JI.jiin** by way of an all-rows scan with no residual conditions, and the grouping identifier in field 1. Aggregate Intermediate Results are computed locally, then placed in Spool 3. The size of Spool 3 is estimated with low confidence to be 1 row. The estimated time for this step is 0.03 seconds.
  - 4) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1 (group\_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with low confidence to be 1 row. The estimated time for this step is 0.04 seconds.
  - 5) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.  
-> The contents of Spool 1 are sent back to the user as the result of statement 1.

## Example 3: All Inner Joins With Aggregation in Join Index Definition

The following aggregate join index definition specifies inner joins on tables *t1*, *t2*, and *t4*.

```
CREATE JOIN INDEX jiin_aggr AS
  SELECT a1, e1, SUM(x1) AS total
  FROM t1, t2, t4
  WHERE a1=a2
  AND   e1=e4
  AND   a1>1
  GROUP BY 1, 2;
```

You would expect the Optimizer to use join index *jiin\_aggr* in its plan for the following query because it has the same join term as the query.

The bold EXPLAIN report text indicates that the Optimizer does use *jiin\_aggr* in its plan:

```
EXPLAIN SELECT a1, e1, SUM(x1) AS total
  FROM t1, t2, t4
```

```

WHERE a1=a2
AND e1=e4
AND a1>2
GROUP BY 1, 2;

*** Help information returned. 13 rows.
*** Total elapsed time was 1 second.

```

**Explanation**

- 1) First, we lock a distinct HONG\_JI."pseudo table" for read on a RowHash to prevent global deadlock for HONG\_JI.jiin\_aggr.
- 2) Next, we lock HONG\_JI.jiin\_aggr for read.
- 3) We do an all-AMPS RETRIEVE step from HONG\_JI.jiin\_aggr by way of an all-rows scan with a condition of ("HONG\_JI.jiin\_aggr.a1 > 2") AND ("HONG\_JI.jiin\_aggr.a1 >= 3") into Spool 1 (group\_amps), which is built locally on the AMPS. The size of Spool 1 is estimated with no confidence to be 1 row. The estimated time for this step is 0.03 seconds.
- 4) Finally, we send out an END TRANSACTION step to all AMPS involved in processing the request.  
 -> The contents of Spool 1 are sent back to the user as the result of statement 1. The total estimated time is 0.03 seconds.

## Example 4: All Inner Joins With Aggregation in Join Index Definition

You would not expect the Optimizer to use join index *jiin\_aggr* (see “[Example 3: All Inner Joins With Aggregation in Join Index Definition](#)” on page 583) in its plan for the following query because the condition *b1=b2* AND *f1=f4* is not covered by the join index defined by *jiin\_aggr*. As a result, the Optimizer specifies a full-table scan to retrieve the specified rows.

The EXPLAIN report text indicates that the Optimizer does *not* choose *jiin\_aggr* to cover the query:

```

EXPLAIN SELECT a1, e1, SUM(x1) AS total
    FROM t1, t2, t4
    WHERE b1=b2
        AND f1=f4
        AND a1>2
    GROUP BY 1, 2;

```

**Explanation**

- 1) First, we lock a distinct HONG\_JI."pseudo table" for read on a RowHash to prevent global deadlock for HONG\_JI.t4.
- 2) Next, we lock a distinct HONG\_JI."pseudo table" for read on a RowHash to prevent global deadlock for HONG\_JI.t2.
- 3) We lock a distinct HONG\_JI."pseudo table" for read on a RowHash to prevent global deadlock for HONG\_JI.t1.
- 4) We lock HONG\_JI.t4 for read, we lock HONG\_JI.t2 for read, and we lock HONG\_JI.t1 for read.
- 5) We do an all-AMPS RETRIEVE step from HONG\_JI.t1 by way of an all-rows scan with a condition of ("HONG\_JI.t1.a1 > 2") into Spool 4 (all\_amps), which is duplicated on all AMPS. The size of Spool 4 is estimated with no confidence to be 2 rows. The estimated time for this step is 0.03 seconds.
- 6) We do an all-AMPS JOIN step from HONG\_JI.t2 by way of an all-rows scan with no residual conditions, which is joined to Spool 4 (Last Use). HONG\_JI.t2 and Spool 4 are joined using a product join, with a join condition of ("b1 = HONG\_JI.t2.b2"). The result goes into Spool 5 (all\_amps), which is duplicated on all AMPS. The size of Spool 5 is estimated with no confidence to be 3 rows. The estimated time for this step is 0.04 seconds.
- 7) We do an all-AMPS JOIN step from HONG\_JI.t4 by way of an all-rows scan with no residual conditions, which is joined to Spool 5 (Last Use). HONG\_JI.t4 and Spool 5 are joined using a product join, with a join condition of ("f1 = HONG\_JI.t4.f4"). The result goes into Spool 3 (all\_amps), which is built locally on the AMPS. The size of Spool 3 is estimated with no confidence to be 2 rows. The

- estimated time for this step is 0.04 seconds.
- 8) We do an all-AMPs SUM step to aggregate from Spool 3 (Last Use) by way of an all-rows scan, and the grouping identifier in field 1. Aggregate Intermediate Results are computed globally, then placed in Spool 6. The size of Spool 6 is estimated with no confidence to be 2 rows. The estimated time for this step is 0.05 seconds.
  - 9) We do an all-AMPs RETRIEVE step from Spool 6 (Last Use) by way of an all-rows scan into Spool 1 (group\_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with no confidence to be 2 rows. The estimated time for this step is 0.04 seconds.
  - 10) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.  
-> The contents of Spool 1 are sent back to the user as the result of statement 1.

## Example 5: More Inner Joined Tables in Aggregate Join Index Definition Than in Query

The following aggregate join index definition specifies inner joins on tables *t1*, *t3*, and *t6* using conditions that exploit a foreign key-primary key relationship between table *t1* and tables *t3* and *t6*, respectively.

```
CREATE JOIN INDEX jiin_aggr AS
  SELECT d1, k1, SUM(x1) AS total
    FROM t1, t3, t6
   WHERE d1=d3
     AND k1=k6
  GROUP BY 1, 2;
```

You would expect the Optimizer to include join index *jiin\_aggr* in its access plan for the following query even though *jiin\_aggr* is defined with an inner joined table, *t6*, that the query does not reference. This is acceptable to the Optimizer because of the foreign key-primary key relationship between *t1* and the extra table, *t6*, on columns *k1* and *k6*, which have a foreign key-primary key relationship and are explicitly defined as a foreign key and as the unique primary index for their respective tables.

The bold EXPLAIN report text indicates that the Optimizer does select *jiin\_aggr* for the query plan:

```
EXPLAIN SELECT d1, SUM(x1)
  FROM t1, t3
 WHERE d1=d3
 GROUP BY 1;

*** Help information returned. 17 rows.
*** Total elapsed time was 1 second.
```

### Explanation

- 
- 1) First, we lock a distinct HONG\_JI, "pseudo table" for read on a RowHash to prevent global deadlock for **HONG\_JI.jiin\_aggr**.
  - 2) Next, we lock **HONG\_JI.jiin\_aggr** for read.
  - 3) We do an all-AMPs SUM step to aggregate from **HONG\_JI.jiin\_aggr** by way of an all-rows scan with no residual conditions, and the grouping identifier in field 1. Aggregate Intermediate Results are computed locally, then placed in Spool 3. The size of Spool 3 is estimated with low confidence to be 1 row. The estimated time for this step is 0.03 seconds.
  - 4) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1 (group\_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with low confidence to be 1 row. The estimated time for this step is 0.04 seconds.
  - 5) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.  
-> The contents of Spool 1 are sent back to the user as the result of statement 1.

## Example 6: Join Index Left Outer Joined on Six Tables

The following join index definition left outer joins table t1 with, in succession, tables  $t_2, t_3, t_4, t_5$ , and  $t_6$  on a series of equality conditions made on foreign key-primary key relationships among the underlying base tables.

```
CREATE JOIN INDEX jiout AS
  SELECT a1, b1, c1, c2, d1, d3, e1, e4, f1, g1, h1, i1, j1, j5,
         k1, k6, x1
    FROM t1
   LEFT OUTER JOIN t2 ON a1=a2 AND b1=b2 AND c1=c2
   LEFT OUTER JOIN t3 ON d1=d3
   LEFT OUTER JOIN t4 ON e1=e4 AND f1=f4
   LEFT OUTER JOIN t5 ON g1=g5 AND h1=h5 AND i1=i5 AND j1=j5
   LEFT OUTER JOIN t6 ON k1=k6;
```

Even though the following query references fewer tables than are defined in the join index, you would expect the Optimizer to include join index  $ji\_out$  in its access plan because all the extra outer joins are defined on unique columns and the extra tables are the inner tables in the outer joins.

The bold EXPLAIN report text indicates that the Optimizer does select  $ji\_out$  for the query plan:

```
EXPLAIN SELECT a1, b1, c1, SUM(x1)
  FROM t1, t2
 WHERE a1=a2
   AND b1=b2
   AND c1=c2
 GROUP BY 1, 2, 3;

*** Help information returned. 18 rows.
*** Total elapsed time was 1 second.
```

### Explanation

- ```
-----
1) First, we lock a distinct HONG_JI."pseudo table" for read on a
RowHash to prevent global deadlock for HONG_JI.jiout.
2) Next, we lock HONG_JI.jiout for read.
3) We do an all-AMPS SUM step to aggregate from HONG_JI.jiout by way
of an all-rows scan with no residual conditions, and the grouping
identifier in field 1. Aggregate Intermediate Results are
computed locally, then placed in Spool 3. The size of Spool 3 is
estimated with high confidence to be 2 rows. The estimated time
for this step is 0.03 seconds.
4) We do an all-AMPS RETRIEVE step from Spool 3 (Last Use) by way of
an all-rows scan into Spool 1 (group_amps), which is built locally
on the AMPS. The size of Spool 1 is estimated with high
confidence to be 2 rows. The estimated time for this step is 0.04
seconds.
5) Finally, we send out an END TRANSACTION step to all AMPS involved
in processing the request.
-> The contents of Spool 1 are sent back to the user as the result of
statement 1.
```

## Example 7: Many More Tables Referenced by Join Index Definition Than Referenced by Query

The following join index definition specifies all inner joins on tables *t1*, *t2*, *t3*, *t4*, *t5* and *t6* and specifies equality conditions on all the foreign key-primary key relationships among those tables.

```
CREATE JOIN INDEX ji_in AS
  SELECT a1, b1, c1, c2, d1, d3, e1, e4, f1, g1, g5, h1, i1, j1,
         k1, k6, x1
    FROM t1, t2, t3, t4, t5, t6
   WHERE a1=a2
     AND b1=b2
     AND c1=c2
     AND d1=d3
     AND e1=e4
     AND f1=f4
     AND g1=g5
     AND h1=h5
     AND i1=i5
     AND j1=j5
     AND k1=k6;
```

Even though 6 tables are referenced in the join index definition, and all its join conditions are inner joins, you would expect the Optimizer to include join index *ji\_in* in its query plan for the following query, which only references 2 of the 6 tables, because all the conditions in the join index definition are based on foreign key-primary key relationships among the underlying base tables.

The bold EXPLAIN report text indicates that the Optimizer does select *ji\_in* for the query plan:

```
EXPLAIN SELECT a1, b1, c1, SUM(x1)
   FROM t1, t2
  WHERE a1=a2
    AND b1=b2
    AND c1=c2
 GROUP BY 1, 2, 3;

*** Help information returned. 18 rows.
*** Total elapsed time was 1 second.
```

### Explanation

- 
- 1) First, we lock a distinct HONG\_JI."pseudo table" for read on a RowHash to prevent global deadlock for **HONG\_JI.ji\_in**.
  - 2) Next, we lock **HONG\_JI.ji\_in** for read.
  - 3) We do an all-AMPS SUM step to aggregate from **HONG\_JI.ji\_in** by way of an all-rows scan with no residual conditions, and the grouping identifier in field 1. Aggregate Intermediate Results are computed locally, then placed in Spool 3. The size of Spool 3 is estimated with high confidence to be 2 rows. The estimated time for this step is 0.03 seconds.
  - 4) We do an all-AMPS RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1 (group\_amps), which is built locally on the AMPS. The size of Spool 1 is estimated with high confidence to be 2 rows. The estimated time for this step is 0.04 seconds.
  - 5) Finally, we send out an END TRANSACTION step to all AMPS involved in processing the request.  
-> The contents of Spool 1 are sent back to the user as the result of statement 1.

## Example 8: Using a Join Index That Has an Extra Inner Join In Its Definition

The following example illustrates how the Optimizer uses a join index with an extra inner join when the connections among the tables in its definition are appropriately defined.

Suppose you have the following table definitions:

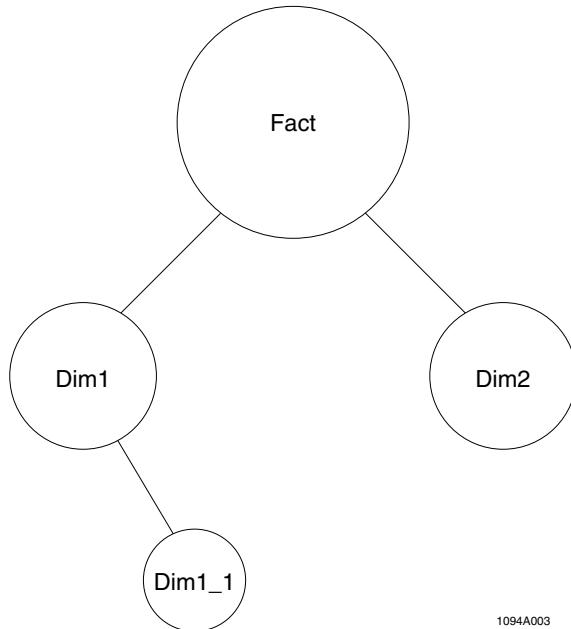
```
CREATE SET TABLE fact (
    f_d1 INTEGER NOT NULL,
    f_d2 INTEGER NOT NULL,
    FOREIGN KEY (f_d1) REFERENCES WITH NO CHECK OPTION dim1 (d1),
    FOREIGN KEY (f_d2) REFERENCES WITH NO CHECK OPTION dim2 (d2))
UNIQUE PRIMARY INDEX (f_d1,f_d2);

CREATE SET TABLE dim1 (
    a1 INTEGER NOT NULL,
    d1 INTEGER NOT NULL,
    FOREIGN KEY (a1) REFERENCES WITH NO CHECK OPTION dim1_1 (d11))
UNIQUE PRIMARY INDEX (d1);

CREATE SET TABLE dim2 (
    d1 INTEGER NOT NULL,
    d2 INTEGER NOT NULL)
UNIQUE PRIMARY INDEX (d2);

CREATE SET TABLE dim1_1 (
    d11 INTEGER NOT NULL,
    d22 INTEGER NOT NULL)
UNIQUE PRIMARY INDEX (d11);
```

The following graphic sketches the dimensional relationships among these tables:



In the following join index definition, the *fact* table is joined with dimension tables *dim1* and *dim2* by foreign key-primary key joins on *fact.f\_d1=dim1.d1* and *fact.f\_d2=dim2.d2*. Dimension table *dim1* is also joined with its dimension subtable *dim1\_1* by a foreign

key-primary key join on `dim1.a1=dim1_1.d11`. A query on the fact and `dim2` tables uses the join index `ji_all` because of its use of foreign key-primary key relationships among the tables:

```
CREATE JOIN INDEX ji_all AS
  SELECT (COUNT(*) (FLOAT)) AS countstar, dim1.d1, dim1_1.d11,
         dim2.d2, (SUM(fact.f_d1) (FLOAT)) AS sum_f1
    FROM fact, dim1, dim2, dim1_1
   WHERE ((fact.f_d1 = dim1.d1)
        AND (fact.f_d2 = dim2.d2))
        AND (dim1.a1 = dim1_1.d11)
  GROUP BY dim1.d1, dim1_1.d11, dim2.d2
 PRIMARY INDEX (d1);
```

Note that the results of the aggregate operations COUNT and SUM are both typed as FLOAT (see “[Restrictions on Join Index Aggregate Functions](#)” on page 591).

As the bold text in the following EXPLAIN report indicates, the Optimizer uses the join index `ji_all` for its query plan in this situation because even though table `fact` is the parent table of both tables `dim1` and `dim2`, those tables are joined with fact on foreign key columns in the join index definition. Similarly, `dim1` is joined to `dim1_1` in the join index definition on a foreign key column.

```
EXPLAIN SELECT d2, SUM(f_d1)
  FROM fact, dim2
 WHERE f_d2=d2
 GROUP BY 1;

*** Help information returned. 18 rows.
*** Total elapsed time was 1 second.
```

#### Explanation

- 
- 1) First, we lock a distinct HONG\_JI.”pseudo table” for read on a RowHash to prevent global deadlock for **HONG\_JI.ji\_all**.
  - 2) Next, we lock **HONG\_JI.ji\_all** for read.
  - 3) We do an all-AMPs SUM step to aggregate from **HONG\_JI.ji\_all** by way of an all-rows scan with no residual conditions, and the grouping identifier in field 1. Aggregate Intermediate Results are computed globally, then placed in Spool 3. The size of Spool 3 is estimated with no confidence to be 3 rows. The estimated time for this step is 0.08 seconds.
  - 4) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1 (group\_amps), which is built locally on the AMPS. The size of Spool 1 is estimated with no confidence to be 3 rows. The estimated time for this step is 0.07 seconds.
  - 5) Finally, we send out an END TRANSACTION step to all AMPS involved in processing the request.  
-> The contents of Spool 1 are sent back to the user as the result of statement 1.

## Examples of Exceptions to the General Rules for Extra Tables in the Join Index Definition

“[Example 1](#)” on page 590 and “[Example 2](#)” on page 591 illustrate cases that are exceptions to the general coverage rules for extra tables in a join index definition (see “[Restriction on Coverage by Join Indexes When a Join Index Definition References More Tables Than a Query](#)” on page 577).

## Example 1

In the following example, table *t9* is the parent table of tables *t7* and *t8*. Generally, this relationship disqualifies a join index from covering any query with fewer tables than are referenced in the definition for that index. However, because *t7* and *t8* are joined on the FK columns (*y7=x8*) in the join index definition, the Optimizer uses the index *ji* to cover the query, as you can see by looking at the bold text in the EXPLAIN report:

```
CREATE SET TABLE t7(
    x7 INTEGER NOT NULL,
    y7 INTEGER NOT NULL,
    z7 INTEGER NOT NULL,
CONSTRAINT r7 FOREIGN KEY (y7)
    REFERENCES WITH NO CHECK OPTION t9 (y9))
PRIMARY INDEX (x7);

CREATE SET TABLE t8(
    x8 INTEGER NOT NULL,
    y8 INTEGER NOT NULL,
    z8 INTEGER NOT NULL,
CONSTRAINT r8 FOREIGN KEY (x8)
    REFERENCES WITH NO CHECK OPTION t9 (x9));

CREATE SET TABLE t9(
    x9 INTEGER NOT NULL UNIQUE,
    y9 INTEGER NOT NULL,
    z9 INTEGER NOT NULL)
UNIQUE PRIMARY INDEX(y9);

CREATE JOIN INDEX ji AS
    SELECT x7, y7, x8, y8, x9, y9
    FROM t7, t8, t9
    WHERE y7=x8
    AND y7=y9
    AND x8=x9;

EXPLAIN SELECT x7, y7, x8, y8
    FROM t7, t8
    WHERE y7=x8
    AND x7>1;

*** Help information returned. 14 rows.
*** Total elapsed time was 1 second.
```

### Explanation

- 
- 1) First, we lock a distinct **HONG\_JI**.*"pseudo table"* for read on a RowHash to prevent global deadlock for **HONG\_JI.ji**.
  - 2) Next, we lock **HONG\_JI.ji** for read.
  - 3) We do an all-AMPS RETRIEVE step from **HONG\_JI.ji** by way of an all-rows scan with a condition of (  
    "**HONG\_JI.ji.x1 > 1**") into Spool 1 (group\_amps), which is built locally on the AMPS. The size of Spool 1 is estimated with no confidence to be 3 rows. The estimated time for this step is 0.06 seconds.
  - 4) Finally, we send out an END TRANSACTION step to all AMPS involved in processing the request.  
-> The contents of Spool 1 are sent back to the user as the result of statement 1. The total estimated time is 0.06 seconds.

## Example 2

As with “[Example 1](#) on page 590”, this example demonstrates how a join index can be used to cover a query when one table in the join index definition is a parent of two others if the tables are joined on foreign key-primary key relationships. *t9* is the parent table of both *t7* and *t10*. But because *t7* and *t10* are joined with *t9* on the same PK column, by transitive closure, *t7* and *t10* are joined on *y7=x10*. The Optimizer does select join index *ji* to cover the query, as the bold text in the EXPLAIN report for the example query demonstrates:

```

CREATE SET TABLE t10(
    x10 INTEGER NOT NULL,
    y10 INTEGER NOT NULL,
    z10 INTEGER NOT NULL,
    CONSTRAINT r10 FOREIGN KEY ( x10 )
        REFERENCES WITH NO CHECK OPTION t9 ( y9 )
    PRIMARY INDEX x10;

CREATE JOIN INDEX ji AS
    SELECT x7, y7, x10, y10, x9, y9
    FROM t7, t10, t9
    WHERE y7=y9
    AND   x10=y9;

EXPLAIN SELECT x7, y7, x10, y10
    FROM t7, t10
    WHERE y7=x10
    AND   x7>1;

*** Help information returned. 14 rows.
*** Total elapsed time was 1 second.

```

### Explanation

- 
- 1) First, we lock a distinct HONG\_JI.“pseudo table” for read on a RowHash to prevent global deadlock for HONG\_JI.ji.
  - 2) Next, we lock HONG\_JI.ji for read.
  - 3) We do an all-AMPS RETRIEVE step from HONG\_JI.ji by way of an all-rows scan with a condition of (“HONG\_JI.ji.x1 > 1”) into Spool 1 (group\_amps), which is built locally on the AMPS. The size of Spool 1 is estimated with no confidence to be 3 rows. The estimated time for this step is 0.06 seconds.
  - 4) Finally, we send out an END TRANSACTION step to all AMPS involved in processing the request.  
-> The contents of Spool 1 are sent back to the user as the result of statement 1. The total estimated time is 0.06 seconds.

## Restrictions on Join Index Aggregate Functions

The following restrictions apply to aggregate functions used to define aggregate join indexes:

- Only the COUNT and SUM functions, in any combination, are valid.
- COUNT DISTINCT and SUM DISTINCT are not valid.
- To avoid overflow problems, always type the COUNT and SUM columns in a join index definition as FLOAT or DECIMAL(38,0) when the data type of the base table column is INTEGER or FLOAT, and as DECIMAL(38,xx) when the data type of the base table column is DECIMAL(NN,xx).

IF you ...	THEN the system ...
do not define an explicit data type for a COUNT or SUM column that has a data type of INTEGER or FLOAT	assigns the FLOAT data type to it.
do not define an explicit data type for a COUNT or SUM column that has a data type of DECIMAL	assigns the DECIMAL(38,xx) data type to it
define a COUNT or SUM column that has a data type of INTEGER as anything other than FLOAT and DECIMAL(38,0)	returns an error and does not create the aggregate join index.
define a COUNT or SUM column that has a data type of DECIMAL as anything other than DECIMAL(38,xx)	returns an error and does not create the aggregate join index.
define a COUNT or SUM column that has a data type of FLOAT as anything other than FLOAT	returns an error and does not create the aggregate join index.

## Restrictions on Sparse Join Index WHERE Clause Predicates

The following restrictions apply to WHERE clause join conditions specified in a join index definition:

- Data types for any columns used in a join condition must be drawn from the same domain because neither explicit nor implicit data type conversions are permitted.
- Multiple join conditions must be connected using the AND logical operator.  
 The OR operator is *not* a valid way to connect multiple join conditions in a join index definition.
- You cannot specify independent inequality WHERE clause join conditions in a join index definition.

The following rules apply to the use of inequality join conditions in a join index definition WHERE clause:

- Inequality join conditions are supported only if they are ANDed to at least one equality join condition.
- Inequality join conditions can be specified only for columns having the same data type in order to enforce domain integrity.
- The only valid comparison operators for an inequality join condition are the following:
  - <
  - <=
  - >
  - >=

The following join index definition is valid because the WHERE clause inequality join condition on o\_totalprice and c\_acctbal it specifies is ANDed with the previous equality join condition on o\_custkey and c\_custkey:

```
CREATE JOIN INDEX ord_cust_idx AS
  SELECT c_name, o_orderkey, o_orderdate
  FROM orders, customer
  WHERE o_custkey = c_custkey
    AND o_totalprice > c_acctbal;
```

The following join index definition is not valid because the WHERE clause inequality join condition has no logical AND relationship with an equality join condition:

```
CREATE JOIN INDEX ord_cust_idx AS
  SELECT c_name, o_orderkey, o_orderdate
  FROM orders, customer
  WHERE o_totalprice > c_acctbal;
```

## Restrictions on Join Index ORDER BY Clauses

The following restrictions apply to ORDER BY clauses specified in a join index definition:

- Sort order is restricted to ASC.  
DESC is not valid.
- Aggregate columns and expressions are *not* permitted as a *column\_name* or *column\_position* specification.
- Supported data types for *column\_name* are restricted to the following:
  - DATE
  - BYTEINT
  - DECIMAL
 

The DECIMAL type is valid only for columns of four or fewer bytes.
  - INTEGER
  - SMALLINT

## Restrictions on Load Utilities

You cannot use FastLoad, MultiLoad, or the Teradata Parallel Transporter operators LOAD and UPDATE to load data into base tables that have join indexes because those indexes are not maintained during the execution of these utilities. If you attempt to load data into base tables with join indexes using these utilities, the load operation aborts and the system returns an error message to the requestor.

To load data into a join-indexed base table, you must drop all defined join indexes on that base table before you can run FastLoad, MultiLoad, or the Teradata Parallel Transporter operators LOAD and UPDATE.

Be aware that you cannot drop a join index to enable MultiLoad or FastLoad batch loads until any requests that access that index complete processing. Requests place locks on any join indexes they access, and the system defers processing of any DROP JOIN INDEX requests against locked indexes until their locks have all been released.

Load utilities like Teradata Parallel Data Pump, BTEQ, and the Teradata Parallel Transporter operators INSERT and STREAM, which perform standard SQL row inserts and updates, *are* supported for join-indexed tables, as are multirow INSERT ... SELECT and MERGE requests.

# Improving Join Index Performance

Depending on the complexity of the joins or the cardinalities of base tables that must be aggregated, join indexes optimize the performance of certain types of workloads.

## Selecting the Primary Index for Join Indexes

As with any other Teradata Database table, you must define a primary index for any join index you create unless it is column-partitioned. While the primary index for any join index must be defined as a NUPI, the column on which it is defined need not be non-unique; in fact, the usual preferences for defining a unique rather than a non-unique column set as the primary index for a base table also apply to join indexes.

Because join indexes are not part of the logical model for a database, and because they are de facto denormalized database objects, there might not be an overriding reason to define one column set over another as the primary index apart from how much more evenly one set might distribute join index table rows than another.

It is usually important to use the join key as the primary index for single-table join indexes, but otherwise there is no compelling theoretical reason to select one unique (or, if not unique, highly singular) column set over another as the primary index. If you create a join index to support a row-partitioned base table, you should consider creating it using row partitioning. This is valid only if the join index is not also row-compressed. You can also create a row-partitioned join index for a base table that does not have row partitioning. It is always possible that such a join index might provide an alternative organization of the data that facilitates access based on partitions. For example, suppose you have a nonpartitioned base table designed to handle efficient joins on its primary index. You might also want to create a row-partitioned join index on the table that optimizes fast row-partition-based access to the data. The rules for creating a row-partitioned join index are generally the same as those for creating a row-partitioned base table.

If a join index is designed to support range queries, you should consider specifying a row-partitioned primary index for it. Note that you cannot specify a row-partitioned primary index for a row-compressed join index.

Join index primary indexes are defined analogously to base table primary indexes: with the CREATE JOIN INDEX statement.

## Selecting Secondary Indexes for Join Indexes

Join indexes are often more high-performing if they have one or more secondary indexes defined on them. The Optimizer adds a join index (even a partially covering join index) to its query plan whenever it can, and just because you define it to have a particular most likely use and access path, there is no reason to believe that the same join index might not also be useful for other, unplanned, queries. The Optimizer will join a base table that is unrelated to a join index *with* that join index if the query plan can be made more cost effective by doing so.

A secondary index on a join index cannot be defined as UNIQUE, even though the column set on which it is defined *is* unique. This rule is enforced because of the way indexes on join index tables are handled internally.

See “CREATE JOIN INDEX” in *SQL Data Definition Language* for further information about creating join indexes and using secondary indexes with them.

## Statistics and Other Demographic Data for Join Indexes

Collect statistics on the indexes of a join index to provide the Optimizer with the information it needs to generate an optimal plan.

The COLLECT STATISTICS (Optimizer Form) statement collects demographics, computes statistics from them, and writes the resulting data into individual entries for each individual base table and join index table on the system.

As far as the Optimizer is concerned, a multitable join index and the base tables it supports are entirely separate entities. You must collect statistics on multitable join index columns separately from the statistics you collect on their underlying base table columns because the column statistics for multitable join indexes and their underlying base tables are *not* interchangeable.

On the other hand, it is generally preferable to collect statistics on the underlying base table of a single-table join index and not directly on the join index columns. See “[Guidelines for Collecting Statistics on Single-Table Join Indexes](#)” on page 596 and “[Collecting Statistics on Base Table Columns Instead of Single-Table Join Index Columns](#)” on page 596 for further details on collecting statistics for single-table join indexes. Note that the derived statistics framework supports bidirectional inheritance of statistics between a non-sparse single-table join index and its underlying base table. See *SQL Request and Transaction Processing* for details.

## Guidelines for Collecting Statistics on Multitable Join Indexes

The guidelines for collecting statistics on the relevant columns are the same as those for any regular join query that is frequently executed or whose performance is critical. The only difference with join indexes is that the join result is persistently stored and maintained by the AMP software without user intervention.

Note the following guidelines for collecting statistics for join indexes.

- To improve the performance of creating a join index and maintaining it during updates, collect statistics on its base tables immediately prior to creating the join index.
- Collect statistics on all the indexes defined on your join indexes to provide the Optimizer with the information it needs to generate an optimal plan.
- Collect statistics on additional join index columns that frequently appear in WHERE clause search conditions, especially when the column is the sort key for a value-ordered join index because the Optimizer can then use that information to more accurately compare the cost of using a NUSI-based access path in conjunction with range or equality conditions specified on the sort key column.

- In general, there is no benefit in collecting statistics on a join index for joining columns specified in the join index definition itself. Statistics related to these columns should be collected on the underlying base tables rather than on the join index.

The only time you gain an advantage by collecting statistics on a join column of the join index definition is when that column is used as a join column to other base tables in queries where the join index is expected to be used in the Optimizer query plan.

## Guidelines for Collecting Statistics on Single-Table Join Indexes

For most applications, you should collect the statistics on base table columns rather than on single-table join index columns. See “[Collecting Statistics on Base Table Columns Instead of Single-Table Join Index Columns](#)” on page 596 for further information.

You can collect and drop statistics on the primary index columns of a single-table join index using an INDEX clause to specify the column set. For example, suppose a join index has the following definition.

```
CREATE JOIN INDEX OrdJIdx8 AS
  SELECT o_custkey, o_orderdate
  FROM Orders
  PRIMARY INDEX (o_custkey, o_orderdate)
  ORDER BY (o_orderdate);
```

Then you can collect statistics on the index columns as a single object as shown in the following example.

```
COLLECT STATISTICS ON OrdJIdx8 INDEX (o_custkey, o_orderdate);
```

Note that you can *only* use columns specified in a PRIMARY INDEX clause definition if you use the keyword INDEX in your COLLECT STATISTICS statement.

A poorer choice would be to collect statistics using the column clause syntax. To do this, you must perform two separate statements.

```
COLLECT STATISTICS ON OrdJIdx8 COLUMN (o_custkey);
```

```
COLLECT STATISTICS ON OrdJIdx8 COLUMN (o_orderdate);
```

It is always better to collect the statistics for a multicolumn index on the index itself rather than individually on its component columns because its selectivity is much better when you collect statistics on the index columns as a set.

For example, a query with a predicate like WHERE x=1 AND y=2 is better optimized if statistics are collected on INDEX (x,y) than if they are collected individually on column x and column y.

The same restrictions hold for the DROP STATISTICS statement.

## Collecting Statistics on Base Table Columns Instead of Single-Table Join Index Columns

The Optimizer substitutes base table statistics for single-table join index statistics when no demographics have been collected for its single-table indexes. Because of the way single-table join index columns are built, it is generally best *not* to collect statistics directly on the index

columns and instead to collect them on the corresponding columns of the base table. This optimizes both system performance and disk storage by eliminating the need to collect the same data redundantly.

You might need to collect statistics on single-table join index columns instead of their underlying base table columns if you decide not to collect the statistics for the relevant base table columns for some reason. In this case, you should collect statistics directly on the corresponding single-table join index columns.

Note that the derived statistics framework can use bidirectional inheritance of statistics between base tables and their underlying non-sparse single-table join indexes, so the importance of collecting statistics on base table columns rather than non-sparse single-table join index columns is no longer as important as it once was (see *SQL Request and Transaction Processing* for details).

## Guidelines for Collecting Statistics On Single-Table Join Index Columns

The guidelines for selecting single-table join index columns on which to collect statistics are similar to those for base tables. The primary factor to consider in all cases is whether the statistics provide better access plans. If they do not, consider dropping them. If the statistics you collect produce *worse* access plans, then you should always report the incident to Teradata support personnel.

When you are considering collecting statistics for a single-table join index, it might help to think of the index as a special kind of base table that stores a derived result. For example, any access plan that uses a single-table join index must access it with a direct probe, a full table scan, or a range scan. With this in mind, consider the following factors when deciding which columns to collect statistics for.

- Always consider collecting statistics on the primary index. This is particularly critical for accurate cardinality estimates.
- 

IF an execution plan might involve ...	THEN collect statistics on the ...
search condition keys	column set that constitutes the search condition predicate.
joining the single-table index with another table	join columns to provide the Optimizer with the information it needs to best estimate the cardinalities of the join.

•	IF a single-table join index is defined ... with an ORDER BY clause	THEN you should collect statistics on the ... order key specified by that clause.
	without an ORDER BY clause and the order key column set from the base table is not included in the <i>column_name_1</i> list	order key of the base table on which the index is defined. This action provides the Optimizer with several essential baseline statistics.

- If a single-table join index column appears frequently in WHERE clause predicates, you should consider collecting statistics on it as well, particularly if that column is the sort key for a value-ordered single-table join index.

## Join Index Storage

The storage organization for join indexes supports a row compressed format to reduce storage space.

If you know that a join index contains groups of rows with repeating information, then its definition DDL can specify repeating groups, indicating the repeating columns in parentheses. The column list is specified as two groups of columns, with each group stipulated within parentheses. The first group contains the non-repeating columns and the second group contains the repeating columns.

You can elect to store join indexes in value order, ordered by the values of a 4-byte column. Value-ordered storage provides better performance for queries that specify selection constraints on the value ordering column. For example, suppose a common task is to look up sales information by sales date. You can create a join index on the sales table and order it by sales date. The benefit is that queries that request sales by sales date only need to access those data blocks that contain the value or range of values that the queries specify.

### Physical Join Index Row Compression

Compression refers to a logical row compression in which multiple sets of nonrepeating column values are appended to a single set of repeating column values. This allows the system to store the repeating value set only once, while any nonrepeating column values are stored as logical segmental extensions of the base repeating set.

A physical join index row has:

- A required fixed part, specified by the *column\_1* list, that is stored only once.  
When a *column\_2* list is specified, the columns specified in the *column\_1* list are the data that is compressed.
- An optional repeated part, specified a the *column\_2* list, that is stored as often as needed.

For example, if a logical join result has  $n$  rows with the same fixed part value, then there is one corresponding physical join index row that includes  $n$  repeated parts in the physical join index. A physical join index row represents one or more logical join index rows. The number of logical join index rows represented in the physical row depends on how many repeated values are stored.

Compression is only done on rows inserted by the same INSERT statement. This means that newly inserted rows are *not* added as logical rows to existing compressed rows.

When the number of repeated values associated with a given fixed value exceeds the system row size limit, the join index row is automatically split into multiple physical rows, each having the same fixed value but different lists of repeated values. Note that a given logical join index result row cannot exceed the system 64KB row size limit.

## Guidelines for Using Row Compression With Join Index Columns

A column set with a high number of distinct values cannot be row compressed because it rarely (and in the case of a primary key, never) repeats. Other column sets, notably foreign key and status code columns, are generally highly non-distinct: their values repeat frequently. Row compression capitalizes on the redundancy of frequently repeating column values by storing them once in the fixed part of the index row along with multiple repeated values in the repeated part of the index row.

Typically, primary key table column values are specified as the repeating part and foreign key table columns are specified in the fixed part of the index definition.

[“Using Outer Joins to Define Join Indexes” on page 559](#) and [“Creating Join Indexes Using Outer Joins” on page 561](#) are examples of how you can use row compression to take advantage of repeated column values.

A row-compressed join index might be used for a query when the join index would be used if it were not row compressed. Check the EXPLAIN for a query to see if join indexes are being used.

## Determining the Space Overhead for a Hash or Join Index

The ROWID, if included, is always 10 bytes long for hash indexes. For join indexes, the space overhead for the ROWID is 10 bytes for 2-byte partitioning and 16 bytes for 8-byte partitioning.

The following equation provides an estimate of the space overhead required for an uncompressed hash index. Double the result if you define fallback on the index.

$$\text{Uncompressed hash or join index size} = N \times (F + O)$$

where:

This symbol ...	Defines the ...
N	cardinality of the base table.

This symbol ...	Defines the ...
F	length of the columns, including base row identifier information.
O	<p>row overhead.</p> <p>Assume the following row overhead for a join index depending on whether it is partitioned or not.</p> <ul style="list-style-type: none"> <li>• 12 bytes for an nonpartitioned join index</li> <li>• 14 bytes for a join index with 2-byte partitioning</li> <li>• 20 bytes for a join index with 8-byte partitioning</li> </ul>

The following equation provides an estimate of the space overhead required for a row-compressed hash or join index. Double the result if you define fallback on the index.

$$\text{Row-compressed hash or join index size} = U \times (F + O + (R \times A))$$

where:

This symbol...	Defines the ...
F	<p>length of the fixed column <i>column_name</i>.</p> <p>For a hash index, this length includes the base row identifier information in the fixed column.</p>
R	<p>length of a single repeating column <i>column_name</i>.</p> <p>For a hash index, this length includes the base row identifier information in the repeating column.</p>
A	average number of repeated columns for a given value in <i>column_1_name</i> .
U	number of unique values in the specified <i>column_1_name</i> .
O	<p>row overhead.</p> <p>Assume the following row overhead for a hash or join index depending on whether it is partitioned or not.</p> <ul style="list-style-type: none"> <li>• 12 bytes for an nonpartitioned hash or join index</li> <li>• 14 bytes for a join index with 2-byte partitioning</li> <li>• 20 bytes for a join index with 8-byte partitioning</li> </ul>

## Considerations for Measuring Disk Space

Keep the following rules of thumb in mind whenever you perform these calculations.

- Implement your measurements over a very large number of rows to avoid distorting the figures with table overhead issues.
- An INTEGER column uses 4 bytes.

A SUM(INTEGER) column uses 8 bytes because the system always casts it to FLOAT.

- If you have not explicitly specified a COUNT column in your aggregate join index definition, add 4 bytes to the definition to account for the required COUNT column.

## Example Measurement 1

Table A and Table B both have 60 million 100-byte rows. An aggregate join index with two INTEGER columns and one SUM(INTEGER) column.

From these figures, the computed size of the aggregate join index is as follows:

$$\text{Row size} = (2 \times 4 \text{ INTEGER bytes}) + 4 \text{ COUNT bytes} + 8 \text{ SUM bytes} + 14 \text{ overhead bytes} = 34 \text{ bytes}$$

FOR this type of join index ...	THERE is one join index row ...
non-aggregate	for every matching row in the base table.
aggregate	per group of rows in the join of the base tables.

The permanent space specification is as follows:

Current Permanent Space (Bytes)	Peak Permanent Space (Bytes)	Maximum Permanent Space (Bytes)
2,041,595,904	2,041,595,904	0

Using these figures, you can compute a measured row size:

$$\text{Row size} = \frac{2\,041\,595\,904}{60\,000\,000} = 34.03 \text{ bytes}$$

Because the measured and computed disk spaces are identical within a narrow confidence interval, it is safe to conclude that the disk space formula is accurate for an aggregate join index.

## Example Measurement 2

This example examines the size of a simple join index created for the same tables used in “[Example Measurement 1](#)” on page 601. There is a savings of 8 bytes because there is no COUNT column and no SUM(INTEGER) column.

The computed row size is as follows:

$$\text{Row size} = 34 \text{ bytes} - (4 \text{ INTEGER bytes} + 4 \text{ SUM(INTEGER) bytes}) = 26 \text{ bytes}$$

The permanent space specification is as follows.

Current Permanent Space (Bytes)	Peak Permanent Space (Bytes)	Maximum Permanent Space (Bytes)
1,561,216,000	1,561,216,000	0

Using these figures, you can compute a measured row size.

$$\text{Row size} = \frac{1\,561\,216\,000}{60\,000\,000} = 26.02 \text{ bytes}$$

As with “[Example Measurement 1](#)” on page 601, the measured and computed disk spaces are identical, and you can conclude that the disk space formula is accurate for a simple join index.

## Value-Ordered Storage of Join Index Rows

The distribution of the subtable rows for a join index across the AMPs is controlled by its NUPI. By default, join index subtable rows are sorted locally on each AMP by the row hash value of the NUPI column set. You can also specify that rows be stored in value-order.

### Value Ordering and Range Queries

You can specify value ordering by means of the optional ORDER BY clause in the join index definition. Sorting a join index by values, as opposed to row hash values, is especially useful for range queries involving the sort key. Value ordering is limited to a single numeric or DATE columns with a size of 4 bytes or less.

For example, the following join index rows are hash-distributed using *c\_name* and are value-ordered on each AMP using *c\_custkey* as the sort key.

```
CREATE JOIN INDEX OrdCustIdx AS
  SELECT (o_custkey
          ,c_name)
         ,
        (o_status
         ,o_date
         ,o_comment)
  FROM Orders LEFT JOIN Customer
  ON o_custkey = c_custkey
  ORDER BY o_custkey
  PRIMARY INDEX (c_name);
```

## Hash Indexes

Hash indexes are file structures that share properties in common with both single-table join indexes and secondary indexes (see “[Comparison of Hash Indexes and Single-Table Join Indexes](#)” on page 603).

From an architectural perspective, the incorporation of auxiliary structures as a transparently embedded element of the hash index column set is what most distinctly distinguishes hash indexes from single-table join indexes. These auxiliary structures are components of the base table, and are added to the hash index definition by default if they are not explicitly declared by the CREATE HASH INDEX column set definition. Because it is not clear what the default auxiliary structures Teradata Database uses when it creates a hash index, you should always consider creating an equivalent single-table join index in preference to a hash index. Also,

multi-value compression from the base table may be carried over to a join index, but it is not carried over to a hash index.

If the columns you specify for the hash index column set duplicate the default auxiliary structure columns, then those columns are not added redundantly. The auxiliary structures provide indexed access to base table rows.

If you do not specify a partition key explicitly with the BY clause of the CREATE HASH INDEX statement, then the system adds this auxiliary pointer data to the hash index rows automatically and then uses it to partition them.

## Comparison of Hash Indexes and Single-Table Join Indexes

The reasons for using hash indexes are similar to those for using single-table join indexes. Not only can hash indexes optionally be specified to be distributed in such a way that their rows are AMP-local with their associated base table rows, they can also provide a transparent direct access path to those base table rows to complete a query only partially covered by the index. This facility makes hash indexes somewhat similar to secondary indexes in function. Hash indexes are also useful for covering queries so that the base table need not be accessed at all.

The most apparent external difference between hash and single-table join indexes is in the syntax of the SQL statements used to create them. The syntax for CREATE HASH INDEX is similar to that for CREATE INDEX. As a result, it is simpler to create a hash index than to create a functionally comparable single-table join index.

The following list summarizes the similarities of hash and single-table join indexes.

- The primary function of both is to improve query performance.
- Both are maintained automatically by the system when the relevant columns of their base table are updated by a DELETE, INSERT, or UPDATE statement.
- Both can be the object of any of the following SQL statements.
  - COLLECT STATISTICS (Optimizer Form)
  - DROP STATISTICS
  - HELP INDEX
  - SHOW HASH INDEX
- Both receive their space allocation from permanent space and are stored in distinct tables.
- Both can be hash- or value-ordered. You must drop and rebuild all value-ordered (but not hash-ordered) hash and join indexes after you run the Reconfig utility (see *Support Utilities*).
- Both can be row compressed.
- Both can be FALBACK protected.
- Both can be used to transform a complex expression into a simple index column. This transformation permits you to collect statistics on the expression, which the Optimizer can then use to make single-table cardinality estimates for a matching complex column predicate specified on the base table and for mapping a query expression that is identical to an expression defined in the join index, but is found within a non-matching predicate (see *SQL Request and Transaction Processing* for details).

- Neither can be queried or directly updated.
- A hash index cannot have a partitioned primary index, but a single-table join index can.
- A hash index must have a primary index, but a single-table join index can be created with or without a primary index if it is column-partitioned.
- A hash index cannot be column-partitioned, but a single-table join index can be column-partitioned.
- Neither can be used to partially cover a query that contains a TOP *n* or TOP *m* PERCENT clause.
- Neither can be implemented with row compression if they specify a UDT in their select list because both create an internal column1 and column2 index when compressed.
- Neither can be defined using the system-derived PARTITION column.
- Both share the same restrictions for use with the MultiLoad, FastLoad, and Archive/Recovery utilities.

The following table summarizes the important differences between hash and join indexes.

Hash Index	Join Index
Indexes one table only.	Can index multiple tables.  This is not true for column-partitioned join indexes, which can only index a single table.
A logical row corresponds to one and only one row in its referenced base table.	A logical row can correspond to either of the following, depending on how the join index is defined: <ul style="list-style-type: none"> <li>One and only one row in the referenced base table.</li> <li>Multiple rows in the referenced base tables.</li> </ul>
Column list cannot specify aggregate or ordered analytical functions.	Select list can specify aggregate functions.
Can specify a UDT column in its column list.	Can only specify a UDT in its select list if it is not row-compressed.
Cannot have a non-unique secondary index.	Can have a non-unique secondary index.
Supports transparently added, system-defined primary index columns that point to the underlying base table rows.	Does not add underlying base table row pointers implicitly.  Pointers to underlying base table rows can be created explicitly by defining one element of the column list using the keyword ROWID.  Note that you can only specify ROWID in the outermost SELECT of the CREATE JOIN INDEX statement. See “CREATE JOIN INDEX” in <i>SQL Data Definition Language Detailed Topics</i> .
Cannot be specified for NoPI or column-partitioned base tables.	Can be specified for both NoPI and column-partitioned base tables.

Hash Index	Join Index
Cannot be column partitioned.	Can be column partitioned.
Cannot be row partitioned.	Primary index of uncompressed row forms and column-partitioned join indexes, can be row partitioned.
Cannot be defined on a table that also has triggers.	Can be defined on a table that also has triggers.
Column multi-value compression, if defined on a referenced base table, is <i>not</i> added transparently by the system and cannot be specified explicitly in the hash index definition.	Column multi-value compression, if defined on a referenced base table, is added transparently by the system with no user input, but cannot be specified explicitly in the join index definition.
Index row compression is added transparently by the system with no user input.	Index row compression, if used, must be specified explicitly in the CREATE JOIN INDEX request by the user.

It is possible to define a join index that has nearly the identical functionality to a hash index. The only essential differences between hash and join indexes is their respective DDL creation syntax. Otherwise, the functionality of hash indexes is a proper subset of the functionality of join indexes.

## Summary of Hash Index Functions

A hash index always has at least one of the following functions.

- Replicates all, or a vertical subset, of a single base table and partitions its rows with a user-specified partition key column set, such as a foreign key column to facilitate joins of very large tables by hashing them to the same AMP.
- Provides an access path to base table rows to complete partial covers.

The AMP software updates hash indexes automatically, so the only task a DBA must perform is to keep the statistics on hash index or base table columns current (see “[Collecting Statistics on Hash Index Columns](#)” on page 608).

## Similarities of Hash Indexes to Base Tables

In many respects, a hash index is identical to a base table.

For example, you perform any of the following statements against a hash index:

- COLLECT STATISTICS (Optimizer Form)
- DROP HASH INDEX
- DROP STATISTICS (Optimizer Form)
- HELP HASH INDEX
- SHOW HASH INDEX

You *cannot* do the following things with hash indexes:

- Query or update hash index rows.

For example, if ordCustHdx is a hash index, then the following query is not legal:

```
SELECT o_status, o_date, o_comment
FROM ordCustHdx;
```

- Store and maintain arbitrary query results.
- Create secondary indexes on its columns.
- Create them on either a NoPI or column-partitioned table.

## Similarities of Hash Indexes to Secondary Indexes

Hash indexes are file structures that can be used either to resolve queries by accessing the index instead of its underlying base table or to enhance access performance when they do not cover a query by providing a secondary access path to requested base table rows. They can either substitute *for* or point *to* base table rows.

Because of these properties, hash indexes are useful for queries where the index structure contains all the columns referenced by a query, thereby *covering* the query, and making it possible to retrieve the requested data from the index rather than accessing its underlying base tables. For obvious reasons, an index with this property is often referred to as a covering index.

Hash indexes put in double duty by also providing pointers to base table rows to facilitate their access in situations where the index does *not* cover the query. This application of hash indexes is similar to how the Optimizer uses secondary indexes when it creates its access plans.

Because the distribution of rows to AMPs of hash index rows is under user control through the specification of an explicit partition key in the CREATE HASH INDEX statement, rows can be distributed in various ways, depending on the requirements of an application.

Distribution of secondary index rows, on the other hand, is not under user control. Because of this, they are less adaptable to the specific requirements of an individual application than hash indexes.

Both secondary and hash indexes provide access paths to base table rows and can be selected by the Optimizer to cover SQL queries. Both also share a similar DDL syntax that is very different from the syntax used to create single-table join indexes.

The key difference between hash and secondary indexes is that the hash index partition key is user-selectable, which often makes it more useful for processing queries.

## Hash Index Applications

Hash indexes are useful for queries where the index table contains the columns referenced by a query, thereby allowing the Optimizer to cover it by planning to access the index rather than its underlying base table. An index that supplies all the table columns requested by a query is said to cover that table for that query and, for obvious reasons, is referred to as a covering index. Note that query covering is not restricted to hash indexes: other indexes can also cover queries.

Hash indexes can be defined on a table in place of secondary indexes. This usage makes more sense when you distribute the hash index row to the AMPs so that it facilitates a wider range of query processing than a secondary index might. Because hash indexes can potentially carry more of an update burden than secondary indexes, you should not define them on a table

when a secondary index would serve the same intended function. Keep in mind that this depends on how the updates are done. For example, a single row update might be faster with a secondary index, but an INSERT ... SELECT might be faster with a hash index.

## Compression of Hash Index Rows

Compression refers to a logical *row* compression in which multiple sets of non-repeating column values are appended to a single set of repeating column values. This allows the system to store the repeating value set only once, while any non-repeating column values are stored as logical segmental extensions of the base repeating set.

When the following is true, the system automatically compresses the rows of a hash index:

- The ORDER BY column set is specified in the *column\_1* list and none of the columns in the order key is unique.
- The primary index columns are specified in the *column\_1* list.

As the following table indicates, the primary index columns for a hash index are always part of the *column\_1* list, whether specified explicitly or not.

IF you create the index ...	THEN the primary index columns ...
without a BY clause	are part of the <i>column_1</i> list by default.
with a BY clause	must be part of the <i>column_1</i> list because all columns specified in the BY clause column list must also be specified in the <i>column_1</i> list.

Rows having the same values for the order key are compressed into a single physical row having fixed and repeating parts. If the columns do not fit into a single physical row, they spill over to additional physical rows as is necessary.

The fixed part of the row is made up of the explicitly-defined columns that define the *column\_1* list. The repeating part is composed of the remaining, implicitly-defined columns.

The system only compresses row sets together if they are inserted by the same INSERT statement. This means that rows that are subsequently inserted are *not* appended as logical rows to existing compressed row sets, but rather are compressed into their own self-contained row sets.

## Fallback With Hash Indexes

You can define fallback for hash indexes. The criteria for deciding whether to define a hash index with fallback are similar to those used for deciding whether to define fallback on base tables.

If you do not define fallback for a hash index and an AMP is down, then the following additional factors become critical.

- The hash index cannot be used by the Optimizer to solve any queries that it covers.
- The base tables on which the hash index is defined cannot be updated.

## Restrictions for NoPI Tables

You cannot create a hash index on a NoPI or a column-partitioned table because hash indexes are defined with transparently added, system-defined primary index columns from the underlying base table primary index that point to the underlying base table rows, and NoPI tables do not have a primary index.

## Restrictions on Load Utilities

You cannot use FastLoad, MultiLoad, or the Teradata Parallel Transporter operators LOAD and UPDATE to load data into base tables that have hash indexes because those indexes are not maintained during the execution of these utilities. If you attempt to load data into base tables with hash indexes using these utilities, the load operation aborts and the system returns an error message to the requestor.

To load data into a hash-indexed base table, you must drop all defined hash indexes on the table before you can run FastLoad, MultiLoad, or the Teradata Parallel Transporter operators LOAD and UPDATE.

Load utilities like Teradata Parallel Data Pump, BTEQ, and the Teradata Parallel Transporter operators INSERT and STREAM, which perform standard SQL row inserts and updates, *are supported for hash-indexed tables.*

## Restriction on Partial Coverage of Queries Containing a TOP *n* or TOP *m* PERCENT Clause

A hash index cannot be used to partially cover a query that specifies the TOP *n* or TOP *m* PERCENT option.

## Compression of Hash Indexes at the Block Level

Because hash indexes are primary tables, they can be compressed at the block level. See “[Data Block](#)” on page 248, “[Reserved Query Bands for Managing the Block-Level Compression and Storage Temperature of Newly Loaded Data](#)” on page 694, and “[Compression Types Supported by Teradata Database](#)” on page 695 for more information about data block compression.

## Further Information

Consult *SQL Data Definition Language* for more detailed information on creating and using hash indexes to enhance the performance of your database applications.

# Collecting Statistics on Hash Index Columns

When there are no statistics for a hash index, the Optimizer uses the statistics of the corresponding base table rows, just as it does for a single-table join index.

The guidelines for selecting hash index columns on which to collect statistics are similar to those for base tables and join indexes. The primary factor to consider in all cases is whether

the statistics provide better access plans. If they do not, consider dropping them. If the statistics you collect produce *worse* access plans, you should always report the incident to Teradata support personnel.

When you consider collecting statistics for a hash index, it might help to think of the index as a special kind of base table that stores a derived result. For example, any access plan that uses a hash index must access it with a direct probe, a full table scan, or a range scan.

With this in mind, consider the following when deciding which columns to collect statistics for.

•

IF an execution plan might involve ...	THEN collect statistics on the ...
search condition keys	column set that constitutes the search condition predicate.
joining the hash index with another table	join columns to provide the Optimizer with the information it needs to best estimate the cardinalities of the join.

•

IF a hash index is defined ...	THEN you should collect statistics on the ...
with a BY clause or ORDER BY clause (or both)	primary index and order keys specified by those clauses.
without a BY clause	primary index column set of the base table on which the index is defined.
without an ORDER BY clause and the order key column set from the base table is not included in the <i>column_name_1</i> list	order key of the base table on which the index is defined.  This action provides the Optimizer with several essential baseline statistics, including the cardinality of the hash index.

- If a hash index column appears frequently in WHERE clause predicates, you should consider collecting statistics on it as well, particularly if that column is the sort key for a value-ordered hash index.
- Consider collecting statistics on the base table columns that are also part of the hash index rather than collecting statistics on the hash index itself.

## General Guidelines

You should collect statistics on appropriate columns of a hash index frequently just as you would for any base table or join index. *For most applications, you should collect the statistics on base table columns rather than on hash index columns.* See “[Collecting Statistics on Base Table Columns Instead of Hash Index Columns](#)” on page 610 for further information.

When you create a hash index with a BY clause, you can collect and drop statistics on those columns using an INDEX clause to specify the column set. For example, suppose a hash index has the following definition.

```
CREATE HASH INDEX OrdHIdx8 (o_custkey, o_orderdate) ON orders
  BY (o_custkey, o_orderdate)
    ORDER BY (o_orderdate);
```

Then you can collect statistics on the partitioning columns as shown in the following example.

```
COLLECT STATISTICS ON OrdHIdx8 INDEX (o_custkey, o_orderdate);
```

Note that you can *only* use columns specified in a BY clause definition if you use the keyword INDEX in your COLLECT STATISTICS statement.

A poorer choice would be to collect statistics using the column clause syntax. To do this, you must perform two separate statements.

```
COLLECT STATISTICS ON OrdHIdx8 COLUMN (o_custkey);
```

```
COLLECT STATISTICS ON OrdHIdx8 COLUMN (o_orderdate);
```

It is always better to collect the statistics for a multicolumn index on the index itself rather than individually on its component columns because its selectivity is much better when you collect statistics on the index columns as a set.

For example, a query with a predicate like WHERE `x=1 AND y=2` is better optimized if statistics are collected on INDEX (x,y) than if they are collected individually on column x and column y.

The same restrictions hold for the DROP STATISTICS statement.

## Collecting Statistics on Base Table Columns Instead of Hash Index Columns

The Optimizer substitutes base table statistics for hash index statistics when no demographics have been collected for its hash indexes. Because of the way hash index columns are built, it is generally best not to collect statistics directly on the index columns and instead to collect them on the corresponding columns of the base table. This optimizes both system performance and disk storage by eliminating the need to collect the same data redundantly.

Collect statistics on a hash index or non-sparse join index directly in the following scenarios:

- If you decide not to collect the statistics for the relevant base table columns for some reason, then you should collect them directly on the corresponding hash index columns.
- If the hash index is row or column partitioned, collect single-column and multicolumn PARTITION statistics directly on the hash index.
- Collect SUMMARY statistics directly on the hash index as the summary attributes such as database block size, row size, etc. differ from the underlying base table.

# Hash Index Definition Restrictions

This topic lists several restrictions on hash index definitions. For complete details about hash index syntax, see the documentation for CREATE HASH INDEX in *SQL Data Definition Language*.

## Restrictions for NoPI Tables

You cannot create a hash index on a NoPI or column-partitioned table because hash indexes are defined with transparently added, system-defined primary index columns from the underlying base table primary index that point to the underlying base table rows, and NoPI tables do not have a primary index.

## Restrictions on Number of Hash Indexes Defined Per Base Table

The maximum number of secondary, hash, and join indexes that can be defined for a table is 32, in any combination. This includes the system-defined unique secondary indexes used to implement PRIMARY KEY and UNIQUE constraints.

Each composite NUSI that specifies an ORDER BY clause counts as 2 consecutive indexes in this calculation (see “Importance of Consecutive Indexes for Value-Ordered NUSIs” on [page 485](#)). You cannot define hash, or any other, indexes on global temporary trace tables. See “CREATE GLOBAL TEMPORARY TRACE TABLE” in *SQL Data Definition Language Detailed Topics*.

For example, suppose you have 4 tables, each with multiple secondary, hash, and join indexes defined on them.

- *Table\_1* has 32 secondary indexes and no hash or join indexes.
- *Table\_2* has 16 secondary indexes, no hash indexes, and 16 join indexes.
- *Table\_3* has 10 secondary indexes, 10 hash indexes, and 12 join indexes.
- *Table\_4* has no secondary or hash indexes, but has 32 join indexes.

Each of these combinations is valid, but they all operate at the boundaries of the defined limits.

Note that if any of the secondary indexes defined on tables 1, 2, or 3 is a NUPI defined with an ORDER BY clause, the defined limits are exceeded, and the last index you attempt to create on the table will fail. Because each NUPI defined with an ORDER BY clause counts as 2 consecutive indexes in the count against the maximum of 32 per table, you could define only 8 of them on *Table\_2*, for example, if you also defined 16 join indexes on the table.

## Restrictions on Number of Columns Per Referenced Base Table

The maximum number of columns that can be specified in a hash index is 64.

## Restrictions on the Data Type of Hash Index Columns

You cannot include columns with the following data types in a hash index definition:

- XML
- BLOB
- CLOB
- BLOB-based UDTs
- CLOB-based UDTs
- XML-based UDTs
- ARRAY
- VARRAY
- Period
- Geospatial
- JSON

## Restrictions on the Use of the System-Derived PARTITION[#Ln] Column in a Hash Index Definition

You cannot use the system-derived PARTITION[#Ln] column in the definition of a hash index.

## Restriction on Number of Hash Indexes Selected Per Base Table

The Optimizer can use several hash indexes for a single query, selecting one or more multitable join indexes as well as additional hash indexes for its join plan. The hash indexes selected depend on the structure of the query, and the Optimizer might not choose all applicable hash indexes for the plan. Always examine your EXPLAIN reports to determine which hash indexes *are* used for the query plans generated for your queries. If a hash index you think should have been used by a query was not included in the query plan, try restructuring the query and then EXPLAIN it once again.

The join planning process selects a multitable join index to replace any individual table in a query when the substitution further optimizes the query plan. For each such table replaced in the join plan by a multitable join index, as many as two additional hash indexes can also be added if their inclusion reduces the size of the relation to be processed, provides a better distribution, or offers additional covering.

The limit on the number of hash indexes substituted per individual table in a query is enforced to limit the number of possible combinations and permutations of table joins in the Optimizer search space during its join planning phase. The rule helps to ensure that the optimization is worth the effort: in other words, that the time spent generating the query plan does not exceed the accrued performance enhancement.

# Hash and Join Index Interactions With Other Teradata Database Systems and Features

This topic describes how hash and join indexes interact with the following Teradata Database systems and features.

## Summary

The following table summarizes the Teradata Database features that do not support hash or join indexes.

Feature	Reason Not Supported	Recommended Workaround
Triggers	Triggers are handled by the Resolver, while hash indexes are handled by the Optimizer.	Do not define triggers and hash indexes on the same base tables.
Permanent journal recovery	Recovery process does not rebuild hash and join indexes.	Rebuild hash and join indexes after the permanent journal recovery completes.
MultiLoad	Utilities do not maintain hash and join indexes.	<ol style="list-style-type: none"><li>1 Ensure that no queries are running against tables that use the hash or join index to be dropped.</li><li>2 Drop hash and join indexes before loading or restoring the base tables.</li><li>3 Recreate hash and join indexes after loading or restoring the base tables.</li></ol>
FastLoad		
Teradata Parallel Transporter LOAD and UPDATE operators		
Archive/Recovery		
<ul style="list-style-type: none"><li>• NoPI tables</li><li>• Column-partitioned tables</li></ul>	Hash indexes automatically add the primary index of their underlying base tables to their definition, and NoPI tables have no primary index.	None.

## Triggers

You cannot define triggers and hash indexes on the same table; however, you *can* define triggers and join indexes on the same table.

Whichever feature is defined first for a table blocks the other from being created and returns an error message to the requestor.

## Permanent Journal Recovery

You can use ROLLBACK or ROLLFORWARD utility commands to recover base tables with hash or join indexes defined on them; however, the indexes are not rebuilt during the recovery process.

Instead, any such hash or join index is marked as non-valid, and you must drop and recreate it before the Optimizer can use it again in a query plan.

When a hash or join index has been marked not valid, the SHOW HASH INDEX and SHOW JOIN INDEX statements display a special status message to inform you that the index has been so marked.

## Load Utilities

You cannot use FastLoad, MultiLoad, or the Teradata Parallel Transporter operators LOAD and UPDATE to load data into base tables that have hash or join indexes because those indexes are not maintained during the execution of these utilities (see *Teradata FastLoad Reference*, *Teradata MultiLoad Reference*, and *Teradata Parallel Transporter Reference* for details).

If you attempt to load data into base tables with hash or join indexes using these utilities, an error message returns and the load does not continue.

To load data into hash- or join-indexed base table, you must drop all defined hash or join indexes before you can run FastLoad, MultiLoad, or the Teradata Parallel Transporter operators LOAD and UPDATE.

Load utilities like BTEQ, Teradata Parallel Data Pump, and the Teradata Parallel Transporter operators INSERT and STREAM, which perform standard SQL row inserts and updates, are supported for hash- and join-indexed tables (see *Basic Teradata Query Reference*, *Teradata Parallel Data Pump Reference*, and *Teradata Parallel Transporter Reference* for details).

You cannot drop a hash join or index to enable batch data loading by utilities such as MultiLoad and FastLoad as long as queries are running that access that index. Each such query places a lock on the index while it is running, so it blocks the completion of any DROP JOIN INDEX or DROP HASH INDEX transactions until the lock is removed.

Furthermore, as long as a DROP JOIN INDEX or DROP HASH INDEX transaction is running, batch data loading jobs against the underlying tables of the index cannot begin processing because of the EXCLUSIVE locks DROP JOIN INDEX and DROP HASH INDEX place on the base table set that defines them.

## Teradata Parallel Data Pump

You *can* use the Teradata Parallel Data Pump utility, which performs standard SQL row inserts and updates, to load data into base tables that have a hash or join index defined on them because those indexes are properly maintained during its execution. See *Teradata Parallel Data Pump Reference* for details.

## Archive and Recovery

You cannot use the Teradata Archive/Recovery utility to archive or restore hash and join indexes, though archiving is supported for a base table or database that has associated hash or join indexes defined on it.

Before you can restore such a base table or database, you must drop the hash and join indexes that are associated with it, and then recreate the dropped indexes after the recovery operation completes. Otherwise, the Optimizer cannot use those indexes for a query plan.

See *Teradata Archive/Recovery Utility Reference* for details.

## Tradeoffs for Join or Hash Indexes

Join and hash indexes can also have properties that make other solutions more optimal. Remember: it is always important to test and validate join and hash indexes on a test system before putting them into a production environment. Keep these potential downside characteristics in mind whenever you consider creating a join or hash index that you think might be useful for your application workload.

- Balance usage broadness with access efficiency. The more columns defined for a join or hash index, the longer the access time for processing the index. The more tables included in a hash or join index definition, the longer and more burdensome the required maintenance time.
- You cannot define row partitioning for a hash index, nor can you define row or column partitioning for a row-compressed join index.  
However, you can define row or column partitioning for an uncompressed join index.  
Note that you cannot collect statistics on the PARTITION column of such a join index.
- You cannot use the MultiLoad, FastLoad, or Restore utilities against tables that have join or hash indexes defined on them. See “[Load Utilities](#)” on page 614, “[Archive and Recovery](#)” on page 614, *Teradata Archive/Recovery Utility Reference*, *Teradata FastLoad Reference*, and *Teradata MultiLoad Reference* for further information.  
For many bulk loading applications, you can instead FastLoad rows into a staging table which you then MERGE (using error logging) into the target table. See “[MERGE](#)” in *SQL Data Manipulation Language* for details.
- You cannot define triggers on tables that have hash indexes defined on them; however, you can define join indexes and triggers on the same table. See “[Triggers](#)” on page 613.



# CHAPTER 12 Designing for Database Integrity

---

This chapter focuses on two important database design issues related to the integrity of databases: semantic and physical data integrity. First and foremost, enforcing database integrity means getting the correct results. This is fundamental.

The Data Warehousing Institute reports that businesses in the United States alone lose \$600 billion dollars each year due to bad data (Eckerson, 2002), spending as much as 10% to 20% of their operating revenue just to scrap and rework poor quality data (Stahl, 2004). The same TDWI white paper notes that 78% of survey respondents believe their organization needs more education about the importance of data quality and how to maintain and improve it.

In a 2005 follow-up study (Russom, 2005; Whiting, 2006), TDWI reports that 82.5% of the organizations surveyed “...continue to perceive their data as good or okay. However, half of the practitioners surveyed warn that data quality is worse than their organization realizes... Two-thirds of respondents have studied the problems of data quality, while *less than half have studied its benefits.*”

Furthermore, when asked if their company has suffered losses, problems, or costs due to poor quality data, 53% say yes, only 11% say no, and an alarming 36% have not even studied the issue (Russom, 2005; Whiting, 2006, where the relevant pie chart on page 42 is labelled “Database Debacles”).

Shilakes and Tylman (1998) estimate that data cleansing accounts for anywhere between 30 and 80% of the development time for a typical data warehouse. Most organizations do their best to fix bad data at the source, long before it arrives in the data warehouse (see English, 1997; Lee, et al., 2006; Olson, 2002; Piattini et al., 2006; Redman, 1997, 2001). But even if only 1% of the errors remain by the time data reaches the data warehouse, that still accounts for roughly \$6 billion dollars of that \$600 billion dollar total lost each year because of bad data. Russom (2008) also calls attention to the fact that the majority of enterprises focus their data quality efforts on customer data to the exclusion of other important data domains such as product, financial, and asset data. He writes, “Customer-oriented data quality techniques and tools can be retrofitted to operate on other data domains, but with limited success.”

Of course, the quality of data that is received from different sources can vary immensely. Efforts are currently underway to develop tools for data cleaning that allow users to estimate the reliability of uncertain data using Bayesian statistical methods, which can be used both to develop probability models for uncertain data and to improve the quality of uncertain data.

# Sources of Data Quality Problems

## Data Warehousing Institute 2002 Survey

The following table reports the results of an industry survey of sources of data quality problems:

Source	Percentage
Data entry by employees	76
Changes to source systems	53
Data migration or conversion projects	48
Mixed expectations by users	46
External data	34
System errors	26
Data entry by customers This includes typographical errors and information typed into the wrong fields made when entering data into forms on the World Wide Web.	25
Other	12

Source: Wayne W. Eckerson, *Data Quality and the Bottom Line: Achieving Business Success Through a Commitment to High Quality Data*, The Data Warehousing Institute, 2002.

## Data Warehousing Institute 2005 Follow-Up Survey

Three years later, a TDWI survey of many of the same industry sources reported the following list of most frequent contributors to data quality problems in their organizations:

Source	Percentage
Data entry by employees	75
Inconsistent definitions for common terms	75
Data migration or conversion projects	46
Mixed expectations by users	40
External data	38
Data entry by customers This includes typographical errors and information typed into the wrong fields made when entering data into forms on the World Wide Web. Note that human error can also have a significant effect on the security of your site (Liginlal, Sim, and Khansa, 2009).	26

Source	Percentage
System errors	25
Changes to source systems	20
Other	7

Source: Philip Russom, *Taking Data Quality to the Enterprise Through Data Governance*, The Data Warehousing Institute, 2005.

The percentages do not add to 100 because they represent the relative number of survey respondents who reported the associated data quality problem as a significant source of data quality problems in their organization, not the percentage of all errors contributed by each source.

The surveys cover the same error sources except that the 2005 survey adds the category “Inconsistent definitions for common terms,” which tied with “Employee data entry” as the number one source of data quality problems in the organization.

Data Quality Problem Source	2002 Survey Score	2005 Survey Score	Percent Change
Data entry by employees	76	75	-1
Inconsistent definitions for common terms	Ø	75	0
Data migration or conversion problems	48	46	-2
Mixed expectations by users	46	40	-6
External data	34	38	4
Data entry by customers	25	26	1
System errors	26	25	-1
Changes to source systems	53	20	-33
Other	12	7	-5

Sources:

- Wayne W. Eckerson, *Data Quality and the Bottom Line: Achieving Business Success Through a Commitment to High Quality Data*, Seattle, WA: The Data Warehousing Institute, 2002.
- Philip Russom, *Taking Data Quality to the Enterprise Through Data Governance*, Chatsworth, CA: The Data Warehousing Institute, 2005.

The category “inconsistent definitions for common terms” was not measured in the 2002 survey.

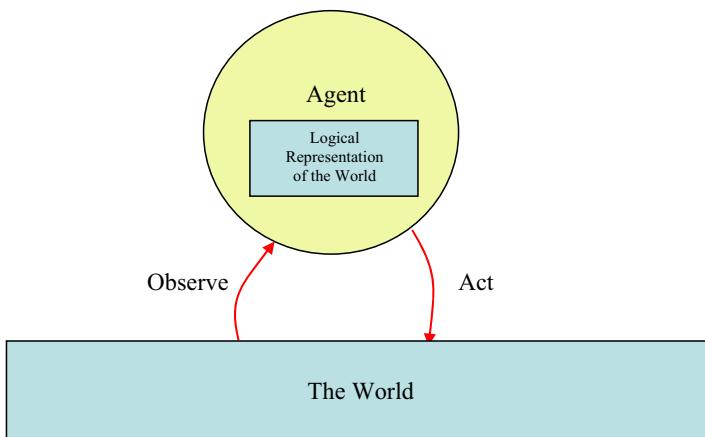
Note that over three years time, the percent change, whether positive or negative, is, with two possible exceptions, not significant. The exceptions are the relatively small 6% drop in “Mixed expectations by users” and the significantly large 33% drop in “Changes to source systems.”

It is interesting to note, however, that of the 79% of the organizations surveyed that have a data quality initiative in place, the team leading that initiative in the 2005 survey is most likely to be the data warehousing group, whereas in the 2002 survey, the data warehousing team was second to the IT department in terms of which was the more likely leader of the initiative. Unfortunately, a whopping 42% of those surveyed had no plans to institute a data governance initiative, while a mere 8% had such an initiative already in place. 33% had an initiative under consideration, while another 17% had such a plan in either its design or implementation phase.

## The Problem of Capturing the External World as Data

Fundamental to the problem of data quality is the fact that it is virtually impossible to capture the reality of the external world as data. At the same time, the Closed World Hypothesis (see “[The Closed World Assumption](#)” on page 630 and “[The Closed World Assumption Revisited](#)” on page 677) tells us that if an otherwise valid tuple does not appear in the body of a relation, then the proposition representing that tuple must be false. In other words, the assumption is made that facts not known to be true in a relational database are false.

The following diagram is from an early draft of a book by Kowalski (2011) on computational logic for people who are not computing professionals.



1101A339

Think of the *Agent* as a business analyst or user, the *Logical Representation of the World* as the database, and *The World* as reality external to the database.

Still following Kowalski (2011), though with numerous modifications, the relationship between logic and the world can be represented from two points of view:

- The world perspective.
- The logical representation of the world.

From the world perspective, logical sentences represent selected world features.

From the logical perspective, the world provides semantics for logical sentences. A world structure is a collection of individuals and the relationships among them, and only true sentences are useful to a business analyst or user, which is why only true sentences are stored

in a database. Note how closely this corresponds to the Entity-Relationship model of Chen (1976).

A world structure corresponds to a single, static state of the world. In the relational model, this corresponds to a *relation value*. See “[Relations, Relation Values, and Relation Variables](#)” on page 627.

An atomic sentence is true in a world structure if, and only if, the relationship it expresses holds in the world structure, and otherwise it is false.

The difference between such atomic sentences and the world structure they represent is that in a world structure the individuals and the relationships between them have a kind of external existence that is independent of language. Atomic sentences, on the other hand, are just symbolic expressions that stand for such external relationships.

When studied closely, it becomes apparent that the philosophical nature of data itself is important enough to warrant serious investigation, particularly as applied to issues of database management. See Kent (2000) for a highly recommended book-length study of this problem. Much of what Kent describes is the disconnect between the fuzzy logic and sets that underlie the real world and the classical set theory and logic that underlie the data in relational and other databases, although he never frames his arguments in those terms. While some research has been made into the concept of fuzzy databases (see Raju and Majumdar, 1988 and Buckles and Petry, 1995, for example), the majority of research in this area has involved the use of fuzzy queries of relational databases and the use of fuzzy logic in data mining. The current state of research makes the problem of a clean, easy-to-use application of fuzzy sets and fuzzy logic to database systems seem intractable, in no small part because fuzzy logic suffers from the same difficulties as other multivalued logics. See Klir, St. Clair, and Yuan (1997) for an introduction to fuzzy set theory and fuzzy logic that builds on classical set theory and logic.

Also see Date and Darwen (1998, 2000) for a briefer and somewhat more formal study of the semantic limitations of data, which they couch in terms of the inability of data represented by an internal predicate to capture the full semantics of the external world, or external (to the database) predicate.

This is not a problem of any particular vendor implementation, nor even of the relational model: it is inherent in data *as data*.

## Data Quality Problems Compromise Information Security

A recent study (Whitman, 2003) identifies software failures and errors, human error, and hardware failures and errors as 3 of the top 6 threats to the information security of the enterprise, with software failures and errors and human error ranking as the second and third most important factors, respectively.

Although semantic integrity problems are far from being the principal result of software failures and errors, the Whitman study specifically calls out data integrity compromises as a fundamental consequence of software failures and errors. A recent paper by Khansa and Liginlal (2009) surveys the quantifiable vulnerabilities of failing to ensure the confidentiality, integrity, and availability of information.

## The Role of Human Error in Creating Bad Data

Numerous ergonomic studies have demonstrated that human error is inevitable for any system of arbitrary complexity (Bailey, 1983; Dörner, 1996; Reason, 1990), and that certainly includes the computer-human interface (Norman, 1988). The majority of data entry errors fall into the category of “valid, but not correct,” including typographical errors. Consider keypunch errors, for example. While it is possible to minimize keypunch errors if optimum working conditions, such as adequate lighting and ergonomic furniture, are enforced and if employee vigilance is optimized by means of adequate rest periods, minimal visual and auditory distractions, and a generally non-hostile working environment, some keypunch errors are inevitable. Constraints cannot completely eliminate this category of errors. After all, it is not possible to prevent all errors from being recorded in the database, but it *is* possible to exclude certain categories of errors by implementing a relatively small number of robust error prevention constraints.

The bottom line, as Bailey (1983) has noted, is that it is far more cost-efficient to *prevent* human errors than it is to detect and correct them after they have been made.

For example, database constraints can easily validate data entries and enforce referential integrity, both of which are commonly reported sources of data errors. CHECK constraints can prevent a keypunch operator from successfully typing values into a table column that are outside the range of values permitted for that column by enterprise business rules, and referential integrity constraints can prevent child table rows from becoming orphaned as the result of a mistaken deletion of a parent table row or update to a parent table primary, or alternate, key. Cohen et al. (2009) point out that constraints are not only important for maintaining data integrity, but also because they capture dependencies among data items, which can then, for example, be used by the Optimizer to generate query plans.

## Database Constraints and Enterprise Business Rules

Constraints are a physical implementation of the business rules under which an enterprise operates. Integrity constraints are a method of restricting database updates to a set of specified values or ranges of values. They ensure not only that bad data does not get into the database, but also that intertable relationships do not become corrupted by the improper deletion or updating of data from the existing database.

The basic types of integrity constraints are:

- Semantic
- Physical

Teradata Database also provides a referential constraints, which provides the Optimizer with a means for devising better query plans, but which does *not* enforce the referential integrity of the database. See “CREATE TABLE” in *SQL Data Definition Language* for more information.

## Definitions

Term	Definition
Business rule	<p>A component of the business model that defines specific conditional modes of activity.</p> <p>Business rules are expressed in natural language and are represented in a relational database by integrity constraints. They are the ultimate determining factor for proper database design.</p>
Constraint	<p>A predicate that must evaluate to TRUE if a database DELETE, INSERT, or UPDATE operation is to be permitted.</p>
Integrity constraint	<p>A component of the logical database model that formalizes business rules by specifying the boundary conditions and ranges permitted for various database parameters.</p> <p>Integrity constraints are one of the four components of an integrity rule and are expressed in the language of databases: tables, columns, rows, and so on.</p>
Integrity rule	<p>A set of rules for ensuring the integrity of a database. Each integrity rule is composed of a:</p> <ul style="list-style-type: none"> <li>• Name</li> <li>• Integrity constraint set</li> <li>• Checking time</li> <li>• Violation response</li> </ul> <p>The checking time specifies the processing point at which the constraint is checked. In the ANSI/ISO SQL standard, a checking time has either of the following possible values:</p> <ul style="list-style-type: none"> <li>• Immediate</li> <li>• Deferred</li> </ul> <p>All Teradata Database constraints have an immediate checking time. Deferred constraint checking, which is never a good way to ensure integrity, is not permitted.</p> <p>The violation response specifies the action to be taken when an integrity constraint is violated. In the ANSI/ISO SQL standard, a violation response has either of the following possible values:</p> <ul style="list-style-type: none"> <li>• Reject</li> <li>• Compensate</li> </ul> <p>All Teradata Database constraints have a reject violation response. Compensate violation responses are not permitted.</p> <p>Integrity rules are specified by the SQL CREATE TABLE and ALTER TABLE statements.</p>

## Semantic Data Integrity Constraints

Semantic integrity constraints enforce the logical meaning of the data and its relationships. Physical integrity constraints enforce the physical integrity of the data, for example ensuring that the bit sequence 00000001 is stored as 00000001 and not 00000010.

The emphasis of this chapter is on declarative semantic constraints: constraints that are part of the definition of the database itself. It is also possible to implement procedural support for

integrity using database features such as triggers and stored procedures. The principal drawbacks to procedural enforcement of database integrity are performance, the added programming requirements and, critically, the increased opportunity for errors in enforcement introduced by these methods. If a declarative constraint alternative is available, it is almost always a better choice than a trigger (which can introduce serious sequential timing issues) or stored procedure.

The least favorable method for enforcing database integrity is through application programming. Besides the increased programming burden and its likely introduction of errors into integrity enforcement, application programming introduces the additional fault of application specificity. If one application enforces database integrity and another does not, or if two programs enforce integrity in different, perhaps contradictory ways, then a still greater chance of corrupting the database results. Worse still, application-based database integrity cannot affect ad hoc inserts, deletes, and updates to the database, and as a result place still further burdens on the DBA to find other mechanisms of preventing database corruption.

As Godfrey et al. (1998, p. 253) write, “Integrity constraints exist in practically all database applications today. Note that the integrity constraints [...] are expressed declaratively, and in the same logical language with which we express facts and rules. By having integrity constraints explicitly represented, they are available to all applications alike. A declarative expression of ICs is important, because non-declarative specifications are difficult to express and maintain. Indeed, this same lesson lead to the advent of relational databases themselves: a declarative specification of facts and rules (views) is easier to express and maintain.”

The point of relational databases is to be application-neutral, thus serving all applications equally well.

These constraints, in addition to the other constraints defined for the system during the process of normalization, such as functional and multivalued dependencies, have the added advantage of facilitating query and join optimization. The richer the constraint set specified for a database, the more opportunities there are to enhance query optimization.

The principal types of declarative constraints described are as follows:

- Column-level constraints
- Table-level constraints
- Database constraints

The recommendation for constraints is to specify them anywhere you can, being mindful of the performance debit their enforcement accrues. If you cannot trust your data, there is little point in maintaining the databases that contain it.

You can never declare semantic database constraints on columns defined with the XML, JSON, BLOB, or CLOB data types.

If performance issues make declarative constraints an unattractive option, then you should *always* implement integrity constraints by some other mechanism. The performance savings attained by implementing integrity constraints outside the database are often just transferred from the database to the application, negating any actual performance gains realized by not implementing the constraints declaratively.

The most important consideration must be that database integrity is consistently enforced.

## Physical Data Integrity Constraints

Teradata Database ensures physical data integrity by means of an end-to-end integrity checking strategy that checksums user data from system memory on down to disk.

See the following topics for more information about physical data integrity constraints:

- “Physical Database Integrity” on page 664
- “Disk I/O Integrity Checking” on page 666
- “Detecting Data Corruption Using Disk I/O Integrity Checksums” on page 667
- “Integrity Checking Using a Checksum” on page 668
- “Disk I/O Integrity Level Settings Based on Table Type” on page 669
- “About Reading or Repairing Data from Fallback” on page 672

## Logical Integrity Constraints

The pursuit of theory is frequently perceived as being unnecessary in the day-to-day practice of administering relational database management systems, but a small dose of theory is essential to understanding the issues you encounter when designing and maintaining databases. If you are interested in pursuing the logical foundations of database integrity constraints, the paper by Godfrey et al. (1998) is a good overview of the subject, though it is grounded in the theory of logic databases rather than ordinary relational systems. Also see the books by Date (2007) and de Haan and Koppelaars (2007).

Concepts that otherwise appear to be complicated and confusing turn out to be rather obvious once the underlying principles have been mastered. As an anonymous wit once observed, “the gap between theory and practice is not as wide in theory as it is in practice.” In other words, without understanding the formal building blocks of relational technology, it is easy to land oneself in considerable trouble in real world practice.

The extent to which practical relational database management can be formalized using the principles of set theory and formal logic is seldom appreciated. The correspondence between relational set theory and relational database theory is not always direct. While it is true that relational database theory is based on the relations of set theory, it necessarily introduces the additional constructs listed below.

- Database relations are typed, while set theory relations are not. Note that data types, or domains, are also a form of constraint.
- Database relations are not ordered left-to-right, while set theory relations are.
- Dependency theory is critical to database relations, while no corresponding theory exists for set theory relations.
- Relation variables are critical to database relations, while no corresponding theory exists for set theory relations. As a result, the concept of integrity as it is understood for relational database management is foreign to set theory relations.

When an everyday practice such as database management can be abstracted to such a degree, it becomes trivial to determine whether a particular operation against the database is logically correct or not. The importance of this capability is far from trivial. Indeed, it has been argued that *all* logical differences are big differences, and this contention is surely true with respect to any database upon which the correctness of decision making for an enterprise relies, because if the information in the database is not correct, it is not just worthless, it seriously endangers the viability of the enterprise it is designed to support.

This principle has been attributed to the philosopher and logician Ludwig Wittgenstein, though the remark appears to have been made to Wittgenstein's colleague Peter Geach in a private communication, not in a public forum or a published text. Even so, its truth is unassailable. Hugh Darwen has added the following important corollary to this principle: "All logical mistakes are big mistakes" as well as the following additional conjecture: "All non-logical (psychological) differences are small differences." (see, for example, Date, 2002; Darwen, 2004). Also see Halpern et al., 2001 for a concise review of the importance of logic in the development of computer science. Section 3 of their paper is called "Logic as a Database Query Language."

## Definitions of Terms

Before proceeding further, some definitions for terms used in predicate logic and the predicate calculus are in order.

Term	Definition
Assertion	See "Proposition".
Existential quantifier	The symbolic quantifier $\exists$ of predicate logic, signifying the logically equivalent English language phrases "for some," "for any," and "there exists."
Identity predicate	The symbolic operator $=$ of predicate logic, signifying the logically equivalent English language phrases "is identical to" or "is equal to."
Inference rules	The rule set of a formal system that determines the steps of reasoning that are valid for proving logical propositions.
Predicate	A truth-valued function.  The attributes of a relation (columns), as well as the relation heading (relation variable) itself, can be represented formally as logical predicates.  This is true whether an explicit constraint is defined over the column or not, because when no explicit constraint is defined, the implicit constraint specified by the data type for the column specifies its minimum, essential, constraint. You cannot, for example, insert the character string 'character string' into a column typed as INTEGER without first converting the string into an integer value.  This type of constraint is known as a domain constraint (see " <a href="#">Domain Constraints</a> " on page 633).
Predicate calculus	The set of inference rules by which propositions in predicate logic are proven.

Term	Definition
Predicate logic	The study of statement validity using the truth-functional operators of the propositional calculus, the universal and existential quantifiers, and the identity predicate.
Proposition	An assertion that can be proven unequivocally to be either true or false.  In a relational table, or relation variable, all rows are assumed to be true propositions by default, because if they were false, they would have been prevented from entry in the database by the various integrity constraints, both implicit and explicit, defined on that database. Each proposition (tuple) in a relation is an instantiation of its relation variable predicate that evaluates to TRUE.  This important property is sometimes called the Closed World Assumption (see “ <a href="#">The Closed World Assumption</a> ” on page 630).  Logical propositions have wide application throughout computer science. See Hoare (2003) for a historical review.
Truth-valued function	A function that evaluates unequivocally to either TRUE or FALSE.
Universal quantifier	The symbolic quantifier $\forall$ of predicate logic, signifying the English language phrase “for all.”

## Relations, Relation Values, and Relation Variables

As the relational model has matured, careful analysis has continued to reveal facets of its definition that are obvious in hindsight, but had remained undiscovered and unstated until recently. An important example of such a revelation of the obvious is the relation variable, or relvar, first described by Date and Darwen (1998), which actually derives from a statement made by Codd (1970), where he writes that a database is “a collection of time-varying relations … of assorted degrees. As time progresses, each  $n$ -ary relation may be subject to insertion of additional  $n$ -tuples, deletion of existing ones, and alteration of components of any of its existing  $n$ -tuples.”

Note that relations, as defined in set theory, do *not* vary over time. A relation is a *value*, and values do not vary as a function of time. The number 3, for example, is always the number 3, and never 2 or 13 or 724.

Variables, on the other hand, take on different values as a function of time, so it is formally correct to state that the value of a relation variable changes as a function of time (the distinctions between the definitions of value and variable are based on those found in the text by Cleaveland (1986). What Codd almost certainly meant to say was something like the following: a database is a collection of time-varying relation variables of assorted degrees.

The distinction between a relation variable and its associated relation value is an important one because without it, the application of the axioms of traditional set theory to database relations is suspect at best. As Kent (2000, pp. 104–105) writes, “…one ought to be very cautious about claims of various models being based on “the axioms of traditional set theory.” That set theory deals entirely with extensional sets: a set is determined entirely by its

population. There is simply no notion of a set with changing population; each different population constitutes a different set.”

Also see Date, 1992f, for an explicit comparison of the definition of a relation in set theory vis-a-vis its definition in the database relational model. Note that Date had not yet developed the notion of a relation variable when he wrote this paper, and where he compares the two flavors of relation definitions, he confounds the notions of relation *value* and relation *variable* in the database relational model. For example, he states that relations in the database relational model are time-varying, while those in set theory are not. In fact, it is relation *values* that are time-varying, while relation variables are *not* (setting aside the issue of adding or dropping columns from a database table for the sake of the theoretical distinction). The paper is nevertheless well worth reading for its explicit clarification of the differences between the two relation types.

In the context of business rules and their relationship with integrity constraints, the concept of the relvar is useful because it provides a formal, yet simple, framework in which to situate the idea of integrity constraints. Note that the relvar concept applies equally well to views (virtual relvars) and base tables (base relvars).

The first principle captured by the concept of the relvar is that the value of a relation is orthogonal to its defining variable. The easiest way to explain this is by analogy. A variable in any programming language represents the possible values that can be taken for that variable, but in itself it is not those values: it is, instead, a place holder for them. The values represented by the program variable can take on any number of values, but the variable itself is always an abstraction.

Think of a relvar as the heading definition for the attributes of a relation: it is the ANDing of all the column headings, including their constraints, defined for the relation. In terms of logic, a relvar is the predicate for a relation. More specifically, a relvar is the *internal* predicate for a relation. The real world that relvar represents is its *external* predicate. Any lack of correspondence between the external and internal predicates for a relvar is the defining characteristic of the data quality issue (see the introduction to this chapter for more information) as well as the inspiration for the maxim that a database can only enforce consistency, not truth. Unless otherwise stated, the phrase *relvar predicate* refers to the internal relvar predicate in this book, while each individual instantiation of that relvar, a tuple (or row), is a proposition for that predicate that evaluates to TRUE.

The relation *value*, then, is the net value of all the data contained by the relation. Each time the relation is updated, its value changes, but its relation *variable* does not. In a very real sense, a relation becomes an entirely new relation when an update occurs, but its relation variable does not change.

For example, consider the following equation:

$$x + y = 7$$

If you set the variable  $x=4$ , then the value for variable  $y$  must be 3. If you then change the value of the variable  $x$  to 3, you have updated  $x$ , but you have not changed the value of 4 to 3, you have just changed the value of the variable  $x$  from 4 to 3. This holds for any other valid number over the domain of  $x$ . Another way to think of this is as follows: values are not time-varying. A 3 is always a 3 and never a 4. On the other hand, a variable can represent any

number of different values over the course of time. An old joke also illustrates this point by confusing variables with values: “Do you realize that  $2 + 2 = 5$  for large values of 2”?

The integrity of this update relationship is maintained by the relvar predicate, which is defined by the logical ANDing of all the integrity constraints defined on the relation in question. The result of enforcing the relvar predicate is what Date and Darwen (1998, 2000) call The Golden Rule: no statement can leave any relvar with a value that violates its relvar predicate. This means that no update operation can ever cause any database constraint to evaluate to FALSE. In other words, the integrity of the database cannot be violated. The Golden Rule is one of the fundamental principles of relational database theory.

It follows that tuples (“rows” in everyday language: see “[Definitions](#)” on page 75) are instantiations of the relvar predicate that represent various propositions about the relation they constitute. When you think of a row as a proposition, it immediately follows that it must also be a *true* proposition because, by definition, false propositions (wrong, or corrupt data in everyday language) are not permitted in a database that enforces integrity constraints (see “[The Closed World Assumption](#)” on page 630). If they were, the database would not represent facts and the Golden Rule would be violated. In other words, if a given tuple is an element of a particular relvar, then that tuple, by default, satisfies its predicate by evaluating to TRUE.

Unfortunately, the representation of the real world by a database is only as good as its input, and any database will permit the insertion of factually wrong data as long as that data does not violate the constraints specified for the base table or view through which the table is updated (see “[Sources of Data Quality Problems](#)” on page 618 and “[The Role of Human Error in Creating Bad Data](#)” on page 622). In other words, a database can only enforce consistency, not truth. All truths are consistent, but not all consistent things are true.

This is an important consideration: the database itself cannot know whether its information corresponds to real world facts (see “[Sources of Data Quality Problems](#)” on page 618). It can, however, constrain the data with which it is updated by ensuring that no business rules are violated, and it is these constraints that provide the mechanism for ensuring the success of the data integrity rules enforced by relational database management systems.

## The Logic of Database Integrity

To ensure that no row in the database represents a logically false proposition, each table in the database must be capable of evaluating its own predicate, or relvar, for its truth value, and the only way it can enforce this integrity is to enforce the set of business rules defined by the enterprise it supports. In practice, this means that the predicate for any table is its sole criterion for update acceptability (McGoveran and Date, 1994). From this, it follows that the formal meaning of any table is defined by the logical ANDing of its component column and table constraints. This logically ANDed constraint set is also the best approximation the database management system can make to any table predicate.

From this, it follows that the meaning of an entire database, its predicate, can be represented formally as the logical ORing of its table constraints logically ANDed with the set of all database constraints.

Although most workers in database management are not familiar with this treatment of databases as systems of logical constraints, the concept is at least as old as the paper of

Kanellakis et al. (1990), which is usually credited with being the foundation paper for the concept of constraint databases (see, for example, Gunther et al., 1997; Kuijpers and Revesz, 2004; Kuper and Wallace, 1996; Kuper et al., 2000; Ramakrishnan and Stuckey, 1997; Revesz, 2002).

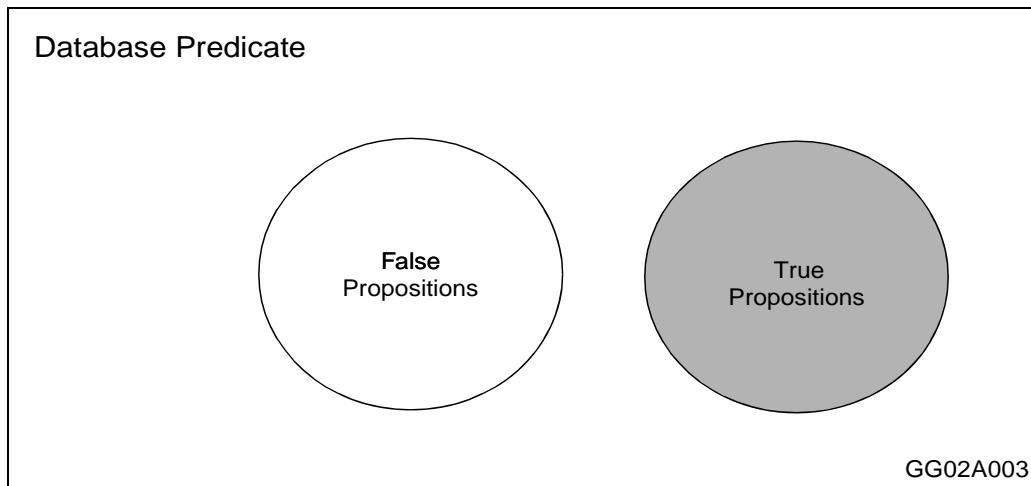
## The Closed World Assumption

The Closed World Assumption (Reiter, 1978, 1984), or CWA, asserts that if an otherwise valid tuple does not appear in the body of a relation, then the proposition representing that tuple must be false. In other words, the assumption is made that facts not known to be true in a relational database are false (Minker, 1996). This was the most important of three assumptions put forth by Reiter. The others are: the *Unique Name Assumption*, which states that any item in a database has a unique name and, further, that objects with different names are not the same; and the *Domain Closure Assumption*, which states that there are no objects other than those within the database. Two closely related rules are the *Completed Database Rule* and the *Negation As Finite Failure Rule*, both introduced by Clark (1978).

The CWA has become one of the fundamental principles of relational database theory.

In symbolic logic, such an assertion is called a completion axiom. It follows that a relation contains all of, and *only* those, tuples whose corresponding propositions are true. In other words, a relation body contains only those tuples that satisfy the completion axiom for its relvar. By generalization, the complete set of completion axioms for a given database defines its CWA, and the set of all tuples in a given database must also satisfy its CWA.

The following Venn diagram illustrates this point:



Note that, by the definition of the CWA, the sets of false and true propositions for a database do not intersect. See “[The Closed World Assumption Revisited](#)” on page 677 for an application of the CWA to missing information in relational databases and the paradox that nulls present to the definition of a well-formed completion axiom.

# How Relational Databases Are Built From Logical Propositions

The following example illustrates how a real database is built from logical propositions.

## An Airline Reservation System

As a concrete example of the logical underpinnings of database constraints, consider an airline reservation. The following proposition can be stated about any reservation, and a particular instance of the proposition can be represented by a unique row in a table:

The reservation identified by reservation number (*reservation\_number*), made for flight (*arrival\_flight\_number*), has a scheduled arrival date (*arrival\_date*) and time (*arrival\_time*) at gate (*arrival\_gate\_number*) and a scheduled departure flight of (*depart\_flight\_number*) with departure date (*depart\_date*) and time (*depart\_time*) from gate (*depart\_gate\_number*).

The variables enclosed within parentheses in this proposition are placeholders for the entire domain of values they represent. When you substitute actual values for these placeholders, you produce a proposition about a specific airline reservation. For example, suppose you have the following set of values:

(88079, 317, 10/19/2001, 13:59, 60, 1138, 10/25/2001, 05:40, 82)

When these values are substituted into the predicate framework, they produce the following logical proposition:

The reservation identified by reservation number 88079, made for flight 317, has a scheduled arrival date 10/19/2001 and time 13:59 at gate 60 and has a scheduled departure flight of 1138 with departure date 10/25/2001 and time 05:40 from gate 82.

This proposition, which represents an instantiation of the table predicate, might be represented in a *flight\_reservations* table by the following row.

*flight\_reservations*

res_num	a_flt_num	a_date	a_time	a_gate	d_flt_num	d_date	d_time	d_gate
PK								
88079	317	10/19/2001	13:59	60	1138	10/25/2001	05:40	82

Several obvious constraints can be developed for the columns and table supporting this proposition.

Stated as simple sentences, these are the following.

- Column constraints:
  - Reservation numbers must be unique values drawn from a defined integer domain.
  - Arrival and departure flight numbers must be unique values drawn from a defined integer domain.

- Arrival and departure dates must be unique values drawn from a defined date domain.
- Arrival and departure times must be unique values drawn from a defined time domain.
- Arrival and departure gate numbers must be unique values drawn from a defined integer domain.
- Table constraints:
  - Departure dates must be greater than or equal to arrival dates.
  - Departure times must be greater than arrival times if they occur on the same date.
  - Arrival flight numbers and departure flight numbers cannot be equal if they occur on the same date.

And so on.

## Inclusion Dependencies

Inclusion dependencies were initially described by a number of people (Codd, 1979; Smith and Smith, 1977; Zaniolo, 19779), but were first formalized as inclusion dependencies by Fagin (1981). Inclusion dependencies are a generalization of referential constraints. As such, they provide the foundation upon which referential integrity is based. In common with functional dependencies (see “[Functional, Transitive, and Multivalued Dependencies](#)” on [page 80](#)), inclusion dependencies represent one-to-many relationships (see “[One-to-Many Relationships](#)” on [page 69](#)); however, inclusion dependencies typically represent relationships between relations (see “[Database-Level Constraints](#)” on [page 635](#)), while functional dependencies always represent relationships between the primary key of a relation variable and its attributes.

Suppose you have the following table definitions:

Using the notation R.A, where R is the name of a relation variable and A is the name of one of its attributes, you can write the following inclusion dependency:

`supplier_parts.part_num → parts.part_num`

parts					supplier_parts	
PK	part_name	color	weight	city	FK	FK
part_num					supp_num	part_num

This inclusion dependency states that the set of values appearing in the attribute *part\_num* of relation variable *supplier\_parts* must be a subset of the values appearing in the attribute *part\_num* of relation variable *parts*. Notice that this defines a simple foreign key-primary key relationship, though it is only necessary, in order to write a proper referential integrity

relationship, that the right hand side (RHS) indicates any candidate key of the specified relation variable, not necessarily its primary key (see “[Foreign Key Constraints](#)” on page 644 for more information about this).

The left hand side (LHS) and RHS of a dependency relationship are not required to be a foreign key and a candidate key, respectively. This is merely required to write a correct inclusion dependency expression of a referential integrity relationship.

## Inference Axioms for Inclusion Dependencies

Casanova, Fagin, and Papadimitriou (1984) produced a complete set of inference axioms for inclusion dependencies. Their axioms are described in the following table:

Axiom	Formal Expression					
Reflexive rule	$A \rightarrow A$					
Projection and Permutation rule	IF    AB $\rightarrow$ CD    THEN    A $\rightarrow$ C    AND    B $\rightarrow$ D					
Transitivity rule	IF    A $\rightarrow$ B    AND    B $\rightarrow$ C    THEN    A $\rightarrow$ C					

## Semantic Integrity Constraint Types

Business rules are expressed in relational databases by means of various types of integrity constraints. Four types of integrity constraint are supported:

- Domain
- Column
- Table
- Database

These four types of constraints are implemented in distinct ways in Teradata database, though the differences between column and table constraints are subtle.

### Domain Constraints

Fundamentally, a domain is a data type. Data types act as simple constraints by not allowing you to, for example, insert a 20 character string into a field typed INTEGER.

The domain constraints for your databases are developed from the ATM Domains form (see “[Constraints Form](#)” on page 142).

### Column-Level Constraints

Column-level constraints define more elaborate integrity rules for columns than those defined by simple domain constraints. For example, a column constraint on a column typed as INTEGER might further specify that only values between 0 and 49999 or between 99995 and 99999 are permitted for that column.

A column constraint specifies a simple predicate that applies to one column only. For example, if some business rule states that employee numbers in the *employee* table must be between 10001 and 32001 inclusive. You could specify this rule in the CREATE TABLE statement used to define the *emp\_no* column in the *employee* table as follows:

```
CONSTRAINT emp_no CHECK (emp_no >= 10001 AND emp_no <= 32001)
```

Notice that the only column referred to by this constraint is the employee number column, *emp\_no*.

The column constraints for your databases are developed from the ATM Constraints form (see “[Constraints Form](#)” on page 142).

You cannot declare column-level CHECK constraints on any column defined with the XML, JSON, BLOB, CLOB, BLOB-based UDT, CLOB-based UDT, XML-based UDT, ARRAY/VARRAY, Period, or Geospatial data types.

You can declare CHECK and UNIQUE constraints on row-level security constraint columns. When you define these constraints for a column, they apply to all rows in the table, not just to rows that are user-visible. You can also declare UNIQUE constraints on distinct and structured UDT columns as long as they are not based on XML, JSON, BLOB, or CLOB data.

You cannot declare database constraints other than NULL or NOT NULL for global temporary trace tables. See “CREATE GLOBAL TEMPORARY TRACE TABLE” in *SQL Data Definition Language*.

You can also define column-level and object-level access constraints, or security privileges, on tables (for details, see *Security Administration* and *SQL Data Control Language*).

## Row-Level Security Constraints

You can define row-level security constraints on tables. Row-level security is not a semantic constraint, but row-level security constraints and semantic constraints may interact. For details about row-level security, see *Security Administration*.

## Table-Level Constraints

Table-level constraints specify sets of relationships among values within a row. A table-level constraint can be either single-row or multirow, and applies to table columns rather than table rows.

For example, you might specify a table constraint on the *flight\_reservations* table (see “[An Airline Reservation System](#)” on page 631) that requires the value for the scheduled arrival date, *a\_date*, to be less than or equal to the value for the scheduled departure date, *d\_date*. This is a single-row table constraint.

The SQL definition for this constraint looks like this:

```
CONSTRAINT arrive_date_check CHECK (a_date >= d_date)
```

Notice that this constraint refers to two columns, *a\_date* and *d\_date*; therefore, must be a table constraint.

You probably want to ensure that the value for *res\_num* is unique and non-null, so you would define a uniqueness constraint on the table for reservation numbers. In this case, *res\_num* is the primary key for the *flight\_reservations* table, so the constraint is called a primary key constraint.

The SQL definition looks like this.

```
res_num INTEGER NOT NULL CONSTRAINT pKey PRIMARY KEY
```

Primary key constraints are a subset of uniqueness constraints: all primary keys are also unique, but all unique columns are not primary keys because a relation can only have one primary key. Because any unique column set is, by definition, also a candidate key, uniqueness constraints are sometimes referred to as key constraints. The primary key constraint on *res\_num* is also a multirow table constraint because it enforces the rule that every row in the table has at least one unique value: its reservation number.

Suppose reservation number is not the primary key for the *flight\_reservations* table, but your business rules require the values for *res\_num* to be unique. To do this, you write a constraint that enforces uniqueness on the *res\_num* column. The SQL definition looks like this:

```
res_num INTEGER NOT NULL CONSTRAINT ResNumUnique UNIQUE
```

The table constraints for your databases are developed from the ATM Constraints form (see “[Constraints Form](#)” on page 142).

You cannot declare table-level CHECK constraints on any column defined with XML, BLOB, CLOB, UDT, Period, or JSON data types.

You cannot specify constraints other than NULL or NOT NULL for global temporary trace tables. See “[CREATE GLOBAL TEMPORARY TRACE TABLE](#)” in *SQL Data Definition Language*.

## Database-Level Constraints

Database-level constraints specify some sort of functional determinant between the key and its dependent attributes (see “[Functional, Transitive, and Multivalued Dependencies](#)” on page 80) as well as the functional determinants among two or more tables (see “[Inclusion Dependencies](#)” on page 632).

The most commonly observed database-level constraint between tables is the primary key-foreign key relationship whose enforcement is referred to as maintaining referential integrity. This constraint specifies that you cannot delete a row having primary key value *x* from table *X* as long as any foreign key value in table *Y* has value *x* on the column set that defines the referential integrity relationship between those columns in tables *X* and *Y*. In other words, if you have foreign key value *x*, then you must also have primary key value *x* if referential integrity is defined for those keys and tables.

The specific table-level constraint syntax used to define the common PK-FK referential integrity constraint is `FOREIGN KEY (referencing_column_set) REFERENCES referenced_table_name (referenced_primary_key_column_name_set)`.

Though less frequently used, it is also possible to specify and enforce database-level constraints on non-PK-non-FK column relationships if the columns defining those relationships are alternate keys.

The specific constraint used to define an alternate key referential integrity constraint is a foreign key constraint with column-level syntax `REFERENCES ... table_name ... alt_key_name` and table-level syntax `FOREIGN KEY (referencing_column_set) REFERENCES referenced_table_name (referenced_alt_key_name)`, where `alt_key_name` refers to an alternate key column set in the parent table.

The database-level constraints for your databases are developed from the ATM Constraints form (see “[Constraints Form](#)” on page 142).

You cannot declare database-level CHECK constraints on any column defined with XML, BLOB, CLOB, UDT, Period, or JSON data types.

You cannot specify constraints other than NULL or NOT NULL for global temporary trace tables. See “[CREATE GLOBAL TEMPORARY TRACE TABLE](#)” in *SQL Data Definition Language*.

## Semantic Constraint Specifications

A name and a data type must be specified for each column defined for a table. Each specified column can be further defined with one or more attribute (see *SQL Data Types and Literals*) or constraint definitions.

There are several different specifications for constraints, some of which apply to multiple categories of constraints.

You cannot declare semantic database constraints on columns defined with XML, BLOB, CLOB, BLOB-based UDT, CLOB-based UDT, XML-based UDT, ARRAY/VARRAY, Period, JSON, or Geospatial data types.

The following constraints are SQL column definition attributes that specify column-level integrity constraints:

- UNIQUE constraint definition on a single column.

UNIQUE constraints are implemented as USIs.

If a row-level security-protected table is defined with a UNIQUE constraint, enforcement of the UNIQUE constraint does not execute any row-level security policy defined for the table.

UNIQUE constraints are applicable to *all* rows in a row-level security-protected table, not just to user-visible rows.

You can specify UNIQUE constraints on columns having a UDT data type as long as the UDT is not based on XML, BLOB, or CLOB data.

You cannot specify UNIQUE constraints on columns having any of the following data types:

- XML

- BLOB
- CLOB
- XML-based UDT
- BLOB-based UDT
- CLOB-based UDT
- ARRAY
- VARRAY
- Period
- Geospatial
- JSON

You cannot define a UNIQUE constraint on a row-level security constraint column of a row-level security-protected table.

If you do not specify either an explicit PRIMARY INDEX or NO PRIMARY INDEX, Teradata Database converts any UNIQUE constraints you define to either a unique primary index or a unique secondary index, depending on whether a primary key is also defined for the table (see “[Primary Index Defaults](#)” on page 263 for details).

IF PrimaryIndexDefault is set to D or P and a CREATE TABLE request specifies this constraint on a column set without also specifying either a PRIMARY INDEX or NO PRIMARY INDEX option ...

PRIMARY INDEX

THEN Teradata Database converts the ...

column set defined as the primary key to the unique primary index for the table.

Any additional column sets defined with UNIQUE constraints are redefined as unique secondary indexes.

PRIMARY INDEX and UNIQUE constraints are implemented as unique secondary indexes. They are also explicitly redefined as unique secondary indexes in the SQL create text for the table definition.

UNIQUE

first column set defined with a UNIQUE constraint to the unique primary index for the table.

Any other column sets defined with UNIQUE constraints are redefined as either unique secondary indexes.

The primary index defaults described in the preceding table do not apply to column-partitioned tables, where the default is always NO PRIMARY INDEX regardless of the setting of the DBS Control parameter PrimaryIndexDefault.

- CHECK constraint definition on a single column.

CHECK constraints are not implemented as indexes.

If a row-level security-protected table is defined with a CHECK constraint, enforcement of the constraint does not execute any security policy defined for the table.

CHECK constraints are applicable to *all* rows in a row-level security-protected table, not just to user-visible rows.

You cannot define a CHECK constraint on a row-level security constraint column of a row-level security-protected table.

You cannot specify CHECK constraints on columns having any of the following data types:

- XML
- BLOB
- CLOB
- XML-based UDT
- BLOB-based UDT
- CLOB-based UDT
- ARRAY/VARRAY
- UDT
- Period
- Geospatial
- JSON
- PRIMARY KEY constraint definition on a single column.

PRIMARY KEY constraints are implemented as USIs.

If a PRIMARY KEY constraint is defined where either or both the parent and child table are row-level security-protected, execution of the referential integrity constraint does not execute any security policy UDFs defined for the constraints on the table. Execution continues as if the tables were not row-level security-protected.

You can specify PRIMARY KEY constraints on columns having a UDT data type.

You cannot specify PRIMARY KEY constraints on columns having any of the following data types:

- XML
- BLOB
- CLOB
- XML-based UDT
- BLOB-based UDT
- CLOB-based UDT
- ARRAY/VARRAY
- Period
- Geospatial
- JSON

You cannot define a PRIMARY KEY constraint on a row-level security-protected column.

If you do not specify an explicit PRIMARY INDEX or NO PRIMARY INDEX option, Teradata Database converts any PRIMARY KEY constraint you define for a table to a unique primary index (see “[Primary Index Defaults](#)” on page 263 for complete details).

- REFERENCES constraint definition on a single column.

REFERENCES constraints are not implemented as indexes.

If a REFERENCES constraint is defined where either or both the parent and child table are row-level security-protected, execution of the referential integrity constraint does not execute any security policy UDFs defined for the constraints on the table. Execution continues as if the tables were not row-level security-protected.

You cannot specify foreign key REFERENCES constraints on columns having any of the following data types.

- XML
- BLOB
- CLOB
- XML-based UDT
- BLOB-based UDT
- CLOB-based UDT
- UDT
- ARRAY/VARRAY
- Period
- Geospatial
- JSON

Temporal tables do not support foreign key REFERENCES constraints for standard or batch referential integrity.

See *ANSI Temporal Table Support* and *Temporal Table Support* for details.

The following constraints are SQL table definition attributes that specify table-level and intertable integrity constraints:

- CHECK constraint definition on a composite column set.
- FOREIGN KEY ... REFERENCES constraint definition on a composite column set.
- PRIMARY KEY constraint definition on a composite column set.
- UNIQUE constraint definition on a composite column set.

You cannot specify constraints other than NULL or NOT NULL for global temporary trace tables. See “CREATE GLOBAL TEMPORARY TRACE TABLE” in *SQL Data Definition Language*.

## Performance Issues for Referential Integrity Constraints

The following set of topics describes some of the more important performance issues that are included in enforcing referential integrity.

For more information on referential integrity, see “[The Referential Integrity Rule](#)” on page 95 and “[Foreign Key Constraints](#)” on page 644.

## Benefits of Referential Integrity

Benefit	Description
Maintains data consistency	Teradata Database enforces integrity relationships between tables based on the definition of a PK or a FK.
Maintains data integrity	When performing INSERT, UPDATE, and DELETE requests, Teradata Database maintains data integrity between referencing and referenced tables.
Increases development productivity	It is not necessary to code applications to enforce referential constraints because Teradata Database automatically enforces referential integrity.
Requires fewer programs to be written	Teradata Database prevents update activities from violating referential constraints. Teradata Database enforces referential integrity in all environments; you need no additional programs.

## Overhead Costs of Referential Integrity

Overhead costs includes building the reference index subtables and inserting, updating, and deleting rows in the referencing and referenced tables. Overhead for inserting, updating, and deleting rows in the referencing table is similar to that of USI subtable row handling.

Teradata Database redistributes a row for each reference to the AMP containing the USI or reference index subtable entry. Processing differs after that, and most of the additional cost is in message handling.

When implementing tables with referential integrity, consider the following factors.

- Most importantly, the performance impact to update operations, which is frequently slowed when a referential integrity constraint must be enforced.
- INSERT performance slows for table because any referential integrity constraints defined for the table must be enforced.
- The cost of extra disk space for table maintenance resulting from referential integrity constraints.
- The cost of extra disk space for reference index subtables versus savings on program maintenance and increased data integrity.
- The cost of DML integrity validity checking in applications versus the cost of not checking.

## Join Elimination and Referential Integrity

Join elimination is a process undertaken by the Optimizer to eliminate redundant joins based on information from referential integrity constraints.

The following conditions eliminate a join.

- A referential integrity relationship exists between the two tables.
- Request conditions are conjunctive, meaning they are ANDed rather than ORed.

This means that if any single condition in an ANDed set fails, the entire condition fails.

- The request does not contain reference columns from the PK table, other than the PK columns, including the SELECT, WHERE, GROUP BY, HAVING, ORDER BY columns.
- PK columns in the WHERE clause appear only in PK-FK joins.

IF...	THEN...
the preceding conditions are met	<ul style="list-style-type: none"> <li>• the PK join is removed from the query.</li> <li>• all references to the PK columns in the query are mapped to the corresponding foreign key columns.</li> </ul>
foreign key columns are nullable	Teradata Database adds a NOT NULL condition to the request.

## Standard Referential Integrity and Batch Referential Integrity

In standard referential integrity, whether you are doing row-at-time updates or set-processing INSERT ... SELECT requests, each child row is separately matched to a row in the parent table, one row at a time. A separate SELECT request against the parent table is performed for each child row. Depending on your demographics, Teradata Database might select parent rows more than once.

With batch referential integrity, all of the rows within a single request, even if only one row is affected, are spooled, sorted, and their references checked in a single operation, as a join to the parent table. Depending on the number of rows in the INSERT ... SELECT request, batch referential integrity could be considerably faster, compared to checking each parent-child relationship individually.

For row-at-time updates, there is very little difference between standard referential integrity and batch referential integrity. But if you plan to load primarily using INSERT ... SELECT requests, batch referential integrity is recommended.

## Referential Constraints

To maximize the usefulness of join elimination, you can specify Referential Constraints that Teradata Database does not enforce.

You can specify the REFERENCES WITH NO CHECK OPTION option to specify CREATE TABLE and ALTER TABLE statements with Referential Constraints, and the Optimizer can use the constraints without incurring the penalty of database-enforced referential integrity.

But when you use a REFERENCES WITH NO CHECK OPTION clause, Teradata Database does not enforce the Referential Constraints that you define. This means that a row having a non-null value for a referencing column can exist in a table even if an equal value does not exist in a referenced column. When you specify Referential Constraints, Teradata Database does not return error messages that would otherwise occur when RI constraints are violated.

If you specify a column name for a Referential Constraint, it need not refer to the single column PK of the referenced table or to a single column alternate key in the referenced table defined as UNIQUE, though best practice dictates that it should. The key acting as the PK in

the referenced table need not be explicitly declared to be unique using the PRIMARY KEY or UNIQUE keywords or by declaring it to be a USI in the table definition.

The candidate key must always be unique even if it is not explicitly declared to be so, otherwise the referential integrity can produce incorrect results, and corrupt your databases if you do not take care to ensure that data integrity is maintained.

Specifying Referential Constraints relies heavily on your knowledge of your data. If the data does not actually satisfy the Referential Constraints that you provide, then requests can easily produce incorrect results.

The Optimizer can use the Referential Constraints without incurring the penalty of database-enforced referential integrity, but with the likelihood that Teradata Database can return corrupted result data.

## Naming Constraints

Naming constraints is a good practice to follow because it permits a programmer to debug an embedded SQL or stored procedure application by fetching the name of a violated constraint from the SQLSTATE area. This makes debugging more difficult because it can be very difficult to determine which constraints belong to which system-defined names.

Row-level security constraints must be named. CHECK, UNIQUE, FOREIGN KEY, and PRIMARY KEY constraint specifications should be named.

Constraint names must conform to the rules for Teradata Database object names and be unique among all other constraint, primary index, and secondary index names specified on the table.

The characters used to name a constraint can be any of the following:

- Uppercase and lowercase alphabetic characters
- Integers
- Any of the following special characters.
  - -
  - #
  - \$

Teradata Database does not assign system-generated names to unnamed constraints.

## CHECK Constraints

CHECK constraints are the most general type of SQL constraint specifications. Depending on its position in the CREATE or ALTER TABLE SQL text, a CHECK constraint can apply either to an individual column or to multiple columns.

Teradata Database derives a table-level index partitioning CHECK constraint from the partitioning expression for a PPI table. The text for this derived constraint cannot exceed 16,000 characters; otherwise, the system aborts the request and returns an error to the requestor. See “[Partitioning CHECK Constraints](#)” on page 376 and “[Partitioning CHECK Constraints for Single-Level Partitioning](#)” on page 362 for more information about this.

The following rules apply to all CHECK constraints.

- You can define CHECK constraints at column-level or at table-level.
- The specified predicate for a CHECK constraint must be a *simple* boolean search condition.  
Subqueries, aggregate expressions, and CASE expressions are not valid search conditions for CHECK constraint definitions.
- You cannot specify CHECK constraints at any level for volatile tables or global temporary trace tables.
- Be aware that a combination of table-level, column-level, and WITH CHECK OPTION on view constraints can create a constraint expression that is too large to be parsed for INSERT and UPDATE requests.
- Teradata Database tests CHECK constraints for character columns using the current session collation.

As a result, a CHECK constraint might be met for one session collation, but violated for another, even though the identical data is inserted or updated.

The following is an example of the potential importance of this. A CHECK constraint is checked on insert and update of a base table character column, and might affect whether a sparse join index defined with that character column gets updated or not for different session character collations, in which case different request results might occur if the index is used in a query plan compared to the case where there is no sparse join index to use.

- Teradata Database considers unnamed CHECK constraints specified with identical text and case to be duplicates, and returns an error when you submit them as part of a CREATE TABLE or ALTER TABLE request.

For example, the following CREATE TABLE request is valid because the case of *f1* and the case of *F1* are different.

```
CREATE TABLE t1 (f1 INTEGER, CHECK (f1>0), CHECK (F1>0));
```

The following CREATE TABLE request, however, is *not* valid because the case of the two unnamed *f1* constraints is identical. This request aborts and returns an error to the requestor.

```
CREATE TABLE t1 (f1 INTEGER, CHECK (f1>0), CHECK (f1>0));
```

- The principal difference between defining a CHECK constraint at column-level or at table-level is that column-level constraints cannot reference other columns in their table, while table-level constraints, by definition, *must* reference other columns in their table.
- Columns defined with a data type from the following list cannot be a component of a CHECK constraint.
  - XML
  - BLOB
  - CLOB
  - XML-based UDT
  - BLOB-based UDT
  - CLOB-based UDT
  - UDT

- ARRAY/VARRAY
- Period
- Geospatial
- JSON
- You cannot define a CHECK constraint on a row-level security constraint column of a row-level security-protected table.
- If a row-level security-protected table is defined with one or more CHECK constraints, the enforcement of those constraints does not execute any UDF security policies that are defined for the table. The enforcement of the CHECK constraint applies to the entire table. This means that CHECK constraints apply to *all* rows in a row-level security-protected table, not just to the rows that are user-visible.

The following rules apply only to column-level CHECK constraints.

- You can specify multiple column-level CHECK constraints on a single column.  
If you define more than one *unnamed* distinct CHECK constraint for a column, Teradata Database combines them into a single column-level constraint.  
However, Teradata Database handles each *named* column-level CHECK constraint separately, as if it were a table-level named CHECK constraint.
- A column-level CHECK constraint cannot reference any other columns in its table.

The following rules apply only to table-level CHECK constraints.

- A table-level constraint usually references at least two columns from its table.
- Table-level CHECK constraint predicates cannot reference columns from other tables.
- You can define a maximum of 100 table-level constraints for a table at one time.

## Foreign Key Constraints

Foreign key constraints permit you to specify referential primary key-foreign key relationships between a unique column set in the current base table and an alternate key column set in a different base table. The FOREIGN KEY keywords are required for table-level foreign key definitions but cannot be used for column-level foreign key definitions;. If you specify FOREIGN KEY, then you must also specify a REFERENCES clause. Teradata Database uses referential integrity constraints to enforce referential integrity (see “[The Referential Integrity Rule](#)” on page 95 and “[Inclusion Dependencies](#)” on page 632) and to optimize SQL requests (see *SQL Request and Transaction Processing*). The overhead of enforcing standard referential integrity constraints is summarized in “[Referential Integrity Constraint Checks](#)” on page 653 and “[Overhead Costs of Standard Referential Integrity](#)” on page 653.

You can also use a foreign key definition to specify any of the following referential constraint types.

- Standard Referential Integrity
- Batch Referential Integrity
- Referential Constraints
- Temporal Relationship Constraints

Temporal tables can only be defined for so-called “soft” referential integrity relationships, or Referential Constraints, and for Temporal Relationship Constraints, and do not permit foreign key constraints for Standard or Batch Referential Integrity. See *ANSI Temporal Table Support* and *Temporal Table Support* or for details.

When you specify the REFERENCES WITH NO CHECK OPTION phrase, Teradata Database does not enforce the defined referential integrity constraint. This implies the following things about child table rows:

- A row having a value, meaning that the implication is false if the referencing column is null, for a referencing column can exist in a table even when no equivalent parent table value exists.
- A row can, in some circumstances, match *multiple* rows in its parent table when the referenced and referencing column values are compared.

This can happen because the candidate key acting as the primary key for the referenced table in the constraint need not be explicitly declared to be unique. See the list of rules for Referential Constraints later in this topic for details.

The various types have different applications as described in the following table:

Referential Constraint Type	Application
Referential Integrity Constraint (see “CREATE TABLE” in <i>SQL Data Definition Language</i> )	<ul style="list-style-type: none"> <li>• Tests each individual inserted, deleted, or updated row for referential integrity.</li> <li>• If insertion, deletion, or update of a row would violate referential integrity, then AMP software rejects the operation and returns an error message.</li> <li>• Permits special optimization of certain queries.</li> </ul>
Batch Referential Integrity Constraint (see “CREATE TABLE” in <i>SQL Data Definition Language</i> )	<ul style="list-style-type: none"> <li>• Tests an entire insert, delete, or update batch operation for referential integrity.</li> <li>• If insertion, deletion, or update of any row in the batch violates referential integrity, then parsing engine software rolls back the entire batch and returns an abort message.</li> <li>• Permits special optimization of certain queries.</li> </ul>
Referential Constraint (see “CREATE TABLE” in <i>SQL Data Definition Language</i> )	<ul style="list-style-type: none"> <li>• Does not test for referential integrity.</li> <li>• Assumes that the user somehow enforces referential integrity in a way other than the normal declarative referential integrity constraint mechanism.</li> <li>• Permits special optimization of certain queries.</li> </ul>

Referential Constraint Type	Application
<p>Temporal Relationship Constraint (TRC) (see <i>ANSI Temporal Table Support</i> and <i>Temporal Table Support</i>)</p>	<ul style="list-style-type: none"><li>• Does not test for referential integrity.</li><li>• Assumes that the user somehow enforces referential integrity in a way other than the normal declarative referential integrity constraint mechanism.</li><li>• Permits special optimization of certain queries.</li><li>• TRC relationships can only be defined at the table level.</li><li>• TRC relationships cannot be defined on self-referencing tables.</li></ul>

Referential Constraints do not enforce primary key-foreign key constraints between tables, so they avoid the overhead of RI enforcement by the system as practiced by standard and batch referential integrity constraints. Their only purpose is to provide the Optimizer with a means for devising better query plans. Referential Constraints should be used only for situations for which referential integrity is either not important or is enforced by other means, because its use implicitly instructs the system to trust the validity of all DML requests made against the affected columns and not to check the specified referential integrity relationships.

**Caution:** If referential integrity errors occur, data corruption can occur. Erroneous results can be returned if a DML request specifies a redundant RI join and the primary key-foreign key rows do not match.

For more information see “[Inclusion Dependencies](#)” on page 632 and “CREATE TABLE” in *SQL Data Definition Language*.

The table on the following pages summarizes the differences among the different referential constraints.

Referential Constraint Type	CREATE/ALTER TABLE Definition Clause	Description	Does It Enforce Referential Integrity?	Level of Referential Integrity Enforcement	Join Elimination Optimizations?	Derived Statistics Propagated Between Child and Parent Tables?	Pros	Cons
Standard Referential Integrity Constraint	REFERENCES	ANSI/ISO SQL:2011 compliant. Integrity enforcement done using a Referential Index (see “ <a href="#">Sizing a Reference Index Subtable</a> ” on page 866).	Yes	Row	Yes	Yes	<ul style="list-style-type: none"> <li>ANSI/ISO compliant.</li> <li>Logs RI violations in an error table.</li> <li>Efficient for low volume updates.</li> </ul>	Not efficient for medium to large updates because it is enforced one row at a time.
Batch Referential Integrity Constraint	REFERENCES WITH CHECK OPTION	Teradata extension to ANSI/ISO SQL:2011 standard. All or nothing for update operations. Enforced using the following methods: <ul style="list-style-type: none"> <li>Joining new child table keys to parent table to ensure they exist.</li> <li>Joining deleted parent table keys to child table to ensure they do not exist.</li> </ul>	Yes	Implicit transaction	Yes	Yes	Efficient for medium to large updates because uses the Optimizer to determine best way to make the join.	<ul style="list-style-type: none"> <li>Can be slower than regular RI for small updates.</li> <li>Does not log RI violations in an error table.</li> </ul>

Referential Constraint Type	CREATE/ALTER TABLE Definition Clause	Description	Does It Enforce Referential Integrity?	Level of Referential Integrity Enforcement	Join Elimination Optimizations?	Derived Statistics Propagated Between Child and Parent Tables?	Pros	Cons
Referential Constraint (“soft referential constraint”)	REFERENCES WITH NO CHECK OPTION	<p>Teradata extension to ANSI/ISO SQL:2011 standard.</p> <p>No RI enforcement by Teradata Database. You must ensure the integrity of the relationship yourself.</p> <p>Not necessary to define a UNIQUE constraint for the primary or alternate key column set.</p>	No	None	Yes	Yes	No cost for enforcing RI or uniqueness.	<ul style="list-style-type: none"> <li>Incorrect results or data corruption can occur if the RI constraint is not valid.</li> <li>Not an issue if you are certain about the integrity of the relationship.</li> </ul>
Temporal Relationship Constraint (“TRC constraint”)								<ul style="list-style-type: none"> <li>For information on temporal relationship constraints, see <i>ANSI Temporal Table Support</i> and <i>Temporal Table Support</i>.</li> </ul>

When you specify the REFERENCES WITH NO CHECK OPTION phrase, Teradata Database does not enforce the defined referential integrity constraint. This implies the following things about child table rows.

- A row having a non-null value for a referencing column can exist in a table even when no equivalent parent table value exists.
- A row can, in some circumstances, match *multiple* rows in its parent table when the referenced and referencing column values are compared.

This can happen because the candidate key acting as the primary key for the referenced table in the constraint need not be explicitly declared to be unique. See the list of rules for Referential Constraints later in this topic for details.

Referential Constraints do not enforce primary key-foreign key constraints between tables, so they avoid the overhead of RI enforcement by the system as practiced by standard and batch referential integrity constraints. Their only purpose is to provide the Optimizer with a means for devising better query plans. Referential Constraints should be used only for situations for which referential integrity is either not important or is enforced by other means, because its use implicitly instructs the system to trust the validity of all DML requests made against the affected columns and not to check the specified referential integrity relationships.

**Caution:** If referential integrity errors occur, data corruption can occur. Erroneous results can be returned if a DML request specifies a redundant RI join and the primary key-foreign key rows do not match.

For more information see “[Inclusion Dependencies](#)” on page 632 and “CREATE TABLE” in *SQL Data Definition Language*.

## Rules for Both Column-Level and Table-Level Foreign Key Constraints

The following rules apply to both column- and table-level FOREIGN KEY ... REFERENCES constraints:

- Teradata Database does not support the following ANSI/ISO SQL:2011 referential action options for FOREIGN KEY ... REFERENCES constraints:
  - MATCH {FULL | PARTIAL | SIMPLE}
  - ON UPDATE {CASCADE | SET NULL | SET DEFAULT | RESTRICT | NO ACTION}
  - ON DELETE {CASCADE | SET NULL | SET DEFAULT | RESTRICT | NO ACTION}
- The specified *column\_name* list must be identical to a set of columns in the referenced table that is defined as one of the following.
  - PRIMARY KEY
  - UNIQUE
  - A unique secondary index

This rule is not mandatory for Referential Constraints. See “[Foreign Key Constraints](#)” on page 644 for details.

The specified *table\_name* refers to the referenced table, which must be a user base data table, not a view.

- A maximum of 64 foreign keys can be defined for a table and a maximum of 64 referential constraints can be defined for a table.

Similarly, a maximum of 64 *other* tables can reference a *single* table. Therefore, there is a maximum of 128 Reference Indexes that can be stored in the table header per table.

The limit on Reference Indexes includes both references to and from the table and is derived from 64 references to other tables plus 64 references from other tables = 128 Reference Indexes.

However, only 64 Reference Indexes are stored per Reference Index subtable for a table, those that define the relationship between the table as a parent and its children.

Column-level CHECK constraints that reference alternate keys in other tables do not count against this limit.

- Each individual foreign key can be defined on a maximum of 64 columns.
  - Foreign key constraints cannot be defined on columns defined with any of the following data types:
    - XML
    - BLOB
    - CLOB
    - XML-based UDT
    - BLOB-based UDT
    - CLOB-based UDT
    - UDT
    - ARRAY/VARRAY
    - Period
    - Geospatial
    - JSON
  - Foreign key constraints cannot be defined on a global temporary trace table.
  - Note the following attributes of foreign key constraints:
    - They can be null.
    - They are rarely unique.
- An example of when a foreign key would be unique is the case of a vertical partitioning of a logical table into multiple tables.
- Each column in the foreign key constraint must correspond with a column of the referenced table, and the same column name must not be specified more than once.
  - The referencing column list should contain the same number of column names as the referenced column list. The  $i^{\text{th}}$  column of the referencing list corresponds to the  $i^{\text{th}}$  column identified in the referenced list. The data type of each referencing column must be the same as the data type of the corresponding referenced column.
  - The user issuing the CREATE TABLE request that specifies a foreign key constraint must either have the REFERENCE privilege on the referenced table or on all specified columns of the referenced table.

- If REFERENCES is specified in a *column\_constraint*, then *table\_name* defines the referenced table. Note that *table\_name* must be a base table, not a view.
- Referential constraints are not supported for global temporary, global temporary trace, or volatile tables.
- While it is possible to create a child table at a time that its parent table does not yet exist, a REFERENCES constraint that makes a forward reference to a table that has not yet been created cannot qualify the parent table name with a database name.  
In other words, the forward-referenced parent table that has not yet been created must be assumed to be “contained” in the same database as its child table that is currently being created.
- You cannot define a foreign key constraint on a row-level security-protected column.

## Rules for Column-Level Foreign Key Constraints Only

The following rules apply to column-level foreign key constraints only.

- You cannot specify the keywords FOREIGN KEY or specify a referencing column set in the definition of a column-level foreign key constraint.  
The referencing column for a column-level foreign key constraint is the column on which the foreign key constraint is defined by default.
- You cannot define a multicolumn foreign key constraint using the column-level foreign key syntax.
- If you omit *column\_name*, the referenced table must have a single-column primary key, and the specified foreign key column references that primary key column of the referenced table.
- If you specify *column\_name*, it must refer to the single-column primary key of the referenced table or to a single-column alternate key in the referenced table defined as UNIQUE.

This rule does *not* apply for Referential Constraints. In such cases, the candidate key acting as the primary key in the referenced table need not be *explicitly* declared to be unique using the PRIMARY KEY or UNIQUE keywords or by declaring it to be a USI in the table definition.

However, the candidate key in the relationship actually must *always* be unique even if it is not explicitly declared to be so. This is true by definition. See “[Identifying Candidate Primary Keys](#)” on page 90 and “[PRIMARY KEY Constraints](#)” on page 654 for details.

The uniqueness rule for candidate keys is necessary because the Optimizer always assumes that candidate keys are unique when it generates its query plans, and if that assumption is not true, it is not only possible to produce incorrect results to queries, but to corrupt databases. Because of this, you must always ensure that all candidate key values are unique even if they are not explicitly declared to be so.

As is always true when you specify Referential Constraints, you must assume responsibility for ensuring that any candidate key in a referential integrity relationship is unique, just as you must assume responsibility for ensuring that the referential relationship it anchors holds true in all cases, because Teradata Database enforces neither constraint.

## Rules for Table-Level FOREIGN KEY ... REFERENCES Constraints Only

The following rules applies to table-level FOREIGN KEY ... REFERENCES constraints only.

- You must specify a complete FOREIGN KEY (*referencing\_column\_set*) REFERENCES (*referenced\_table\_name*) specification for the foreign key definition.
- If you do not specify the optional *referenced\_column\_set* in the foreign key definition, Teradata Database assumes that the columns in the referencing column set have the identical names as the implied columns in the referenced column set.
- If the referenced column set columns do not have the same names as their counterparts in the *referencing\_column\_set* list, you must specify their names using the FOREIGN KEY (*referencing\_column\_set*) REFERENCES *referenced\_table\_name* (*referenced\_column\_set*) syntax.
- The optional keywords WITH CHECK OPTION and WITH NO CHECK OPTION define a foreign key constraint as being either a batch referential integrity constraint or a Referential Constraint, respectively.  
If you specify neither set of keywords, the foreign key constraint defines a traditional foreign key constraint by default.
- A maximum of 100 table-level constraints can be defined for any table.

## Rules for FOREIGN KEY ... REFERENCES Referential Constraints Only

Other than their not actually enforcing referential integrity, most of the rules for Referential Constraints are identical to those documented by “[Rules for Both Column-Level and Table-Level Foreign Key Constraints](#)” on page 649 and by “[Rules for Table-Level FOREIGN KEY ... REFERENCES Constraints Only](#)” on page 652.

The exceptions are documented by the following set of rules that apply specifically to the specification and use of Referential Constraints.

- You can specify standard RI, batch RI, and Referential Constraints in the same table, but not for the same column set.
- You can specify Referential Constraints for both of the following constraint types:
  - FOREIGN KEY (*FK\_column\_set*) REFERENCES (*parent\_table\_PK\_column\_set*)
  - (*NFK\_column\_set*) REFERENCES (*parent\_table\_AK\_column\_set*)  
where NFK indicates non-foreign key and *parent\_table\_AK\_column\_set* indicates an alternate key in the parent table.
- Referential Constraint references count toward the maximum of 64 references permitted for a table referenced as a parent even though they are not enforced by the system.
- INSERT, DELETE, and UPDATE requests are not permitted against tables with unresolved, inconsistent, or non-valid Referential Constraints. This rule is identical to the rule enforced for standard and batch RI.
- The candidate key acting as the primary key in the referenced table in the constraint need not be explicitly declared to be unique using the PRIMARY KEY or UNIQUE keywords or by declaring it to be a USI in the table definition.

## Referential Integrity Constraint Checks

Teradata Database performs referential integrity constraint checks whenever any of the following things occur:

- A referential integrity constraint is added to a populated table.
- A row is inserted or deleted.
- A parent or foreign key is modified.

The following table summarizes these actions:

Action Taken	Constraint Check Performed
INSERT into parent table	None.
INSERT into child table	Must have matching parent key value if the foreign key is not null.
DELETE from parent table	Abort the request if the deleted parent key is referenced by any foreign key.
DELETE from child table	None.
UPDATE parent table	Abort the request if the parent key is referenced by any foreign key.
UPDATE child table	New value must match the parent key when the foreign key is updated.

## Overhead Costs of Standard Referential Integrity

By implementing standard referential integrity, you incur certain overhead costs that can have a negative effect on performance. The following table lists the various referential integrity overhead operations that affect performance.

Operation	Description
Insert, update, or delete rows in a referencing table.	<p>Overhead is similar to that for USI subtable row handling.</p> <p>A redistributed spool for each reference is dispatched to the AMP containing the subtable entry.</p> <p>BYNET traffic incurs the majority of the cost of this operation.</p>
Insert a row into a referencing table	<p>A referential integrity check is made against the Reference Index subtable.</p> <ul style="list-style-type: none"> <li>• If the referenced column is in the Reference Index subtable, the count in the Reference Index subtable is incremented.</li> <li>• If the referenced column is not in the Reference Index subtable, Teradata Database first checks the Reference Index subtable to verify that the referenced column exists.</li> <li>• If it does, an entry with a count of 1 is added to the Reference Index subtable.</li> </ul>
Delete a row from a referencing table	<p>A referential integrity check is made against the Reference Index subtable and its count for the referenced field is decremented.</p> <p>When the count decrements to 0, then the subtable entry for the Referenced field is deleted.</p>

Operation	Description
Update a referenced field in a referencing table	Overhead is similar to that for changing the value of a USI column. A referential integrity check is made against the Reference Index subtable. Both the inserting-a-row and deleting-a-row operations execute on the Reference Index subtable, decrementing the count of the old Referenced column value and incrementing the count of the new Reference column value.
Delete a row from a referenced table	The Reference Index subtable is checked to verify that the corresponding Referenced column does not exist. When nonexistence is confirmed, the row is deleted from the Referenced table. No BYNET traffic is involved because the Referenced column is the same value in the Referenced table and the Reference Index subtable.

## PRIMARY KEY Constraints

PRIMARY KEY constraints specify the primary key column set in a table definition. Teradata Database uses primary keys to enforce both row uniqueness and referential integrity.

Whether a PRIMARY KEY constraint is treated as a column-level constraint or a table-level constraint depends on whether the primary key is simple or composite.

The following rules apply to PRIMARY KEY constraints.

- Only one primary key can be defined per table.
- The following table explains the column limits for column-level and table-level primary key constraints.

IF the PRIMARY KEY constraint is ...	THEN you must define it at this level ...
simple, or defined on a single column	column. You can define a simple PRIMARY KEY constraint at table-level, but there is no reason to do so.
composite, or defined on multiple columns	table. Defining a table-level PRIMARY KEY constraint is the only way you can create a multicolumn primary key.

- Defining a primary key for a table is never required, though it is recommended for documentation purposes as part of a policy of enforcing data integrity in those cases where the logical primary key is not chosen to be the unique primary index.
- A PRIMARY KEY constraint cannot be defined on the same column set as the set used to define the unique primary index for a table.  
If you attempt to define a PRIMARY KEY constraint on the same column set that defines the primary index for a table, the request aborts and returns a message to the requestor.
- A primary key can be defined on a maximum of 64 columns.

PRIMARY KEY constraints are treated as ...	When the primary key is defined on this many columns ...
column-level constraints	1
table-level constraints	> 1

A maximum of 100 table-level constraints can be defined for any table.

- When a PRIMARY KEY constraint is defined, it is implemented physically as either a UNIQUE NOT NULL secondary index or as a single-table join index.

If you attempt to define a PRIMARY KEY constraint on the same column set that defines the primary index for a table, the request aborts and returns a message to the requestor.

Note that in physical database design, candidate keys, whether chosen to be a primary key or not, are always defined internally as either a UNIQUE NOT NULL secondary index or as a single-table join index.

- You cannot define a PRIMARY KEY constraint for a NoPI table.

If you attempt to do so, Teradata Database does not return an error, but instead converts the PRIMARY KEY constraint specification to a UNIQUE NOT NULL secondary index specification. As a result of this conversion, if you submit a SHOW TABLE request on such a table, the create text that the system returns does not show a PRIMARY KEY specification, but instead returns a UNIQUE NOT NULL secondary index specification on the column set that had been specified for the primary key.

Note that this is different from the ordinary *implementation* of a PRIMARY KEY constraint column set as a UNIQUE NOT NULL secondary index because Teradata Database actually changes the stored create text for a NoPI table defined with a PRIMARY KEY constraint, it does not simply implement the constraint as a USI.

- PRIMARY KEY constraints cannot be defined on columns defined with any of the following data types.
  - XML
  - BLOB
  - CLOB
  - XML-based UDT
  - BLOB-based UDT
  - CLOB-based UDT
  - ARRAY/VARRAY
  - Period
  - Geospatial
  - JSON
- PRIMARY KEY constraints cannot be defined on a global temporary trace table.
- You cannot define a PRIMARY KEY on a row-level security-protected column.

Other than providing documentation, PRIMARY KEY constraints *per se* have little value except for establishing referential integrity constraints (see “[Foreign Key Constraints](#)” on page 644). Even this usage is generally of no use, because the logical primary key for a table is frequently declared to be the unique primary index of the physical table it models, and you cannot declare a column set to be both a unique primary index *and* a primary key.

For applications where a logical single-column primary key is *not* chosen to be the unique primary index, use the UNIQUE NOT NULL constraint in its place.

## UNIQUE Constraints

UNIQUE constraints specify that the column set they modify must contain unique values. Teradata Database implements UNIQUE constraints as either a unique secondary indexes or as a single-table join index.

The following rules apply to UNIQUE constraints:

- UNIQUE constraints should always be specified with a NOT NULL attribute specification. Otherwise, it is possible for a single null to be inserted into a uniquely constrained column. The semantics of a unique null “value” are uncertain at best, and almost certainly violate the intent of the uniqueness constraint.
- UNIQUE constraints can be defined at column-level (simple) or at table-level (composite).

The following table explains the column limits for column-level and table-level primary key constraints.

IF the UNIQUE constraint is ...	THEN you must define it at this level ...
simple, or defined on a single column	column. You can define a simple UNIQUE constraint at table-level, but there is no reason to do so.
composite, or defined on multiple columns	table. Defining a table-level constraint is the only way you can create a multicolumn UNIQUE constraint.

- Column-level UNIQUE constraints refer only to the column on which they are specified.
- Table-level UNIQUE constraints can be defined on multiple columns by specifying a column name list.
- A table-level UNIQUE constraint can be defined on a maximum of 64 columns.
- A maximum of 100 table-level constraints can be defined for any table.
- You can define a UNIQUE constraint for a column-partitioned table.  
You cannot define a UNIQUE constraint for a non-partitioned NoPI table.  
If you attempt to do so, Teradata Database does not return an error, but instead converts the UNIQUE constraint specification to a UNIQUE NOT NULL secondary index specification. As a result of this conversion, if you submit a SHOW TABLE request on such

a table, the create text that the system returns does not show a UNIQUE constraint specification, but instead returns a UNIQUE NOT NULL secondary index or single-table join index specification on the column set that had been specified for the UNIQUE constraint.

Note that this is different from the ordinary *implementation* of a UNIQUE constraint column set as a UNIQUE NOT NULL secondary index because Teradata Database actually changes the stored create text for a NoPI table defined with a UNIQUE constraint, it does not simply implement the constraint as a USI.

- UNIQUE constraints cannot be defined on columns defined with any of the following data types:
  - XML
  - BLOB
  - CLOB
  - XML-based UDT
  - BLOB-based UDT
  - CLOB-based UDT
  - ARRAY/VARRAY
  - Period
  - Geospatial
  - JSON
- UNIQUE constraints cannot be defined on a global temporary trace table.
- If a row-level security-protected table is defined with a UNIQUE constraint, enforcement of the constraint does not execute any security policy defined for the table. UNIQUE constraints are applicable to *all* rows in a row-level security-protected table, not just to user-visible rows.
- You cannot define a UNIQUE constraint on a row-level security constraint column of a row-level security-protected table.

## Semantic Constraint Enforcement

The enforcement of a constraint depends on how the base table on which it is defined is accessed. If the base table is accessed directly, then its column and table constraints are always enforced. Date (2001a) calls this “The Golden Rule,” which he defines as follows: No update operation must ever assign to any database a value that causes its database predicate to evaluate to false. This, of course, is a generalization of the Closed World Assumption (see “[The Closed World Assumption](#)” on page 630 and “[The Closed World Assumption Revisited](#)” on page 677).

By this definition, the checking time must always be immediate for any update; that is, the constraint check is made at statement boundaries, not deferred for checking at transaction (COMMIT time) boundaries. If this were not the case, then inconsistent, or false, data could be entered into the database, even if only for a brief time and even if the inconsistency were, as

it must be, private to the transaction in question. It would still be possible for a query that followed this inconsistent update within the boundaries of an explicit transaction to report erroneous information. Teradata Database does not support deferred integrity checking.

If a base table is accessed by means of a view, then the enforcement of any WHERE clause constraints specified in the view definition depends on whether the view is also defined WITH CHECK OPTION or not (see “[Semantic Integrity Constraints for Updatable Views](#)” on page 659).

Views inherit the constraints of their underlying base tables; therefore, base table constraints cannot be violated by updating through a view. However, additional constraints defined by means of a WHERE clause specification in a view definition *can* be bypassed if the view is not also defined WITH CHECK OPTION. The result is not a violation of any constraints defined on underlying base tables, but the insertion of a row that cannot be seen from that view.

## Updatable Cursors and Semantic Database Integrity

Positioned updates using updatable cursors can present semantic integrity problems unless strict locking protocols are observed. Cursor updates (as used here, the word *update* also includes deletes) must be executed within the same transaction as the SELECT statement that defined the cursor.

Because Preprocessor2 does not support large objects, you cannot use cursors of any kind on columns defined with the XML, BLOB, or CLOB data types.

### Between-Transaction Integrity Issues

By default, updatable cursors use READ locks, which are implemented as row hash locks internally. READ locks are adequate for preserving database integrity because they prevent access to the database by users who attempt to either change the definitions of database structures or to update table data. As a result, stored data cannot be changed while an open cursor is also manipulating the same data in such a way that it updates the same fields of the same rows. This is sometimes referred to as repeatable read mode. Programmers can also upgrade the locks used by updatable cursors to WRITE or even EXCLUSIVE locks if they so desire.

The potentially most severe problems for database integrity regarding updatable cursors are presented by the use of ACCESS or CHECKSUM locks, because both of these locks permit rows to be updated by other users while an open cursor is manipulating them, thus providing greater concurrency (note that CHECKSUM locks *are* ACCESS locks). The difference is that CHECKSUM locks compute a checksum value for updated rows and compare it with the checksum value computed for the row at the time the cursor accessed it, while ACCESS locks provide no such integrity check.

CHECKSUM locks provide better integrity than simple ACCESS locks because they compute a checksum value for each row updated by an open cursor (see the chapter “[Transaction](#)

Processing” in *SQL Request and Transaction Processing* for details). When the current transaction commits and an updated row is written to disk, the checksum value for that row is compared with the value computed for the row at the time it was accessed and if the values differ, the update is not permitted.

There is a finite probability that pre- and post-updated rows will produce an identical checksum value, so this option is only advisable for those situations where performance concerns significantly override integrity concerns. The most important fact to understand about the various locking options for positioned updates via updatable cursors is that there is no integrity risk with READ and more restrictive locks, but there is a finite integrity risk with CHECKSUM locks, and an almost certain integrity risk with ACCESS locks.

## Within-Transaction Integrity Issues

“[Between-Transaction Integrity Issues](#)” on page 658 describes the integrity risks presented by concurrent updating of a table by different transactions. This topic describes the risks associated with cursor conflicts that occur within an individual transaction.

Cursor conflicts occur in the following situations:

- Two cursors are open on the same table at the same time. One attempts a positioned update, but the other has already made a positioned update on the same row, thereby modifying the table.
- While a cursor is open, a searched update is made on a row, and then the cursor attempts a positioned update on that row.
- While a cursor is open, a positioned update is made on a row through that cursor and then a searched update is attempted on the same row.

All three of these updates are dirty because the same row was updated twice without the transaction having been committed.

Note that the system permits all three of the dirty updates to occur and then returns a cursor conflict warning to the requesting program. Resolution of the integrity breach is left to the user.

Because of the multiple possible integrity risks associated with declaring and opening multiple cursors within the same transaction, you should code your applications so those risks cannot occur.

## Semantic Integrity Constraints for Updatable Views

Besides defining constraints on your base tables, you can specify additional constraints on a user-by-user basis through updatable views by specifying constraints in a WHERE clause and enforcing them by specifying a WITH CHECK OPTION clause. Because views inherit the constraints defined for their underlying base tables as well as those defined for any

intermediate views, the constraints they inherit from their underlying relations are called derived constraints.

## Specifying Integrity Constraints in an Updatable View Definition

Not all views are updatable. You can tailor update constraints at the user level by specifying user-specific constraints in the WHERE clause of a view as long as you also specify a WITH CHECK OPTION clause.

WITH CHECK OPTION pertains only to updatable views. Views are typically created to restrict which base table *columns* and *rows* a user can access in a table. Base table column projection is specified by the columns named in the *column\_name* list, while base table row restriction is specified by an optional WHERE clause.

The WITH CHECK OPTION clause is an integrity constraint option that restricts the rows in the table that can be affected by an *INSERT or UPDATE* statement to those defined by the WHERE clause. If you do not specify WITH CHECK OPTION, then the integrity constraints specified by the WHERE clause are not checked for updates. Derived constraints inherited from underlying relations are not affected by this circumvention and continue to be enforced. The problem is that the view that updated the row in a way that violates the WHERE clause cannot view the updated row, so it cannot be updated in the future through that view.

WHEN WITH CHECK OPTION is ...	THEN any insert or update made to the table through the view ...
specified	only inserts or updates rows that satisfy the WHERE clause.
not specified	ignores the WHERE clause used in defining the view.

The following rules apply to updatable views and the WITH CHECK OPTION.

- If WITH CHECK OPTION is specified, the view is updatable. Any insert or update to the table through the view is rejected if the WHERE clause predicate evaluates to false.
- If WITH CHECK OPTION is not specified in an updatable view, then any WHERE clause contained in the view definition is ignored for any insert or update action performed through the view. In other words, the specified integrity constraints in the view definition are ignored, so you should always specify a WITH CHECK OPTION clause to define constrained updatable views unless you have an extraordinary reason not to.
- You can define nested views that only reference a single base table, which might allow the views to be updatable. In this case, the specification of a WITH CHECK OPTION clause in the view definition permits the WHERE clause constraints for that view, as well as those defined for any underlying views, to be exercised in the constraint on INSERT or UPDATE. See “[Updatable View Inheritance](#)” on page 662 for more information.

The following request creates a view of the *employee* table so that it provides access only to the names and job titles of the employees in department 300:

```
CREATE VIEW dept300 (Name, JobTitle) AS
    SELECT name, job_title
    FROM employee
    WHERE dept_no = 300
```

```
WITH CHECK OPTION;
```

The WITH CHECK OPTION clause in this example prevents you from using the Dept300 view to update a row in the Employee table for which DeptNo <> 300.

This example shows how using a view in an UPDATE, INSERT, or DELETE statement allows you to add, change, or remove data in the base table set on which the view is defined.

Consider the following *staff\_info* view, which provides a personnel clerk with retrieval access to employee numbers, names, job titles, department numbers, sex, and dates of birth for all employees except vice presidents and managers:

```
CREATE VIEW staff_info (number, name, position,
    department, sex, dob) AS
    SELECT employee.empno, name, jobtitle, deptno, sex, dob
    FROM employee
    WHERE jobtitle NOT IN ('Vice Pres', 'Manager')
    WITH CHECK OPTION ;
```

If the owner of *staff\_info* has the insert privilege on the *employee* table, and if the clerk has the insert privilege on *staff\_info*, then the clerk can use this view to add new rows to *employee*.

For example, this request inserts a row into the underlying *employee* table that contains the specified information:

```
INSERT INTO staff_info (number, name, position, department, sex, dob)
    VALUES (10024, 'Crowell N', 'Secretary', 200, 'F', 'Jun 03
    1960');
```

**Note:** The constraint on *staff\_info* illustrated by the following WHERE clause applies to any insert using this view that includes the WITH CHECK OPTION phrase.

```
...
WHERE jobtitle NOT IN ('Vice Pres', 'Manager')
...
```

Therefore, the preceding INSERT statement would fail if the *position* entered for employee Crowell was 'Vice Pres' or 'Manager'.

**Note:** If this view were defined without the WITH CHECK OPTION, and a user had the UPDATE privilege on the table, that user could update a job title to 'Vice Pres' or 'Manager'. The user would be unable to access the changed row through the view.

The following statement changes the department number typed for Crowell in the preceding INSERT request from 200 to 300:

```
UPDATE staff_info
    SET department = 300
    WHERE number = 10024;
```

Performing the following DELETE request removes the row for employee Crowell from the *staff\_info* table.

```
DELETE
    FROM staff_info
    WHERE number = 10024;
```

Views are a useful method for permitting selected users to have restricted access to base table data. However, as the preceding examples suggest, granting another user insert, update, and

delete privileges on a view means relinquishing some control over your data. Carefully consider granting such privileges.

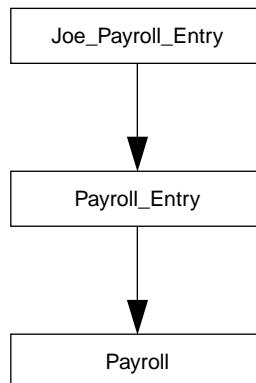
The default is *not* to constrain updated or inserted values unless the view definition explicitly includes WITH CHECK OPTION.

## Updatable View Inheritance

All base table integrity constraints are enforced without exception. Another way of stating this is to say that updatable views inherit the integrity constraints of their underlying base tables.

Because views can be nested, it is also true that updatable views inherit the integrity constraints of their underlying updatable views (when such relationships exist) as derived constraints in addition to inheriting the integrity constraints of their underlying base tables.

For example, suppose you have two nested views defined on the *payroll* table as follows.



GG02A005

When user Joe logs on, he is assigned the view *joe\_payroll\_entry* defined with the following WITH CHECK OPTION integrity constraint.

```
WHERE dept_no = 1350
WITH CHECK OPTION
```

Joe can only update values for department 1350.

View *joe\_payroll\_entry* is defined on top of view *payroll\_entry*, which restricts the ability to update payroll for any employee who earns a base salary of 200,000 USD or greater. This constraint is specified by the following WHERE clause in the view definition:

```
WHERE base_salary < 200000
WITH CHECK OPTION
```

Because of this derived constraint, Joe can only update annual salaries less than 200,000 USD in department 1350.

View *payroll\_entry* is defined on top of base table *payroll*, which restricts the ability to update payroll to only those employees who are US citizens or who have valid work visas. This constraint is specified by the following column-level CHECK constraint in the table definition:

```
visa_code CHARACTER(2)
```

```
CONSTRAINT ok2work CHECK (visa_code IN ('US', 'WV'))
```

Because of this additional derived constraint, Joe can only update salaries of US citizens or individuals with valid work visas who work in department 1350 and earn annual salaries less than 200,000 USD.

## Summary of Fundamental Database Principles

This chapter and [Chapter 5: “The Normalization Process”](#) has presented a number of fundamental principles of relational database management. The following summary of those rules, including several introduced for the first time, is based in part on Date (2005, 2009):

Principle	Definition
Entity integrity rule	<p>The attributes of the primary key of a relation cannot be null. More accurately, this rule applies to any candidate key of a relation, not just to the candidate key chosen to be the primary key.</p> <p>Note that because this rule is explicitly intended to apply to base relations only and not virtual relations (views), it violates the Principle of Interchangeability.</p> <p>See “<a href="#">Rules for Primary Keys</a>” on page 92.</p>
Referential integrity rule	<p>There cannot be any unmatched foreign key values.</p> <p>Restated more formally, assume a primary key value PK in relvar <math>T_1</math> and a candidate key value FK in relvar <math>T_2</math> that references it.</p> <p>The Referential Integrity Rule states that if FK references PK, then PK must exist.</p> <p>Note that the Referential Integrity Rule permits the attributes of FK to be wholly or partly null, which violates the Entity Integrity Rule because FK must reference a candidate key in <math>T_1</math>, and no attribute of a candidate key can be null.</p> <p>See “<a href="#">The Referential Integrity Rule</a>” on page 95.</p>
Information principle	A relational database contains nothing but relation variables. In other words, the information content of a relational database at any given instant is represented as explicit values (recall that nulls are <i>not</i> values) in attribute positions in tuples in relations.
Closed world assumption	If a tuple $t$ could appear in relation variable $R$ at a given instant, but does not appear in that relvar, then the logical proposition $p$ that corresponds to $t$ must evaluate to FALSE at that time. See “ <a href="#">The Closed World Assumption</a> ” on page 630 and “ <a href="#">The Closed World Assumption Revisited</a> ” on page 677.
Principle of interchangeability	No arbitrary or unnecessary distinctions shall be made between base relations (base tables) and virtual relations (views).

Principle	Definition
Principles of normalization	<ul style="list-style-type: none"> <li>• A relation variable that is not in 5NF should be decomposed into a set of 5NF projections.</li> <li>• The decomposition must be non-loss.</li> <li>• The decomposition must preserve dependencies.</li> <li>• Every projection on the non-5NF relvar must be required to reconstruct the original relvar from those projections.</li> <li>• The decomposition should stop as soon as all of its relation variables are in 5NF (or, situation permitting, 6NF).</li> </ul>
Principle of orthogonal design	<p>Let <math>T_1</math> and <math>T_2</math> be distinct relation variables contained within the same database.</p> <p>There must not be a non-loss decomposition of <math>T_1</math> and <math>T_2</math> such that the relation variable constraints for some projection of <math>T_1</math> and some projection of <math>T_2</math> in those decompositions are such as to permit the same tuple to appear in both of the projections.</p>
Assignment principle	After value $v$ is assigned to variable $V$ , the comparison $V=v$ <i>must</i> evaluate to TRUE.
Golden rule	<p>No update operation can ever cause any database constraint to evaluate to FALSE.</p> <p>In other words, no statement can leave any relvar with a value that violates its relvar predicate.</p> <p>See “<a href="#">Relations, Relation Values, and Relation Variables</a>” on page 627.</p>
Principle of the identity of indiscernibles	<p>Every entity has its own identity.</p> <p>In more formal terms, let <math>E_1</math> and <math>E_2</math> be any two entities. If there is no way to distinguish between <math>E_1</math> and <math>E_2</math>, then <math>E_1</math> and <math>E_2</math> are identical: they are one thing, not two.</p>

## Physical Database Integrity

Physical database integrity checking mechanisms usually detect data corruption caused by lost writes or bit, byte, and byte string errors. Most hardware devices protect against data corruption automatically by means of various error detection and correction algorithms. For example, bit- and byte-level corruption of disk I/O is usually detected, and often corrected, by error checking and correcting mechanisms at the level of the disk drive hardware, and if the corruption is detected but cannot be corrected, the pending I/O request fails.

Similarly, bit- and byte-level corruption of an I/O in transit might be detected by various parity or error checking and correcting mechanisms in memory and at each intermediate communication link in the path. Again, if the corruption is detected, but cannot be corrected, the pending I/O request fails.

## CHECKSUM Integrity Checking and Physical Database Integrity

Unfortunately, not all hardware devices have end-to-end error detection and correction schemes in place, and it is possible for corrupted data to be written to the database. To minimize this problem, Teradata Database provides a mechanism for DBAs to selectively enforce physical data integrity at the level of disk I/O. When implemented, the feature checksums everything from system memory to disk, thus enforcing true end-to-end data integrity checking (see “[Disk I/O Integrity Checking](#)” on page 666 and “[Integrity Checking Using a Checksum](#)” on page 668).

You can specify various levels of corruption detection rigor by selecting among mechanisms that use various degrees of sampled checksums, including full checksums, to detect disk I/O corruption. Teradata Database controls the levels of physical integrity enforcement in two complimentary ways:

- The default sampled checksum rigor is user-definable at the system-level by means of the DBS Control utility.
- The specification of the checksum rigor at the level of individual base tables is user-definable in DDL by means of the ALTER TABLE, CREATE HASH INDEX, CREATE JOIN INDEX, and CREATE TABLE statements.

## FALLBACK Protection and Physical Database Integrity

Fallback protection is another important data integrity mechanism. Fallback works by writing the same data to two different AMPs within a cluster. If the AMP that manages the primary copy of the data goes down, you can still access the fallback version from the other AMP.

Keep in mind that if you specify fallback for a table, you double the amount of disk space required to store the same quantity of data. The amount of disk space required by a table is also doubled if you configure your system for RAID1 mirroring. This means that if you configure your disk for RAID1 mirroring and also specify fallback protection for a table, you actually quadruple the amount of disk space required to store the same quantity of data.

Also be aware that when a table is defined with fallback, it imposes a performance penalty for all DELETE, INSERT, and UPDATE operations on the table because each such operation must be executed twice in order to update both the primary table and its fallback table.

The system defaults to bringing Teradata Database up when AMPs are down on the assumption that any down AMPs can run in fallback mode. If your site does not use fallback for its critical tables, you probably want to keep Teradata Database down in this situation. To enable the logic to keep Teradata Database down, you can use the Maximum Fatal AMPs option from the SCREEN command of the CTL DBS Control utility (see *Utilities: Volume 1 (A-K)* for documentation of the SCREEN command).

Teradata Database also provides a means for using fallback to deal with read errors caused by bad data blocks (see “[About Reading or Repairing Data from Fallback](#)” on page 672).

# Disk I/O Integrity Checking

Not all problems with data integrity are the result of keypunch errors or semantic integrity violations. Problems originating in disk drive and disk array firmware can also corrupt user data, typically at the block or sector levels. Block- and sector-level errors are the most common origins of disk I/O corruption encountered in user data.

A major problem with handling this type of data corruption is that it generally is not detected until some time after it has occurred. As a result, queries against the corrupted data return semantically correct, but factually incorrect answer sets, and update or delete operations can either miss relevant rows or can change them in error. Once the system detects the corruption, the affected AMP is typically taken offline, various utilities such as ScanDisk and CheckTable are performed, and the data is either repaired or reloaded. Each of these actions either removes access to, or reduces the availability of, the data warehouse to its users until corrections have been made.

## Levels of Disk I/O Integrity Checking

To protect against physical data corruption, Teradata Database permits you to select various levels of disk I/O integrity checking of your table, hash index, and join index data. Secondary index subtables assume the level of disk I/O integrity checking that is defined for their parent table or join index. Furthermore, you can specify corruption checking levels at both system and table levels. These checks detect corruption of disk blocks (checksum sampling can also detect some forms of bit and byte corruption) using one of the following integrity methods, which are ranked in order of their ability to detect errors:

- 1 Full end-to-end checksums.  
Detects lost writes and most bit, byte, and byte string errors.
- 2 Statistically sampled partial end-to-end checksums.  
Detects lost writes and intermediate levels of bit, byte, and byte string errors.
- 3 No checksum integrity checking.  
Detects some forms of lost writes using standard file system metadata verification.

## Disk I/O Integrity Checking Detects and Logs Errors But Does Not Fix Them

This feature detects and logs disk I/O errors: it does not fix them. When the system detects data corruption, it removes the affected AMP from service and you must then take the appropriate measures to repair the corrupted data.

# Detecting Data Corruption Using Disk I/O Integrity Checksums

If both data and corruption patterns are random, then the likelihood that a single 64-bit checksum sample erroneously matches a single corrupted disk sector is only 1 in  $2^{64}$ . Obviously, neither your data nor any potential corruption of it is random, so the likelihood of a false negative is somewhat smaller than 1 in  $2^{64}$ , but the efficiency of detecting corruption is still excellent.

For a checksum to fail to detect block-level corruption, the corrupt data from which it is derived would need to match identically the data in the original, uncorrupted sector.

For example, assume that all but one of the 64-bit values in a data block is unique. The following table shows the effect of varying the probability of occurrence of the nonunique value and the number of corrupted sectors. The cell values are the likelihoods that the checksum does not catch an error. Shaded cells indicate that all corruption is detected.

Number of Corrupted Sectors	Probability of a Repeated Value (Percent)					
	1	2	5	10	20	50
1	0.010	0.040	0.250	1.000	4.000	25.000
2	0.000	0.000	0.001	0.010	0.160	6.250
3	0.000	0.000	0.000	0.000	0.006	1.563
4	0.000	0.000	0.000	0.000	0.000	0.391
5	0.000	0.000	0.000	0.000	0.000	0.098
10	0.000	0.000	0.000	0.000	0.000	0.000
20	0.000	0.000	0.000	0.000	0.000	0.000

The more words used to generate the checksum value, the better able that checksum is to detect disk I/O errors; however, the rate of increased corruption detection diminishes rapidly, and there is often little to be gained by using higher checksum sample percentages for most applications. See “[Integrity Checking Using a Checksum](#)” on page 668 for more about this.

Because CPU utilization increases as a function of the number of words used to generate a checksum, several different levels of checking are provided so you can adjust disk I/O integrity checking at the level of the entire system or of individual tables as needed, balancing integrity checking against system performance.

You can specify system-wide checksum defaults for various table types using the Checksum Levels fields of the DBS Control utility Disk I/O Integrity fields. You can also specify the system-wide checksum level definitions for the LOW, MEDIUM, and HIGH values using the same fields. See *Utilities: Volume 1 (A-K)* and *Database Administration* for details.

# Integrity Checking Using a Checksum

## Disk I/O and File System Data Structures

Teradata Database disk I/Os read from and write to the following data and file structures:

- Data blocks (DBs)
- Cylinder indexes (CIs)
- Master index (MI)
- File information block (FIB)
- DEPOT
- WAL Data Blocks (WDBs)
- WAL Cylinder Indexes (WCIs)

The file system first folds each sampled 64-bit check data segment into a 32-bit chunk using a barrel shift method that maximizes the minimum distance between the folded bytes to optimize the likelihood of detecting bursty corruption of adjacent bytes while conserving storage space in the cylinder index. It then computes the checksum by XORing the individual sample set together. See the illustration of how this XORing to compute a checksum is done in the topic [“Creating and Verifying File System Checksums” on page 668](#).

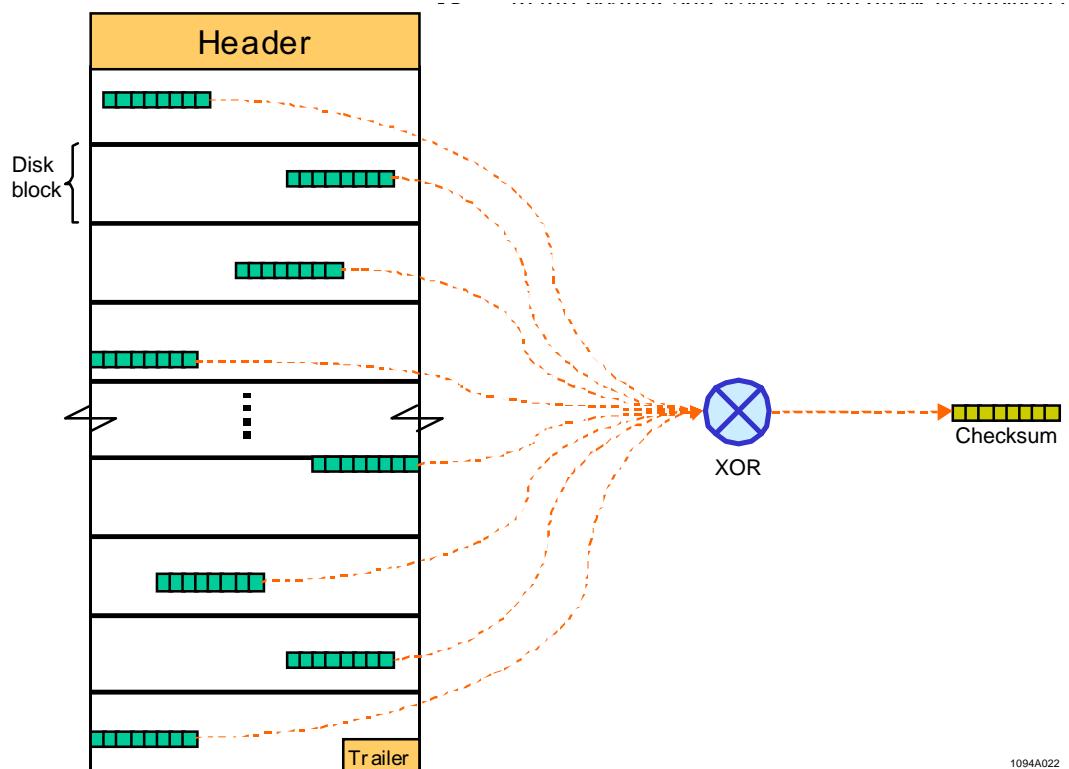
Reads and writes to file system structures are not made in smaller units than the file system structures listed. Because of this, a full checksum of the data in units of a DB and CI is sufficient to verify its integrity when a read completes on one of these structures.

## Creating and Verifying File System Checksums

Creating or verifying a full checksum of file system structures is a very CPU-intensive operation. To reduce CPU cycles, you can specify a sampled statistical checksum for a base table, where only a portion of the data is XORed. The sampling algorithm that generates the checksum from the block optimizes the detection of as many data corruption cases as possible using as few CPU cycles as possible. Even when sampling only one 64-bit word per disk block to generate a checksum, roughly 99.9% of disk block-level corruption can be detected as well as some bit, byte, or byte string corruption.

The system varies the location of the words to be checked in each disk sector of a file system block to avoid favoring certain portions of a disk sector over others. The system samples 64-bit words from each disk sector when it generates the checksum.

The following drawing abstracts the process of computing a checksum by sampling a single word per disk block and then XORing all those individual samples into a single checksum value.



Because the number of 64-bit words sampled per disk block affects performance, the number of words to sample is tunable at both system and table levels. However, if you change this value for an individual base table, the affected checksum file structures are not recalculated to match the new setting until the next time the block is modified, unless you also specify IMMEDIATE in the checksum option clause of an ALTER TABLE request. The typical applications for the IMMEDIATE option are 1) to turn checksums off for the table in question or 2) to initiate checking immediately when disk array problems have been noted. see “ALTER TABLE” in *SQL Data Definition Language*.

After an ALTER TABLE request without the IMMEDIATE option has been executed, the file system recalculates checksums only when the file system structure is again written to during the normal activity of database I/Os. At that time, the system recalculates the checksum using the newly specified sample algorithm.

## Disk I/O Integrity Level Settings Based on Table Type

You can choose the level of disk I/O integrity checking to perform based on the table type by means of the DBS Control utility.

## Default Definitions for Checksum Levels

You can use these keywords to specify table-level checksums for user data tables, join indexes, and hash indexes. You can either redefine their specific meanings using the DBS Control utility or use their Teradata Database-defined default meanings as defined in the following table.

Note that you *must* use the default settings for the NONE and ALL keywords. Only the meanings for LOW, MEDIUM, and HIGH are user-definable.

Disk I/O Integrity Checking Level	Checksum Sample Count Percentage	Comments
DEFAULT	Depends	Uses the system-wide default setting for this type of table as defined by the DBS Control utility.  See “ <a href="#">Eligible Table Types for Setting System-Wide Disk I/O Integrity Checking Levels Using DBS Control</a> ” on page 671, and <i>Utilities: Volume 1 (A-K)</i> for more information.
NONE	0	Checksum integrity checking disabled.  Detects some forms of lost writes.  A lost write is a write of a file system block that received successful status on completion, but the write either never actually occurred or it was written to an incorrect location. As a result, subsequent reads of the same block return the old data.
LOW	2	Samples one word (2%) of data per disk sector to compute the checksum value.  Detects all forms of lost write and whole disk sector corruption.  Lowest likelihood of detecting bit, byte, or byte string corruption.
MEDIUM	33	Samples one third (33%) of the data per disk sector to compute the checksum value.  Detects all forms of lost write and whole disk sector corruption.  Higher likelihood of detecting bit, byte, or byte string corruption than LOW sampling level.
HIGH	67	Samples two thirds (67%) of the data per disk sector to compute the checksum value.  Detects all forms of lost write and whole disk sector corruption.  Higher likelihood of detecting bit, byte, or byte string corruption than MEDIUM sampling level.
ALL	100	Uses all (100%) data to compute the checksum value.  Detects 99.99999998% of all forms of data corruption.  Strongly recommended for support of the read from fallback feature.

You can specify system-wide integrity levels at the table type level as well as on an individual base table basis. See “[Eligible Table Types for Setting System-Wide Disk I/O Integrity Checking Levels Using DBS Control](#)” on page 671, and *Utilities: Volume 1 (A-K)* for more information.

## DDL Statements for Specifying Individual Table-Level Integrity Checking Levels

You define base table-specific integrity checking values using the appropriate CREATE or ALTER statement drawn from the following list.

- CREATE HASH INDEX
- CREATE JOIN INDEX
- CREATE TABLE
- ALTER TABLE

Note that the secondary index subtables defined using either a CREATE INDEX request or a CREATE TABLE request automatically inherit their CHECKSUM I/O integrity level from their parent table or join index. See *SQL Data Definition Language* for information about specifying disk I/O integrity using these statements.

## Eligible Table Types for Setting System-Wide Disk I/O Integrity Checking Levels Using DBS Control

You can select different disk I/O integrity levels for the following table types using the DBS Control utility.

Table Type	Description
System	All nonjournal dictionary tables, session tables, and the WAL log. There are additional system-level defaults for System Journal tables and System Login tables.
User base tables	All user data tables, including the following. <ul style="list-style-type: none"> <li>• Stored procedures</li> <li>• Join indexes</li> <li>• Secondary index subtables</li> <li>• User-defined functions</li> <li>• Hash indexes</li> <li>• Associated fallback tables</li> </ul>
Permanent journal tables	All user permanent journal tables.
Temporary tables	All temporary tables including the following: <ul style="list-style-type: none"> <li>• Global temporary tables</li> <li>• Spool tables</li> <li>• Volatile tables</li> </ul>

## CPU Utilization As a Function of Number of Checksum Words

The performance of disk I/O integrity checking is an explicit tradeoff between the amount of data sampled and CPU utilization. As the number of words per disk block used to generate a checksum increases, the probability of detecting bit, byte, and byte string corruption

increases, but the rate of increase of detection slows. The additional computation required to generate more accurate checksum values also causes CPU utilization to increase.

Because it is not possible to know how many words need to be checked per disk block to ensure both optimal detection of corruption and minimal CPU utilization, the number of words to check is user-tunable. You can specify sample counts ranging from 0 to 64 64-bit words per disk block at both system and table levels. The default sample count is one word per disk block.

## About Reading or Repairing Data from Fallback

When the file system detects a hardware read error (see “[Conditions That Support Reading or Repairing Data from Fallback](#)”), the AMP that owns the bad data block sends a message to each of the other AMPs in its cluster. This message contains the tableID and rowID range of the unreadable data block. Each of the other AMPs in the cluster then reads its fallback rows in that range and sends them to the requesting AMP. There, the rows from each AMP are reconstructed as a valid data block that can be used instead of the unreadable data block. In some cases, the system can repair the damage to the primary data dynamically. In other cases, attempts to modify rows in the bad data block fail. Instead, the system substitutes an error-free fallback copy of the corrupt rows each time the read error occurs.

To avoid the continued overhead of data block substitution when dynamic repair cannot be performed, rebuild the primary copy of table data manually from the fallback copy using the Table Rebuild utility. For details, see *Utilities: Volume 2 (L-Z)*. You cannot use Table Rebuild to rebuild a hash index, join index, or USI, so if the corruption occurs in a hash index, join index, or USI subtable, you must drop the index and recreate it.

To detect all such read errors, the integrity checking level for the table or index should be set to ALL (see “[Disk I/O Integrity Checking](#)” on page 666).

### Conditions That Support Reading or Repairing Data from Fallback

Reading or repairing data from fallback is limited to the following table, subtable, and data block types:

- Hashed tables.
- Primary hash index, join index, and USI subtable data blocks.

Reading or repairing data from fallback cannot be done for the following table types:

- Unhashed tables.
- BLOB, CLOB, or XML subtables.
- NUSI subtables, whether their parent table is defined with fallback or not.

Reading or repairing data from fallback can only be used when:

- All AMPs in the fallback cluster are up.
- Requests do not try to modify the data in the bad data block.

# CHAPTER 13 Designing for Missing Information

---

The efficacy and usefulness of the manner in which missing information is handled by SQL is in the eye of the beholder. Perhaps no other topic in relational database management generates as much controversy as does the dispute over the proper way to deal with missing information in relational databases. Independent of the semantic difficulties SQL presents when dealing with nulls, missing information also makes exploratory data analysis far more difficult than the identical data mining operation performed on data free of missing values.

The problems resulting from recording and manipulating missing information are concerned principally with the sometimes ambiguous, and often counterintuitive, meaning of nulls as reported by various types of database queries.

This chapter examines the various semantic ambiguities with nulls in order to make designers aware of the potential problems of interpretation they present. The principal thrust is to make you aware of the inconsistencies in the way SQL handles missing information. Careful use of nulls can provide benefits that cannot otherwise be achieved: at the same time, their careless use can present serious problems not only for the integrity of your databases, but also for the accuracy of the information you retrieve from them.

The recommendation for nulls is not to use them if you can avoid doing so. Constrain as many of your columns as possible to be NOT NULL. A carefully considered database design is often all that is required to avoid recording most nulls.

## Semantics of SQL Nulls

For most applications, an SQL null means that the value of interest is not known. The only exceptions to this occur when nulls are used to represent the empty set. SQL does not support the empty set.

### Types of Missing Values

The following list touches on most of the common uses of SQL nulls:

- Value is unknown
- Value is not applicable
- Value does not exist
- Value is not defined
- Value is not valid
- Value is not supplied
- Value is the empty set

The semantics, properties, and behavior of each of these null types are different, but SQL treats them identically, including all 14 of the ANSI/X3/SPARC “null manifestations” and their 8 submanifestations. Codd (1990) has proposed a revision of SQL logic that includes a second type of null: value not applicable, which would mean extending the current 3-valued first-order predicate logic of SQL (3VL, having the possible predicate evaluations TRUE, FALSE, and UNKNOWN, the latter called MAYBE by Codd) to a 4-valued first-order predicate logic (4VL, having the possible predicate evaluations TRUE, FALSE, UNKNOWN, and NOT APPLICABLE, the latter two called MAYBE BUT APPLICABLE (designated with an A-mark) and MAYBE BUT NOT APPLICABLE (designated with an I-mark), respectively).

The problems with the 3VL now supported by SQL are well documented (see Date, 1990ab, 1992abc; McGoveran, 1993-94; Pascal, 2000), and the formal arguments used to justify the deprecation of 3VLs are not repeated in any detail here. Arguments supporting 3VL have also been made (see E.F. Codd in Codd and Date, 1993; Fesperman, 1998; Fesperman, 1998-2003; Johnston, 1995ab), and detailed rejoinders from 2VL (no nulls allowed) supporters have been published in response (see C.J. Date in Codd and Date, 1993; Date, Darwen, and McGoveran, 1995, 1997). It is perhaps worth noting that trivalent logics are categorized as *deviant* logics, meaning that they capture alternative forms of reasoning (see, for example, Haack, 1975; 1996).

## Inconsistencies in How SQL Treats Nulls

Recall that the defined semantics for SQL nulls is as the simple representation for missing information (see “[Semantics of SQL Nulls](#)” on page 673). A null is to be interpreted as missing information and nothing more. There are a number of areas in which the SQL treatment of nulls is inconsistent with this definition, and the purpose of this section is to summarize those inconsistencies.

The following summary of the inconsistencies in the treatment of nulls by SQL is intended to assist you with the interpretation of results that might otherwise be misleading:

- The empty set (notated symbolically as  $\emptyset$ ), which is a well-defined *value* in set theory, evaluates as null in SQL. The empty set is an actual value in mathematics, not a place holder for missing information. The union of the empty set with any set  $S$  yields  $S$ . In other words, the empty set is the identity under union. In SQL, however, nulls are used to indicate both missing information (and many other things) as well as the empty set. To demonstrate that nulls and the empty set are *not* the same thing in SQL, try to submit the following request and see what happens.

```
SELECT column_name
  FROM table_name
    UNION
      NULL;
```

Furthermore, in set theory,  $\emptyset = \emptyset$ , while in SQL,  $\text{NULL} \neq \text{NULL}$ .

For example, the evaluation of an empty character string in SQL returns null rather than the mathematically correct empty set. Similarly, aggregation over empty sets report null, which is meant to mean that the value is missing, but there are known results having real values for such operations on empty sets.

The problem with empty sets is also true for the outer join, where the extended columns in the join result are denoted by nulls, but are actually empty sets. As a result, nulls are used to represent both empty sets and missing information in the same report.

- Nulls have no value by definition, but sort as if they were all equal to one another.
- Similarly, when Teradata Database makes duplicate row checks, it treats nulls as if they were equal to one another.
- A null unique primary index is valid in Teradata Database. In this situation, the non-value NULL is treated as if it were a unique value, which it is not, because by definition, NULL is neither a value nor unique.

There can be only one null UPI per table, and if a table defined with a NUPI has very many null primary indexes, the distribution of those rows across the AMPs will be very skewed.

- With the exception of COUNT(\*), SQL aggregate operations ignore nulls, while SQL arithmetic operations do not.
- With the exception of CASE, COALESCE, and NULLIF, nulls are not valid predicate conditions in SQL expressions.
- With the exception of CASE, COALESCE, and NULLIF, SQL expressions cannot neither return nor operate on nulls.
- When a CASE, COALESCE, or NULLIF expression returns a null literal, it has INTEGER data type. All other nulls are untyped.
- Null is a relatively common European family name, so its use as a literal default for name columns can produce incorrect query results if there are instances of the family name Null in those columns.
- Structured UDTs can have null attributes, but the semantics of null attributes are different from the semantics of a null field in a column. For example, consider a structured UDT that is composed of a single attribute. If that attribute is set null, the field value in that column is *not* treated as a null with respect to the UDT column value.

You must explicitly place a null marker into the UDT column for the column “value” to be considered null.

The semantics of a null data type, whether partially or wholly null, are difficult to grasp. It can be said that a null attribute represents a missing type definition, but nulls are defined in ANSI/ISO SQL to represent missing *values*, not missing type definitions. The semantics of a UDT are what its designer defines them to be, of course, but from a logical perspective, it would seem that the best semantic definition of a null UDT would be *undefined*.

Using technology borrowed from object-oriented programming languages, it might be said that nulls superficially appear to be *overloaded* in SQL because multiple markers having different semantics are all subsumed under the same name: null. However, unlike the case for overloaded functions in object-oriented languages, it is *not* possible to discriminate among the various semantic possibilities a given null marker presents to a user or routine that must distinguish its intended semantics from among the myriad possible interpretations.

# Bivalent and Higher-Valued Logics

This topic presents a brief overview of the relevant predicate logic underlying the database relational model and the place of nulls in that logic.

A two-valued logic (2VL) has 2 truth values: TRUE and FALSE, while a multivalued logic (MVL) has the truth values TRUE and FALSE plus an additional number, where the number of truth values depends on the individual logic.

## First Order Predicate Logic and Bivalent Logic

In his original statement of the database relational model, Codd (1970) explicitly stated that it was based on the foundation of first order predicate logic. The classic first order predicate logic is based on bivalent logic (a bivalent, or Boolean, logic has exactly two truth values: TRUE and FALSE, as noted in the previous topic). Higher-valued logics have three or more truth values. For example, SQL uses a 3VL having the following three truth values: TRUE, FALSE, and UNKNOWN), and though the theory of higher-valued logics has been studied by mathematicians and philosophers, no true multivalued logics are used for real world applications.

Upon close examination, it is apparent that the logic supporting SQL is actually a bivalent logic (2VL) with additional ad hoc support for nulls. Note that the inconsistent treatment of missing values in SQL is common to all vendors: it is not an implementation problem per se, but rather a problem with the way SQL itself is defined.

Indeed, Codd did not add nulls to the relational model until 1979 (see section 2.3 of Codd, 1979). Codd did not originate the use of nulls to represent missing information in relational systems. The Project MAC Advanced Interactive Management System at the Massachusetts Institute of Technology pioneered their use as early as 1970 (see Goldstein and Strnad, 1970, 1971; Strnad, 1970, 1971 for details).

Perhaps significantly, the pre-System R relational system pioneered in the early 1970s at the IBM Peterlee Scientific Centre in the UK, which was based on solid principles drawn from the relational algebra, did not (see Notley, 1972; Hall, Hitchcock, and Todd, 1975; Todd, 1976 for details).

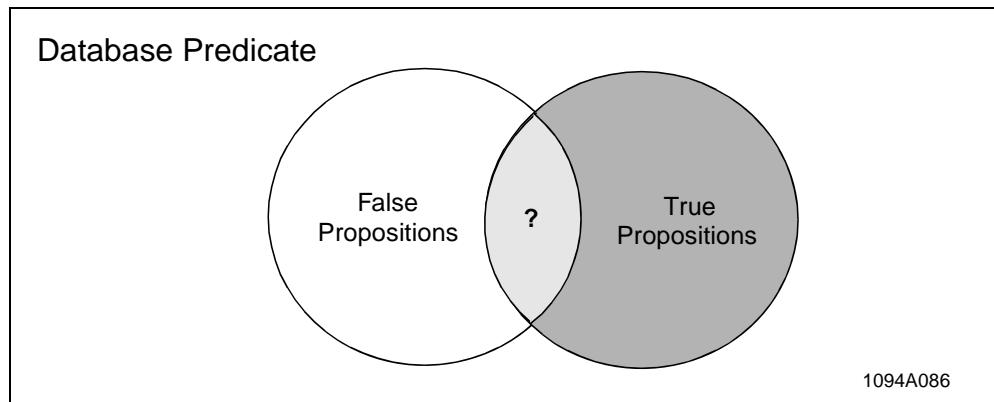
Pearson (2006), in reviewing the problems that missing data present to data mining, notes that the use of nulls in SQL databases “introduces significant practical complications” because of the faults of applying 3VL to a language that is based on 2VL. For example, the truth value NOT TRUE is not equivalent to the truth value FALSE in 3VLs, but the SQL language can deal only with truth values of TRUE and FALSE. SQL dialects typically evaluate a truth value of *unknown* as if it were FALSE.

Schafer and Graham (2002) review the statistical literature on dealing with missing data and evaluate the various methodologies that have been developed to handle the issue. Unfortunately, their recommendation of the methods of maximum likelihood estimation and Bayesian multiple imputation for estimating missing data values are of no use to data warehousing because they do not deal with the specifics of detail data. For example, how

might one do a maximum likelihood estimate of a missing employee home address, or of an undisclosed employee age (see “[The Closed World Assumption Revisited](#)” on page 677)?

## The Closed World Assumption Revisited

Recall that any tuple in a relation body is assumed to conform to the Closed World Assumption of Raymond Reiter (see “[The Closed World Assumption](#)” on page 630). With the addition of nulls to the database relational model, the CWA can no longer be assumed to be true. As an example, consider the following Venn diagram.



In contrast with the Venn diagram in the topic “[The Closed World Assumption](#)” on page 630, there is overlap between the set of false propositions and the set of true propositions, represented by the darker shading. Better put, there is *potential* overlap between those sets. For example, suppose the *employee* table has an *emp\_age* column that permits nulls. The company hires an employee who does not disclose her age in the application form. All other information for the employee is entered.

Because the age for this employee is missing, the tuple that represents her can neither be evaluated as true nor as false, so it falls into the ambiguous intersection of the Venn diagram.

At some point, it is discovered that this employee is 15 years old, which violates a constraint on the *emp\_age* column that requires all employees to be at least 18 years old. The proposition for this employee now evaluates as FALSE, and the tuple that represents her is no longer valid in the database. Nevertheless, that tuple was treated as valid from the time its data was entered until the time the age of the employee was discovered to violate the constraint specified for the *emp\_age* column, and that information was used to produce any number of reports which are, in retrospect, known to be non-valid.

Also see Date (2007) for a review of the Closed World Assumption and its role in designing databases.

## Number of Logical Operators Supported for Bivalent and Trivalent Logics

In a bivalent logic, there are exactly four possible monadic, or single-operand, logical operators, as indicated in the following table. Note that the numberings have no meaning other than to distinguish the operators from one another.

This operator number ...	Performs these mappings of TRUE and FALSE values ...
1	$T \rightarrow T$ $F \rightarrow T$
2	$T \rightarrow F$ $F \rightarrow F$
3	$T \rightarrow F$ $F \rightarrow T$
4	$T \rightarrow T$ $F \rightarrow F$

By generalization, it is also true that there are exactly 16 possible dyadic, or two-operand, logical operators.

The formulas that produce the numbers of monadic and dyadic operators are provided as follows, where  $n$  represents the valence of the logic in question. For a bivalent logic, its value is 2; for a trivalent logic, its value is 3.

$$\text{Number of monadic operators} = n^n$$

$$\text{Number of dyadic operators} = n^{n^2}$$

By performing the calculations for a trivalent logic, you find that 3VLs have 27 possible monadic operators and 19,683 possible dyadic operators. These figures, of course, are redundant: they represent the number of possible operators, not the number of useful operators.

SQL, like most programming languages, overdetermines its operators. For example, the function performed by the = operator in the following two equations is not identical internally, but the language uses the same symbol for the operation in both cases.

- ‘value’ = ‘value’
- 2 = 2

The issue raised here is whether SQL supports a sufficient number of trivalent logical operators to fully support a 3VL, and the answer to that question is negative.

Nevertheless, they indicate that SQL does not support anywhere near the number of logical operators required to support a coherent 3VL. Imagine the difficulty of trying to support the 4VL that Codd (1990) advocates, for which there is a possible 256 monadic operators and over four billion dyadic operators (4,294,967,296 to be exact). See Bolc and Borowik (1992, 2000) for examples of how complicated many-valued logics are.

## SQL Support for a Consistent Trivalent Logic

The treatment of nulls by SQL is not only inconsistent, but it is also sometimes incorrect from a real world perspective (see Date, 2005 or Date, 2007, for thorough reviews).

Rubinson (2007) argues that Date misinterprets the semantics of the example he uses to establish that SQL can return erroneous results from databases that contain nulls. After making the case that the example query Date uses incorrectly assumes 2VL, but that SQL actually uses 3VL and does return the correct, albeit confusing, result, Rubinson goes on to argue that the important point the error of interpretation Date makes demonstrates is that “SQL’s use of nulls and three-valued logic introduces a startling amount of complexity into seemingly straightforward queries” and “since the presence of a single null value (sic) taints the entire database, one must generally assume three-valued logic. Consequently, the burden is on us to carefully review our queries to ensure that they mean what we intend.” His overriding argument is that “[p]roper design techniques, then, naturally minimize the number of nulls in the database.” Both Date (2007b) and Grant (2007) respond that it is Rubinson, not Date, who misinterprets SQL semantics, while otherwise agreeing that Rubinson is correct in his argument that the use of 3VL in SQL is an error.

The purpose of this description of the problems inherent in SQL missing values is to help you to understand what the most important inconsistencies are and how you can avoid some of them through careful database design.

## Alternatives To Nulls for Representing Missing Information

Unlike every other feature of the database relational model, the treatment of missing information is not based on a formal foundation. As a result, even though the usual explanations of nulls often appear to make common sense, any detailed investigation reveals a number of potentially serious problems with SQL nulls.

The following sections describe some proposed alternative solutions to the missing value problem:

- [“Systematic Use of Default Values” on page 679](#)
- [“Redesigning the Database to Eliminate the Need for Nulls” on page 680](#)

### Systematic Use of Default Values

Because SQL has a built-in, nonprocedural, method to handle default values (see the default value control phrases “DEFAULT” and “WITH DEFAULT” in *SQL Data Types and Literals*), no special coding is required on the part of users. Note that if no defaults are specified for a column and it is not defined as NOT NULL, then SQL automatically inserts nulls to represent any information missing from an INSERT request.

## Redesigning the Database to Eliminate the Need for Nulls

In the first order predicate calculus, propositions can have one of two values: they can either be true or they can be false. Recall that the rows of a relational table correspond to logical propositions that evaluate to TRUE (see “[How Relational Databases Are Built From Logical Propositions](#)” on page 631), because all false propositions are excluded from the database by means of various integrity constraints.

One can build a set of axioms deriving from such a set of true propositions in addition to a set of well-defined inference rules (referred to as a calculus in formal logic). Additional true propositions, formally known as theorems, can be derived from this calculus. These derived propositions correspond to valid relational queries if and only if the following statements about the original propositions are true:

- The initial set of rows represents only true propositions.
- All operations on a relation that contains tuples corresponding to these true propositions obey the formal inference rules of bivalent logic.

If either of these statements is false, then the correctness (truth of the derived propositions) of any query made against the data cannot be guaranteed. Because SQL nulls represent data values that cannot be asserted, it is not possible to know whether rows that contain nulls represent true propositions or not, hence, the tables that contain them are not true relations in the mathematical sense.

Noting that one of the most frequently provided justifications null supporters give for their use is that maintaining rows with missing values is often a necessary and useful thing to do in the real world, Pascal asks, why it would be considered useful to record what he calls “partial nothings” in the database if it is obviously nonsensical to maintain “full nothings”? Pascal then uses the following extreme example to support his case against nulls.

### Taking the “Partial Nothing” Argument To Its Logical Extreme

Consider the following classic case study: an employee table that records the salary of each employee currently working for the enterprise. It is not uncommon for salaries to be unknown at a given point, so an employee row might contain a null to hold the place of the true salary of that employee. At the time the rows is inserted into the table, the salary value is missing information.

At a given instant, this employee table might look like this:

employee				
emp_num	emp_name	dept_num	hire_date	salary
PK				

**employee**

214	Smith	32	09-12-1989	56150
447	Lau	15	05-30-1993	?
103	Hossein	09	09-13-1984	29775
500	Nakamura	11	06-09-1997	84932
713	Schroeder	24	10-29-2001	?

New hire Schroeder has not yet been assigned a salary and Lau has just been promoted, but her new salary has not yet been determined.

Pascal argues that in reality, this “partial nothing” information is no more useful than the following “partial nothing” table, for which only the primary key values known for each row.

**employee**

emp_num	emp_name	dept_num	hire_date	salary
PK				
214	?	?	?	?
447	?	?	?	?
103	?	?	?	?
500	?	?	?	?
713	?	?	?	?

Pascal argues that the problem with the “partial nothing” table is that its semantics are perceived as something other than what they truly are. The intended meaning of the nulls in the *salary* column derives from the true proposition that all employees receive a salary. The formal semantics of the table, however, assert only that a salary amount exists for each employee.

As a result, the table actually represents a mix of two different categories of assertion:

- Those rows with nulls that apply to all employees (“a salary amount exists for all employees”).
- Those rows that describe only those employees whose salaries are known (“all employees receive a salary”).

Another way of saying this is that multiple, inconsistent, informal predicates are being mapped (incorrectly!) into a single formal predicate. In this particular case, some rows (those containing nulls as place holders for the salary value) apply to all employees, while others (those that contain values for employee salaries) apply to a subset of that population. No single predicate can possibly apply to both situations.

## Solving the Problem By Taking Simple Projections

The particular problem with nulls in this table is readily resolved with a minor change in the logical design of the database. Because the attributes in this *employee* table are a combination of those that apply to all tuples and those that apply only to some tuples, the relationship is similar to that of a simple entity supertype and subtype, the difference being that the attribute in question, *salary*, is not unique: all employees receive a salary.

The following tables resolve the problem of multiple simultaneous semantics for the original *employee* table by taking projections from the original *employee* table:

employee				employee_salary	
emp_num	emp_name	dept_num	hire_date	emp_num	salary
PK				PK	
214	Smith	32	09-12-1989	214	56150
447	Lau	15	05-30-1993	103	29775
103	Hossein	09	09-13-1984	500	84932
500	Nakamura	11	06-09-1997		
713	Schroeder	24	10-29-2001		

## Manipulating Nulls With SQL

As described in “[Types of Missing Values](#)” on page 673, SQL nulls are often mistakenly interpreted in ways that extend significantly beyond their single valid meaning, which is simply unknown (no value is present).

These properties make the use and interpretation of nulls in SQL problematic. The following sections outline the behavior of nulls for various SQL operations to help you to understand how to use them in data manipulation statements and to interpret the results those statements effect.

See “[Manipulating Nulls With SQL](#)” on page 682, and “[NULL Literals](#)” on page 686 for details about manipulating nulls with SQL.

## Logical and Arithmetic Operations on Nulls

Nulls are not valid as predicate conditions in SQL other than for the unique example of CASE expressions (see “[Nulls and CASE Expressions](#)” on page 685).

You cannot solve for the value of a null because, by definition, it *has* no value. For example, the expression *value* = NULL has no meaning and therefore can never be true (however, see “[Null](#)”

[Sorts as the Lowest Value in a Collation](#) on page 687 for a counterexample). A query that specifies the predicate WHERE value = NULL is not valid because it can never be true or false. The meaning of the comparison it specifies is not only unknown, but unknowable.

If you want to search for fields that do or do not contain nulls, you must use the operators IS NULL or IS NOT NULL (see “[Searching for Nulls Using a SELECT Request](#)” on page 687, “[Excluding Nulls From Query Results](#)” on page 688 and “[Searching for Nulls and Nonnulls In the Same Search Condition](#)” on page 688).

The difference is that when you use a mathematical operator like  $=$ , you specify a comparison between values or value expressions, whereas when you use the IS NULL or IS NOT NULL operators, you specify an existence condition. Note that even though IS NULL and IS NOT NULL return truth values, their operands are *not* truth values; therefore they are not logical operators and do not count against the number of possible logical operators for a trivalent logic calculated in “[Number of Logical Operators Supported for Bivalent and Trivalent Logics](#)” on page 677.

## Nulls and Arithmetic Operators and Functions

If an operand of any *arithmetic* operator or function is null, then the result of the operation or function is usually null. The following table provides some illustrations and some exceptions:

WHEN the expression is ...	THEN the result is ...
5 + NULL	null
LOG(NULL)	null
NULIFZERO(NULL)	null
ZEROIFNULL(NULL)	0
COALESCE(NULL, 6)	6

## Nulls and Comparison Operators

If either operand of a *comparison* operator is null, then the result is unknown and an error is returned to the requestor. The following examples indicate this behavior.

WHEN the expression is ...	THEN the result is ...	AND this error message returns to the requestor ...
5 = NULL	Unknown	3731
5 <> NULL		
NULL = NULL		
NULL <> NULL		
5 = NULL + 5		

Note that if the argument of the NOT operator is unknown, the result is also unknown. This evaluation translates to FALSE as a final boolean result.

## Nulls and Aggregate Functions

With the important exception of COUNT(\*), aggregate functions ignore nulls in their arguments. This treatment of nulls is very different from the way arithmetic operators and functions treat them, and is one of the major inconsistencies in the way SQL deals with nulls.

This behavior can result in apparent nontransitive anomalies. For example, if there are nulls in either column A or column B (or both), then the following expression is virtually always true.

$$\text{SUM}(A) + (\text{SUM } B) <> \text{SUM } (A+B)$$

In other words, for the case of SUM, the result is never a simple iterated addition if there are nulls in the data being summed.

The only exception to this is the case in which the values for columns A and B are *both* null in the same rows, because in those cases the entire row is not counted in the aggregation. This is a trivial case that does not violate the general rule.

The same is true, the necessary changes being made, for all the aggregate functions except COUNT(\*), which does include nulls in its result.

If this property of nulls presents a problem, you can perform either of the following workarounds, each of which produces the desired result of the aggregate computation

$$\text{SUM}(A)+\text{SUM}(B) = \text{SUM}(A+B).$$

- Define all NUMERIC columns as NOT NULL DEFAULT 0.
- Use the ZEROIFNULL function nested within the aggregate function to convert any nulls to zeros for the computation, for example

$$\text{SUM}(\text{ZEROIFNULL}(x) + \text{ZEROIFNULL}(y))$$

## Nulls and DateTime and Interval Data

The general rule for managing nulls with DateTime and Interval data environments is that, for individual definitions, the rules are identical to those for handling numeric and character string values.

WHEN any component of this type of expression is null ...	THEN the result is ...
Value	null.
Conditional	FALSE.
CASE	as it would be for any other CASE expression.

DateTime or Interval values are defined to be either atomically null or atomically non-null. For example, you cannot have an interval YEAR TO MONTH value in which YEAR is null and MONTH is not.

## Nulls and CASE Expressions

The ANSI/ISO SQL-2008 definitions for the CASE expression and its related expressions NULLIF and COALESCE specify that these expressions can return a null. Because of this, their behavior is an exception to the rules for all other predicates and expressions.

The rules for null usage in CASE, NULLIF, and COALESCE expressions are as follows.

- If no ELSE clause is specified in a CASE expression and no WHEN clause evaluates to TRUE, then NULL is returned by default.
- Nulls and expressions containing nulls are valid as CASE conditions. The following examples are valid.

```

SELECT CASE NULL
        WHEN 10
        THEN 'TEN'
    END;

SELECT CASE NULL + 1
        WHEN 10
        THEN 'TEN'
    END;

SELECT CASE column_1
        WHEN NULL
        THEN 'NULL'
    END
FROM table_1;

SELECT CASE column_1
        WHEN NULL + 1
        THEN 'NULL'
    END
FROM table_1;

SELECT CASE
        WHEN column_1 = NULL
        THEN 'NULL'
    END
FROM table_1;

SELECT CASE
        WHEN column_1 = NULL + 1
        THEN 'NULL'
    END
FROM table_1;

```

The following example is valid.

```

SELECT CASE
        WHEN column_1 IS NULL
        THEN 'NULL'
    END
FROM table_1;

```

- In contrast to the situation for WHEN clauses in a CASE expression, nulls and expressions containing nulls are valid as THEN clause conditions. The following example is valid.

```
SELECT CASE
    WHEN column_1 = 10
        THEN 'NULL'
    END
FROM table_1
```

## NULLIF and COALESCE

The behavior of the CASE shorthand expressions NULLIF and COALESCE is the same as that for CASE with respect to nulls.

See “NULLIF” and “COALESCE” in *SQL Functions, Operators, Expressions, and Predicates* for further information.

# NULL Literals

The keyword NULL is sometimes available as a special construct similar to, but not identical with, a literal. In this case, the keyword NULL represents the SQL null placeholder for a value logically in an SQL request, but is *not* the same marker that the system stores to indicate missing information.

## Rules for Using NULL as a Literal

The literal NULL can be used in the following ways.

- As a CAST source operand, for example.

```
CAST (NULL AS value)
```

- As a CASE result, for example.

```
CASE expression
    THEN NULL
END
```

or

```
CASE expression
    THEN expression
    ELSE NULL
END
```

- As an item specifying a null is to be placed in a column on INSERT or UPDATE.
- As a default column definition specification, for example.

```
DEFAULT NULL
```

- As an explicit SELECT item, for example.

```
SELECT NULL
```

This usage is a Teradata extension to the ANSI/ISO SQL-2008 standard because it does not specify a FROM clause.

- As an operand of a function, for example.

```
SELECT TYPE(NULL)
```

This usage is a Teradata extension to the ANSI/ISO SQL-2008 standard because it does not specify a FROM clause.

## Data Type of NULL Literals

When you use NULL as an explicit SELECT item or as the operand of a function, its data type is INTEGER. For example, if you perform SELECT TYPE(NULL), then the data type of NULL is returned as INTEGER.

In all other cases NULL has no data type because it has no value.

## Hashing on Nulls

Even though nulls have no external value, they do have an internal value that can be processed by the Teradata Database hashing algorithm. If a NUSI for a table permits nulls, there is an increased probability of an uneven distribution of the rows for that table across the AMPs of a system because all rows having a null primary index hash to the same AMP (see “[Number of Null Rows](#)” on page 425).

If the number of rows having a null primary index is sufficiently large, then significant skew occurs, making efficient parallel processing difficult to achieve. This is not a problem for UPIs because there can be no more than one null UPI per table.

Although indexes are not a logical concept, and therefore are not part of the relational model, Teradata primary indexes are built from in-row values, so a null UPI provides a mechanism for what is effectively a potential duplicate row to be stored in the system as long as the column set remains null.

## Null Sorts as the Lowest Value in a Collation

When rows are sorted using an ORDER BY clause, nulls sort as the lowest value.

If any row has a null in the column being grouped, then all rows having a null are placed into one group. In other words, though it is syntactically incorrect to formulate the predicate NULL = NULL in a DML request because nulls have no value and therefore cannot be equated, it is also true that when SQL sorts nulls, the implicit result is that NULL = NULL.

## Searching for Nulls Using a SELECT Request

The IS NULL operator tests for the presence of nulls in a specified column. For example, to search for the names of all employees who have a null in the *dept\_no* column, you could type the following request.

```
SELECT name
  FROM employee
 WHERE dept_no IS NULL;
```

This query produces the names of all employees having a null in the *dept\_no* column. Because all employees contained in the *employee* table have been assigned to a department, no rows would be returned for this particular example.

## Searching for Nulls and Nonnulls In the Same Search Condition

To search for nulls and non-nulls within the same predicate, the search condition for nulls must be specified separately from any other search conditions.

For example, to select the names of all employees with the job title of ‘Vice Pres,’ ‘Manager,’ or null, you could type the following SELECT statement:

```
SELECT name, job_title
  FROM employee
 WHERE job_title IN ('Manager' OR 'Vice Pres')
   OR job_title IS NULL;
```

## Excluding Nulls From Query Results

The IS NOT NULL operator excludes rows having a null in a specified column from the results of a query.

For example, to search for the names of all employees with non-nulls in the *job\_title* column, you could type the following request.

```
SELECT name
  FROM employee
 WHERE job_title IS NOT NULL;
```

The result of this query is the names of all employees with a non-null in the *job\_title* column. For this particular example, the names of all employees are returned because every employee has, by definition, been assigned a job title.

## Nulls and the Outer Join

The outer join is a join that retains rows from one or both of the joined tables, depending on whether it is declared to be a left, right, or full outer join, respectively. This means that rows that do not match the rows produced by the inner join are extended in the outer join result by reporting nulls in the nonmatching columns. From the perspective of set theory, these nonmatching column values are not missing values, but the empty set, which is a real value. Because SQL represents the empty set as a null, the semantics of outer joins must be evaluated carefully whenever there are actual missing values in the inner join portion of the operation.

Ambiguities presented by nulls in the outer join also make query optimization more difficult, because certain key methods of query optimization such as transitive closure often cannot be achieved.

## Semantics of Nulls in the Outer Join

The motivation behind the outer join is to preserve information that is otherwise lost in an inner join. There can be no doubt that this is an important issue, but the SQL solution to the problem presents some inconsistencies regarding the semantics of the nulls reported by an outer join. The following contrast of simple inner and outer natural joins on the same two tables is based on an example developed by Date (1992c). The respective joins are between the following *supplier* and *supplier\_parts* tables:

supplier				supplier_parts		
suppl_num	suppl_name	status	city	suppl_num	part_num	quantity
PK				PK		
S2	Jones	10	Paris	S2	P1	300
S5	Adams	30	Athens	S2	P2	400

Here is the result of the inner join of the *supplier* and *supplier\_parts* tables.

suppl_num	suppl_name	status	city	part_num	quantity
S2	Jones	10	Paris	P1	300
S2	Jones	10	Paris	P2	400

This join outcome has no information about supplier S5, therefore, it is said to have lost that information.

Here is the result of the natural outer join of the *supplier* and *supplier\_parts* tables:

suppl_num	suppl_name	status	city	part_num	quantity
S2	Jones	10	Paris	P1	300
S2	Jones	10	Paris	P2	400
S5	Adams	30	Athens	?	?

This join outcome retains information about supplier S5, therefore it is said to preserve that information. Note that the result reports nulls for the *part\_num* and *quantity* columns for supplier S5.

Think back to the definition of null semantics that was presented in “[Semantics of SQL Nulls](#)” on page 673. Recall that the unequivocal definition for an SQL null is that it represents an

unknown value. In this case, however, the values for both *part\_num* and *quantity* for supplier S5 *are* known, and those values are both the empty set. Because SQL does not support empty sets directly, it substitutes nulls in their place. This is also true for the result of the length determination of an empty character string (see “[Inconsistencies in How SQL Treats Nulls](#)” on page 674).

The important thing to remember with respect to the semantics of outer join nulls is that nulls represent both the empty set (when used to extend the inner join to preserve information that would otherwise be lost) and semantically correct SQL nulls (when used to represent unknown information in the inner join portion of the result). The determination of which null is which is left to the user.

Note that in set theory, the empty set is sometimes referred to as the *null set*. It is extremely important to understand that the word null in this context means something entirely different from its meaning in the SQL language. The null set is simply a set that contains no members. It is, therefore, empty, which is why this set is also referred to as the empty set.

Also note that there is only one empty set in set theory, analogous to how there is only one value of 3 in the set of real numbers, only one value of 3 in the set of cardinal numbers, and so on. Therefore, it is referred to as *the* empty set, not *an* empty set. Note, too, that studies of the properties of the empty set are sometimes referred to as *nullology*. Again, this term has absolutely nothing to do with nulls as they are defined by the SQL language.

# CHAPTER 14 Database-Level Capacity Planning Considerations

---

This chapter describes some of the database-level considerations for capacity planning.

[Chapter 15: “System-Level Capacity Planning Considerations,”](#) describes file system- and hardware-oriented considerations for capacity planning.

## Capacity Planning

When there is no legacy database to build on, capacity planning can be a difficult enterprise to undertake. Fortunately, this is rarely an issue for contemporary users because at least part of their corporate databases are almost always maintained in electronic form. Keep in mind that much of the information presented in this chapter assumes you have a legacy system to draw upon for making your sizing estimations.

Capacity planning should begin with the idea of making the most frequently accessed data available at all times. With the relatively low priced, large capacity disk storage units commonly used for data warehousing applications, the nature of the emphasis on this factor has changed from offloading as much historical data as possible to archival storage toward developing the capability of keeping all data forever online and accessible to the warehouse.

In a data warehouse that maintains massive quantities of history data, the volume of data is typically inversely proportional to its use. In other words, there is an enormous amount of cool history data that is accessed lightly, and a relatively lesser volume of hot and warm data that is accessed frequently.

The following, somewhat loose, default definitions apply to the commonly described temperature bands.

Temperature	Default Definition
COLD	The 20% of data that is least frequently accessed.
WARM	The remaining 60% of data that falls between the COLD and HOT bands.
HOT	The 20% of data that is most frequently accessed.
VERY HOT	Data that you or Teradata Virtual Storage think should be added to the Very Hot cache list and have its temperature set to very hot when it is loaded using the TVSTemperature query band.

**Note:** Teradata Virtual Storage tracks data temperatures at the level of cylinders, not tables. Because the file system obtains its temperature information from Teradata Virtual Storage, it also handles temperature-related compression at cylinder level. See *Teradata Virtual Storage* for more information about this.

The file system can change the compressed state of the data in an AUTOTEMP table at any time based on its temperature. Cylinders in an AUTOTEMP table become eligible for temperature-based block-level compression only when they reach or fall below the threshold defined for COLD temperature-based block level compression. See “TempBLCThresh” in *Utilities: Volume 1 (A-K)* for more information about the temperature settings that you can use for temperature-based block-level compression.

Temperature-based thresholds for the block-level compression of AUTOTEMP tables work as defined by the following table.

IF data blocks are initially ...	AND then become ...	THEN the file system ...
block-level compressed	warmer than the defined threshold for compression	decompresses them.
not block-level compressed	colder than the defined threshold for decompression	compresses them.

For tables that are not defined with BLOCKCOMPRESSION=AUTOTEMP, you must control their block-level compression states yourself using Ferret commands or, if a table is not populated with rows, you can use one of the TVSTemperature query bands to specify the type of block-level compression to use for the newly loaded rows. If temperature-based block-level compression is disabled but block-level compression is enabled, Teradata Database treats AUTOTEMP tables the same as MANUAL tables.

For all of the data in a table to be block compressed or decompressed at once in an AUTOTEMP table, Teradata Virtual Storage must become aware that *all* cylinders in the table have reached the threshold specified by the DBS Control parameter TempBLCThresh. This would occur in the following case. Suppose the threshold value for TempBLCThresh is set to WARM.

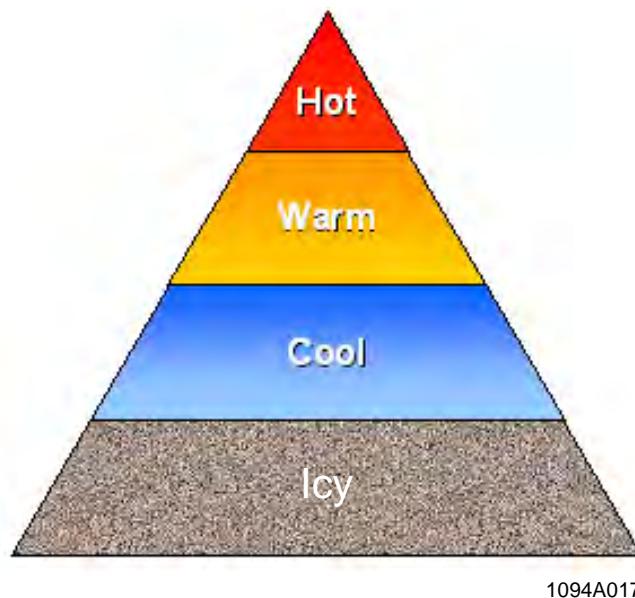
IF <i>all</i> of the cylinders in the table ...	THEN they <i>all</i> become eligible for ...
reach or fall below the WARM or COLD thresholds	block-level compression.
reach or exceed the HOT or VERY HOT thresholds	decompression.

Because of this, the best practice is not to use the AUTOTEMP option, or not to use any form of temperature-based block-level compression for a table that you think requires compression consistency for the entire table.

## Extended Data Lifetimes

The lifetime of data is now being extended for a number of different reasons. Users often have varied performance requirements for time-based or historical data: recent data might be accessed frequently, while older data is accessed less often. Call these conceptual access rates hot, warm, cool, and icy, respectively, ranking from most frequently accessed to least frequently accessed. Keep in mind that these data access states are largely conceptual. Cool and icy have a loose correspondence with the temperature-based block-level compression state of COLD and the warm and hot states loosely correspond with the identically named temperature-based block-level compression states.

In a warehouse with massive historical databases, the volume of data is typically inversely proportional to the data usage, as illustrated by the following graphic, where the ordinate represents the relative warmth of the data and the abscissa represents the volume of data represented by the respective measures of data warmth.



1094A017

In this picture, the temperature of the data reflects its access rate. The optimal storage for hot, warm, and cool data is online disk that is directly accessible to the data warehouse.

See “CREATE TABLE” and “ALTER TABLE” in *SQL Data Definition Language* for information about how you can specify the way an individual table deals with block-level compression based on the temperature of the data.

## Cool and Icy Data

Typically, there is a vast quantity of cool historical data in the data warehouse that is accessed lightly and a lesser volume of hot and warm data that is accessed frequently.

**Note:** The distinction between cool and icy data is conceptual and has no direct parallel to temperature-based block-level compression.

While cool data might be accessed lightly on average, it still has temporal hot spots, such as those that occur when performing comparative analyses of sales data between different time

periods. Similarly, if the relational schema is modified by adding new columns or indexes, or if column data types are changed, then the affected data cannot be considered to be dormant. Finally, if data is periodically accessed and somehow recast to make historical data relevant within the current business context, then it is not dormant.

Truly dormant data is rarely, if ever, accessed, and is typically retained for various national and international regulatory reasons such as the Sarbanes-Oxley Act in the United States, Bill 198 in Canada, or the Eighth Company Law Directive 1984/253/EEC in the European Union rather than for operational reasons. As a result, the data stored in the warehouse is frequently a mix of both important and unimportant data (Unimportant from the perspective of the day-to-day operation of running the enterprise, not from a legal perspective), and a flexible management system is required to allocate the appropriate availability, reliability, and privacy levels for the data as its usage changes across time.

Icy, or truly dormant, data is a good candidate for alternative storage such as tape or optical disk. The treatment of icy data is not the subject of this chapter.

## **Warm, Hot, and Very Hot Data**

**Note:** The distinction between warm, hot, and very hot data is somewhat conceptual and is only loosely parallel to the WARM and HOT categories of temperature-based block-level compression. Very Hot data is data that you or Teradata Virtual Storage determine should be added to the Very Hot cache or that is currently contained on Very Hot cylinders.

Warm and hot data typically constitute what is often called the operational data store. Quoting Fan and Pederson (2003), an operational data store "...is a collection of data in support of an organization's need for up-to-date, operational, integrated, collective information. [An operational data store...] is a purely operational construct to address the operational needs of a corporation.

Hot and warm data are both important in the extended lifetime of data, particularly with respect to multi-value compression (see "[Compression Types Supported by Teradata Database](#)" on page 695), as is cool data to a lesser degree.

## **Reserved Query Bands for Managing the Block-Level Compression and Storage Temperature of Newly Loaded Data**

The BlockCompression and TVSTemperature reserved query bands enable you to control various aspects of loading new data into tables. For more information on the TVSTemperature query band, see *Teradata Virtual Storage*.

**Note:** These query band names are reserved for use by Teradata Database and by Teradata partners and should not be used by customers.

The BlockCompression reserved query band controls the compression of data loaded into a primary data table, join index, or hash index at the data block level using the following data loading methods:

- Archive and Restore

*Teradata Archive/Recovery Utility Reference*

- DUL csp command  
See *Utilities: Volume 1 (A-K)*
- FastLoad  
See *Teradata FastLoad Reference*
- MultiLoad  
See *Teradata MultiLoad Reference*
- JDBC Embedded FastExport  
See *Teradata JDBC Driver User Guide*
- JDBC Embedded FastLoad  
See *Teradata JDBC Driver User Guide*
- Teradata Parallel Transporter Load, Update, and Export operators  
See *Teradata Parallel Transporter Reference*
- CREATE HASH INDEX requests  
See *SQL Data Definition Language*
- CREATE JOIN INDEX requests  
See *SQL Data Definition Language*
- CREATE TABLE ... AS ... WITH DATA  
See *SQL Data Definition Language*
- INSERT requests into an empty table  
See *SQL Data Manipulation Language*
- INSERT ... SELECT requests into an empty table  
See *SQL Data Manipulation Language*

The BlockCompression reserved query band indicates whether data being loaded into an empty table that is using a form of manual compression should be compressed at the data block level or not. The term *manual compression* as used here refers to column-level compression methods such as multi-value compression and algorithmic compression. See “[Block-Level Compression](#)” on page 704 for more information about this. The TVSTemperature reserved query band controls the desired TVS-managed temperature level for the cylinders that store the newly loaded data for both manually-managed tables and temperature-based-managed tables, which can be specified to be Cold, Warm, or Hot. See “[Temperature-Based Block-Level Compression](#)” on page 705 for more information about this.

## Compression Types Supported by Teradata Database

Compression reduces the physical size of stored information. The goal of compression is to represent information accurately using the fewest number of bits. Compression methods are either logical or physical. Physical data compression re-encodes information independently of

its meaning, while logical data compression substitutes one set of data with another, more compact set.

Compression is used for the following reasons.

- To reduce storage costs
- To enhance system performance

Compression reduces storage costs by storing more logical data per unit of physical capacity. Compression produces smaller rows, resulting in more rows stored per data block and fewer data blocks.

Compression enhances system performance because there is less physical data to retrieve per row for queries. Also, because compressed data remains compressed while in memory, the FSG cache can hold more rows, reducing the size of disk I/O.

Most forms of compression are transparent to applications, ETL utilities, and queries. This can be less true of algorithmic compression, because a poorly performing decompression algorithm can have a negative effect on system performance, and in some cases a poorly written decompression algorithm can even corrupt data.

Experience with real world customer production databases with very large tables indicates that compression produces performance benefits for a table even when more than 100 of its columns have been compressed.

Teradata Database uses several types of compression.

FOR this database element...	Compression refers to...
column values	<p>the storage of those values one time only in the table header, not in the row itself, and pointing to them by means of an array of presence bits in the row header. It applies to:</p> <ul style="list-style-type: none"><li>• Multi-value compression See “<a href="#">Multi-Value Compression</a>” on page 699.</li><li>• Algorithmic compression See “<a href="#">Algorithmic Compression</a>” on page 700.</li></ul> <p>You cannot apply either multi-value compression or algorithmic compression to row-level security constraint columns.</p>
hash and join indexes	<p>a logical row compression in which multiple sets of nonrepeating column values are appended to a single set of repeating column values. This allows the system to store the repeating value set only once, while any nonrepeating column values are stored as logical segmental extensions of the base repeating set.</p> <p>See “<a href="#">Row Compression</a>” on page 709.</p>
data blocks	<p>the storage of primary table data, or join or hash index subtable data. Secondary Index (SI) subtable data cannot be compressed.</p> <p>See “<a href="#">Block-Level Compression</a>” on page 704.</p> <p>There are no restrictions on using block-level compression for a row-level security-protected table.</p>

FOR this database element...	Compression refers to...
partition containers	<p>the autocompression method set determined by Teradata Database to apply to a container of a column-partitioned table or join index when you have not specified the NO AUTO COMPRESS option at the time the object was created.</p> <p>See “<a href="#">Autocompression</a>” on page 300 for further information about autocompression for column-partitioned tables and join indexes.</p>

Row compression, multi-value compression, block-level compression, and autocompression are lossless methods, meaning that the original data can be reconstructed exactly from the compressed forms, while algorithmic compression can be either lossless or lossy, depending on the algorithm used.

There is a small initial cost, but even for queries made against small tables, compression is a net win if the chosen compression method reduces table size.

For compressed spool files, if a column is copied to spool with no expressions applied against it, then the system copies just the compressed bits into the spool file, saving both CPU and disk I/O size. Once in spool, compression works exactly as it does in a base table. There is a compress multi-value in the table header of the spool that stays in memory while the system is operating on the spool. When algorithmic compression is carried to spool files, the compressed data is carried along with the compress bits

The column attributes COMPRESS and NULL (see *SQL Data Types and Literals*) are useful for minimizing table storage space. You can use these attributes to selectively compress as many as 255 distinct, frequently repeated column values (*not* characters), to compress all nulls in a column, or to compress both.

The limit of 255 values is approximate because there is also a limit on the number of bytes or characters per column that can be multi-value compressed. These limits vary for different types of character data, as the following table explains.

FOR this type of data ...	THE maximum storage per column is approximately ...
• BYTE	4,093 bytes
• KanjiSJIS • GRAPHIC • Latin • Unicode	8,188 characters

## Identifying Compressed Tables for Further Opportunities to Save Space

This topic discusses no compression, single-value compression (SVC), and multi-value compression (MVC).

To determine whether your existing tables present further opportunities for compression, you must first identify which of your current tables are compressed in some way.

This SELECT request identifies all such tables in databases *aaaa*, *bbbb*, and *cccc* that are 15GB or greater in size, as specified by the phrase HAVING Current\_Perm > 15000000000. You should customize this specification to the requirements of your site.

```

SELECT dbt.DATABASENAME, dbt.TABLENAME,
       MAX(CASE WHEN (compressvaluelist IS NOT NULL)
                  THEN (CASE WHEN INDEX(compressvaluelist,',') > 0
                               THEN '3. MVC '
                               ELSE '2. SVC '
                           END)
                  ELSE '1. NONE'
               END) AS Compress_Type,
       MIN(pds.Current_Perm) AS Current_Perm
  FROM DBC.columns AS dbt, (SELECT t.DATABASENAME, t.TABLENAME,
                                         SUM(ts.CurrentPerm) AS Current_Perm
                                        FROM DBC.Tables AS t, DBC.TableSize AS ts
                                       WHERE t.DATABASENAME = ts.DATABASENAME
                                         AND t.TABLENAME = ts.TABLENAME
                                         AND ts.TABLENAME <> 'ALL'
                                         HAVING Current_Perm > 15000000000
                                         GROUP BY 1,2) AS pds
 WHERE dbt.DATABASENAME IN ('aaaa','bbbb','cccc')
   AND dbt.DATABASENAME = pds.DATABASENAME
   AND dbt.TABLENAME = pds.TABLENAME
 -- HAVING Compress_Type = '1. NONE'
 GROUP BY 1,2
 ORDER BY 1,3,4 DESC, 2;
    
```

The GROUP BY and ORDER BY conditions in this request order the data within each database by table size and compression type.

Note that the result set indicates that the only forms of compression found in databases *aaaa*, *bbbb*, and *cccc* in tables 15GB or greater in size are identified as NONE, SVC, and MVC, where NONE indicates no compression, SVC indicates single-value compression, and MVC indicates multi-value compression.

NONE identifies a condition where no values are compressed, so no space savings are gained from implementing multi-value compression; SVC identifies a condition that should change to MVC if possible to take advantage of the space savings offered by compressing more values in the table columns; MVC identifies a condition where at least some space savings are achieved through multi-value compression.

Having a condition of MVC does not mean that an optimal multi-value compression state has been achieved for a table, however. You should make a serious effort to analyze the data from tables having a Compress\_Type of MVC to determine whether still more space savings can be realized by multi-value compressing additional values in additional columns.

DatabaseName	TableName	Compress_Type	Current_Perm
-----	-----	-----	-----
aaaa	table_a	NONE	47,898,088,960
aaaa	table_e	SVC	16,797,542,912
aaaa	table_i	MVC	19,040,798,720
aaaa	table_j	MVC	18,537,593,856

aaaa	table_k	MVC	18,333,077,504
aaaa	table_l	MVC	15,982,028,800
bbbb	table_1	NONE	20,157,521,408
bbbb	table_2	SVC	113,774,842,368
bbbb	table_3	SVC	29,932,489,728
bbbb	table_4	SVC	19,060,238,848
bbbb	table_5	MVC	240,413,969,920
bbbb	table_6	MVC	191,052,663,296
bbbb	table_7	MVC	106,940,743,680
bbbb	table_8	MVC	102,247,339,520
bbbb	table_9	MVC	77,317,453,824
cccc	table_o	NONE	66,612,497,920
cccc	table_p	NONE	24,427,379,712
cccc	table_q	NONE	19,321,673,216
cccc	table_t	SVC	16,848,895,488
cccc	table_x	MVC	52,900,937,728
cccc	table_y	MVC	21,144,027,648

This topic is based on the article *Workload Toolkit—Part 4—Compression Analysis* on Teradata Developer Exchange (<http://developer.teradata.com/>) by David Roth of the Teradata Professional Services organization.

## Multi-Value Compression

Multi-value compression (MVC) is a form of logical data compression in which compressed values are stored in the table header not in the row itself (see “[Teradata Database Mechanisms for Multi-Value Compression](#)” on page 711). Each table has up to 1 MB of table header. MVC is a lossless method.

MVC is valid for permanent, global temporary, and volatile tables.

Besides storage capacity and disk I/O size improvements, MVC has the following performance impacts:

- Improves table scan response times for most configurations and workloads
- Provides moderate to little CPU savings

You may need to limit the number of values you compress depending on table specifications for factors such as journaling, fallback, logging, disk I/O integrity, and indexes.

You can use MVC to compress columns with these data types:

- Any numeric data type
- BYTE
- VARBYTE
- CHARACTER
- VARCHAR

- DATE

To compress a DATE value, you must specify the value as a Date literal using the ANSI DATE format (DATE 'YYYY-MM-DD'). For example:

```
COMPRESS (DATE '2000-06-15')
```

- TIME and TIME WITH TIME ZONE
- TIMESTAMP and TIMESTAMP WITH TIME ZONE

To compress a TIME or TIMESTAMP value, you must specify the value as a TIME or TIMESTAMP literal. For example:

```
COMPRESS (TIME '15:30:00')
COMPRESS (TIMESTAMP '2006-11-23 15:30:23')
```

In addition, you can use COMPRESS (NULL) for columns with these data types:

- ARRAY
- Period
- Non-LOB distinct or structured UDT

**Note:** Compressing many values in a table can cause the dictionary cache to overflow. If this happens and your dictionary cache is less than the default value of 1 MB, increase your dictionary cache to 1 MB.

With multi-value compression, no decompression is required to access compressed data values. This removes the CPU cost tradeoff common to many compression implementations.

**Note:** You cannot apply MVC to row-level security columns.

For more information about row-level security, see *Security Administration*.

Specify options in CREATE TABLE and ALTER TABLE requests to control multi-value compression (for more information, see *SQL Data Definition Language*). You can compress column values using MVC individually or in combination with algorithmic compression.

## Algorithmic Compression

You can use algorithmic compression to compress table columns with the following data types:

- ARRAY
- BYTE
- VARBYTE
- BLOB
- CHARACTER
- VARCHAR
- CLOB
- JSON, with some restrictions listed below
- TIME and TIME WITH TIME ZONE
- TIMESTAMP and TIMESTAMP WITH TIME ZONE
- Period types

- Distinct UDTs, with some restrictions listed below
- System-defined UDTs, with some restrictions listed below

These restrictions apply:

- You cannot use ALC to compress columns that have a data type of structured UDT.
- The TD\_LZ\_COMPRESS and TD\_LZ\_DECOMPRESS system functions compress all large UDTs including UDT-based system types such as Geospatial, XML, and JSON. However, if you write your own compression functions, the following restrictions apply:
  - Custom compression functions cannot be used to compress UDT-based system types (except for ARRAY and Period types).
  - Custom compression functions cannot be used to compress distinct UDTs that are based on UDT-based system types (except for ARRAY and Period types).
- You cannot write your own compression functions to perform algorithmic compression on JSON type columns. However, Teradata provides the JSON\_COMPRESS and JSON\_DECOMPRESS functions that you can use to perform ALC on JSON type columns.
- You cannot use ALC to compress temporal columns:
  - A column defined as SYSTEM\_TIME, VALIDTIME, or TRANSACTIONTIME.
  - The DateTime columns that define the beginning and ending bounds of a temporal derived period column (SYSTEM\_TIME, VALIDTIME, or TRANSACTIONTIME).

You can use ALC to compress Period data types in columns that are nontemporal; however, you cannot use ALC to compress derived period columns.

For details about temporal tables, see *Temporal Table Support* and *ANSI Temporal Table Support*.

- You cannot specify multi-value or algorithmic compression for a row-level security constraint column.

For more information about row-level security, see *Security Administration*.

You can apply algorithmic compression to referential integrity columns.

Depending on the implementation, algorithmic compression can be either physical or logical, though most implementations use physical data compression. Algorithmic compression can be either lossy or lossless, depending on the algorithm used.

Teradata provides the following system-defined external UDFs for algorithmic compression and decompression.

UDF Name	Function
TransUnicodeToUTF8	<p>Compresses the specified Unicode character data into UTF-8 format. The result data type is VARBYTE(64000). This is useful when the characters are in the ASCII script (U+0000 to U+007F) because UTF-8 uses one byte to represent these characters and Unicode (UTF-16) uses two bytes. TransUnicodeToUTF8 can only compress characters from the ASCII script of Unicode (U+0000 to U+007F).</p>

UDF Name	Function
TransUTF8ToUnicode	Uncompresses data that was compressed using TransUnicodeToUTF8.  The result data type is VARCHAR(32000) CHARACTER SET UNICODE.
LZCOMP	Compresses the specified Unicode character data using Lempel-Ziv coding.  The result data type is VARBYTE(64000).
LZDECOMP	Uncompresses Unicode data that was compressed using LZCOMP.  The result data type is VARCHAR(32000) CHARACTER SET UNICODE.
LZCOMP_L	Compresses the specified Latin character data using Lempel-Ziv coding.  The result data type is VARBYTE(64000).
LZDECOMP_L	Uncompresses Latin data that was compressed using LZCOMP_L.
CAMSET	Compresses the specified Unicode character data into partial byte 4-bit values for numeric characters or partial 5-bit values for alphabetic characters using a proprietary Teradata algorithm and an embedded services function.  The result data type is VARBYTE(64000).  You should not use this UDF to compress CHARACTER SET GRAPHIC character data.
DECAMSET	Uncompresses the Unicode character data that was compressed using CAMSET.  The result data type is VARCHAR(32000) CHARACTER SET UNICODE.
CAMSET_L	Compresses the specified Latin character data in the range U+0000 to U+00FF into partial byte 4-bit values for numeric characters or partial 5-bit values for alphabetic characters using a proprietary Teradata algorithm and an embedded services function.  The result data type is VARBYTE(64000).  You should not use this UDF to compress CHARACTER SET GRAPHIC character data.
DECAMSET_L	Uncompresses the Latin character data that was compressed using CAMSET_L.  The result data type is VARCHAR(64000) CHARACTER SET LATIN.

UDF Name	Function
TD_LZ_COMPRESS	<p>Compresses the specified ALC-supported data type or predefined type data using Lempel-Ziv coding.</p> <p>Valid column data types include distinct UDTs, distinct BLOB-, CLOB-, and XML-based UDTs, ARRAY, VARAY, Period, CHARACTER, VARCHAR, BLOB, CLOB, XML, JSON, Geospatial, BYTE, and VARBYTE.</p> <p>For a column that specifies both multi-value compression and algorithmic compression, only distinct UDTs based on predefined types that are valid for multi-value compression can be algorithmically compressed using TD_LZ_COMPRESS. This includes all numeric types, Period types, BLOB, CLOB, XML, distinct BLOB-based UDTs, distinct CLOB-based UDTs, distinct XML-based UDTs, Geospatial, DATE, CHARACTER, VARCHAR, BYTE, and VARBYTE types.</p> <p>The parameter data type of TD_LZ_COMPRESS must match the data type of TD_LZ_DECOMPRESS exactly.</p> <p>The result data type must match the data type of the UDT or predefined type column exactly.</p>
TD_LZ_DECOMPRESS	<p>Uncompresses data that was compressed using TD_LZ_COMPRESS.</p> <ul style="list-style-type: none"> <li>For algorithmic compression, the return data type of TD_LZ_DECOMPRESS must match the data type of the compressed column exactly.</li> <li>For algorithmic compression, the VARBYTE return length for TD_LZ_COMPRESS must match the length of the TD_LZ_DECOMPRESS VARBYTE parameter exactly.</li> </ul>
TS_COMPRESS	Compresses TIME and TIMESTAMP data.
TS_DECOMPRESS	Uncompresses TIME and TIMESTAMP data that was compressed using TS_COMPRESS.
JSON_COMPRESS	Compresses JSON data.
JSON_DECOMPRESS	Uncompresses the JSON data that was compressed using JSON_COMPRESS.

## Related Topics

Topic	Reference
How to code scalar UDFs to perform algorithmic compression on column data	<i>SQL External Routine Programming</i>
How to create the SQL definition for an algorithmic compression UDF	CREATE FUNCTION (External Form) in <i>SQL Data Definition Language</i>
How to specify those scalar UDFs in a table definition	ALTER TABLE and CREATE TABLE in <i>SQL Data Definition Language</i>
Guidelines for selecting an algorithmic compression functions	<a href="http://developer.teradata.com/extensibility/articles/selecting-an-alc-compression-algorithm">http://developer.teradata.com/extensibility/articles/selecting-an-alc-compression-algorithm</a>

Topic	Reference
Evaluating algorithmic compression UDFs	download the test suite from <a href="http://downloads.teradata.com/download/extensibility/algorithmic-compression-test-package">http://downloads.teradata.com/download/extensibility/algorithmic-compression-test-package</a>

## Block-Level Compression

Block-level compression enables any data block to be stored in compressed form. BLC applies to primary data tables, BLOB, CLOB, and XML subtables, fallback tables, join index subtables, hash index subtables, reference index subtables, journal tables, and even tables that do not survive restarts. Block-level compression is independent of any row compression, multi-value compression, or algorithmic compression applied to the same data. Block-level compression is a lossless method.

The goals of block-level compression are to save storage space and to reduce disk I/O bandwidth. While the number of physical I/O operations might be reduced by BLC, the number of logical I/O operations typically does not change. Block-level compression can use significantly more CPU to compress and decompress data dynamically, so whether query performance is enhanced with block-level compression depends on whether performance is more limited by disk I/O bandwidth or CPU usage.

Block-level compression enables a slight performance gain due to the reduced I/O traffic for hardware platforms that are CPU rich; however, for hardware platforms that are not CPU rich, block-level compression often has a negative performance effect.

Permanent journals and tables that do not survive restarts (spool, redrive, volatile, global temporary, and MultiLoad work) can be compressed at the data block level if the DBS Control setting specific to that table type is enabled (see *Utilities: Volume 1 (A-K)* for information).

**Note:** You can apply block-level compression to row-level security-protected tables.

The following site provides guidelines for evaluating the impact of block-level compression on performance and on disk space: <http://developer.teradata.com/extensibility/articles/block-level-compression-evaluation-with-the-blc-utility>. The site also provides a link to download the Block Level Compression Evaluation utility.

The compression method used by software block-level compression is the public domain algorithm known as zlib (see <http://zlib.net> for more information).

Block-level compression is controlled by:

- The Ferret utility
- DBS Control parameters (see *Utilities: Volume 1 (A-K)*)
- The BLOCKCOMPRESSION clause of ALTER TABLE, CREATE TABLE, CREATE HASH INDEX, or CREATE JOIN INDEX
- The BlockCompression reserved query band (for more information, see “[Reserved Query Bands for Managing the Block-Level Compression and Storage Temperature of Newly](#)

["Loaded Data" on page 694](#) and "SET QUERY\_BAND" in *SQL Data Definition Language Detailed Topics*).

In addition to software block-level compression, Teradata Database also offers hardware-based block-level compression for systems that have installed the required hardware. The advantage of hardware-based block-level compression is that compression and decompression of data presents very little CPU resource contention. Hardware-based block-level compression uses the Exar Corporation Lempel-Ziv-Stac algorithm.

For a comprehensive discussion of block-level compression, see *Database Administration*.

## Temperature-Based Block-Level Compression

Temperature-based block-level compression uses the temperature of data as maintained by Teradata Virtual Storage to determine what to compress. For example, COLD or WARM data might be compressed automatically, while HOT data might be decompressed automatically if it was previously compressed.

Temperature-based block-level compression applies only to permanent user data tables. The following types of tables and file system structures cannot be managed by temperature-based block-level compression.

- Primary copies of index subtables
- Permanent journal tables
- Data dictionary tables
- BLOB subtables
- Cylinder indexes
- Geospatial index subtables
- Snapshot/restore log subtables
- Tables that do not survive restarts (global temporary, volatile, spool, redrive, and MultiLoad work)

Teradata Virtual Storage tracks the cylinder temperature metrics for all tables when temperature-based block-level compression is enabled. Temperatures are in effect system-wide, and Teradata Virtual Storage takes all temperatures into account when it determines which data on a system is associated with which temperature.

Once that determination has been made, the file system judges if a table is using temperature-based block-level compression and if so, applies selective block-level compression or decompression based on the cylinder temperatures. For example, when the DBS Control parameter TempBLCThresh is set to COLD, the threshold defined for COLD applies to all tables in the system, so if the cylinders of an AUTOTEMP table are in that COLD temperature band, they are eligible for block-level compression. Similarly, if TempBLCThresh is set to HOT, the cylinders in a AUTOTEMP table would become eligible to be decompressed only if their temperature met or exceeded that threshold.

The file system cannot compress the cylinders of a MANUAL table using either ordinary block-level compression or temperature-based block-level compression. Instead, you must use the Ferret commands COMPRESS or UNCOMPRESS to change the compression status of an

existing MANUAL table or subtable manually. You can also use the BlockCompression query band to load rows into new MANUAL tables or subtables using the appropriate block-level compression.

You can activate temperature-based block-level compression for a new MANUAL table by using an ALTER TABLE request to change its BLOCKCOMPRESSION definition from MANUAL to AUTOTEMP. You can also use one of the following TVSTemperature query bands to load data into a new AUTOTEMP table or subtable.

- TVSTEMPERATURE\_PRIMARY
- TVSTEMPERATURE\_PRIMARYCLOBS
- TVSTEMPERATURE\_FALLBACK
- TVSTEMPERATURE\_FALLBACKCLOBS

You can also use the Ferret commands COMPRESS and UNCOMPRESS to manually change the state of an AUTOTEMP table. Note that Ferret returns a warning message to the requestor who attempts to do this.

**Notice:** If the block-level compressed state of the data in an AUTOTEMP table does not match its temperature, the block-level compression changes you make using Ferret might be undone by the system over time. Because of this, using the COMPRESS and UNCOMPRESS commands to change the compression status of an AUTOTEMP table is not recommended.

The same set of subtables that can be compressed with ordinary block-level compression are compressed with temperature-based block-level compression.

- Primary and fallback base table rows.
- Primary and fallback CLOB data.
- Fallback secondary index rows.

There are a number of advantages to managing compression based on the data temperature.

- Large sets of historical data that are not frequently accessed are often kept uncompressed. In some cases, you might not even realize that data is no longer being used. Temperature-based compression automatically locates such data and compresses it.
- Data in partitions of partitioned tables are often not frequently used. Temperature-based compression enables you to compress such partition data automatically.
- By using temperature-based compression with date-partitioned tables where new partitions are constantly being added, but old partitions are retained, you are relieved of determining which partitions are old enough to have become infrequently used.

Temperature-based block-level compression is dynamically controlled by the AutoTempComp background task. This task does things like the following to cylinders that contain a single or multiple user table or to multiple subtables of a single user table.

- Identifies COLD cylinders, determines if their data is already block-level compressed, and if not, compresses it.
- Identifies block-level compressed data that is no longer COLD and decompresses it. Data whose temperature changes from WARM to HOT is also decompressed.

Several DBS Control parameters define boundaries to control AutoTempComp, preventing it from continually compressing and decompressing data that is on a boundary. The TempBLCSpread parameter identifies the minimum spread for such cases. For example, compressing data that is 5% below a defined border between WARM and COLD border data and only decompressing such data when the temperature becomes 5% above the defined border.

The TempBLCThresh parameter defines the temperature at which data is block-level compressed or decompressed. This enables some sites to set the boundary between COLD and WARM, other sites to set the boundary between WARM and HOT, and other sites to set the boundary for only the defined bottom percentage of the very coldest data.

The BLOCKCOMPRESSION table-level attribute of the CREATE TABLE and ALTER TABLE statements categorizes tables into 3 groups, as indicated by the following set of keyword options.

- AUTOTEMP identifies tables that should be automatically managed based on the Teradata Virtual Storage temperature of their data.

If temperature-based block-level compression is disabled but block-level compression is enabled, Teradata Database treats AUTOTEMP tables the same as MANUAL tables.

- MANUAL identifies tables that are not managed automatically like AUTOTEMP tables are. You can specify a TVSTemperature query band option to determine the state of the data when the table is empty. Once a table is populated, the compressed state of the data in the table does not change unless you take specific action to do so using Ferret commands.

**Note:** You can use the non-temperature-related BlockCompression query band to load an empty table with non-temperature-related data to compress.

- NEVER identifies tables that should never be compressed, even if a query band or the applicable DBS Control defaults indicate otherwise.

This means that Teradata Database rejects Ferret commands to manually compress table data, but Ferret commands to decompress table data are valid.

A fourth option, DEFAULT, determines whether the block compression for a table should be defined as AUTOTEMP, MANUAL, or NEVER based on the setting of the DefaultTableMode DBS Control parameter. DEFAULT is the default for all new tables and for tables on a system that is being upgraded.

AUTOTEMP tables normally exist in a mixed compression state because temperature-based block-level compression does not directly apply compression to an entire table all at once. As various cylinders within a table grow warmer or colder over time, the file system either compresses or decompresses those cylinders as is appropriate for their Teradata Virtual Storage temperature.

Cylinders in an AUTOTEMP table become eligible for temperature-based block-level compression only when they reach, become lower than, or exceed the defined threshold for the specified TempBLCThresh option. For all of the data in a table to be block compressed (or decompressed) at once, Teradata Virtual Storage must become aware that all cylinders in the table have reached or exceeded the threshold for the specified TempBLCThresh option. This would only occur if all of the cylinders in the table had not been accessed in some time and

had been classified as COLD with respect to all other cylinders for compression or as HOT for decompression. Because of this, the best practice is not to use temperature-based block-level compression for a table that you think requires compression consistency for the entire table.

As COLD disk cylinders are compressed and consolidated across time, each cylinder can contain a greater number of data blocks. Because of this, fewer cylinders are needed to hold the compressed data. This causes additional cylinders to be categorized as COLD that might not have been considered COLD initially, and the number of allocated cylinders decreases while the number of free cylinders increases.

All these factors cause increasingly more data blocks to be stored in a compressed form, to the point where all cylinders considered by Teradata Virtual Storage to be COLD have been compressed.

When a table is in a state of mixed block-level compression, all of its data remains accessible and can be read or written to as required without incurring any notable performance issues. The only conclusions you should draw about a table with mixed block-level compression are that they do not have the most efficient space utilization at the levels of file system space or database space and the performance of read and update operations on them are often unpredictable. For those tables over which you want to have total control, MANUAL is probably the best block-level compression option.

Because the Teradata Virtual Storage temperature of data is a relative value, if you access 50% of your data daily and 50% every other day, the second group of data becomes classified as COLD even though it is accessed fairly often, and compressing data based on its temperature might compress this COLD data. You might want to mark a table such as this one as being managed manually to avoid unwanted compression.

To use temperature-based block-level compression, you should set the following DBS Control parameters as noted. When a setting is site-dependent, its setting is given as *optional*.

- BlockLevelCompression = ON  
This parameter must also set ON to enable temperature-based block-level compression.
- DisableAutoCylPack=FALSE
- AutoCylPackColddata=TRUE
- EnableTempBLC=TRUE
- DefaultTableMode=MANUAL

The best practice is to modify the option specified for the BLOCKCOMPRESSION attribute for individual tables to AUTOTEMP when required.

- TempBLCThresh=optional
- TempBLCSpread=optional
- TempBLCInterval=optional
- TempBLCIOTHresh=optional
- TempBLCPriority=optional
- TempBLCRescanPeriod=optional

For more information on these parameters, see the *Utilities: Volume 1 (A-K)*.

The permanent table data used most often remains in-memory for faster access. The data stays in VERYHOT cache until other data is used more often and replaces it in the cache. For more information manually setting the data temperature of cylinders so that they stay in-memory, see *Database Administration*.

## Combining Multi-Value, Algorithmic, and Block-Level Compression

You can combine multi-value compression, algorithmic compression, and block-level compression for the same table to achieve better compression, but as a general rule you should not use algorithmic compression with block-level compression because of the possibility of a negative performance impact for other workloads.

## Row Compression

Row compression is a form of logical data compression in which Teradata Database stores a repeating column value set only once, while any non-repeating column values that belong to that set are stored as logical segmental extensions of the base repeating set. Row compression is a lossless method.

Like multi-value compression, there is no decompression necessary to access row compressed data values. For an explanation of row compression, see “CREATE JOIN INDEX” in *SQL Data Definition Language Detailed Topics*.

**Note:** You can use row compression for hash and join indexes that are defined on row-level security-protected columns.

You control the row compression of join indexes with the syntax you use when you create an index (see “CREATE JOIN INDEX” in *SQL Data Definition Language*). The row compression of hash indexes is automatic and requires no special syntax.

## Autocompression

The term *autocompression* refers to a set of compression methods that Teradata Database can apply to a container of a column-partitioned table or join index that has COLUMN format if you do not specify the NO AUTO COMPRESS option for the container when you create the table or join index. All of the compression methods that Teradata Database uses to autocompress container data are lossless. Autocompression is most effective for a column partition with a single column and COLUMN format.

Teradata Database determines how to autocompress a physical row on either a row-by-row basis or on an a priori basis, including not applying autocompression techniques or user-specified compression to one physical row or *any* physical rows in a column partition.

If the autocompression techniques chosen for a physical row do not reduce the size of the physical row, Teradata Database does not compress the physical row. Similarly, when there is no autocompression technique that reduces the size of the physical row, Teradata Database does not compress the values for that physical row.

Teradata Database appends column partition values to a container without any autocompression until the container is full. Only then is the form of autocompression for the container determined and the compression applied to the container data. Teradata Database

appends any column partitions values added at a later time using the existing form of autocompression until the container is again full. When you add more column partition values to the column partition, another container is started and the process repeats.

Note that a container might use the compression techniques of the preceding container when Teradata Database decides to do so. Whether to reuse the autocompression techniques of the preceding container or to use a specific set of autocompression techniques for the current container is determined automatically by Teradata Database. Reusing the autocompression that was used for the preceding cylinder can improve the performance of INSERT ... SELECT requests by avoiding the determination of a specific set of autocompression techniques for each container when Teradata Database thinks it is unnecessary because the compression techniques used by the preceding container are anticipated to match what would be used anyway. In such cases, Teradata Database can just reuse the set of compression techniques used by the preceding container.

Teradata Database determines whether to apply both autocompression and user-specified multi-value and algorithmic compression to a column, and if the user-specified methods do not help to compress a container, the system only applies autocompression.

If you want to ensure that the null compression, multi-value compression, or algorithmic compression you specify is used for a column partition, specify NO AUTO COMPRESS for that partition.

If a container has autocompression, it uses several overhead bytes in the row header as an offset to compression bits, to indicate the autocompression types and their arguments for the container, autocompression bits, for a local value-list dictionary, and for present column partition values. If the container does not have autocompression, it uses 0 or more bytes for present column partition values.

**Note:** You can apply autocompression to row-level security-protected columns.

See “[Autocompression](#)” on page 300 for more information about autocompression.

## Row Header Compression

The term *row header compression* applies only to column-partitioned tables and column-partitioned join indexes, and only when their data is stored in COLUMN format. Row header compression is a lossless method.

Teradata Database packs column partition values into containers up to a system-determined limit. When that limit is reached, it begins packing column partition values into a new container.

The column partition values packed into a container must be in the same combined partition to be packed into a container. The row header occurs once for a container, using the rowID of the first column partition value as the rowID of the entire container, instead of storing a row header for each column partition value. Teradata Database can determine the rowid of a column partition value by its position within the container.

If many column partition values can be packed into a container, row header compression can greatly reduce the space needed for a column-partitioned object compared to the storage required for the same NoPI object without column partitioning.

If only a few column partition values can be packed in a container because of their width, it is possible for there to be a small *increase* in the space needed for a column-partitioned table or NoPI join index compared to the space required by the object without column partitioning. In this case, ROW format may be more appropriate than COLUMN format for storing the object.

If only a few column partition values can be stored using ROW format because the row partitioning is such that only a few column partition values occur for each combined partition, it is possible for there to be a very large increase in the space required to store a column-partitioned object compared to the space required for the same object without column partitioning. In the worst case, the space required to store a column-partitioned object can increase by up to nearly 24 times.

If this occurs, consider one of the following alternatives.

- Alter the row partitioning to produce more column partition values per combined partition.
- Remove column partitioning from the object definition.

## Teradata Database Mechanisms for Multi-Value Compression

### Multi-Value Compression Algorithm

Teradata Database uses a non-adaptive, lossless, corruption-resistant multi-value compression algorithm called the Dictionary Index method to compress values independently on a column-by-column basis. The list of compressible values is stored in the table header as an array, and presence bits are allocated in the row where an index into the array can be stored.

### Determining How Much Table Header Space is Used for Compression

The following procedure is an easy way to determine how much table header space is used for a particular set of column multi-value compressions.

- 1 Create two unpopulated tables that are identical except that one is defined with compression and the other without.
- 2 Perform the following query against both empty tables.

```
SELECT CurrentPerm  
FROM DBC.TableSizeV  
WHERE TableName = 'table_name';
```

where *table\_name* is the name of the compressed or uncompressed table.

- 3 Perform the following calculation.

$$\text{Table Header Spaced Used for Compression} = \text{CurrentPerm}_{\text{compressed}} - \text{CurrentPerm}_{\text{non-compressed}}$$

The result is the space in bytes used for multi-value compression in the table header.

## Meaning of the COMPRESS Attribute in Table DDL

The effect of a COMPRESS specification depends on the format of the definition.

IF COMPRESS is defined with ...	THEN ...
no argument	all nulls for the column are compressed to zero space.
one or more constant arguments	<ul style="list-style-type: none"><li>Each occurrence of a specified constant is compressed to zero space.</li><li>All nulls for the column are compressed to zero space.</li></ul>

The following CREATE TABLE fragment specifies that all occurrences of *cashier*, *manager*, and *programmer* for the jobtitle column as well as all nulls are to be compressed to zero space.

This definition saves 30 bytes for each row whenever an employee has one of the following job titles:

- Null
- Cashier
- Manager
- Programmer

```
CREATE TABLE employee (
    employee_number INTEGER
    ...
    jobtitle      CHARACTER(30) COMPRESS ('cashier',
                                         'manager', 'programmer')
    ...
);
```

## Guidelines for Using Multi-Value Compression

- To change the values that are compressed or to change whether or not a column uses compression, use an ALTER TABLE request. For example, the following turns off the compression in the mycol\_varchar column:

```
ALTER TABLE DB.mytable_vlc
ADD mycol_varchar NO COMPRESS;
```

- You cannot compress more than 255 distinct values for an individual column.
- You cannot create a table with more bytes compressed than there is room to store them in the table header.
- The maximum number of characters that can be listed in a COMPRESS clause is 8,188.
- You cannot compress values in columns that are any of the following:
  - Primary index columns
  - Identity columns
  - Derived table columns
  - Derived period columns

- Row-level security constraint columns
- Referenced primary key columns
- Referencing foreign key columns for standard referential integrity relationships
  - You can compress values in referencing foreign key columns for Batch and Referential Constraint referential integrity relationships.
- You can compress columns that are a component of a secondary index, but MultiLoad operations on a table with a secondary index can take longer if the secondary index column set is compressed.
  - To avoid this problem, drop any compressed secondary indexes before starting the MultiLoad job and then recreate them afterward.
- You can compress columns that are components of a referential integrity relationship.
- You can assign multi-value compression to columns that contain the following types of data:
  - Nulls, including nulls for distinct and structured non-LOB and non-XML UDTs, ARRAY/VARRAY and Period UDT data types
  - Zeros
  - Blanks
  - Constants having any of the data types supported by multi-value compression. See [“Multi-Value Compression” on page 699](#).

## Storing Data Efficiently

### Using Multi-Value Compression

Multi-value compression is one of several techniques available for storing data efficiently. Some of the other most commonly used methods are listed below this topic.

Before using multi-value compression as a technique for storing data efficiently, it is useful to understand how these techniques provide superior alternatives to algorithmic compression in some circumstances. See *SQL External Routine Programming* for general information about how to code algorithmic compression UDFs.

### Specifying DECIMAL and NUMERIC Precisions

Storage requirements for decimal data types are a function of the precision required of the values to be stored. The following table indicates the number of bytes used to store DECIMAL and NUMERIC data types of various precisions.

Number of Precision Digits	Number of Bytes Stored
1 - 2	1
3 - 4	2

Number of Precision Digits	Number of Bytes Stored
5 - 9	4
10 - 18	8

You can achieve optimal space savings by using multi-value compression together with an efficient decimal size. An example would be to define decimal precision such that a 4-byte representation was used instead of an 8-byte representation, for example DECIMAL(9) instead of DECIMAL(10). Then, multi-value compression could be used on the most frequently occurring values to reduce the storage overhead still further.

## Specifying INTEGER Precisions

Like the DECIMAL/NUMERIC data type, the INTEGER family of data types offers different levels of precision that you can harness to reduce the number of bytes used to store integer numbers.

The following table indicates the number of bytes used to store integer numbers of various precisions:

Integer Data Type	Number of Bytes Stored
BYTEINT	1
SMALLINT	2
INTEGER	4
BIGINT	8

You can achieve optimal space savings by using multi-value compression together with an efficient integer data type. An example would be to define the precision such that a 1-byte representation was used instead of a 4-byte representation: for example, BYTEINT instead of INTEGER. Then, multi-value compression could be used on the most frequently occurring values to reduce the storage overhead still further.

## Using Standardized Encodings

There are many standardized encodings of commonly used words. For example, the United States Postal Service (USPS) encodes all U.S. states and possessions using a two-character abbreviation. The lengths of the unabbreviated state and possession strings range from four characters (Guam, Iowa, Ohio, and Utah) to 30 characters (Federated States of Micronesia). While you could define a State column as CHARACTER(30) and compress its values, or as VARCHAR(30), the easiest and most compact method of denoting U.S. states and possessions is to use the two-character USPS encodings.

## Storing NUMBER Data

You can achieve optimal space savings for NUMBER data using either multi-value compression, algorithmic compression, or a combination of both.

NUMBER data is stored as a variable length field from 0 to 18 bytes in width. You can use the NUMBER type to represent both fixed point and floating point decimal numbers, depending on the syntax you use to specify the type. In the following table, the characters *p* and *s* represent precision and scale, respectively.

General Type	Syntax	Functional Description
Fixed point	NUMBER( <i>p,s</i> )	Similar to DECIMAL( <i>p,s</i> )
	NUMBER( <i>p</i> )	Similar to DECIMAL( <i>p</i> )
Floating point	NUMBER(*, <i>s</i> )	Floating point decimal with <i>s</i> fractional digits
	NUMBER	A floating point number
	NUMBER(*)	Same as NUMBER

The following table highlights the approximate equivalences/valid substitutions between the fixed and floating point NUMBER types with other exact (fixed point) and approximate (floating point) types.

You can use this NUMBER type ...	Anywhere you can use this data type ...	With a maximum precision of this many digits ...
Fixed point	<ul style="list-style-type: none"> <li>DECIMAL</li> <li>NUMERIC</li> </ul>	38
Floating point	<ul style="list-style-type: none"> <li>FLOAT</li> <li>REAL</li> <li>DOUBLE PRECISION</li> </ul>	40

Whether a NUMBER type is used to represent a fixed point value or a floating point value, it is stored in the same way and with the same accuracy. See “[Floating Point NUMBER Types](#)” on page 825, “[Non-INTEGER Numeric Data Types](#)” on page 826, and [SQL Data Types and Literals](#) for more information about the NUMBER type.

## Storing VARCHAR, VARBYTE, and VARCHAR(n) CHARACTER SET GRAPHIC Data

You should consider storing VARCHAR, VARBYTE, and VARCHAR(n) CHARACTER SET GRAPHIC data as CLOB or BLOB, or data, respectively, if the columns in question are not accessed regularly. This consideration also applies to storing VARCHAR, VARBYTE, and VARCHAR(n) CHARACTER SET GRAPHIC data as XML data. This is because large objects are stored outside the base table row in subtables (subtable rows have the same row hash value

as their associated base table rows, so both are stored on the same AMP), so there is no performance impact of non-LOB or non-XML queries against tables that have LOB or XML columns, while there is a performance cost to retrieving rows with inline VARCHAR, VARBYTE, and VARCHAR(n) CHARACTER SET GRAPHIC data if that data is not part of the request.

Conversely, character and byte columns that are frequently accessed should be stored as VARCHAR, VARBYTE, and VARCHAR(n) CHARACTER SET GRAPHIC if possible because of the performance cost of retrieving a LOB or XML string from its subtable.

## Storing Character Data

For character data, an alternative to encodings and value compressing fixed-length CHARACTER(*n*) strings is to specify the variable-length VARCHAR or LONG VARCHAR data types. The number of bytes used to store each VARCHAR or LONG VARCHAR column is the length of the data item plus 2 bytes. Contrast this to the fixed-length CHARACTER data type which uses *n* bytes per row, regardless of the actual number of characters in each individual column.

The demographics of the data determine whether VARCHAR, LONG VARCHAR, or CHARACTER plus multi-value compression is more efficient. The most important factors are:

- Maximum column length
- Average column length

Evaluate the following factors when determining which approach to storing the data is the more efficient:

- VARCHAR or LONG VARCHAR are more efficient when the difference of maximum and average column length is high and value compressibility is low.
- Multi-value compression with CHARACTER data is more efficient when the difference of maximum and average column length is low and value compressibility is high.

When neither CHARACTER nor VARCHAR/LONG VARCHAR is clearly a superior choice, use VARCHAR or LONG VARCHAR because their data requires slightly less CPU resource to manipulate than CHARACTER data.

## Typical Uses For Capacity Reclaimed By Multi-Value Compression

Following are some of the ways that reclaimed data capacity can be put to use:

- On an existing system, the introduction of multi-value compression is not likely to free a significant quantity of CPU resources. For those systems having residual CPU capacity, the space savings from multi-value compression might permit more applications to be introduced prior to the next system upgrade. If excess CPU capacity is already exhausted, then alternative ways to use reclaimed capacity must be explored.
- Perhaps the most obvious way to recycle newly liberated capacity is to keep more infrequently accessed historical data online (see “[Capacity Planning](#)” on page 691).

- Another way to use the reclaimed capacity is to improve system availability by declaring fallback for some or all tables.
- Alternatively, the excess capacity can be harnessed to improve performance by adding new indexes to various SQL workloads (see *SQL Request and Transaction Processing* and *Teradata Index Wizard User Guide* for information about automatic evaluation of workloads to develop more high-performing secondary indexes).

## Tradeoffs Between Multi-Value Compression and Storage Requirements for Compressed Values

### About Multi-Value Compression and Net Capacity for Nulls and Values

While multi-value compression removes specified values from row storage, those values do not disappear: they must be stored somewhere. This statement applies only to *values*, not nulls. Null compression is handled by the presence bits in the row header and does not have an impact on the table header.

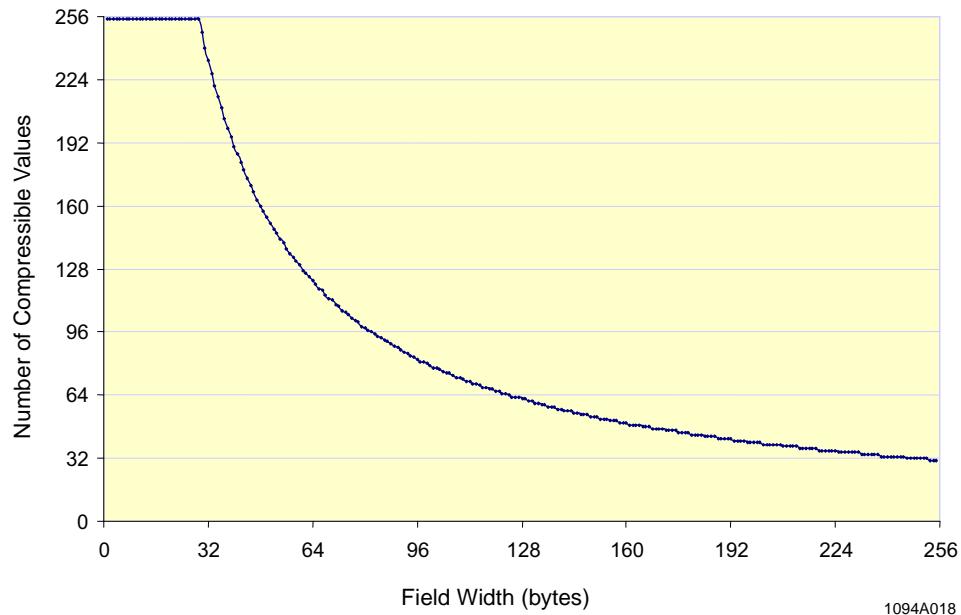
### Storage of Compressed Values

The presence bits in the row header index into field 5 of the table header, where the compressed values are stored, once per column per AMP. This does not apply to algorithmically compressed data, which is stored in place within the row except for algorithmically compressed BLOB, BLOB-related UDT, CLOB, CLOB-related UDT, XML, XML-related UDT, or Geospatial data that is generally stored in subtables. See *SQL External Routine Programming* for more information about algorithmic compression.

This is why the number of AMPs figures into the calculation of compression efficiency (see “[Equation 6: Net Capacity Usage for Multi-Value Compression](#)” on page 727).

Because the size of the table header is limited to 1 MB, there is a limit to how many bytes can be compressed for a given column. If the number of bytes compressed exceeds the maximum row length, then the CREATE or ALTER TABLE statement used to create the new table is not valid and the DDL statement aborts. This is true even if the number of values specified for compression does not exceed the upper limit of 255.

The following graph plots the number of compressible values that can be specified for a column as a function of column width.



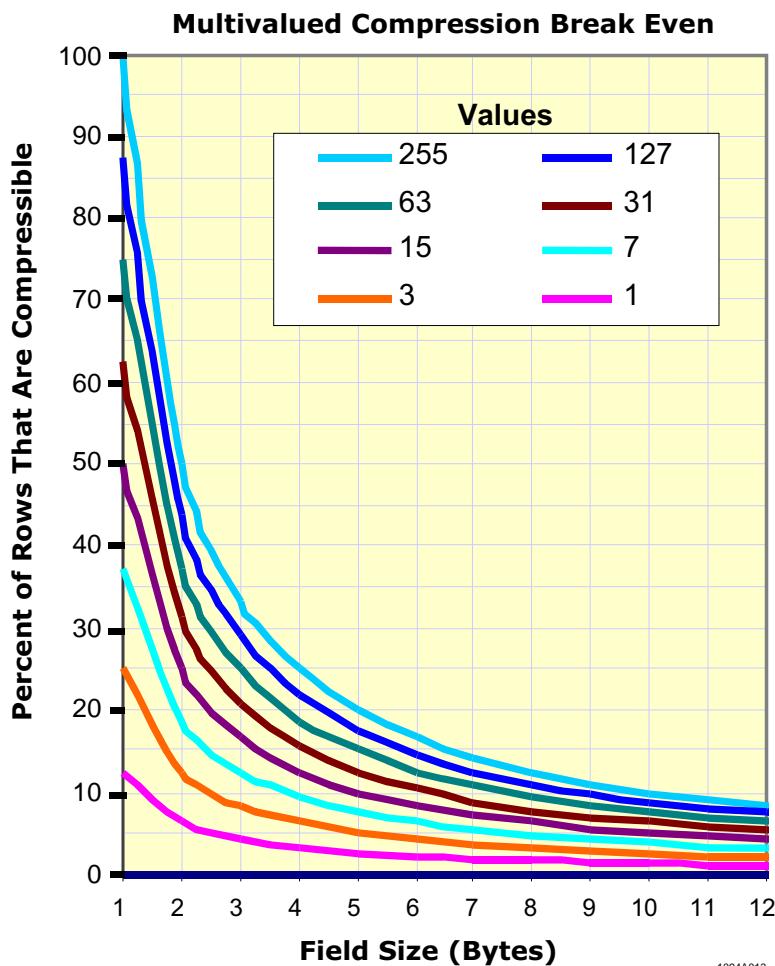
Not surprisingly, the plot clearly indicates that the wider the column, the fewer the number of values that can be compressed for the column. Particularly for wider columns, this means that in order to optimize compression, you must carefully analyze your tables to determine which values occur the most frequently and then limit compression to the top  $n$  values from that list.

## Computing and Interpreting the Break-Even Point for Multi-Value Compression

### About the Multi-Value Compression Break-Even Point

The break-even point for multi-value compression is the point at which there is zero net savings as computed by “[Equation 7: Net Capacity Usage for Multi-Value Compression With Fallback Not Enabled](#)” or “[Equation 8: Net Capacity Usage for Multi-Value Compression With Fallback Enabled](#).“ At this point, the savings in column storage are exactly balanced by the overhead for the compression bits field. In other words, compressing the specified value for a column neither adds to nor subtracts from the net storage required to compress the value.

The following graphic indicates the break-even point for a column for various numbers of distinct values by representing the percentage of compressible rows as a function of the column size width in bytes.



The break-even point for the efficiency of multi-value compression on a column varies as a function of the size of the compressed column. This is obvious. The more data values (larger column size) you can compress, the more storage space you save.

A break-even point represents a boundary condition. If compression of a value or null for a column results in a higher percentage of rows being compressed (expressed as a negative value for net capacity usage) than the break-even point, then it is worth doing; otherwise, it provides no benefit.

## Computing the Break-Even Percentage

Another way to look at breaking even is to compute the percentage of rows in the table that must be compressed in order to realize no net savings. When you can value-compress a higher percentage of rows than this through compression (expressed as a higher value for the break-even percentage), then you should compress the value.

Use the following equation to compute the break-even percentage for multi-value compression on a column.

### Equation 1: Break-Even Percentage

$$\text{Break-even percentage} = \left( \frac{\text{table_header_space} + (\text{fallback_factor} \times \text{row_header_space})}{\text{fallback_factor} \times \text{cardinality} \times \text{field_size}} \right) \times 10$$

where:

Variable name ...	Specifies ...
<i>table_header_space</i>	<ul style="list-style-type: none"> <li>the number of table header bytes used by the compressed column for aligned row format systems.</li> <li>the number of table header bytes used by the compressed column + 2 for packed64 format systems.</li> </ul> <p style="text-align: right;"><math>\text{col} = N</math></p> $\text{table_header_space} = \text{number\_of\_AMPs} \bullet \sum_{\text{col} = 1} \text{field\_size}_{\text{col}}$
	This value indicates the number of additional bytes required in the table header to compress the specified value, nulls, or both.
<i>fallback_factor</i>	<p>whether fallback is enabled for the table.</p> <ul style="list-style-type: none"> <li>1 = fallback not enabled</li> <li>2 = fallback enabled</li> </ul>
<i>row_header_space</i>	<p>the number of presence octets (see “<a href="#">Equation 9: Additional Presence Octets</a>” on page 728) in the table.</p> $\text{row_header_space} = \text{table_cardinality} \bullet (\text{presence\_octets} - 1)$ <p>This value indicates the number of additional bytes required in the row header to compress the specified value, nulls, or both.</p>
<i>cardinality</i>	the number of rows in the table.
<i>field_size</i>	the width of the compressed column in bytes.
<i>field_size<sub>col</sub></i>	the number of table header bytes used by the compressed column + 1.

Field size, fallback, table header space, row header space, and cardinality are all factors in this determination, so the break-even point is different depending on the demographics of the case being evaluated.

# Determining How Much Multi-Value Compression Can Be Realized

## Percent Multi-Value Compression

You can determine how much multi-value compression can be achieved for a set of parameters by calculating its percent compression. Percent multi-value compression is the savings when considering the compressed size in comparison to the uncompressed size. The following tables show the percent compression for a billion row table along several dimensions:

- Column widths of 2, 4, 8, and 16 bytes.
- Number of compressed distinct values per column ranging through 1, 3, 7, 15, 31, 63, 127, and 255 (corresponding to the range maxima expressible by 1 to 8 compress bits in the row header)
- Number of rows with value compressible columns equal to 20, 50, 80, and 100 percent of the cardinality for the table.

The row header overhead for multi-value compression can be larger than the savings realized, thus producing a net increase, or expansion, in capacity. In other words, more storage capacity is required to store the presence bits in each row and the compressed data value in the table header than is returned to the available storage pool by compression. This condition is indicated by the MINUS character in the tables.

The multi-value compression savings expressed are for individual columns. Overall system savings are dependent on the number of columns that are compressed and the individual savings realized from each column.

## Byte Alignment Considerations for Multi-Value Compression

As described in [“Byte Alignment” on page 770](#), rows are always aligned on even byte boundaries. You must take this into consideration when you are determining how many values to compress for a given column because for short rows with all fixed-length columns, compression can actually increase the number of bytes in the table.

Byte alignment effects on compression are important when the following things are true for a table.

- Row lengths are short.
- Most or all columns have fixed length data types.
- Few columns are compressed.

Byte alignment effects on compression are less important when the following things are true for a table.

- Row lengths are highly variable because of variable length columns.
- The number of columns is high and many are compressed.

Consider the following example. Suppose you have a table with these row characteristics:

- 14-byte row header
- 4-byte nullable non-unique primary index column
- 2-byte nullable SMALLINT non-index data column

The total number of bytes for this row is 20.

Because the primary index and SMALLINT columns are both nullable in this scenario, each uses a null presence bit in the row header, so 5 unused presence bits remain in the default presence octet.

Suppose you decide to compress 63 distinct values in the SMALLINT column. This requires an additional 6 presence bits (see “[Presence Bits](#)” on page 780 and “[Number of Presence Bits Required to Represent Compressed Values](#)” on page 784), rolling over into a new presence bits octet. The row header is now 15 bytes wide, where it was previously only 14 bytes, but when a row contains a compressed value for the SMALLINT column, it is  $15 + 4 = 19$  bytes wide, an apparent savings of 1 byte for each such row in the table.

Upon further analysis, you realize that all the rows you thought were 19 bytes wide are actually 20 bytes wide, so no savings are accrued by multi-value compression. The reason the rows expanded from 19 to 20 bytes is the system-enforced even-byte row alignment: the system added a 20<sup>th</sup> filler byte to the row to ensure an even offset.

Suppose that instead of compressing 63 distinct values in the SMALLINT column, you compress only 31. In this case, there is no need to roll over to a second presence octet, so many rows compress to 18 bytes.

You could also make the primary index non-nullable (the recommended practice anyway), which also removes the need to roll over to a second presence octet. In this case, all rows can compress to 18 bytes.

## Calculating Multi-Value Percent Compression

Percent compression is a normalized measure of the increase in total table capacity achieved by multi-value compression. “[Equation 2: Percent Multi-Value Compression](#)” is the equation for computing percent compression.

### Equation 2: Percent Multi-Value Compression

$$\text{Percent multi-value compression} = \left(1 - \frac{\text{table\_capacity}_{\text{compressed}}}{\text{table\_capacity}_{\text{uncompressed}}}\right) \times 100$$

where:

This term ...	Specifies ...
$\text{table\_capacity}_{\text{compressed}}$	the value calculated by “ <a href="#">Equation 4: Table Capacity for Multi-Value Compression</a> ” on page 726.
$\text{table\_capacity}_{\text{uncompressed}}$	the value calculated by “ <a href="#">Equation 3: Uncompressed Table Capacity</a> ” on page 725.

## Percent Savings Realizable With 20% Multi-Value Compression

The following table indicates the percentage of savings that can be realized when 20% of the subject rows can be multi-value compressed. A MINUS sign indicates net capacity expansion rather than savings.

Number of Values	Column Width (bytes)			
	2	4	8	16
1	14	17	18	19
3	7	14	17	18
7	1	11	15	18
15	-	7	14	17
31	-	4	12	16
63	-	1	11	15
127	-	-	9	15
255	-	-	7	14

## Percent Savings Realizable With 50% Multi-Value Compression

The following table indicates the percentage of savings that can be realized when 50% of the subject rows can be multi-value compressed. A MINUS sign indicates net capacity expansion rather than savings.

Number of Values	Column Width (bytes)			
	2	4	8	16
1	44	47	48	49
3	37	44	47	48
7	31	41	45	48
15	25	37	44	47
31	19	34	42	46
63	12	31	41	45
127	6	28	39	45
255	-	25	37	44

## Percent Savings Realizable With 80% Multi-Value Compression

The following table indicates the percentage of savings that can be realized when 80% of the subject rows can be multi-value compressed.

Number of Values	Column Width (bytes)			
	2	4	8	16
1	74	77	78	79
3	67	74	77	78
7	61	71	75	78
15	55	67	74	77
31	49	64	72	76
63	42	61	71	75
127	36	58	69	75
255	30	55	67	74

## Percent Savings Realizable With 100% Multi-Value Compression

The following table indicates the percentage of savings that can be realized when 100% of the subject rows can be multi-value compressed.

Number of Values	Column Width (bytes)			
	2	4	8	16
1	94	97	98	99
3	87	94	97	98
7	81	91	95	98
15	75	87	94	97
31	69	84	92	96
63	62	81	91	95
127	56	78	89	95
255	50	75	87	94

# Calculating the Efficiency of Multi-Value Compression

This topic defines several measures of the efficiency of multi-value compression.

Several factors determine whether it is useful to value compress a column or not. The net capacity usage statistic consolidates these factors and provides an objective measure of how efficient it would be to compress a particular value for a column.

## Table Capacity

Table capacity is a measure of the storage space occupied by a table. Table capacity varies as a function of multi-value compression. The more values compressed, the smaller the computed value for table capacity.

[“Equation 3: Uncompressed Table Capacity”](#) defines raw table capacity in the absence of multi-value compression.

### Equation 3: Uncompressed Table Capacity

$$\text{Table\_Capacity}_{\text{uncompressed}} = (\text{number\_of\_AMPs})(\text{base\_table\_header\_size})$$

$$+ (\text{cardinality})(14 + \text{additional\_nullability\_presence\_bytes})$$

$$+ (\text{cardinality}) \left( \sum_{\text{col}=1}^{\text{col=N}} \text{field\_size}_{\text{col}} \right)$$

where:

This term ...	Specifies the number of ...
<i>number_of_AMPs</i>	AMPs in the system.
<i>base_table_header_size</i>	bytes in the table header.
<i>cardinality</i>	rows in the table.
<i>additional_nullability_presence_octets</i>	extra presence octets required to account for the compressed values.
<i>field_size<sub>col</sub></i>	number of bytes in a compressed column.

[“Equation 4: Table Capacity for Multi-Value Compression”](#) defines table capacity with multi-valued compression enabled for one or more columns.

### Equation 4: Table Capacity for Multi-Value Compression

$$\begin{aligned}
 \text{Table\_capacity}_{\text{compression}} = & \text{ number\_of\_AMPs}(\text{base\_table\_header\_size}) \\
 & + \text{number\_of\_AMPs} \left( \sum_{\text{col}=1}^{\text{col=M}} (\text{FieldSize}_{\text{col}} \times \text{number\_distinct\_values}_{\text{col}}) \right) \\
 & + (\text{number\_of\_AMPs} \times \text{number\_of\_multi-value\_compression\_columns} \times 1 \text{ byte}) \\
 & + \text{cardinality} \times (\text{additional\_presence\_octets} + 14) \\
 & + \text{cardinality} \times \sum_{\text{col}=1}^{\text{col=N}} (\text{FieldSize}_{\text{col}})
 \end{aligned}$$

where:

This term ...	Specifies the number of ...
<i>number_of_AMPs</i>	AMPs in the system.
<i>base_table_header_size</i>	bytes in the table header.
<i>field_size<sub>col</sub></i> *	the number of table header bytes used by the compressed column + 1.
<i>cardinality</i>	rows in the table.
<i>additional_nullability_presence_octets</i>	extra presence octets required to account for the compressed multi-values.
<i>field_size<sub>col</sub></i>	bytes in a compressed column.

### Compression Ratio

The compression ratio is the ratio of uncompressed table capacity to compressed table capacity for multi-value compression. The ratio is a normalized measure of the effectiveness of the multi-value compression. The larger the value for the compression ratio, the more effective the compression.

### Equation 5: Compression Ratio

$$\text{Compression ratio} = \frac{\text{table\_capacity}_{\text{uncompressed}}}{\text{table\_capacity}_{\text{compressed}}}$$

## Relationship Between Percent Multi-Value Compression and Compression Ratio

The values for percent multi-value compression and compression ratio for a given compression calculation are related as indicated in the following two equations. Let PC represent percent compression and CF represent compression factor.

$$\text{Percent compression} = 100 - \frac{100}{\text{CF}}$$

$$\text{Compression ratio} = \frac{100}{100 - \text{PC}}$$

## Net Capacity Usage for Multi-Value Compression

Net capacity usage is a measure of the net savings realized from multi-value compressing column data values.

IF the computed value for net capacity usage is ...	THEN ...
positive	the column is a poor candidate for multi-value compression because it would require more net space to compress the specified value than would be saved in row space.
negative	multi-value compression will produce a net space savings for the column because it reduces the net space used to store the compressed value and recovered row space savings.

### Equation 6: Net Capacity Usage for Multi-Value Compression

$$\begin{aligned} \text{Net capacity usage} &= \text{number\_of\_AMPs} \left( \sum_{\substack{\text{col=M} \\ \text{col=1}}}^{\text{col=N}} (\text{number\_of\_distinct\_values}_{\text{col}} \times \text{FieldSize}_{\text{col}}) \right) \\ &\quad + (\text{number\_of\_AMPs} \times \text{number\_of\_compressed\_columns}) \\ &\quad + (\text{cardinality} \times \text{number\_additional\_compression\_presence\_bytes}) \\ &\quad - \sum_{\substack{\text{col=N} \\ \text{col=1}}}^{\text{col=1}} (\text{rows\_with\_compressed\_values}_{\text{col}} \times \text{FieldSize}_{\text{col}}) \end{aligned}$$

Calculate the value for net capacity usage with “[Equation 7: Net Capacity Usage for Multi-Value Compression With Fallback Not Enabled](#)” on page 727 and “[Equation 8: Net Capacity Usage for Multi-Value Compression With Fallback Enabled](#)” on page 728.

### Equation 7: Net Capacity Usage for Multi-Value Compression With Fallback Not Enabled

$$\text{Net capacity usage} = \text{table\_header\_space} + \text{row\_header\_space} - \text{row\_vacated\_space}$$

where:

This term ...	Specifies the number of ...
<i>table_header_space</i>	<p>table header bytes used by the compressed column set + 2.</p> $\text{table\_header\_space} = \text{number\_of\_AMPs} \cdot \sum_{\text{col} = 1}^{\text{N}} \text{field\_size}_{\text{col}}$ <p>This value indicates the number of additional bytes required in the table header to compress the specified values, nulls, or both.</p>
<i>row_header_space</i>	<p>presence octets (see “<a href="#">Equation 9: Additional Presence Octets</a>” on page 728) in the table.</p> $\text{row\_header\_space} = \text{table\_cardinality} \cdot (\text{presence\_octets} - 1)$ <p>This value indicates the number of additional bytes required in the row header to compress the specified values, nulls, or both.</p>
<i>row_vacated_space</i>	<p>compressed bytes in the compressed column.</p> $\text{row\_vacated\_space} = \sum_{\text{col} = 1}^{\text{N}} (\text{number\_compressed\_rows}_{\text{col}} \cdot \text{field\_size}_{\text{col}})$ <p>This value indicates the number of bytes in the table saved by compression.</p>

### Equation 8: Net Capacity Usage for Multi-Value Compression With Fallback Enabled

If the table has fallback enabled, some of the terms in the equation must be doubled.

$$\text{Net capacity usage} = \text{table\_header\_space} + 2(\text{row\_header\_space}) - 2(\text{row\_vacated\_space})$$

### Presence Octets

The number of additional presence octets for a table is a measure of how many octets (see “[Presence Bits](#)” on page 780) are required to define the multi-value compressibility of all the values for all the columns for a given table beyond those needed to express nullability. “[Equation 9: Additional Presence Octets](#)” presents the formula for calculating the number of additional presence octets in a table.

### Equation 9: Additional Presence Octets

where:

$$\text{Number of additional presence octets} = \text{floor}\left(\frac{\sum_{\text{col}=1}^{\text{col=N}} \text{presence\_bits}_{\text{col}}}{\frac{8 \text{ bits}}{\text{octet}}}\right)$$

This term ...	Specifies ...
$\text{floor}(x)$	<p>a function that returns the greatest integer less than or equal to <math>x</math>.</p> <p>This function is common in the mathematics library of most programming languages. See <i>SQL Functions, Operators, Expressions, and Predicates</i> for documentation of the Teradata SQL implementation of the FLOOR function.</p> <p>The FLOOR function rounds the result of the expression up to the nearest integer, which is the exact number of presence octets required to store the compression information for the given table.</p>
$\sum \text{presence\_bits}_{\text{col}}$	<p>the sum of all the presence bits used for all columns (from column 1 through column N) in the table.</p>
8 bits/octet	<p>8 bits define an octet.</p> <p>This term produces the number of presence octets required to contain the presence bits used by all value compressed columns in the table.</p> <p>Because the value can be a non-integer, it must be rounded up to the next highest integer number.</p>

You can calculate the individual contributions of additional nullability presence octets and additional compressibility presence octets to the total additional presence octets required to compress column values and nulls using “[Equation 10: Additional Presence Octets for Nullability](#)” on page 729 and “[Equation 11: Additional Presence Octets for Multi-Value Compressibility](#)” on page 730.

Note that available presence octet bits are always used exhaustively before adding additional presence octets to the row header.

### Equation 10: Additional Presence Octets for Nullability

$$\text{Number of additional nullability presence octets} = \text{floor}\left(\frac{\sum_{\text{col}=1}^{\text{col=N}} \text{nullability\_presence\_bits}_{\text{col}}}{\frac{8 \text{ bits}}{\text{octet}}}\right)$$

### Equation 11: Additional Presence Octets for Multi-Value Compressibility

$$\text{Number additional compressibility presence octets} = \text{num\_additional\_presence\_octets} - \text{num\_additional\_nullability\_presence\_octets}$$

The following table provides examples of representative figures for multi-value compression.

Number of Presence Bits for Nullability	Additional Presence Octets for Nullability	Residual Bits in Presence Octets After Accounting for Nullability	Number of Presence Bits for Compressibility	Additional Presence Octets for Compressibility
3	0	4	2	0
8	1	7	7	0
8	1	7	8	1
12	1	3	39	5
12	1	3	43	5

### Detailed Calculation of Multi-Value Net Capacity Usage

“[Equation 7: Net Capacity Usage for Multi-Value Compression With Fallback Not Enabled](#)” on page 727 and “[Equation 8: Net Capacity Usage for Multi-Value Compression With Fallback Enabled](#)” on page 728 describe net capacity usage at a high-level. “[Equation 12: Multi-Value Net Capacity Usage](#)” presents the same formula in greater detail.

### Equation 12: Multi-Value Net Capacity Usage

$$\begin{aligned} \text{Net capacity usage} = & \left( (\text{AMPs}) \left( \sum_{\text{col}=1}^{\text{col=N}} \text{field\_size}_{\text{col}}^* \right) \right) + (\text{cardinality})(\text{additional\_presence\_octets}) \\ & - \sum_{\text{col}=1} (\text{rows\_with\_compressed\_values}_{\text{col}})(\text{field\_size}_{\text{col}}) \end{aligned}$$

where:

This term ...	Specifies the number of ...
N	compressed columns in the table.
AMPs	AMPs in the system.
$\text{field\_size}_{\text{col}}^*$	the number of table header bytes used by the compressed column + 1.
<i>cardinality</i>	rows in the table.

This term ...	Specifies the number of ...
<i>additional_presence_octets</i>	presence octets in the table determined by “ <a href="#">Equation 9: Additional Presence Octets</a> ” on page 728.
<i>rows_with_compressed_values</i>	rows in the table having compressed values.
<i>field_size<sub>col</sub></i>	bytes in the compressed column.

## Example Multi-Value Net Capacity Usage Calculation

With the information provided by equations 1, 2, 3, and 4, you are now able to calculate the net capacity usage for a table. This example is for single-valued compression. Consider the following table demographics.

Characteristic	Description
Cardinality	$1 \times 10^9$ rows
Candidate column	<ul style="list-style-type: none"> <li>Base column size is 8 bytes (CHARACTER(8))</li> <li>Calculated column size is 10 bytes (see “<i>field_size<sub>col</sub></i>” in the preceding table).</li> <li>column is nullable</li> <li>10% of rows have a common value for the column</li> <li>5% of rows are null for the column</li> </ul>
Number of AMPs in system	2,000
Additional presence octets	0 Only one column is compressed.
Number of rows having compressed column	$\text{Compressible rows} = ((0.10)(1 \times 10^9)) + ((0.05)(1 \times 10^9)) = 1.5 \times 10^8$

The value for net capacity usage is calculated as follows. Note that because there is only 1 presence octet, the *row\_header\_space* term evaluates to 0, so it has no effect in the computation.

$$\text{Net capacity usage} = ((2000 \text{ AMPs})(10 \text{ bytes})) - ((1.5 \times 10^8)(8 \text{ bytes})) = -1\ 199\ 980\ 000 \text{ bytes}$$

The highly negative result indicates that the column is an excellent candidate for compression.

## Expanded Multi-Value Compression Examples

The following set of examples studies the relative benefits of multi-value compression for different table and system sizes. The calculations evaluate compression as if presence bits were added individually. You can also use this methodology to evaluate the outcome of compressing multiple columns within a table.

## Example 1

This example examines a case that multi-value compresses 100 values.

Fallback is enabled for this table.

The following table provides the specifications for the study.

	Bytes Recovered by Compression Expressed in Different Units of Magnitude			
	Bytes	Megabytes	Gigabytes	Terabytes
Additional Table Header Space	10,010	0	0.00	0.00
Additional Row Header Space	1,750,000,000	1,750	1.8	0.00
Savings from Row Compression	2,000,000,000	2,000	2.0	0.00
<hr/>				
Net Savings from Compression	249,989,990	250	0.2	0.00

Capacity utilization if column not compressed	20,250,000,000
Capacity utilization if column is compressed	20,000,010,010
Percent compression	1
Compression ratio	1.01
Break-Even Percentage of Compressible Rows	8.75
To Break Even, 1 in $n$ Rows Must Be Compressible, Where $n$ Is This Value	11.4

The following table indicates the multi-value net space savings recovered through compression. Notice the relative multi-value net savings *loss* for this case when 100 values are compressed rather than 1 value.

Variable	Condition
Number of AMPs in system	10
Column nullability	Nullable
Null compression	Compressed

Variable	Condition
Cardinality	$1.0 \times 10^9$
Number of rows having compressible values	$1.0 \times 10^8$
Capacity utilization if column is not compressed (bytes)	20,250,000,000
Number of values to compress	100
Column size (bytes)	10
Fallback	Yes
1 in $n$ rows is compressible, where $n$ is this value	10.0
Percentage of rows compressible in table	10
Presence bits for multi-value compression	7
Presence bits for null compression	1

## Example 2

This example examines a case that is identical to the one presented in “[Example 1](#) on [page 732](#) except that it multi-value compresses 200 values rather than 100 values.

Fallback is enabled for this table.

The following table provides the specifications for the study.

	Bytes Recovered by Compression Expressed in Different Units of Magnitude			
	Bytes	Megabytes	Gigabytes	Terabytes
Additional Table Header Space	120	0	0.0	0.00
Additional Row Header Space	250,000,000	250	0.3	0.00
Savings from Row Compression	2,000,000,000	2,000	2.0	0.00
<hr/>				
Net Savings from Compression	1,749,999,880	1,750	1.7	0.00
<hr/>				
Capacity utilization if column not compressed	20,250,000,000			
Capacity utilization if column is compressed	18,500,000,120			
Percent compression	9.00			

	Bytes Recovered by Compression Expressed in Different Units of Magnitude			
	Bytes	Megabytes	Gigabytes	Terabytes
Compression ratio	1.09			
Break-Even Percentage of Compressible Rows	1.25			
To Break Even, 1 in $n$ Rows Must Be Compressible, Where $n$ Is This Value	80.00			

Variable	Condition
Number of AMPs in system	10
Column nullability	Nullable
Null compression	Compressed
Number of values to compress	200
Cardinality	$1.0 \times 10^9$
Number of rows having compressible values	$1.0 \times 10^8$
Capacity utilization if column is not compressed (bytes)	20,250,000,000
Number of values to compress	200
Column size (bytes)	10
Fallback	Yes
1 in $n$ rows is compressible, where $n$ is this value	10.0
Percentage of rows compressible in table	10
Presence bits for multi-value compression	8
Presence bits for null compression	1

The following table indicates the net space savings recovered through multi-value compression. Notice the absolute net savings *loss* for this case when 200 values are multi-value compressed rather than 100 values. Multi-value compression of this column actually increases the space consumed rather than decreasing it.

	Bytes Recovered by Compression Expressed in Different Units of Magnitude			
	Bytes	Megabytes	Gigabytes	Terabytes
Additional Table Header Space	20,010	0	0.0	0.00
Additional Row Header Space	2,000,000,000	175	0.2	0.00
Savings from Row Compression	2,000,000,000	2,000	2.0	0.00
<hr/>				
Net Savings from Compression	-20,010	-2,000	-2.0	0.00

Capacity utilization if column not compressed	20,250,000,000
Capacity utilization if column is compressed	20,250,020,010
Percent compression	0
Compression ratio	1.00
Break-Even Percentage of Compressible Rows	10.00
To Break Even, 1 in $n$ Rows Must Be Compressible, Where $n$ Is This Value	10.00

### Example 3

This example examines a case that is identical to the one presented in “[Example 2](#)” on [page 733](#) except that fallback is not enabled for this table.

The following table provides the specifications for the study.

Variable	Condition
Number of AMPs in system	10
Column nullability	Nullable
Null compression	Compressed

Variable	Condition
Number of values to compress	200
Cardinality	$1.0 \times 10^9$
Number of rows having compressible values	$1.0 \times 10^8$
Capacity utilization if column is not compressed (bytes)	10,125,000,000
Column size (bytes)	10
Fallback	No
1 in $n$ rows is compressible, where $n$ is this value	10.0
Percentage of rows compressible in table	10
Presence bits for multi-value compression	8
Presence bits for null compression	1

The following table indicates the net space savings recovered through multi-value compression. Notice the absolute net savings *loss* for this case when 200 values are compressed rather than 100 values. Multi-value compression of this column actually increases the space consumed rather than decreasing it.

	Bytes Recovered by Compression Expressed in Different Units of Magnitude			
	Bytes	Megabytes	Gigabytes	Terabytes
Additional Table Header Space	20,010	0	0.0	0.00
Additional Row Header Space	1,000,000,000	175	0.2	0.00
Savings from Row Compression	1,000,000,000	2,000	2.0	0.00
<hr/>				
Net Savings from Compression	-20,010	-2,000	-2.0	0.00

Capacity utilization if column not compressed	10,125,000,000
Capacity utilization if column is compressed	10,120,020,010
Percent compression	0
Compression ratio	1.00

	Bytes Recovered by Compression Expressed in Different Units of Magnitude			
	Bytes	Megabytes	Gigabytes	Terabytes
Break-Even Percentage of Compressible Rows	10.00			
To Break Even, 1 in $n$ Rows Must Be Compressible, Where $n$ Is This Value	10.00			

## Example 4

This example examines a case that is identical to the one presented in “[Example 2](#)” on [page 733](#) except that the length of the multi-value compressed column is 40 bytes rather than 10 bytes.

Fallback is enabled for this table.

The following table provides the specifications for the study.

Variable	Condition
Number of AMPs in system	10
Column nullability	Nullable
Null compression	Compressed
Cardinality	$1.0 \times 10^9$
Number of rows having compressible values	$1.0 \times 10^8$
Capacity utilization if column is not compressed (bytes)	80,250,000,000
Number of values to compress	200
Column size (bytes)	40
Fallback	Yes
1 in $n$ rows is compressible, where $n$ is this value	10.0
Percentage of rows compressible in table	10
Presence bits for multi-value compression	8
Presence bits for null compression	1

The following table indicates the net space savings recovered through multi-value compression. Notice the net savings *gain* for this case when the number of bytes in the multi-value compressed column is increased to 40 from 10. This not only demonstrates the obvious fact that greater space savings can be extracted by multi-value compressing wider column values, but that there is a crossover point between where the column width is such that a net

savings cannot be realized by multi-value compression and where the column width is such that a net savings *can* be realized by multi-value compression.

	Bytes Recovered by Compression Expressed in Different Units of Magnitude			
	Bytes	Megabytes	Gigabytes	Terabytes
Additional Table Header Space	80,010	0	0.0	0.00
Additional Row Header Space	2,000,000,000	2,000	2.0	0.00
Savings from Row Compression	8,000,000,000	8,000	8.0	0.01
<hr/>				
Net Savings from Compression	5,999,919,990	6,000	6.0	0.01

Capacity utilization if column not compressed	80,250,000,000
Capacity utilization if column is compressed	74,250,080,010
Percent compression	7
Compression ratio	1.08
Break-Even Percentage of Compressible Rows	2.50
To Break Even, 1 in $n$ Rows Must Be Compressible, Where $n$ Is This Value	40.00

## Example 5

This example examines a case that is identical to the one presented in “[Example 2](#)” on [page 733](#) except that 30% of the rows in the table are multi-value compressible rather than 10% of the rows being multi-value compressible.

Fallback is enabled for this table.

The following table provides the specifications for the study.

Variable	Condition
Number of AMPs in system	10
Column nullability	Nullable

Variable	Condition
Null compression	Compressed
Cardinality	$1.0 \times 10^9$
Number of rows having compressible values	$1.0 \times 10^8$
Capacity utilization if column is not compressed (bytes)	20,250,000,000
Number of values to compress	200
Column size (bytes)	10
Fallback	Yes
1 in $n$ rows is compressible, where $n$ is this value	10.0
Percentage of rows compressible in table	30
Presence bits for multi-value compression	8
Presence bits for null compression	1

The following table indicates the net space savings recovered through multi-value compression. Notice the net savings *gain* for this case when the percentage of multi-value compressible rows in the compressed column is increased to 30 from 10.

	Bytes Recovered by Compression Expressed in Different Units of Magnitude			
	Bytes	Megabytes	Gigabytes	Terabytes
Additional Table Header Space	20,010	0	0.0	0.00
Additional Row Header Space	2,000,000,000	2,000	2.0	0.00
Savings from Row Compression	6,000,000,000	6,000	6.0	0.01
<hr/>				
Net Savings from Compression	3,999,979,990	4,000	4.0	0.00

Capacity utilization if column not compressed	20,250,000,000
Capacity utilization if column is compressed	16,250,020,010
Percent compression	20
Compression ratio	1.25

Bytes Recovered by Compression Expressed in Different Units of Magnitude				
	Bytes	Megabytes	Gigabytes	Terabytes
Break-Even Percentage of Compressible Rows	10.00			
To Break Even, 1 in $n$ Rows Must Be Compressible, Where $n$ Is This Value	10.00			

## Base Table Row Format

The structure of Teradata Database base table rows is slightly different for rows in the following rectangular conditions.

- A table having an nonpartitioned primary index versus a table having a partitioned primary index.
- A system using *byte-packed* format referred to as *packed64* format versus a system using *64-bit byte-aligned* referred to as *aligned row* format.

Row format information is used to make fine-grained row size estimates for capacity planning.

### General Row Structure

The graphics in the two following topics describe the structure of a Teradata Database row in systems using packed64 format and systems using aligned row format. Whether a system stores its data in packed64 or aligned row format depends on the settings on several factors your Teradata support personnel can modify. The size of tables on a system that stores data in packed64 format is generally between 3% and 9% smaller than the size of the same tables on a system that stores data in aligned row format (the average difference is roughly 7% smaller for the packe64d row format). Storing data in packed64 format reduces the number of I/O operations required to access and write rows in addition to saving disk space.

Either 12 or 16 bytes of the row header are devoted to overhead, depending on whether the table has a non-partitioned or a partitioned primary index. The maximum row length is approximately 64 KB (the actual limit is 64,256 bytes) except for spool file rows, which can be up to approximately 1MB long, and this limit is the same for both packed64 and aligned row formats. This maximum length does not include BLOB, CLOB, or XML columns, which are stored in special subtables outside the base table row. See “[Sizing a LOB or XML Subtable](#)” on page 861 for information about BLOB, CLOB, and XML subtables.

The number of characters that can be represented by this number of bytes varies depending on whether characters are represented by one byte, two bytes, or a combination of single-byte and multibyte representations.

There are three general categories of table columns, which are stored in the row in the order listed:

- 1 Fixed length.  
Storage is always allocated in a row for these columns.
- 2 Compressible.  
Includes both value-compressed and algorithmically compressed data and can be stored in any order.

FOR this type of column ...	Storage is allocated ...
Multi-value-compressible	<p>in a row when needed.</p> <p>No disk storage space is required when the column contains a compressed value as specified in the CREATE TABLE DDL text.</p> <p>If a column value is not compressed, then Teradata Database allocates storage space for it.</p>
algorithmically-compressible	<p>in a row when needed.</p> <p>If a column is not multi-value compressed or NULL, Teradata Database allocates storage space for it.</p> <p>Algorithmically-compressed data requires less space.</p>

- 3 Variable length.  
Storage space is allocated in the row depending on the size of the column.
- The structure of Teradata Database base table rows varies slightly for tables having an nonpartitioned primary index versus a partitioned primary index (see “[Row Structure for Packed64 Systems](#)” on page 753 and “[Row Structure for Aligned Row Format Systems](#)” on page 755 for details).
- For a system using the packed64 format, columns are stored in field ID order, with the fixed length fields first followed by the compressed fields, and last by the variable length fields.
- In aligned row format, fixed length columns are stored first, followed by compressed fields, and then by variable length fields. This ordering is the same as the ordering for packed64 format storage with the exception that the columns within each category are stored in decreasing alignment constraint order.
- For example, if a row contains fixed length columns with the types CHARACTER, INTEGER and FLOAT, the floating point numbers are stored first, followed by the INTEGER numbers, and then by the CHARACTER columns regardless of the order in which the columns were defined in the CREATE TABLE request defining the table.
- Teradata Database uses a space optimization routine for aligned format rows that can store a maximum of 7 bytes of data in the potentially unused space that can occur when a column is aligned on a  $0(\text{mod } 8)$  boundary. After the routine fills the available space with candidate data, Teradata Database follows the rules outlined in the preceding paragraphs to complete the remainder of the row data in an aligned format row.

## Sector Alignment

The Teradata File System supports devices that use either a native 512 byte sector size or a native 4KB sector size.

The new 4KB disk drives can be separated into 2 classes: those that support I/O on non-4KB aligned disk boundaries and those that do not support non-4KB aligned I/O. When operating on system with 4KB drives, Teradata Database only performs I/Os on blocks that are full 4KB aligned (in size and length), regardless of the class of the drive.

Teradata Database only supports a homogenous alignment configuration throughout the entire system. On a given system, data on all storage devices is aligned or unaligned, and cannot be both. This also applies across cliques.

## General Row Structure When Compressing Variable Length Columns

The following information describes aspects of general row structure when one or more variable length columns in a table are compressed. The description applies to columns having any of the following data types:

- VARBYTE
- VARCHAR
- VARCHAR(n) CHARACTER SET GRAPHIC
- UDT
- CLOB/BLOB

The description assumes that all variable length compressible columns are treated as an extension of fixed length compressible columns except that decompressed variable length columns store both a length and the actual column data. There is an additional presence bit for an algorithmically compressed column to indicate whether the column data is compressed or not. Teradata Database sets this bit only when data in a column is compressed using algorithmic compression.

When column data is compressed algorithmically, Teradata Database stores it as length: data pairs interleaved with the other compressible columns in the table. When column data is null, Teradata Database does not store a length, so there is no overhead in that case.

The following summary information applies to general row structure elements for the storage of variable length column data for multi-value compression.

- When compression does not apply to a value in a column that specifies multi-value compression, Teradata Database stores the column data in the row as a length and data value pair.

If the length of the column data is  $\leq 255$  bytes, then Teradata Database stores the length in 1 byte; otherwise it stores the length in 2 bytes.

A variable length column (without compression) always has a 2-byte offset associated with it, whereas a compressed variable length column can have its length stored in one or two bytes, depending on whether the column length is  $\leq 255$  or not, respectively.

- Teradata Database stores uncompressed variable length data in the row as a length and data value pair.

The stored length for multi-value compressed data is the length of the *uncompressed* column value.

- If a variable or fixed length multi-value compressed column is compressed or is null, then its values are not stored in the row.

Instead, Teradata Database stores one instance of each compressed value within a column in the table header and references it using the presence bits array for the row.

- If a fixed length multi-value compressed column is not compressed, then Teradata Database stores its values in the row as data, but without an accompanying length. No length is needed because Teradata Database pads fixed length data values to the maximum length defined for their containing column.

The following summary information applies to general row structure elements for the storage of variable length column data for algorithmic compression.

- If a variable length column is not compressed, then Teradata Database stores its values in the row as a length and data value pair.
  - If a fixed length column is not compressed, then Teradata Database stores its values in the row as data, but without an accompanying length.
- Teradata Database pads fixed length data values to the maximum length defined for their containing column.
- For an algorithmically-compressed column with a variable length, Teradata Database stores its values in the row as a length and data value pair.

The stored length for algorithmically-compressed data is the length of the *compressed* column value, not its original, uncompressed length.

- If the length of the column data is  $\leq 55$  bytes, then Teradata Database stores the length in 1 byte; otherwise, it stores the length in 2 bytes.
- For an algorithmically-compressed column with either a variable length or a fixed length, Teradata Database does not store a representation of its data in the row if the column is null, but does indicate its existence in the data using the presence bit array for the row.

The following sets of examples demonstrate the specific effects on row structure for multi-value compression alone (cases 1, 2, and 3 of example 1), algorithmic compression alone (cases 1, 2, and 3 of example 2), and combined multi-value and algorithmic compression (cases 1, 2, and 3 of example 3). In each case, the row structure diagram is based on the structure for a PPI table (see “[Packed64 Row Structure for a Partitioned Table](#)” on page 754) even though the examples are all for nonpartitioned primary index tables.

## Example 1: Algorithmic Compression But No Multi-Value Compression

Suppose you define the following table to uses Huffman encoding algorithms to compress and decompress two of its columns algorithmically. This table specifies algorithmic compression on columns *alc1* and *alc2*, but no multi-value compression. Particularly relevant data for each case is highlighted in **red boldface type**.

```
CREATE TABLE t1 (
    fc1 INTEGER,
    alc1 VARCHAR(10) COMPRESS ALGCOMPRESS huffcomp
                           ALGDECOMPRESS huffdecomp,
```

```

vc1  VARCHAR( 20 ) ,
alc2 VARCHAR(10) COMPRESS ALGCOMPRESS huffcomp
          ALGDECOMPRESS huffdecomp,
vc2  VARCHAR( 20 ) ;

```

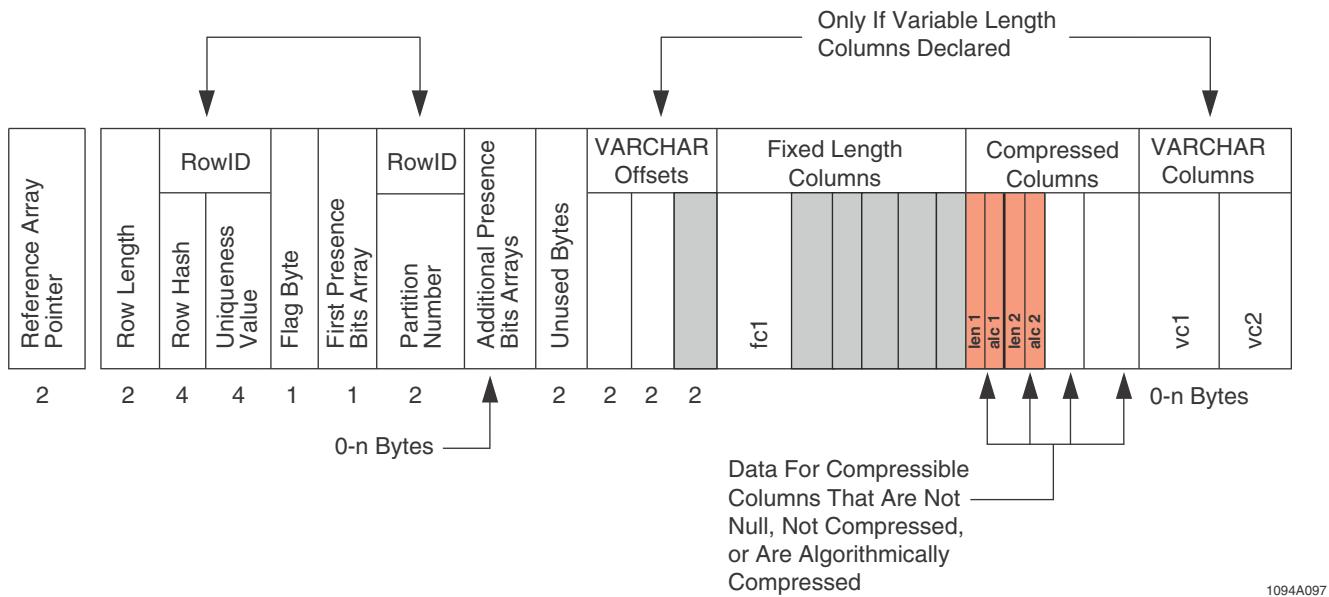
Field 5 of the table header for this table has the following field descriptors.

In- dex	Fld id	Num- set	Off- Comp-	Next Fld Ind	Com- press	UDT or Pres-	Pres- ence	Sort	EvlRepr
					Calc	Sto- ffset	Byte	Bit	
1	1025	20		2	0	Offset Nullable	0	1	AscKey DBC INTEGER
2	1026	-		4	196	Comprs alc+Null1	0	2	NonKey DBC VARCHAR(10) LATIN
3	1027	12		5	0	Var Nullable	0	4	NonKey DBC VARCHAR(20) LATIN
4	1028	-		3	206	Comprs alc+Null1	0	5	NonKey DBC VARCHAR(10) LATIN
5	1029	14		0	0	Var Nullable	0	7	NonKey DBC VARCHAR(20) LATIN

To be as general as possible, each row structure diagram indicates the configuration of fields as they would be if the table had a partitioned primary index, though none of the actual table creation SQL text specifies a PPI. The diagrams also assume the system has a Packed64 row structure.

### Case 1

Suppose you have the same table, but without any algorithmic compression being defined. The row structure for this table looks something like the following diagram.



The following abbreviations are used for the various fields of the diagram for this case.

Abbreviation	Definition
fc1	Uncompressed data for column <i>fc1</i> , a fixed length data type.
len1	Length of the data in column <i>alc1</i> , an algorithmically-compressed variable length data type.
alc1	Algorithmically-compressed data for column <i>alc1</i> , a variable length data type.

Abbreviation	Definition
len2	Length of the data in column <i>alc2</i> , an algorithmically-compressed variable length data type.
alc2	Algorithmically-compressed data for column <i>alc2</i> , a variable length data type.
vc1	Uncompressed data for column <i>vc1</i> , a variable-length data type.
vc2	Uncompressed data for column <i>vc2</i> , a variable-length data type.

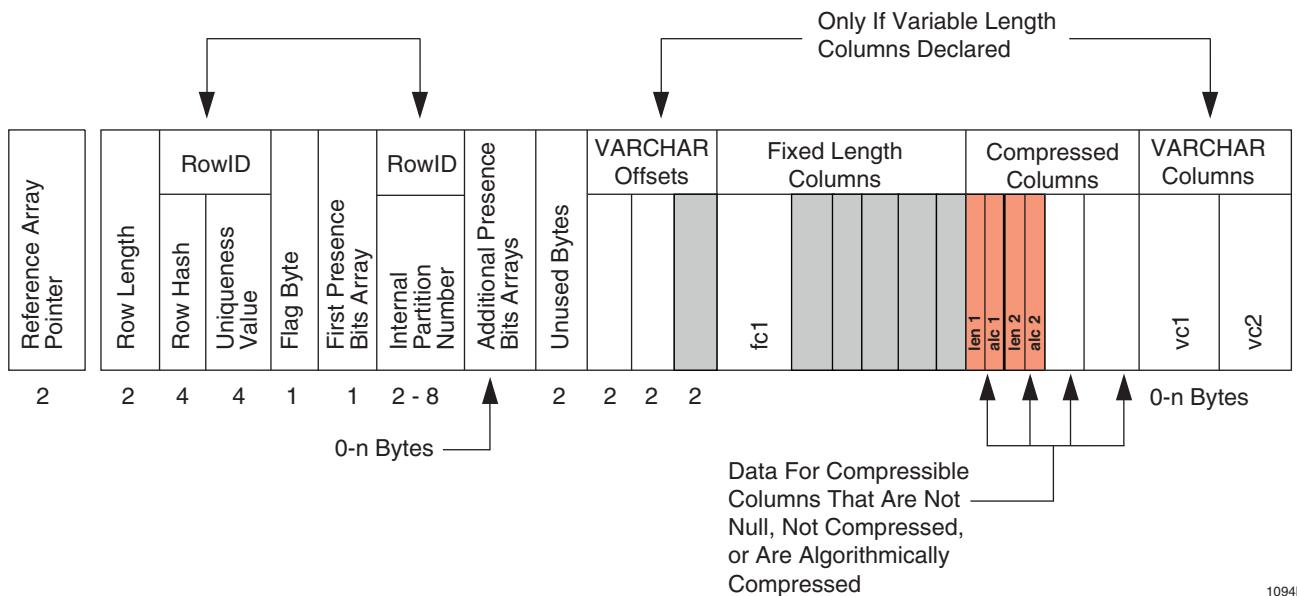
In this case, the data values for columns *alc1* and *alc2* are both stored in the row and are not compressed.

Teradata Database has placed the variable length compressible columns after the fixed length columns in the compressible columns area of the row. This is also reflected in the Field5 descriptor. Because the columns are not compressed, Teradata Database stores lengths with the two *alc* columns. *len1* and *len2* contain the uncompressed lengths of those columns, and the ALC bit is not set.

### Case 2

In the next case, algorithmic compression has been defined, and the data for columns *alc1* and *alc2* is not null. *alc1* and *alc2* are present in the row and compressed. *len1* and *len2* contain the new compressed length. The ALC bit has been set for this case to indicate that the data in columns *alc1* and *alc2* is compressed.

For this case, the row structure for the table superficially looks exactly like the row structure diagram for the previous case.

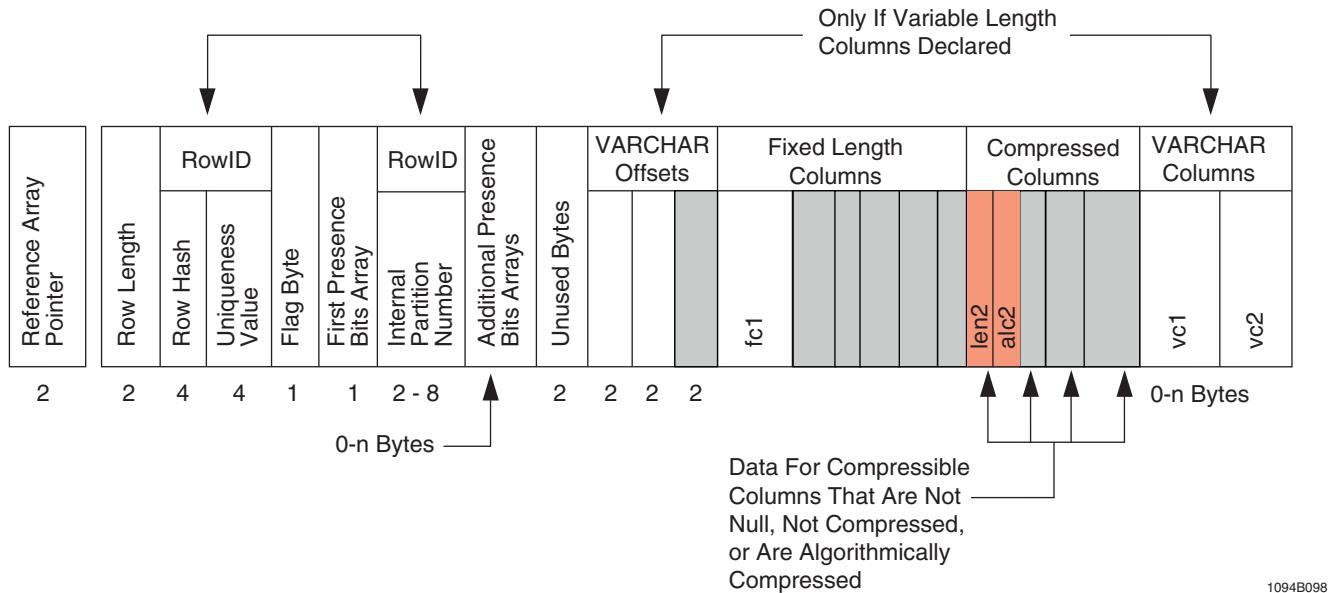


The following abbreviations are used for the various fields of the diagram for this case.

Abbreviation	Definition
fc1	Uncompressed data for column <i>fc1</i> , a fixed length data type.
len1	Length of the data in column <i>alc1</i> , an algorithmically-compressed variable length data type.
alc1	Algorithmically-compressed data for column <i>alc1</i> , a variable length data type.
len2	Length of the data in column <i>alc2</i> , an algorithmically-compressed variable length data type.
alc2	Algorithmically-compressed data for column <i>alc2</i> , a variable length data type.
vc1	Uncompressed data for column <i>vc1</i> , a variable-length data type.
vc2	Uncompressed data for column <i>vc2</i> , a variable-length data type.

### Case 3

In the next case, algorithmic compression has been defined, but the data for columns *alc1* and *alc2* is null. *alc1* and *alc2* are present in the row and compressed. *len1* and *len2* contain the new compressed length. The ALC bit is not set for this case because column *alc1* is null.



The following abbreviations are used for the various fields of the diagram for this case.

Abbreviation	Definition
fc1	Uncompressed data for column <i>fc1</i> , a fixed-length data type.
len1	Not represented. This column is null, so no length for column <i>alc2</i> is stored in the row.

Abbreviation	Definition
alc1	Not represented. This column is null, so no value for <i>alc1</i> is stored in the row.
len2	Length of the data in column <i>alc2</i> , an algorithmically-compressed variable length data type.
alc2	Algorithmically-compressed data for column <i>alc2</i> , a variable length data type.
vc1	Uncompressed data for column <i>vc1</i> , a variable-length data type.
vc2	Uncompressed data for column <i>vc2</i> , a variable-length data type.

## Example 2: Multi-Value Compression But No Algorithmic Compression

This example presents row structure cases based on the following table definition. This table has multi-value compression defined on columns *mvc1* and *mvc2*, but no algorithmic compression. Particularly relevant data for each case is highlighted in **red boldface type**.

```
CREATE TABLE t1 (
    fc1 INTEGER,
    mvc1 VARCHAR(10) COMPRESS ('mars','saturn','jupiter'),
    vc1 VARCHAR(20),
    mvc2 VARCHAR(10) COMPRESS ('Germany','France','England'),
    vc2 VARCHAR(20) );
```

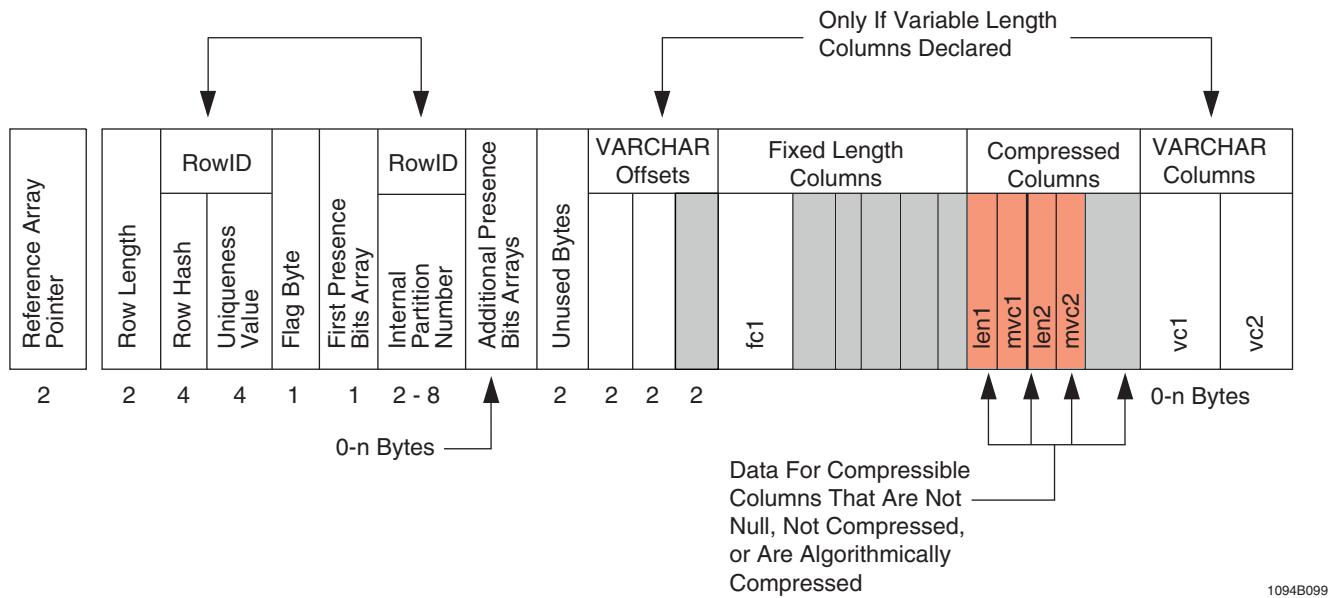
Field 5 of the table header for this table has the following field descriptors.

In- dex	Fld id	Off- set set	Num- ress	Next Comp- Ind	UDT or		Pres- ence	Sort	EvlRepr		
					Fld	Com- press	Calc	Sto- ffset	Byte	Bit	Desc
1	1025	20		2	0	Offset	Nullable	0	1	AscKey	DBC INTEGER
2	1026	-		4	196	<b>Comprs</b>	cmp+Null12	0	<b>2</b>	NonKey	DBC VARCHAR(10) LATIN
3	1027	14		5	0	Var	Nullable	0	5	NonKey	DBC VARCHAR(20) LATIN
4	1028	-		3	206	<b>Comprs</b>	cmp+Null11	0	<b>6</b>	NonKey	DBC VARCHAR(10) LATIN
5	1029	16		0	0	Var	Nullable	0	1	NonKey	DBC VARCHAR(20) LATIN

### Case 1

In this case, the data values for columns *mvc1* and *mvc2* are both stored in the row because they are not compressed for the particular data values they contain.

Teradata Database has placed the variable length compressible columns after the fixed length columns in the compressible columns area of the row. This is also reflected in the Field5 descriptor. *len1* and *len2* contain the uncompressed lengths of those columns.

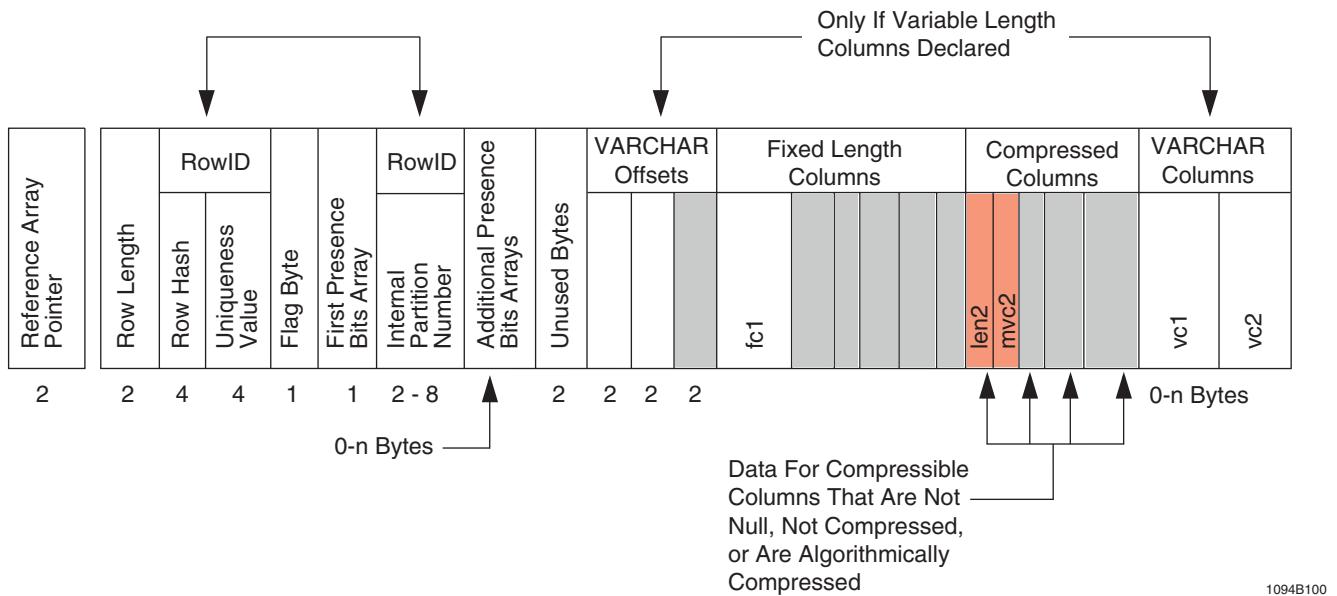


The following abbreviations are used for the various fields of the diagram for this case.

Abbreviation	Definition
fc1	Uncompressed data for column <i>fc1</i> , a fixed-length data type.
len1	Compressed length of the data in column <i>mvc1</i> , a multi-value compressed variable length column.
mvc1	Multi-value compressed data for column <i>mvc1</i> .
len2	Compressed length of the data in column <i>mvc2</i> , a multi-value-compressed variable length column.
mvc2	Multi-value compressed data for column <i>mvc2</i> .
vc1	Uncompressed data for column <i>vc1</i> , a variable-length data type.
vc2	Uncompressed data for column <i>vc2</i> , a variable-length data type.

### Case 2

The next case demonstrates row storage when the column data in the row is to be multi-value compressed. In this case, the value for column *mvc1* is not stored in the row because it is compressed. The value is instead stored in the table header and referenced by the presence bits array for the row.

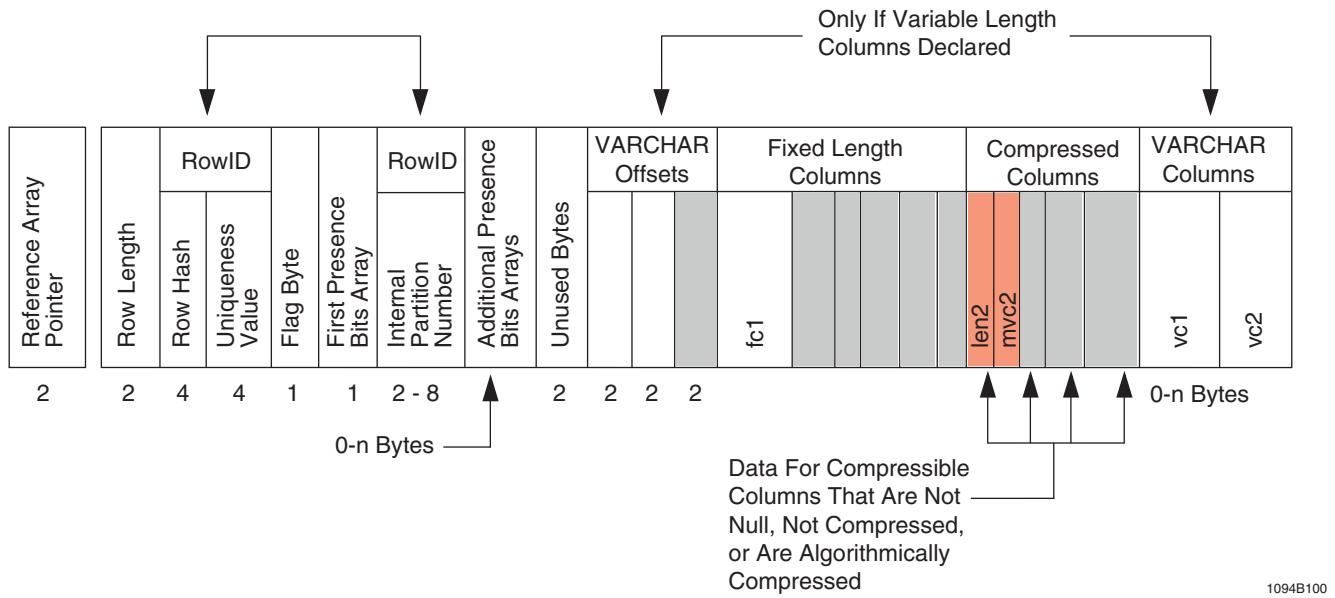


The following abbreviations are used for the various fields of the diagram for this case.

Abbreviation	Definition
fc1	Uncompressed data for column <i>fc1</i> , a fixed-length data type.
len1	Not represented. This column is null, so no length for column <i>mvc1</i> is stored in the row.
mvc1	Not represented. This column is null, so no value for <i>mvc1</i> is stored in the row.
len2	Compressed length of the data in column <i>mvc2</i> , a multi-value-compressed variable-length column.
mvc2	Multi-value compressed data for column <i>mvc2</i> .
vc1	Uncompressed data for column <i>vc1</i> , a variable-length data type.
vc2	Uncompressed data for column <i>vc2</i> , an uncompressed variable-length data type.

### Case 3

The next case demonstrates row storage when the column data for *mvc1* is null. In this case, the value for column *mvc1* is not stored in the row because nulls are always compressed. This demonstrates that nulls never incur a cost overhead for compression.



The following abbreviations are used for the various fields of the diagram for this case.

Abbreviation	Definition
fc1	Uncompressed data for column <i>fc1</i> , a fixed-length data type.
len1	Not represented. This column is null, so no value for <i>mvc1</i> is stored in the row.
mvc1	Not represented. This column is null, so no value for <i>mvc1</i> is stored in the row.
len2	Compressed length of the data in column <i>mvc2</i> , a multi-value compressed variable-length column.
mvc2	Multi-value compressed data for column <i>mvc2</i> .
vc1	Uncompressed data for column <i>vc1</i> , a variable-length data type.
vc2	Uncompressed data for column <i>vc2</i> , a variable-length data type.

### Example 3: Mix of Multi-Value and Algorithmic Compression

This example presents row structure cases based on the following table definition. This table has multi-value compression defined on columns *mvc\_f1* and *mvc\_v1*, and algorithmic compression defined on column *alc1*. Particularly relevant data for each case is highlighted in **red boldface type**.

```
CREATE TABLE t1(
    fc1      INTEGER,
    alc1     VARCHAR(10) COMPRESS ALGCOMPRESS huffcomp
                                         ALGDECOMPRESS huffdecomp,
    vc1      VARCHAR(20),
    mvc_f1   CHARACTER(10) COMPRESS ('Germany', 'France', 'England'),
```

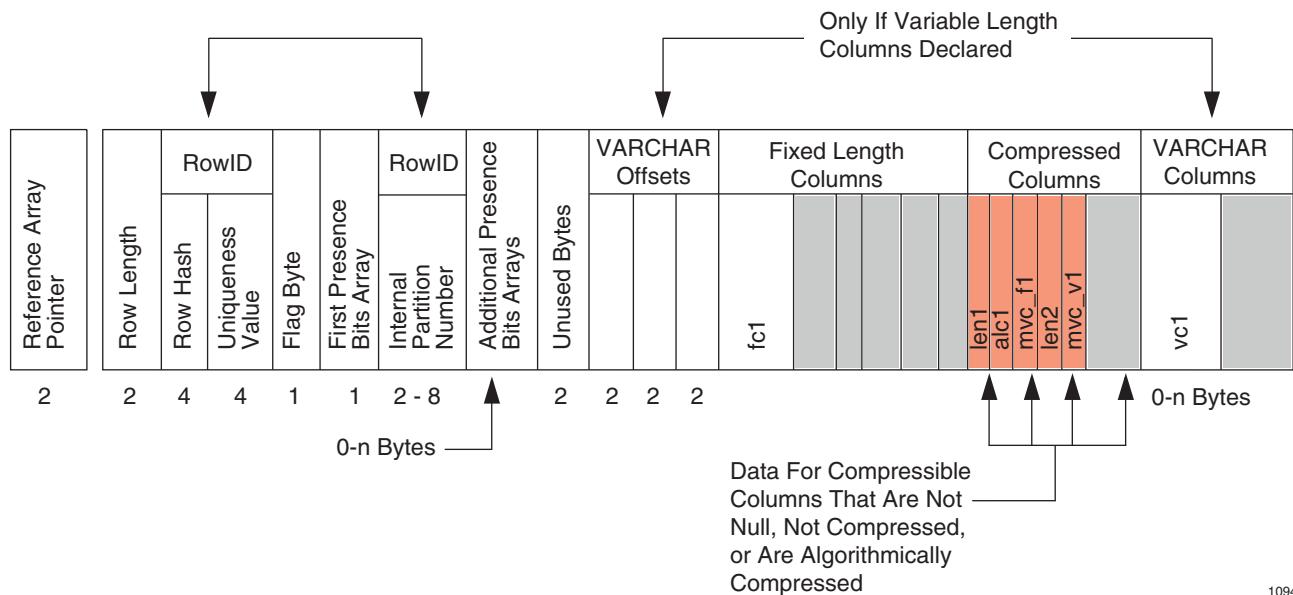
```
mvc_v1 VARCHAR(20) COMPRESS ('Nike', 'Reebok', 'Adidas') );
```

Field 5 of the table header for this table has the following field descriptors.

In- dex	Fld id	Off- set	Num- berset	Next Fld Ind	Com- press	UDT or Com- press	Cal- culation	Sto- rage	Pres- ence	Sort	EvlRepr
1	1025	20		2	0	Offset Nullable	0	1	AscKey DBC INTEGER		
2	1026	-		4	196	Comprs alc+Null1	0	2	NonKey DBC VARCHAR(10) LATIN		
3	1027	14		0	0	Var Nullable	0	5	NonKey DBC VARCHAR(20) LATIN		
4	1028	-		5	226	Comprs cmp+Null2	0	6	NonKey DBC VARCHAR(10) LATIN		
5	1029	-		3	256	Comprs cmp+Null2	1	1	NonKey DBC VARCHAR(20) LATIN		

### Case 1

In this case, no compression has been specified for the particular data values, so data for columns *alc1*, *mvc\_f1*, and *mvc\_v1* is stored in the row.



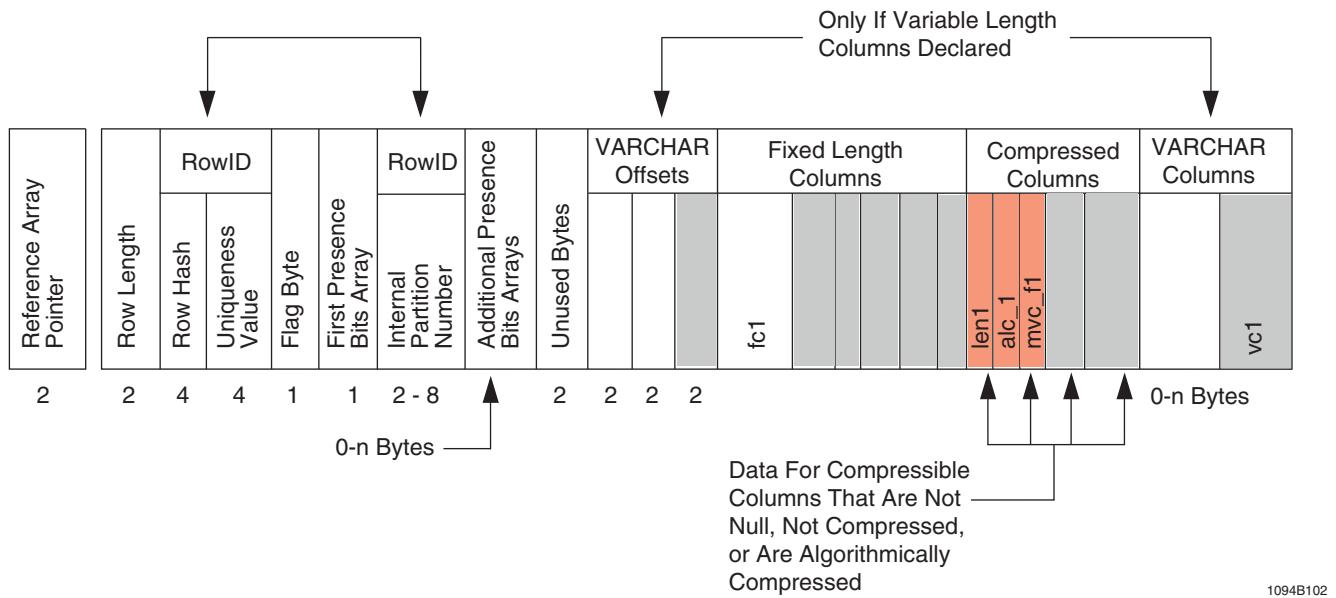
The following abbreviations are used for the various fields of the diagram for this case.

Abbreviation	Definition
fc1	Uncompressed data for column <i>fc1</i> , a fixed length data type.
len1	Compressed length of the data in column <i>alc1</i> , an algorithmically-compressed variable length data type.
alc1	Algorithmically-compressed data for the variable length column <i>alc1</i> .
mvc_f1	Multi-value compressed data for the fixed length column <i>mvc_f1</i> .
len2	Length of the data in column <i>mvc_f1</i> , a multi-value compressed variable length data type.
mvc_v1	Multi-value compressed data for the variable length column <i>mvc_v1</i> .
vc1	Uncompressed data for column <i>vc1</i> , a variable length data type.

Teradata Database has placed the variable compressible columns after the fixed field columns in the compressible columns area. This is also reflected in the Field5 descriptor. The compressed column data is interleaved with other compressible fields. *len1* and *len2* contain the *uncompressed* lengths of *alc1* and *mvc\_v1*, respectively.

### Case 2

In this case, Teradata Database compresses the data for columns *mvc\_v1* and *alc1* because the values for the row are specified in the multi-value for those columns.



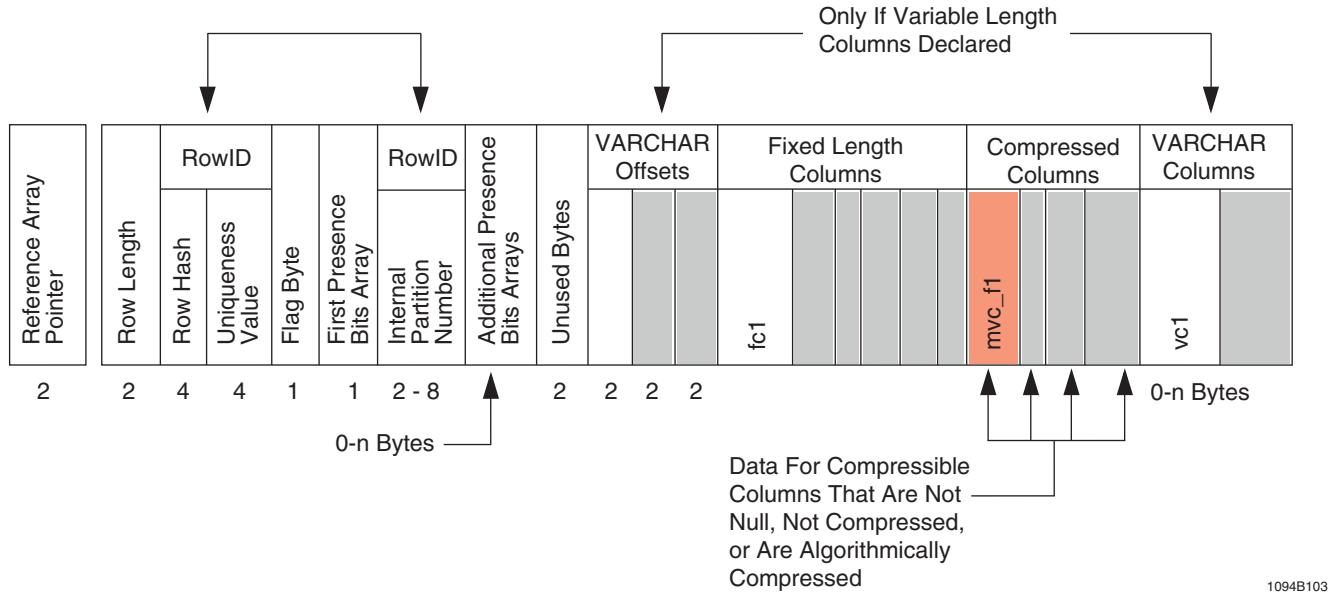
The following abbreviations are used for the various fields of the diagram for this case.

Abbreviation	Definition
fc1	Uncompressed data for column <i>fc1</i> , a fixed length data type.
len1	Compressed length of the data in column <i>alc1</i> , an algorithmically-compressed variable length data type.
alc1	Algorithmically-compressed data for column <i>alc1</i> , a variable length data type.
mvc_f1	Multi-value compressed data for column <i>mvc_f1</i> , a fixed length data type.
mvc_v1	Not represented. This column is multi-value compressed, so no value for <i>mvc1</i> is stored in the row.
vc1	Uncompressed data for column <i>vc1</i> , a variable length data type.

In this case, Teradata Database does not store the value for *mvc\_v1* in the row because it is compressed for the particular data value, so the value is stored in the table header. *len1* contains the compressed length for *alc1*.

### Case 3

In this case, the data for the compressible columns *mvc\_v1* and *alc1* is null, so Teradata Database does not store values for those columns in the row. This example demonstrates that nulls never incur a cost overhead for compression.



The following abbreviations are used for the various fields of the diagram for this case.

Abbreviation	Definition
fc1	Uncompressed data for column <i>fc1</i> , a fixed length data type.
len1	Not represented. This column is null, so no value for <i>alc1</i> is stored in the row.
alc1	Not represented. This column is null, so no value for <i>alc1</i> is stored in the row.
mvc_f1	Multi-value compressed data for column <i>mvc_f1</i> , a fixed length data type.
len2	Not represented. This column is null, so no value for <i>mvc_v1</i> is stored in the row.
mvc_v1	Not represented. This column is null, so no value for <i>mvc_v1</i> is stored in the row.
vc1	Uncompressed data for column <i>vc1</i> , a variable length data type.

## Row Structure for Packed64 Systems

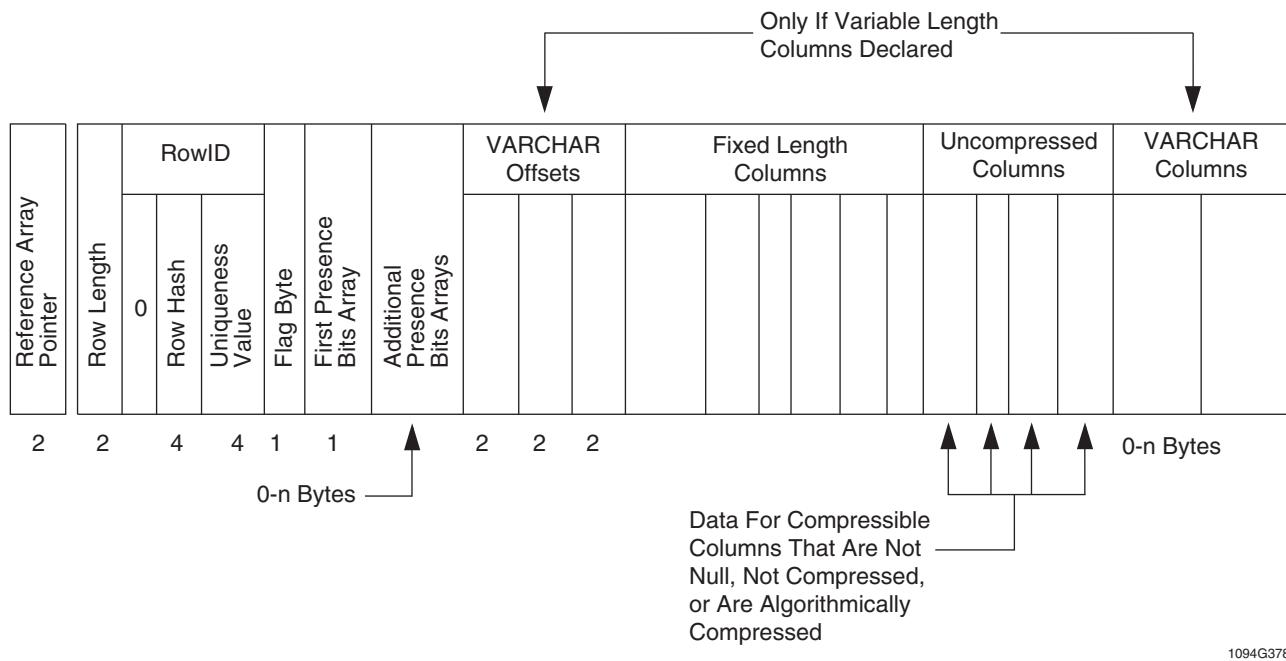
Base table rows are stored in packed format on packed64 format systems, so they need not align on 8-byte boundaries. Because of this, their row structure is simpler than that of

equivalent base table rows on aligned row format systems (see “[Row Structure for Aligned Row Format Systems](#)” on page 755).

Note that the Row Hash value is 4 bytes wide irrespective of the number of hash buckets the system has (see “[Teradata Database Hashing Algorithm](#)” on page 225).

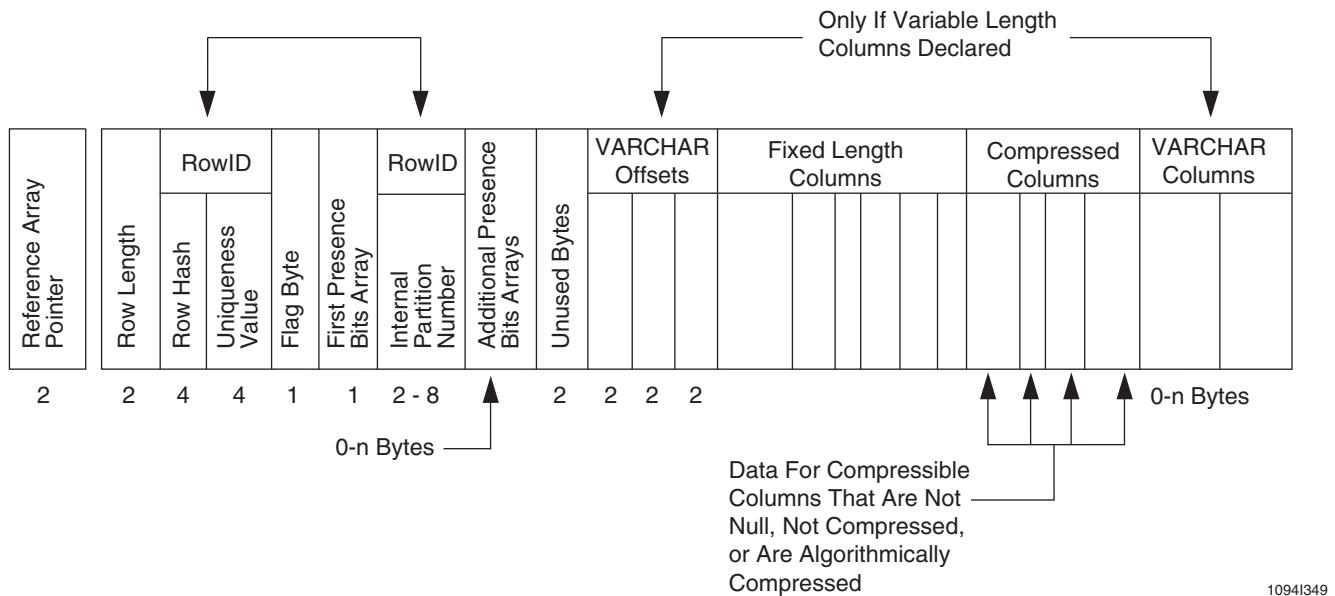
### Packed64 Row Structure for an Nonpartitioned Primary Index Table

The following graphic illustrates the basic structure of a Teradata Database row from a table on a packed64 format system with an nonpartitioned, or traditional, primary index.



### Packed64 Row Structure for a Partitioned Table

The following graphic illustrates the basic structure of a Teradata Database row from a table on a packed64 format system with a partitioned primary index:



The difference between this and the format of a nonpartitioned primary index row is the presence of a an additional 2-byte or 8-byte partition number field, which is also a component of the RowID (PPI table rows are an additional 4 bytes wider if they also specify multi-value compression). It is this field that generates the need for a BYTE(10) data type specification for a RowID. For nonpartitioned primary index tables, the partition number is assumed to be 0, so the rowID of an nonpartitioned primary index table is also logically BYTE(10) (see “[ROWID Columns](#)” on page 800).

## Row Structure for Aligned Row Format Systems

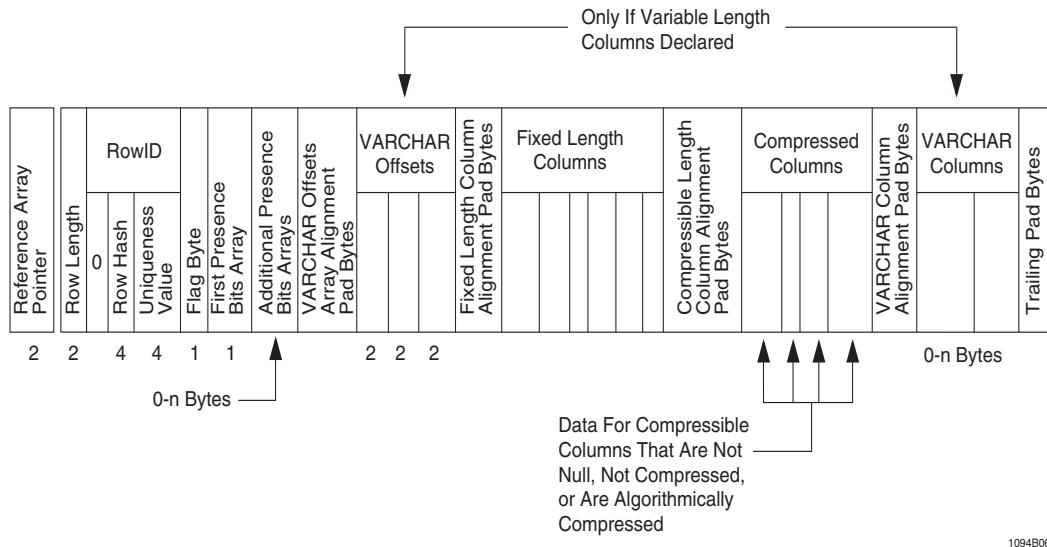
The row structure diagrams for aligned row format systems differs from those of packed64 systems by having five additional pad byte fields to ensure row alignment on an 8-byte boundary. The following table lists each of these pad fields and explains their purpose:

Pad Byte Field Name	Purpose
VARCHAR Offsets Array Alignment Pad Bytes	Aligns VARCHAR offsets array at a 2-byte boundary.
Fixed Length Column Alignment Pad Bytes	Aligns fixed length columns.
Compressible Length Column Alignment Pad Bytes	Aligns value compressible length columns.
VARCHAR Column Alignment Pad Bytes	Aligns variable length columns.
Trailing Pad Bytes	Aligns entire row on an 8-byte boundary.

Note that the Row Hash value is 4 bytes wide irrespective of the number of hash buckets the system has (see “[Teradata Database Hashing Algorithm](#)” on page 225).

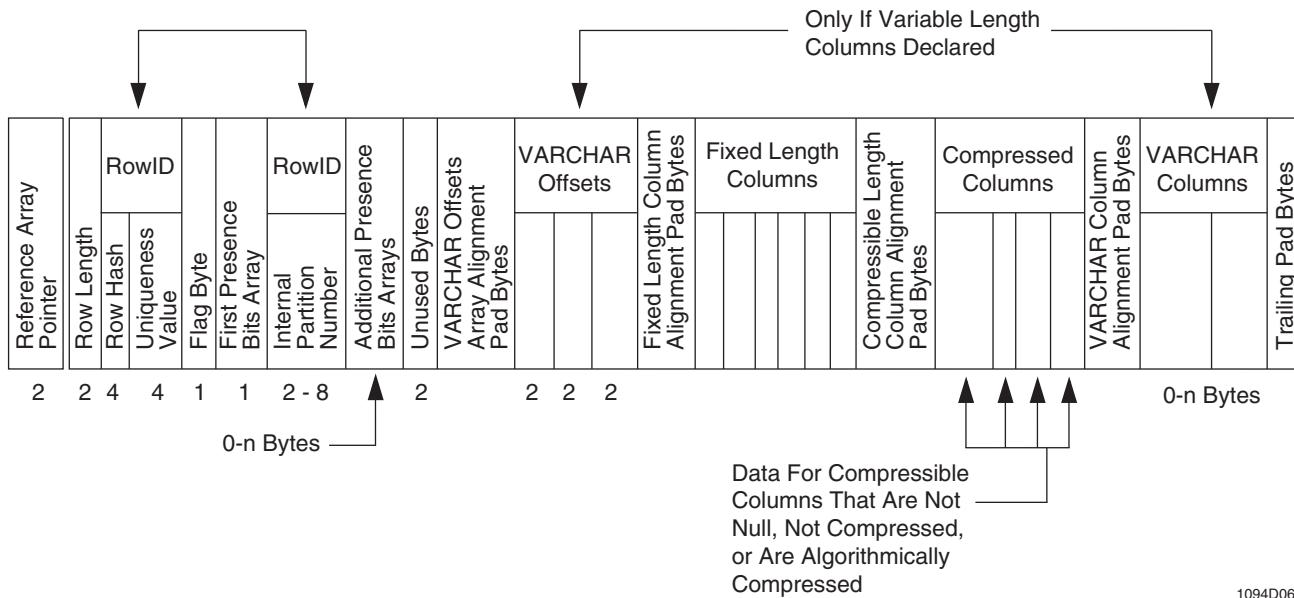
## Aligned Row Structure for an Nonpartitioned Table

The following graphic illustrates the basic structure of a Teradata Database row from a table on an aligned row format system with an nonpartitioned, or traditional, primary index.



## Aligned Row Structure for a Partitioned Table With 65,535 or Fewer Combined Partitions

The following graphic illustrates the basic structure of a Teradata Database row from a table on an aligned row format system with a partitioned primary index that has 65,535 or fewer combined partitions:



The difference between this and the format of an nonpartitioned table row is the presence of a 2-byte or 8-byte partition number field, which is also a component of the RowID (PPI table rows are an additional 4 bytes wider if they also specify multi-value compression. See “[PARTITION Columns](#)” on page 801). It is this field that generates the need for a BYTE(10)

data type specification for a RowID. For nonpartitioned primary index tables, the partition number is assumed to be 0, so the rowID of an nonpartitioned primary index table is also logically BYTE(10) (see “[ROWID Columns](#)” on page 800).

## Containers and Subrows

The data in a column-partitioned table can be stored in containers or subrows. The following table defines these terms.

### Containers and Space

If Teradata Database can pack many column partition values into a container, this form of compression, called *row header compression*, can reduce the space needed for a column-partitioned table or join index compared to the same object without column partitioning.

If Teradata Database can place only a few column partition values in a container because of their width, there can actually be a small *increase* in the space needed for a column-partitioned table or join index compared to the same object without column partitioning. In this case, ROW format may be more appropriate.

If there are only a few column partition values (because it is possible with row partitioning that only a few column partition values occur for each combined partition) and there are many column partitions, there can be a very large *increase* in the space needed for a column-partitioned object compared to the same object without column partitioning. In the worst case, the space required for a table can increase by nearly 24 times.

In this case, consider making one of the following changes:

- Alter the column or row partitioning to allow for more column partition values per combined partition.
- Remove the column partitioning from the table or join index.

### Container Row Contents

A container row must have the same internal partition number and hash bucket for all the column partition values in that container row.

Following are the contents of a container:

- The row header for a container indicates its internal partition number, hash bucket, and uniqueness value for the first column partition value. The row header is the same as for any other physical row, including physical row length, a rowID, flag byte, and first presence byte. The row header for a container is either 14 or 20 bytes long.

There is only one row header for a container, using the rowID of the first column partition value as the rowID of the container instead of there being a row header for each column partition value as is the case for all other row types in Teradata Database. You can determine the rowID of a column partition value by its position within the container.

Note that the first presence byte for a container is not used as a presence byte.

- The first presence byte for a container is not used as a presence byte. It indicates whether autocompression types have been determined for the container.

If the selected autocompression types, which might include applying user-specified compression, do not reduce the size of the container row, the AC bit in the first presence byte in the row header is set to 0 and the container row is not autocompressed.

- The number of column partition values, including values for logically deleted rows, is represented by the container and various offsets to its sections.
- An optional series of column partition values for the local value list compression dictionary.

This is preceded by arrays of offsets if the values are variable length.

- A series of fixed-length column partition values or a series of variable-length column partition values, each prefixed by a 1-byte or 2-byte length.

The series can be empty if the autocompression bits do not indicate that any column partition values are present. If this occurs, the column partition values have all been compressed.

- 0 or more bytes of free space.
- Optional autocompression presence bits, value length compression bits, algorithmic compression bits, and run length bits, depending on the autocompression techniques used in the container row in reverse order of the series of values.

A column container should have thousands of column values for fixed length and short variable-length data types unless the table is overly row-partitioned. The values and presence bits grow closer to one another as they consume the available free space. If there is insufficient free space, Teradata Database expands the row.

For a single-column partition that has COLUMN format (that is, its physical rows are containers), a column partition value is the same as a column value and can have either a fixed or a variable length.

For a multicolunm partition with COLUMN format, a column partition value has a structure similar to a regular row, containing presence and compression bits as 0 or more bytes, offsets to variable-length column values, the values of its fixed-length columns, the values of its uncompressed columns, and the values of its variable-length columns.

A container does not include:

- A row length because the length of a column partition value is handled separately
- An internal partition number
- A hash bucket
- A uniqueness value
- Presence or compression bits
- First presence bit

Teradata Database applies user-specified compression within the multicolunm column partition value and might apply autocompression to the column partition value as a whole.

Column partition values for a single-column or multicolunm column partition can have either fixed or variable length. If the column partition value has variable length, the length is not part of the column partition value. Instead, the length is specified in the container row by

a preceding length field or by using the difference between offsets if the column partition value is in the local value-list dictionary.

A container includes other information such as offsets to the beginning of the series of column partition values.

## Containers and Autocompression

A container with autocompression includes:

- 2 bytes are used as an offset to the compression bits.
- 1 or more bytes indicate the autocompression types and their arguments for the container.
- 1 or more bytes of autocompression bits, depending on the number of column partition values and the autocompression type.
- 0 or more bytes are used for a local value-list dictionary.
- 0 or more bytes are used for present column partition values.

A container without autocompression uses 0 or more bytes for present column partition values. A container can exist without autocompression because:

- You specified NO AUTO COMPRESS for the column partition when you created the table or join index.
- No autocompression types are applicable for the column partition values of the container.

**Note:** Whether a column partitioning level defaults to AUTO COMPRESS or NO AUTO COMPRESS depends on the setting of the AutoCompressDefault cost profile. See *SQL Request and Transaction Processing* for further information about AutoCompressDefault.

## Row Structure for Containers (COLUMN Format)

Teradata Database packs column partition values into a container up to a system-determined limit and then packs the next set of column partition values into a new container. The column partition values within a container must be in the same combined partition to be packed into that container.

A column partition represents one or more table columns of a table. A column partition that has COLUMN format is represented as a series of containers that hold the column partition values of the column partition.

A container consists of a header and column partition values followed by autocompression bits at the end, with free space in between. The free space is allocated in such a way that column partition values and autocompression bits can grow toward each other using the free space without moving around the values and changing the row size.

A newly constructed container in memory starts at the maximum allowed size and, therefore, a large free space. Once it either fills up or there are no more column partition values to add for the current DML request, the container can be reduced in size as described later, and it is then written to disk. If there are still more column partition values to add for this request, Teradata Database starts a new container in memory.

When new column partition values arrive to be appended for a subsequent DML request and there is a last container for the combined partition in which the column partition values are to

be appended, and subsequently insufficient free space is available for appending the next new column partition value in this last container, and the container was reduced in size when last written as described later, Teradata Database expands the container in memory to its maximum allowable size if that would enable a column partition value to be added. If the container becomes full, it can be reduced in size as described later, and it is then written to disk. The process begins again with a new container.

Before writing a container row that has reached its maximum size, either because it is a new container or because it is a last container read from disk that was expanded to the maximum size because it ran out free space, when there are no more column partition values to be added to it by a DML request, there is sufficient free space to add more column partition values, but its free space exceeds a system-determined percentage of its size without the free space, Teradata Database reduces its free space to be within this percentage.

It is possible that free space could occur in the last container for each combined partition. If there are many combined partitions, the sum of the free space could add up to a great deal of unused disk space. Therefore, it is desirable to keep this total unused space to a small percentage of the table or join index size. However, having a reasonable amount of free space in the last container minimizes the copying of a container to a larger memory area in order to accommodate new incoming column partition values such as, for example, by an array INSERT, a small INSERT ... SELECT request, or a large INSERT ... SELECT request to a row partitioned column-partitioned table or join index such that a small number of column partition values are inserted into a combined partition at a time, and most often the container can be written with the same physical row size after the insert operation, which is more efficient than if the physical row size changes.

When writing a container row at the point where it can no longer hold additional column partition values, necessitating that a new container be started, the free space for the container is reduced to 0 or near zero. The last container does not necessarily need to be written if it were just read and there is insufficient free space to add another column partition value but the remaining free space is small. However, if a container has too much remaining free space, Teradata Database must remove the free space, and the last container row must be written.

This does *not* apply to a container for the delete column partition (see “[Container Row for the Delete Column Partition](#)” on page 760) and is autocompressed.

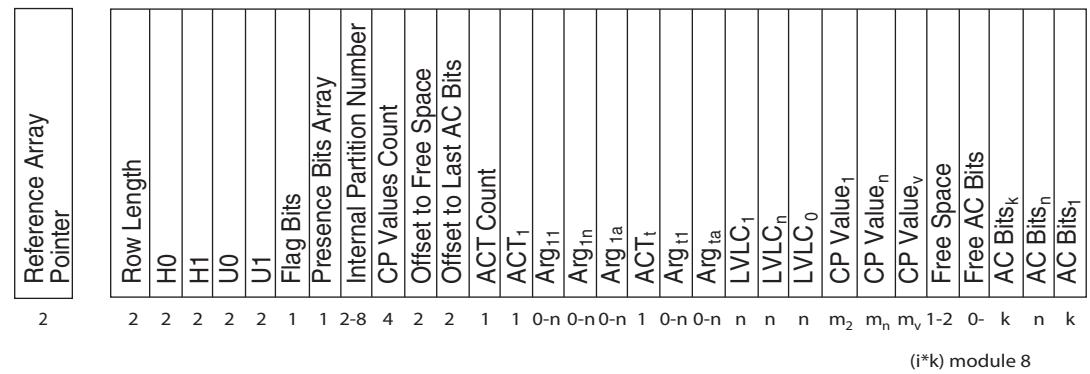
### **Container Row for the Delete Column Partition**

A container row for the delete column partition has fixed-length, single-column, BYTEINT NOT NULL column partition values. This data is constrained to have the values 0 or 1. The container row has the layout as a fixed-length container (see “[Row Structure for Fixed-Length Containers](#)” on page 760).

### **Row Structure for Fixed-Length Containers**

This row structure applies to both fixed-length single-column and multicolunm column partitions unless autocompression compresses the data as variable length.

Offsets in the container row are relative to its beginning.



1094A108

Term	Description
Row Length	Length of the row in bytes.
H0	First row hash field value for the container.
H1	Second row hash field value for the container.
U0	First uniqueness field value for the container.
U1	Second uniqueness field value for the container.
Flag Bits	Two bits indicate whether this container requires 2 bytes or 8 bytes to store its maximum number of combined partitions.
Presence Bits Array	Contains autocompression flags in a container.
Internal Partition Number	Internal partition number for the container. <ul style="list-style-type: none"> <li>The field is 2 bytes long if the maximum combined partition number for the container is <math>\leq 65,535</math>.</li> <li>The field is 8 bytes long if the maximum combined partition number for the container is <math>&gt; 65,535</math>.</li> </ul>
CP Values Count	Number of column partition values represented by this container.
Offset to Free Space	Offset to the first byte of free space.
Offset to Last AC Bits	Offset to the last byte of autocompression bits.
ACT Count	Count of the number of types of autocompression applied to the container. If the autocompression bit is not set in the first presence byte, this field is omitted from the row header.
ACT <sub>1</sub>	First autocompression type applied to this container. If the autocompression bit is not set in the first presence byte, this field is omitted from the row header.
Arg <sub>11</sub>	First argument for this autocompression type. If the autocompression bit is not set in the first presence byte, this field is omitted from the row header.

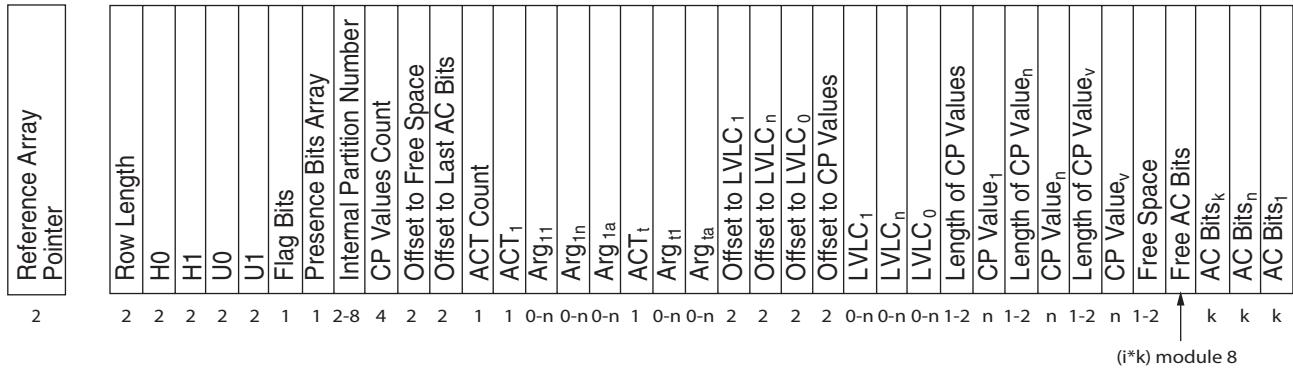
Term	Description
Arg <sub>1n</sub>	$n^{\text{th}}$ argument for autocompression type 1.
Arg <sub>1a</sub>	$a^{\text{th}}$ argument for autocompression type 1.
ACT <sub>t</sub>	$t^{\text{th}}$ autocompression type applied to this container.
Arg <sub>t1</sub>	First argument for autocompression type t.
Arg <sub>ta</sub>	$a^{\text{th}}$ argument for autocompression type t.
Offset to LVLC <sub>1</sub>	Length of LVLC <sub>1</sub> is offset in the next two bytes. Offset to LVLC <sub>1</sub> if the LVLC bit is set in the first presence byte.
Offset to LVLC <sub>n</sub>	Length of LVLC <sub>n</sub> is offset in the next two bytes. Offset to LVLC <sub>n</sub> if the LVLC bit is set in the first presence byte.
Offset to LVLC <sub>o</sub>	Length of LVLC <sub>o</sub> is offset in the next two bytes. Offset to LVLC <sub>o</sub> An ACT argument indicates how many column partition values are in the local value list compression dictionary if the LVLC bit is set in the first presence byte. $o$ is an argument to an autocompression type that specifies LVLC.
Offset to CP Values	Offset to the first column partition value. The offset to this field is $2^*o + \text{size of the autocompression types and their arguments} + \text{the offset to OffsetToLastACBits} + 2$ , where $o$ is an argument of an autocompression type indicating that there is LVLC.
LVLC <sub>1</sub>	First column partition value for the local value list compression dictionary if the LVLC bit is set in the first presence byte.
LVLC <sub>n</sub>	$n^{\text{th}}$ column partition value for the local value list compression dictionary if the LVLC bit is set in the first presence byte.
LVLC <sub>a</sub>	Last column partition value for the local value list compression dictionary if the LVLC bit is set in the first presence byte.
Length of CPValue <sub>1</sub>	If a single-column partition, the maximum length is in the field5 field descriptor. If a multicolpartition, the maximum length is in the multicolpartition descriptor. <ul style="list-style-type: none"> <li>• The field is 1 byte long if the maximum length is <math>\leq 55</math>.</li> <li>• The field is 2 bytes long if the maximum length is <math>&gt; 255</math>.</li> </ul>
CP Value <sub>1</sub>	First present column partition value. The field is $n$ bytes long, where $n$ is the number of bytes in CPValue <sub>1</sub> .

Term	Description
Length of CPValue <sub>n</sub>	If a single-column partition, the maximum length is in the field5 field descriptor.  If a multicol column partition, the maximum length is in the multicol column partition descriptor. <ul style="list-style-type: none"><li>• The field is 1 byte long if the maximum length is <math>\leq 255</math>.</li><li>• The field is 2 bytes long if the maximum length is <math>&gt; 255</math>.</li></ul>
CP Value <sub>n</sub>	$n^{\text{th}}$ present column partition value.  The field is $n$ bytes long, where $n$ is the number of bytes in CPValue <sub>1</sub> .
Length of CPValue <sub>v</sub>	If a single-column partition, the maximum length is in the field5 field descriptor.  If a multicol column partition, the maximum length is in the multicol column partition descriptor. <ul style="list-style-type: none"><li>• The field is 1 byte long if the maximum length is <math>\leq 255</math>.</li><li>• The field is 2 bytes long if the maximum length is <math>&gt; 255</math>.</li></ul>
CP Value <sub>v</sub>	Last present column partition value.
Free Space	<ul style="list-style-type: none"><li>• 1 if AC bit is set or ACTBD is set and it is a nullable single-column partition.  <math>\text{FreeSpace} = (\text{OffsetToLastACBits} - \text{OffsetToFreeSpace})\text{bytes}</math></li><li>• 2 otherwise.  <math>\text{FreeSpace} = (\text{RowLength} - \text{OffsetToFreeSpace})\text{bytes}</math></li></ul>
Free AC Bits	$\text{FreeACBits} = (i \times k) \bmod 8$ bits <ul style="list-style-type: none"><li>• Only included if the autocompression bit is set in the first presence byte.</li><li>• Otherwise set to 0.</li></ul>
AC Bits <sub>j</sub>	Last set of autocompression bits.  $k$ bits, where $k$ is the number of bits needed for compression a column partition value per the autocompression types and their arguments. <ul style="list-style-type: none"><li>• Last set of autocompression bits. Only included if the autocompression bit is set in the first presence byte.</li><li>• Otherwise set to 0.</li></ul>
AC Bits <sub>k</sub>	$k^{\text{th}}$ set of autocompression bits.  $k$ bits, where $k$ is the number of bits needed for compression a column partition value per the autocompression types and their arguments. <ul style="list-style-type: none"><li>• More sets of autocompression bits. Only included if the autocompression bit is set in the first presence byte.</li><li>• Otherwise set to 0.</li></ul>

Term	Description
AC Bits <sub>1</sub>	<p>First set of autocompression bits.</p> <p><math>k</math> bits, where <math>k</math> is the number of bits needed for compression a column partition value per the autocompression types and their arguments.</p> <ul style="list-style-type: none"> <li>• First set of autocompression bits.</li> <li>• Only included if the autocompression bit is set in the first presence byte.</li> <li>• Otherwise set to 0.</li> </ul>

### Row Structure for Variable-Length Containers

This row structure applies to both variable-length single-column and variable length multicolumn partitions. The same structure is also used for fixed-length single-column and multicolumn column partitions if autocompression compresses as variable length for a container. Offsets in a container are relative to its beginning.



where:

Term	Description
Row Length	Length of the row in bytes.
H0	First row hash field value for the container.
H1	Second row hash field value for the container.
U0	First uniqueness field value for the container.
U1	Second uniqueness field value for the container.
Flag bits	Two bits indicate whether this container requires 2 bytes or 8 bytes to store its maximum number of partitions.
Presence bits array	Used for autocompression flags in a container row.

Term	Description
Internal Partition Number	<p>Internal partition number for the container.</p> <ul style="list-style-type: none"> <li>The field is 2 bytes long if the maximum combined partition number for the container is <math>\leq 65,535</math>.</li> <li>The field is 8 bytes long if the maximum combined partition number for the container is <math>&gt; 65,535</math>.</li> </ul>
CP Values Count	Number of column partition values represented by this container.
Offset to Free Space	Offset to the first byte of free space.
Offset to Last AC Bits	Offset to the last byte of autocompression bits.
ACT Count	<p>Count of the number of types of autocompression applied to the row.</p> <p>If the autocompression bit is not set in the first presence byte, this field is omitted from the row header.</p>
ACT <sub>1</sub>	<p>First autocompression type applied to this container.</p> <p>If the autocompression bit is not set in the first presence byte, this field is omitted from the row header.</p>
Arg <sub>11</sub>	<p>First argument for this autocompression type.</p> <p>If the autocompression type has no arguments, this field is omitted from the row header.</p>
Arg <sub>1n</sub>	$n^{\text{th}}$ argument for autocompression type 1.
Arg <sub>1a</sub>	$a^{\text{th}}$ argument for autocompression type 1.
ACT <sub>t</sub>	$t^{\text{th}}$ autocompression type applied to this container.
Arg <sub>t1</sub>	First argument for autocompression type $t$ .
Arg <sub>ta</sub>	$a^{\text{th}}$ argument for autocompression type $t$ .
Offset to LVLC <sub>1</sub>	<p>Length of LVLC<sub>1</sub> is offset in next two bytes.</p> <p>Offset to LVLC<sub>1</sub> if LVLC bit is set in first presence byte.</p>
Offset to LVLC <sub>n</sub>	<p>Length of LVLC<sub>n</sub> is offset in next two bytes.</p> <p>Offset to LVLC<sub>n</sub> if LVLC bit is set in first presence byte.</p>
Offset to LVLC <sub>o</sub>	<p>Length of LVLC<sub>o</sub> is Offset in next two bytes.</p> <p>Offset to LVLC<sub>o</sub>.</p> <p>An ACT argument indicates how many CPValues are in the local VLC dictionary if the LVLC bit is set in 1st presence byte. <math>o</math> is an arg to an ACT specifying LVLC.</p>
Offset to CP Values	<p>Offset to first CPValue.</p> <p>The offset to this field is <math>2 \times o + \text{size of the ACTs and their arguments} + \text{offset to OffsetToLastACBits} + 2</math>, where <math>o</math> is an argument of an ACT indicating there is LVLC.</p>
LVLC <sub>1</sub>	First CPValue for local VLC dictionary if LVLC bit is set in the first presence byte.

Term	Description
LVLC <sub>n</sub>	$n^{\text{th}}$ CPValue for local VLC dictionary if LVLC bit is set in the first presence byte.
LVLC <sub>o</sub>	Last CPValue for local VLC if LVLC bit is set in the first presence byte. o is an argument to an ACT specifying LVLC.
Length of CPValue <sub>1</sub>	If a single-column partition, the maximum length is in the field5 field descriptor.  If a multicolumn partition, the maximum length is in the multicolumn partition descriptor. <ul style="list-style-type: none"><li>• The field is 1 byte long if the maximum length is <math>\leq 55</math>.</li><li>• The field is 2 bytes long if the maximum length is <math>&gt; 255</math>.</li></ul>
CPValue <sub>1</sub>	First present column partition value.  The field is $n$ bytes long, where $n$ is the number of bytes in CPValue <sub>1</sub> .
Length of CPValue <sub>n</sub>	If a single-column partition, the maximum length is in the field5 field descriptor.  If a multicolumn partition, the maximum length is in the multicolumn partition descriptor. <ul style="list-style-type: none"><li>• The field is 1 byte long if the maximum length is <math>\leq 55</math>.</li><li>• The field is 2 bytes long if the maximum length is <math>&gt; 255</math>.</li></ul>
CP Value <sub>n</sub>	$n^{\text{th}}$ present column partition value.  The field is $n$ bytes long, where $n$ is the number of bytes in CPValue <sub>n</sub> .
Length of CPValue <sub>v</sub>	If a single-column partition, the maximum length is in the field5 field descriptor.  If a multicolumn partition, the maximum length is in the multicolumn partition descriptor. <ul style="list-style-type: none"><li>• The field is 1 byte long if the maximum length is <math>\leq 55</math>.</li><li>• The field is 2 bytes long if the maximum length is <math>&gt; 255</math>.</li></ul>
CP Value <sub>v</sub>	Last present column partition value.
Free Space	<ul style="list-style-type: none"><li>• 1 if AC bit is set or ACTBD is set and it is a nullable single-column partition.  FreeSpace = (OffsetToLastACBits – OffsetToFreeSpace)bytes</li><li>• 2 otherwise.  FreeSpace = (RowLength – OffsetToFreeSpace)bytes</li></ul>
Free AC Bits	FreeACBits = $(i \times k)$ modulo 8 bits <ul style="list-style-type: none"><li>• Only included if the autocompression bit is set in the first presence byte.</li><li>• Otherwise set to 0.</li></ul>

Term	Description
AC Bits <sub>j</sub>	<p>Last set of autocompression bits.</p> <p><math>k</math> bits, where <math>k</math> is the number of bits needed for compressing a column partition value per the autocompression types and their arguments.</p> <ul style="list-style-type: none"> <li>• Last set of autocompression bits.</li> <li>• Only included if the autocompression bit is set in the first presence byte.</li> <li>• Otherwise set to 0.</li> </ul>
AC Bits <sub>k</sub>	<p><math>k^{\text{th}}</math> set of autocompression bits.</p> <p><math>k</math> bits, where <math>k</math> is the number of bits needed for compressing a column partition value per the autocompression types and their arguments.</p> <ul style="list-style-type: none"> <li>• More sets of autocompression bits.</li> <li>• Only included if the autocompression bit is set in the first presence byte.</li> <li>• Otherwise set to 0.</li> </ul>
AC Bits <sub>1</sub>	<p>First set of autocompression bits.</p> <p><math>k</math> bits, where <math>k</math> is the number of bits needed for compressing a column partition value per the autocompression types and their arguments.</p> <ul style="list-style-type: none"> <li>• First set of autocompression bits.</li> <li>• Only included if the autocompression bit is set in the first presence byte.</li> <li>• Otherwise set to 0.</li> </ul>

### Row Structure for Subrows (ROW Format)

A column partition that has ROW format is represented as a series of subrows, where each subrow contains a single column partition value of the column partition. Subrows have the same format as regular rows with the following exceptions:

- A regular row contains all the column values of a table row, while a subrow contains only a subset of the column values of a table row (that is, a column value for each of the columns in the column partition).
  - The internal partition number of the row ID of a regular row does not indicate a column partition because regular rows are not column-partitioned.
- The internal partition number of the row ID of a subrow indicates its column partition number.

Teradata Database applies any user-specified compression within the subrow for the column partition value.

### Alignment of Containers and Subrows

Containers and subrows are always packed, even on aligned format systems. The only differences between packed and aligned systems are:

- The length of a container or subrow is a multiple of 8 on 64-bit aligned system.
- The length of a container or subrow is a multiple of 2 on a packed system.

For a container, the file system adds any extra bytes that are required to make the length of the row even to the freespace, not to the end of the row.

On a packed system, the length of a subrow can be odd. If so, the file system adds a byte to the end of the subrow.

## Column Partitioning

Column partitioning is a form of partitioning for multiset tables and for single-table, non-aggregate, non-compressed join indexes. Columnar storage stores the data into a series of containers with usually many values of the column partition packed into each container; alternatively, the values of a column partition can be stored into a series of subrows with one value of the column partition per subrow.

Column partitioning enables sets of table or join index columns to be stored in separate partitions. Row partitioning of primary-indexed tables also enables sets of rows to be stored in separate partitions. Teradata Columnar makes it possible for a table or join index to be column-partitioned, or both column-partitioned and row-partitioned by using multilevel partitioning.

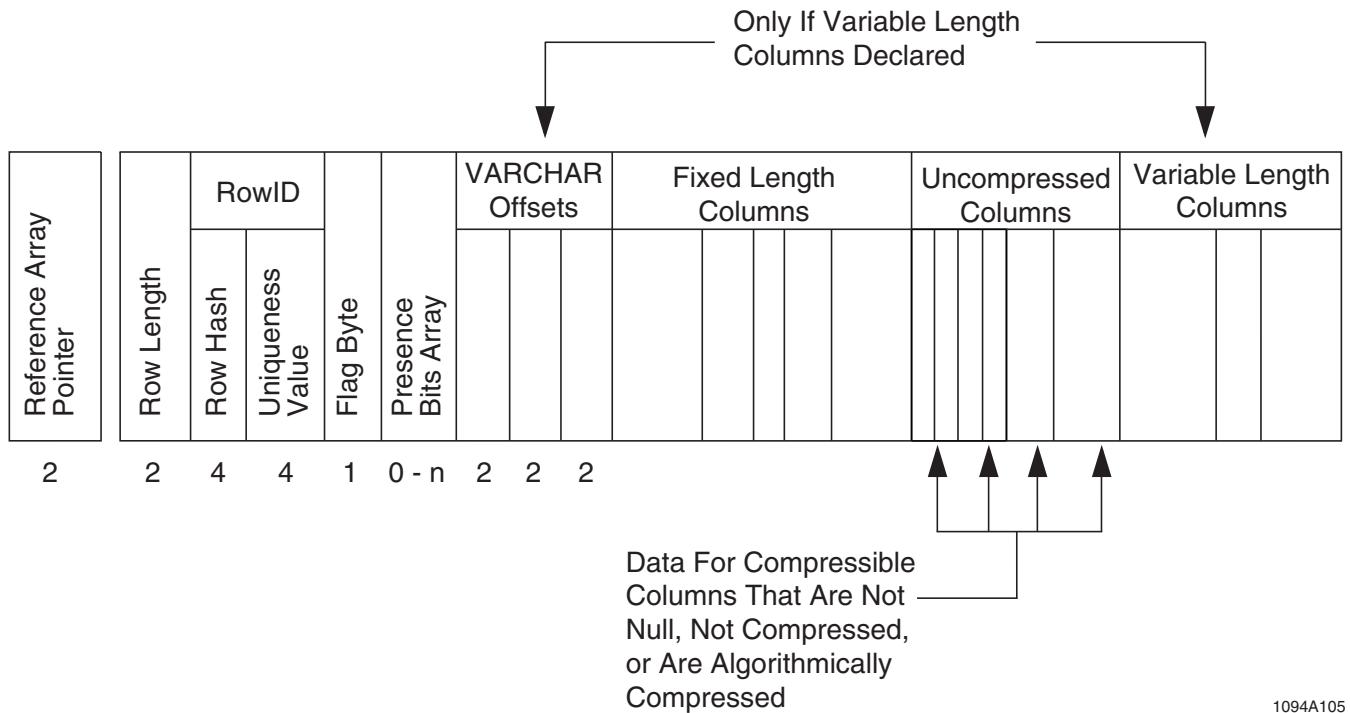
Column partitioning enables the Optimizer to devise efficient searches by using column and row partition elimination based on the columns that are needed by a query. If a table or index column is not needed by a request, the column partition with that column need not be read. If multiple columns are needed for a request, the query plan devised by the Optimizer includes putting projected column values from selected table rows together to form result rows. This can be combined with row partition elimination to further reduce the data that must be accessed to satisfy a request.

Teradata Database can apply various compression techniques to column-partitioned data that can reduce the storage requirements for a table or join index, which can then reduce the I/O requirements for DML requests. When column partitioning is combined with row partitioning, the number of compression opportunities available to the system can increase.

## Multicolumn Column Partition Values for a Container

A series of one or more multicolumn column partition values is stored as a container for a multicolumn partition.

The following diagram shows the layout of a multicolumn column partition value.



The fixed-length, compressible, and variable-length sets of columns are each stored in the order of their field IDs in their respective sets which is the order in which they are specified in the database object defining DDL text.

If there are no columns with user-specified compression and the columns are all fixed length, a multicolumn partition has fixed-length column partition values unless autocompression causes the column partition values to have variable length. Otherwise, the multicolumn partition has variable-length values, which could either be all variable-length columns or a mix of fixed-length and one or more variable-length columns.

The presence bits array is omitted from the container if there are no nullable columns or columns with user-specified compression.

A fixed-length column, whether nullable or not, fixed-length column is not user-specified to be compressed, space is consumed in the row for a fixed-length column in the fixed-length columns area.

## Consumption of Disk Space by Populated and Empty Partitions

With the large number of partitions that can be defined for a table or join index, it is very likely that a high percentage of those partitions are empty at any given time. For example, a table on a 200 AMP system that defines 100,000 combined partitions with 100 rows per unaligned (127.5 KB) data block and 100 data blocks per each combined partition per AMP has 200 billion rows. This is a relatively small number of combined partitions when you consider that the maximum for a table or join index is 9,223,372,036,854,775,807 combined partitions.

If each row were 100 bytes in length, the primary data alone consumes 20 petabytes of disk. That is  $20 \times 10^{15}$  bytes. It is highly unlikely that every combined partition would be populated. When you consider a multidimensional use of multilevel partitioning, you can easily deduce that not all combinations of dimension values actually occur.

In general, a populated combined partition should have either many data blocks per AMP or no data blocks. For the example proposed in the first paragraph, if there is actually only 200 gigabytes of data and each populated combined partition had 100 data blocks per AMP, about 99% of the combined partitions are empty.

## Byte Alignment

Rows on packed64 format systems are always aligned on even-byte boundaries. In other words, rows are never stored with an odd number of bytes. As a result, if a row has an odd byte length, the system adds a filler byte to the end of the row to make its length even. Filler bytes are included in the *CurrentPerm* total for each table column in *DBC.DatabaseSpace*.

During a SysInit operation, Teradata Database reads several flags to determine whether a system should be configured to format its data in a packed64 format or in an aligned row format. Consult your Teradata support representative if you want to change the current setting for your system. Depending on the settings of the row format flags, a freshly initialized system formats its rows in packed64 format by default, while an upgraded system continues to format its rows in aligned row format as the default.

See the table in [“Row Structure for Aligned Row Format Systems” on page 755](#).

A data block contains rows from one table only, and any row from that table is completely contained within a given data block. In the packed64 row format, a row starts on a 2-byte boundary relative to the start of a data block. This ensures alignment of columns only for those tables where the maximum alignment requirement for any column is 2.

In the aligned row format, the maximum alignment requirement for a column in a table can be 1, 2, 4, or 8 bytes. The size of each row must be a multiple of the maximum alignment requirement to ensure that each row starts on a valid boundary within a data block. For the sake of uniformity, all row sizes are multiples of 8, starting on an 8-byte boundary relative to the start of a data block. This requires additional disk space for aligning data and the multiples of 8 filler applies to both small-block-sized systems and large-block-sized systems.

Write Ahead Logging (WAL) introduces a data block header size of 72 bytes. This increase causes 5.5% of existing data blocks to increase in size by one sector (512 bytes).

The 28 byte increase in size of the data block header plus the additional pad characters in the row data that are required to ensure 8-byte boundary alignment decreases the number of rows that can be stored in a data block with respect to the same data stored on pre-WAL systems.

The high level structure of a Teradata Database data block is as follows.

Datablock Header	Compression Header	Row Data	Reference Array	Datablock Trailer
------------------	--------------------	----------	-----------------	-------------------

1094-001A

The size of the data block trailer is 2 bytes for small block-sized systems and 4 bytes for large block-sized systems. The size of the compression header is the same for small-block-sized systems and large-block-sized systems. The size of each element in the reference array is 2 bytes for all systems.

You should take this information into account when you undertake the various table sizing operations performed during capacity planning. This information can also be important if you need to determine the exact amount of compression you can achieve by compressing multiple column values (see [“Byte Alignment Considerations for Multi-Value Compression” on page 721](#)).

## Row Length Characteristics

The following table describes the boundary characteristics for a Teradata Database row on both packed64 and aligned row format systems. The value for minimum length derives from the mandatory 12 or 16 overhead bytes per row.

Characteristic	Description
Maximum length	64 KB (64,256 bytes)
Minimum length	<ul style="list-style-type: none"> <li>• 12 bytes for nonpartitioned primary index tables</li> <li>• 16 bytes for PPI tables</li> </ul>

## Row Components

The following table describes the individual components of a Teradata Database base table row for both packed64 systems and aligned row systems:

Component	Number of Bytes	Description
Reference array pointer	2	The reference array pointer is not part of the physical row. It is a 2-byte entry maintained at the bottom of each data block that points to the first byte of each row in the block.
<b>Row Header</b>		
Row length	2	Specifies the exact length of the row in bytes.
Nonpartitioned Primary Index RowID	8	<p>Contains two physical fields and one virtual field:</p> <ul style="list-style-type: none"> <li>Partition number:           <ul style="list-style-type: none"> <li>Implicitly assumed to have a value of 0 (as indicated by 2 bits in the flag byte).</li> <li>Consumes 0 bytes from the row.</li> </ul> </li> <li>Row hash value</li> <li>Uniqueness value</li> </ul>
Row hash	4	<p>Contains the hashing algorithm-generated row hash value for the row. The value is 4 bytes wide irrespective of the number of hash buckets the system has (see “<a href="#">Teradata Database Hashing Algorithm</a>” on page 225).</p>
Uniqueness value	4	Contains the system-generated uniqueness value for the row.
Internal partition number	0	<p>This field is logical, not physical. It is not stored as a physical value because its value is known to the system always to be 0.</p>
PPI RowID	10 or 16	<p>Contains these physical fields:</p> <ul style="list-style-type: none"> <li>Partition number:           <ul style="list-style-type: none"> <li>Contains a value between 1 and 65,535 or big number that defines the internal partition number for the row.</li> <li>Consumes 2 or 8 bytes from the row.</li> </ul> </li> <li>Row hash value.</li> <li>Uniqueness value.</li> </ul> <p>Note that the partition number field is not contiguous with the row hash and uniqueness value fields; instead, it immediately follows the first byte of the presence bits array.</p>
Row hash	4	<p>Contains the hashing algorithm-generated row hash value for the row. The value is 4 bytes wide irrespective of the number of hash buckets the system has (see “<a href="#">Teradata Database Hashing Algorithm</a>” on page 225).</p>
Uniqueness value	4	Contains the system-generated uniqueness value for the row.

Component	Number of Bytes	Description
Internal partition number	2 or 8	<p>Contains the partition number for a row in a PPI table.</p> <ul style="list-style-type: none"> <li>If the maximum partition is <math>\leq 65,535</math>, partition number is 2 bytes.</li> <li>If the maximum partition is <math>&gt; 65,535</math>, partition number is 8 bytes.</li> <li>If the row does not belong to a PPI table, then this field is logical only, and its value is 0.</li> </ul>
Flag byte	1	<p>Defines whether the 4-byte Partition number field is used or not.</p> <ul style="list-style-type: none"> <li>If the flag is set, the row is from a PPI table and the row header uses the 4-byte partition number field that follows the first presence octet to store its partition number.</li> <li>If the flag is not set, the row is not from a PPI table and the system assumes that the partition number for the rowID is 0.</li> </ul>
<b>Presence Bits Array</b>		
First byte of the presence bit array  See “ <a href="#">Presence Bits</a> ” on <a href="#">page 780</a> for more information.	1  If the table has a PPI, there is an additional 2 or 8 byte overhead in the presence bits array.	Defines column nullability and compressibility.
Additional bytes of presence bit arrays	8 bytes per presence bit array.	<p>Provides additional space to define nullability and compressibility for columns if the first presence bit array is too small to contain all the information for the table.</p> <p>For more information, see “<a href="#">Presence Bits</a>” on <a href="#">page 780</a>.</p>
<b>VARCHAR Offset Array</b>		
Offset array pad bytes  The array pad bytes are only used for aligned row format systems.	Varies	<p>Aligns the offset array at a 2-byte boundary.</p> <p>This component exists for rows on aligned row systems <i>only</i>.</p>
Column offsets	Varies  2 bytes per variable length column.	<p>Only present when variable length columns are defined for a table.</p> <p>Indicates the intrarow location of the column it references.</p>
<b>Fixed Length Columns</b>		
Fixed length column pad bytes <sup>b</sup>	Varies	<p>Aligns the fixed length columns on an 8-byte boundary.</p> <p>This component exists for rows on aligned row systems <i>only</i>.</p>
Fixed length columns	Varies	Contains all non-compressible fixed length columns.

Component	Number of Bytes	Description
<b>Compress Length Columns</b>		
Compressible column pad bytes <sup>b</sup>	Varies	Aligns the value compressible columns on an 8-byte boundary.  This component exists for rows on aligned row systems <i>only</i> .
Compressible columns	Varies	Contains compressible columns with data belonging to one of four categories: <ul style="list-style-type: none"> <li>Not compressed using multi-value compression.</li> <li>Not compressed using algorithmic compression.</li> <li>Compressed using algorithmic compression.</li> <li>Compressible data that is not null.</li> </ul>
<b>Variable-Length (VARCHAR and VARBYTE) Columns</b>		
Variable length column pad bytes <sup>b</sup>	Varies	Aligns the variable-length columns on an 8-byte boundary.  This component exists for rows on aligned row format systems <i>only</i> .
VARCHAR and VARBYTE columns	Varies	Only present when variable-length columns are defined for a table.  Contains all variable length character columns. The column offsets field points to the starting location for each variable length column in the row.
<b>Row Trailer</b>		
Trailing pad bytes	Varies	Aligns the row on an even-byte boundary for packed64 format systems or on an 8-byte boundary for aligned row systems.

## Hash and Join Index Row Structures

The row structure for non-row compressed hash and join indexes is the same as that for a standard data table (with the exception that you cannot compress individual column values for hash indexes (which means that the presence bit array of a hash index indicates only the nullability of a column, not its compressibility), while the row structure for row compressed hash and join indexes is more like that of a secondary index (see “[Sizing a Unique Secondary Index Subtable](#)” on page 862 and “[Sizing a Non-Unique Secondary Index Subtable](#)” on page 864)).

Other than that exception, the only difference is the specific columns included in the base table data columns of the row. For all hash indexes, Teradata Database automatically adds the primary index columns of the base table (if not explicitly specified in the hash index definition) and its row uniqueness value. The system does not automatically add this information to join indexes.

Hash indexes are typically row compressed by default (see “[Compression of Hash Index Rows](#)” on page 607 for details), while you must explicitly specify row compression in a CREATE JOIN INDEX request to compress rows in the join index it creates.

Note that you cannot row compress either of the following hash and join index types:

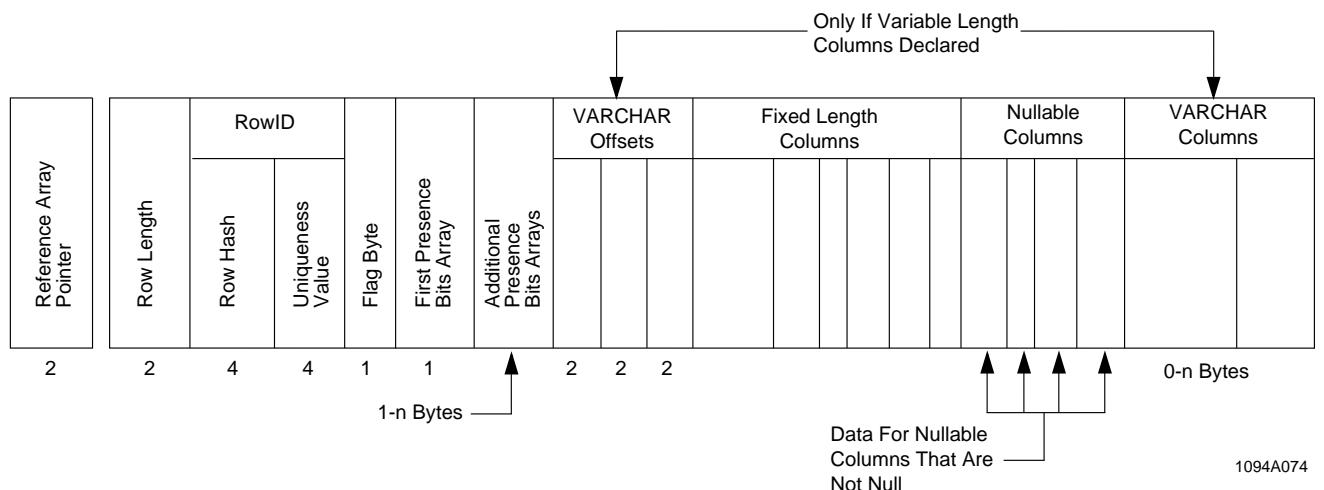
- A hash index defined with a PPI.  
You cannot define *any* hash index with a partitioned primary index.
  - A join index defined with a PPI or with column partitioning.

## Hash and Join Index Row Structure for Packed64 Format Systems

Hash and join index rows on packed64 format systems do not have to align on 8-byte boundaries. Because of this, their row structure is simpler than that of equivalent base table rows on aligned row format systems (see “[Row Structure for Aligned Row Format Systems](#)” on page 755).

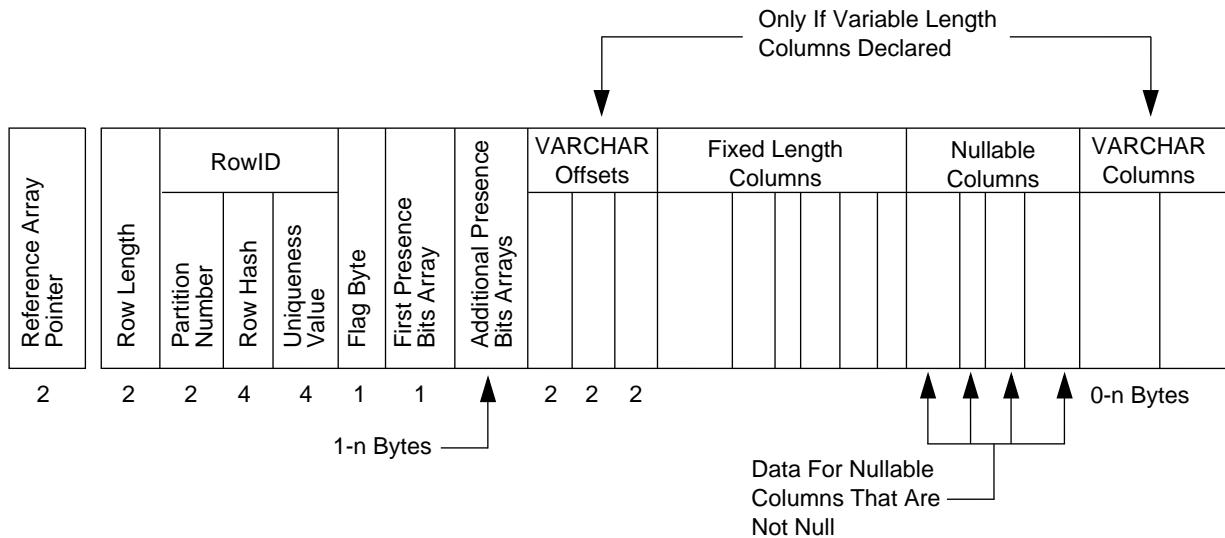
## Packed64 Row Structure for an Uncompressed Hash or Join Index With an Nonpartitioned Primary Index

The following graphic illustrates the basic structure of a non-row compressed Teradata Database hash or join index row from an index defined with an nonpartitioned primary index.



## Packed64 Row Structure for an Uncompressed Join Index With a Partitioned Primary Index

The following graphic illustrates the basic structure of an non-row compressed Teradata Database join index row(hash indexes cannot have a partitioned primary index) from an index defined with a partitioned primary index.



1094A075

The difference between this and the format of a non-partitioned primary index row is the presence of a 2-byte partition number field at the beginning of the RowID. It is this field that generates the need for a BYTE(10) data type specification for a RowID. For nonpartitioned primary index indexes, the partition number is assumed by default to be 0, and is not stored.

### Packed64 Row Structure for a Row Compressed Hash or Join Index With an Nonpartitioned Primary Index

The region labelled as *Index Value* in the graphic of the row format is the *column\_1* value set for the row. It is an abbreviation for the various data type orderings and offsets shown in detail in the row format diagrams for non-row compressed hash and join index rows.

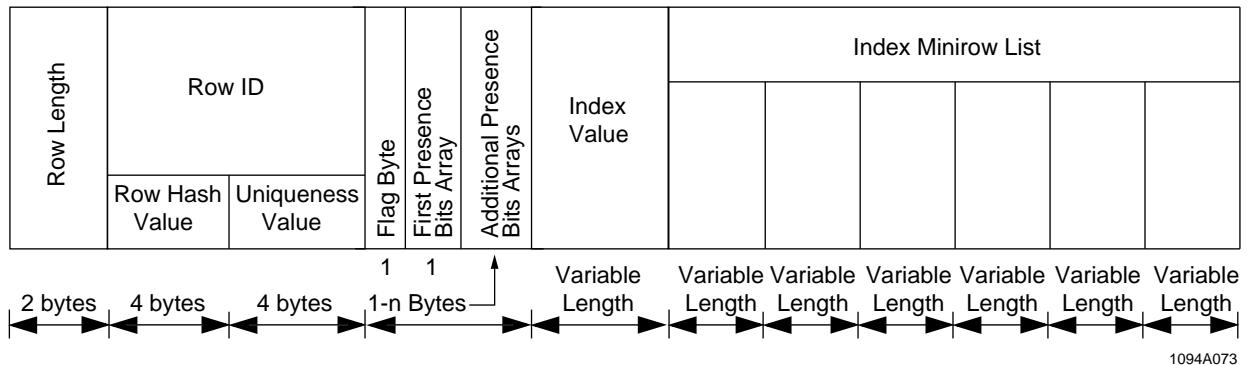
For example, consider the following example CREATE JOIN INDEX text:

```
CREATE JOIN INDEX test AS
    SELECT (a,b), (x,y)
    FROM table1, table2;
```

The system packs columns *a* and *b*, labelled as the *column\_1\_name* list in the CREATE JOIN INDEX syntax diagram, in the last subfield of Field1 of a row compressed join index row the same way that index keys (an *index key* is the set of values stored as “the index” in a secondary index) are packed in the Field1 of a secondary index row (columns *a* and *b* are stored in the area labelled as *Index Value* in the row format graphic).

The system also stores the uncompressed index values (each instance of those *(x,y)* values that has the same *(a,b)* values as a minirow in Field2) in the area labelled as *Index Minirow List* in the row format graphic. These columns are labelled as the *column\_2\_name* list in the CREATE JOIN INDEX syntax diagram.

Each minirow consists of a BYTE length field, an optional presence bits array field, and a *column\_2\_name* value set (see “[Packed64 Row Structure for an Uncompressed Hash or Join Index With an Nonpartitioned Primary Index](#)” on page 775 or “[Packed64 Row Structure for an Uncompressed Join Index With a Partitioned Primary Index](#)” on page 775).



## Packed64 Row Structure for a Hash or Compressed Join Index With a Partitioned Primary Index

Teradata Database does not support PPIs for hash indexes, compressed join indexes, or column-partitioned join indexes.

## Hash and Join Index Row Structure for Aligned Row Format Systems

The hash and join index row structure diagrams for aligned row format systems differ from those of packed64 format systems by having as many as five additional pad byte fields (depending on the data types of the columns defined for *Index Value*) to ensure row alignment on an 8-byte boundary. The following table lists each of these pad fields and explains their purpose:

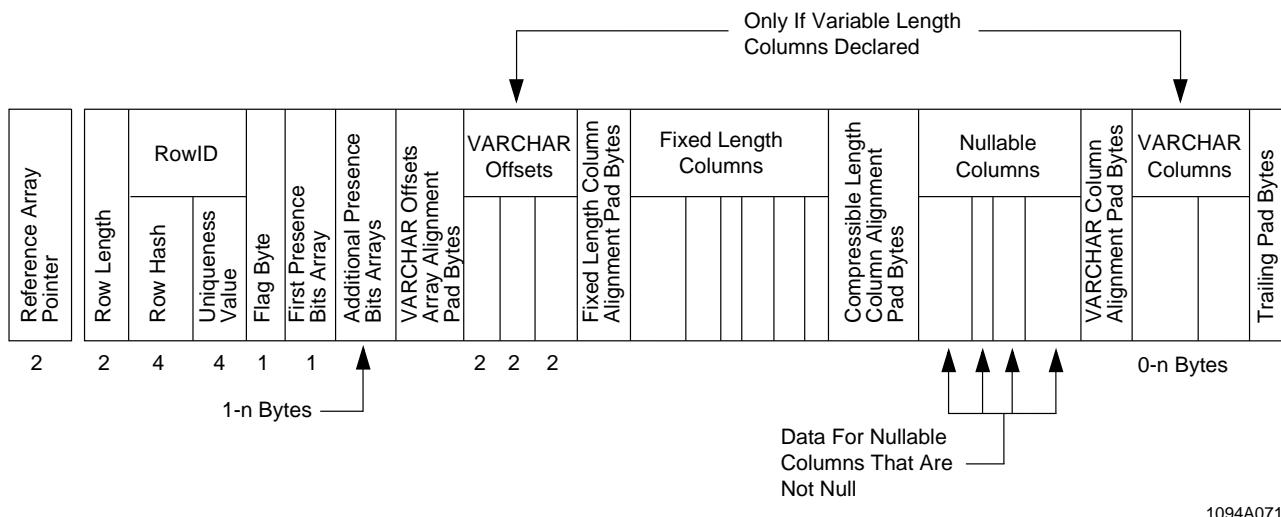
Pad Byte Field Name	Purpose
VARCHAR Offsets Array Alignment Pad Bytes	Aligns VARCHAR offsets array at a 2-byte boundary.
Fixed Length Column Alignment Pad Bytes	Aligns fixed length columns.
Nullable Length Column Alignment Pad Bytes	Aligns nullable length columns.
VARCHAR Column Alignment Pad Bytes	Aligns variable length columns.
Trailing Pad Bytes	Aligns entire row on an 8-byte boundary.

The index columns in a row compressed hash or join index row are stored in packed64 format, adjusted for aligned row format systems by a field of alignment bytes trailing *field1* and another field of alignment bytes trailing *field2* (if necessary to make the entire row align on a modulo(8) boundary). When you row compress a hash or join index, the *column\_2\_name* value set for each row in the index is stored as a minirow. Each minirow consists of a byte length field, an optional presence bits array field, and the *column\_2\_name* value set for the minirow.

Teradata Database does not support PPIs for hash indexes, row-compressed join indexes, or column-partitioned join indexes.

## Aligned Row Format Structure for an Uncompressed Hash or Join Index With an Nonpartitioned Primary Index

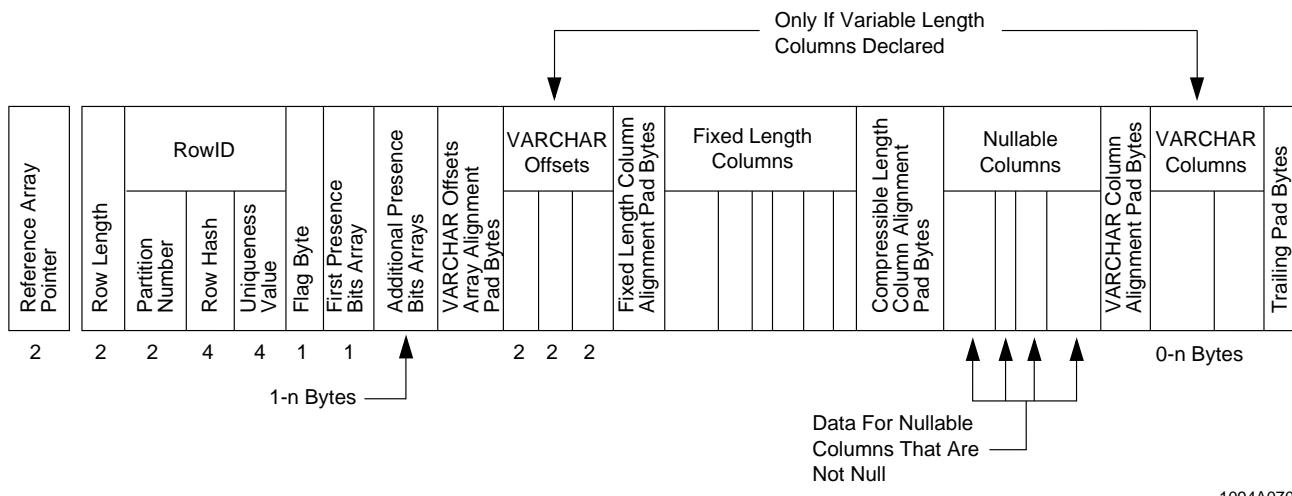
The following graphic illustrates the basic structure of a Teradata Database row from a non-row compressed hash or join index with an nonpartitioned, or standard, primary index:



## Aligned Row Format Structure for an Uncompressed Join Index With a Partitioned Primary Index and 65,535 or Fewer Combined Partitions

Teradata Database does not support PPIs for hash indexes.

The following graphic illustrates the basic structure of a Teradata Database row from a non-row compressed join index (hash indexes cannot have partitioned primary indexes) with a partitioned primary index:



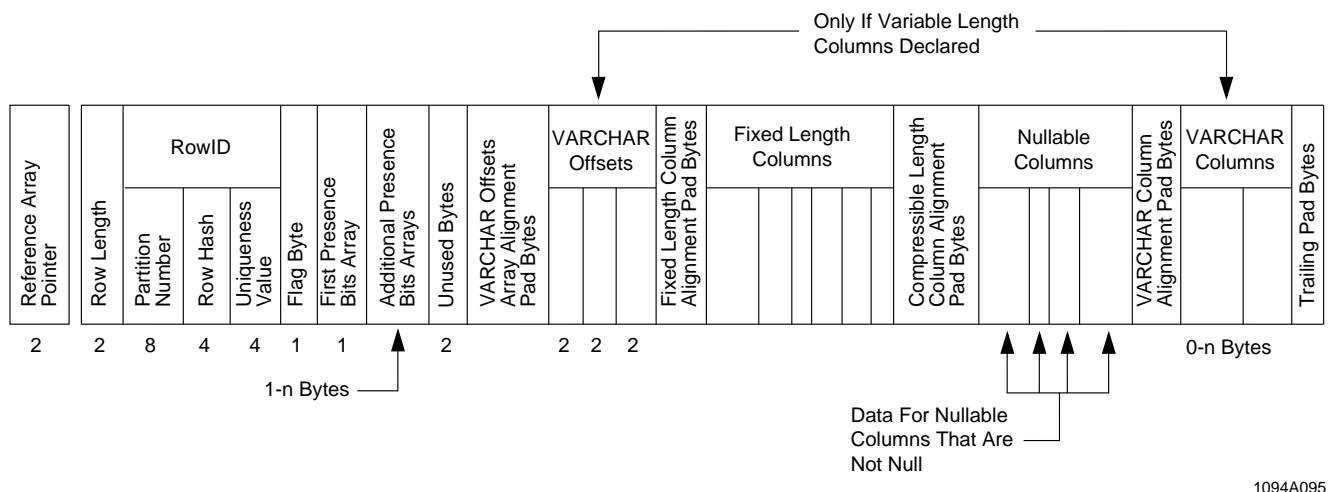
The difference between this and the format of a non-row compressed non-partitioned primary index row is the presence of a 2-byte partition number field at the beginning of the RowID (see “[PARTITION Columns](#)” on page 801). It is this field that generates the need for a BYTE(10) data type specification for a RowID. For nonpartitioned primary index tables, the

partition number is assumed to be 0, so the rowID of an nonpartitioned primary index table is also logically BYTE(10) (see “[ROWID Columns](#)” on page 800).

### **Aligned Row Format Structure for an Uncompressed Join Index With a Partitioned Primary Index and More Than 65,535 Combined Partitions**

Teradata Database does not support PPIs for hash indexes.

The following graphic illustrates the basic structure of a Teradata Database row from a non-row compressed join index (hash indexes cannot have partitioned primary indexes) with a partitioned primary index:



The difference between this and the format of a non-row compressed non-partitioned primary index row is the presence of an 8-byte partition number field at the beginning of the RowID (see “[PARTITION Columns](#)” on page 801). It is this field that generates the need for a BYTE(16) data type specification for a RowID.

### **Aligned Row Format Join Index Row Layout for a Compressed Hash or Join Index With an Nonpartitioned Primary Index**

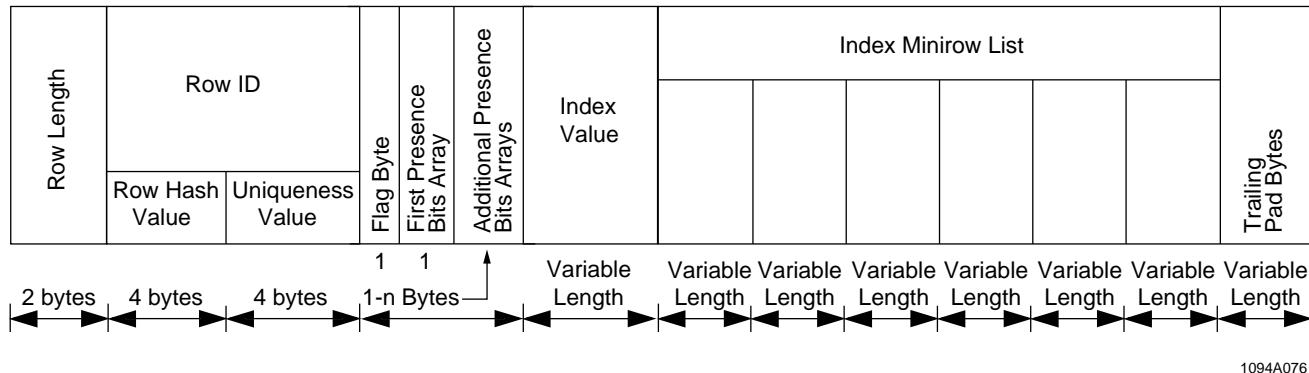
Consider the following example CREATE JOIN INDEX text:

```
CREATE JOIN INDEX test AS
  SELECT (a,b), (x,y)
  FROM table1, table2;
```

Teradata Database packs columns *a* and *b*, labelled as the *column\_1\_name* list in the CREATE JOIN INDEX syntax diagram, in the last subfield of Field1 of a row compressed join index row the same way that index keys (an *index key* is the set of values stored as “the index” in a secondary index) are packed in the Field1 of a secondary index row (columns *a* and *b* are stored in the area labelled as *Index Value* in the row format graphic).

The system also stores the non-row compressed index values (each instance of those *(x,y)* values that has the same *(a,b)* values as a minirow in Field2) in the area labelled as *Index Minirow List* in the row format graphic. These columns are labelled as the *column\_2\_name* list in the CREATE JOIN INDEX syntax diagram.

The region labelled as *Index Value* in the following row format graphic is the *column\_1\_name* value set for the row and should be interpreted as containing the requisite pad bytes depicted in “[Aligned Row Format Structure for an Uncompressed Hash or Join Index With an Nonpartitioned Primary Index](#)” on page 778). It is an abbreviation for the various data type orderings and offsets shown in detail in the row format diagrams for non-row compressed hash and join index rows. The number of trailing pad bytes required by a given row is the number of bytes it takes to make the row end on a modulo(8) boundary.



When you row compress a hash or join index, the *column\_2\_name* value set for each row in the index is stored as a minirow. Each minirow consists of a BYTE length field, an optional presence bits array field, and a *column\_2\_name* value set (see “[Aligned Row Format Structure for an Uncompressed Hash or Join Index With an Nonpartitioned Primary Index](#)” on page 778 or “[Packed64 Row Structure for a Row Compressed Hash or Join Index With an Nonpartitioned Primary Index](#)” on page 776).

### **Aligned Row Format Join Index Row Layout for Compressed Hash and Join Indexes With a Partitioned Primary Index**

Teradata Database does not support PPIs for hash indexes, row-compressed join indexes, or column-partitioned join indexes.

## **Secondary Index Subtable Row Structures**

“[USI Subtable Row Structure](#)” on page 463 documents the row structure for unique secondary index subtables.

“[NUSI Row Structure](#)” on page 468 documents the row structures for non-unique secondary index subtables.

## **Presence Bits**

Presence bits indicate the status of each column with respect to its nullability, multi-value compressibility, and autocompressibility. Compression presence bits are added to the row header of each row to specify how multi-value compression is used for that row.

Each row has at least one octet of presence bits and can have more, depending on the degree of the table and the cumulative number of values compressed. The difference between bytes and octets is conceptual. The 8 bits of a byte define an atomic unit, while each individual bit of an octet is an atomic flag in a bit array. For the purpose of storage capacity analysis, both are treated as bytes. All rows for a given table have the same presence bits defined because nullability and multi-value compressibility are table attributes.

The first bit of the first presence bits octet is always set to 1, so the first octet defines the nullability and multi-value compressibility for no more than seven columns. When necessary, additional presence bit octets are added to the row header. Although only 7 of the bits in the first octet are available for use as presence bits, all 8 bits of successive octets are available. Eight bits are used because 255 compressed values plus null require  $2^8$ , or 256 bit combinations, to be represented.

For a column-partitioned table or join index there is one set of presence bits for each column partition value in the container. If a presence bit is 1, a value is present. If a presence bit is 0, no value is present.

A compression-enabled column for a table has a maximum of 256 presence bits set, depending on how many values are compressed using multi-value compression.

The meaning of the number of presence bits set per column for compression of a single value is provided in the following table.

This type of presence bit field...	Has this many presence bits ...	For this type of column ...
Nullability	0	Non-nullable
	1	Nullable
Compressibility	0	Not compressed or compressed on nulls only.
	1	Compressed on a value

For multi-value compression, the following equation calculates the number of compression presence bits required to define compression for a column in its table header.

```
compression_presence_bits = ceiling(log2(1 + number_of_distinct_values))
```

where:

This term ...	Specifies ...
<i>ceiling(x)</i>	<p>a function that returns the smallest integer greater than or equal to <math>x</math>. This function is common in the mathematics library of most programming languages. See <i>SQL Functions, Operators, Expressions, and Predicates</i> for documentation of the Teradata SQL implementation of the ceiling function, CEIL.</p> <p>The ceiling function rounds the result of the expression up to the nearest integer, which is the exact number of presence bits required to account for both single-valued and multi-value compression information for the given table.</p>
<i>number_of_distinct_values</i>	the number of distinct values to be compressed for the column using multi-value compression.

The total number of presence bits for a given row is the sum of the nullability presence bits and the compressibility presence bits.

$$\text{presence\_bits} = \sum_{\text{col}=1} \text{nullability\_presence\_bits} + \text{compressability\_presence\_bits}$$

The following table expresses the same information in a slightly different way.

Compressible		Nullable	
Bit Value	Meaning	Bit Value	Meaning
0	The column is multi-value compressed.	0	The column is null.
1	A non-compressed column value is present.	1	The column is not null.

An algorithmically compressed column adds an extra bit to indicate whether the column is algorithmically compressed or not, as follows:

Bit Value	Meaning
0	The column is not algorithmically compressed.
1	The column is algorithmically compressed.
1 to 8 bits for each multi-value compressible column (8 bits because 255 compressed values plus null require 28, or 256 bit combinations, to be represented).	1 bit for each nullable column.

The following table provides a comprehensive mapping of the presence bits and their various combinations for the multi-value compression case:

WHEN the presence bits for a column have these values ...		THEN the column ...	AND ...
COMPRESS	NULL		
no bit	no bit	is not compressible	is not nullable.
0	no bit	is compressed	is not nullable.
1	no bit	contains uncompressed column values	is not nullable.
no bit	0	is not compressible	is null.
no bit	1	is not compressible	is not null.
0	0	is compressed	is null.
1	1	is not compressed	is not null.
1	0	is not compressed	is null.
0	1	is not compressed.	is null.

Mappings of COMPRESS bit values for multi-valued compression generalize from this specific case as illustrated by the following table.

IF the presence bit is ...	THEN the data is ...
1	not compressed. The corresponding compress bits are all 0.
0	compressed. <ul style="list-style-type: none"> <li>If the corresponding compress bits are all 0, then the compress value is null.</li> <li>If the corresponding compress bits are not all 0, then the compress value is an index to the compress multi-value array in the table header.</li> </ul>

The following table presents a set of examples that clarifies the correspondence between presence bits and data attribute specifications.

FOR this column definition ...	The presence bits are ...	For these characters ...
col_1 CHAR(1) NOT NULL	none	A
col_1 CHAR(1) NOT NULL COMPRESS ('A')	0	A
	1	B
col_1 CHAR(1) COMPRESS	0	null
	1	A

FOR this column definition ...	The presence bits are ...	For these characters ...
col_1 CHAR(1) COMPRESS ('A')	00	null
	01	A
	10	B
	10	C
col_1 CHAR(1) COMPRESS ('A', 'B', 'C', 'D')	0000	null
	0001	A
	0010	B
	0011	C
	0100	D
	1000	E
col_1 CHAR(1) NOT NULL COMPRESS ('A', 'B', 'C', 'D')	001	A
	010	B
	011	C
	100	D
	000	E

## Number of Presence Bits Required to Represent Compressed Values

Because compression is signaled by populating octets of presence bit fields in the row header, there is a tradeoff between the capacity required to add presence octets and the savings realized from multi-value compression at the base table row level: the size of the bit field required to express multi-value compression increases non-linearly as a function of the number of values to be compressed. With respect to the savings gained by multi-value compression, this tradeoff is almost always worth doing.

The following table indicates the relationship between the number of values (and null, which is always compressed by default) specified for compression per column and the number of compress bits required to indicate that number of values.

Nullable?	Number of Column Values Compressed	Number of Compress Bits in the Row Header Bit Field
Yes	null	1 Shared with the presence bit.
No	1	1
No	2 - 3	2
No	4 - 7	3

Nullable?	Number of Column Values Compressed	Number of Compress Bits in the Row Header Bit Field
No	8 - 15	4
No	16 - 31	5
No	32 - 63	6
No	64 - 127	7
No	128 - 255	8
Yes	1 + null	1
Yes	2 - 3 + null	2
Yes	4 - 7 + null	3
Yes	8 - 15 + null	4
Yes	16 - 31 + null	5
Yes	32 - 63 + null	6
Yes	64 - 127 + null	7
Yes	128 - 255 + null	8

The compress bits value represents an index into the compress multi-value array for that column in the table header. This is illustrated by the following graphic showing that the presence bit pattern 10 points to the compressed character string “Los Angeles” in the table header. Note that because only 3 values are compressed for this column, 2 compression bits are required to uniquely identify them.

The table definition DDL for this illustration is as follows:

```
CREATE TABLE MVCompress (
    StreetAddress VARCHAR(40),
    City          CHARACTER(20) COMPRESS ('New York',
                                         'Los Angeles', 'Chicago') NOT NULL,
    StateCode     CHARACTER(2));
```

Table Header						
Field: StreetAddress VARCHAR(40)			Field: City CHAR(20) COMPRESS NOT NULL			Field: StateCode CHAR(2)
			01 'Chicago', 10 'Los Angeles', 11 'New York'			
			130 Sutter St. San Francisco CA			
			333 Wacker Drive IL			
			5 Times Square NY			
			900 North Michigan Avenue IL			
			135 East 57th NY			
			1525 Howe St. Racine WI			
			304 S. Broadway CA			

1094A019

As you can see, all the possible combinations of bit patterns are used to identify and locate the values. For this particular example, the bit patterns and their corresponding values are those indicated in the following table:

FOR this pattern of presence bits ...	The value for City is ...	AND stored in ...
00	not compressed.	the row.
01	'Chicago' and is compressed	the table header.
10	'Los Angeles' and is compressed	the table header.
11	'New York' and is compressed	the table header.

From the perspective of the row header, the best policy is to compress the highest number of values in each octet bit array, because doing so does not add any more space in the row header (nor does null compression, which is handled by presence bits in the row header for each row). In other words, the best practice is to optimize multi-valued compression for 1, 3, 7, 15, 31, 63, 127, or 255 values. You should evaluate all these possibilities to determine which yields the best compression for a particular column.

You must also take into account the number of bytes compression transfers to the table header, because those bytes count against the maximum row length of 64 KB.

The compressed values are stored adjacent to one another in ascending value order and descending order of field size following the end of Field5 in the table header. See “[Table Header Components](#)” on page 787.

# Table Headers

Each Teradata Database table has an associated subtable called a table header. One copy of each table header is stored on each AMP in the system, having one row per table per AMP. Table headers are composed of a row header, one fixed-length field, and six variable-length fields (see “[Table Header Components](#)” on page 787).

Teradata Database uses table headers internally to maintain various information about each table. Table headers are described here primarily because they are used to store value-compressed column values (see “[Compression Types Supported by Teradata Database](#)” on page 695), and it is difficult to explain how compression works without having at least a superficial understanding of this file structure.

The size limit for a table header is 1 MB, so it is not possible to create a table defined with all possible features (2,048 columns, 32 indexes with 64 columns each (a composite NUSI that specifies an ORDER BY clause counts as 2 indexes in this calculation), 32 BLOB, CLOB, or XML columns, 255 distinct compressed values in each compressible column, and so on) without exceeding the defined byte limit. The maxima stated in [Appendix B: “Teradata System Limits,”](#) are for the individual table parameters only.

## Table Header Components

The following topics describe the individual fields of a Teradata Database table header as it is stored on disk. The maximum length for each field is the value for a thin table header.

### Row Header

See “[Base Table Row Format](#)” on page 740 for a description of Teradata Database row headers.

### Field 1

Fixed length. Includes:

- Row 1 header.
- Offset array for the variable length columns in the table header, Used to locate those columns.
- Table header row format version.
- Internal ID of the database to which this table belongs.
- Internal ID of the database to which space for this table is charged.
- Table creation timestamp.
- Last table update timestamp.
- Last table archive timestamp.
- Primary index flag.
- Table structure version. Updated each time the table description is modified
- Table structure version for host utilities. Incremented each time a structural change is made that makes a database dump obsolete.
- Number of backup tables associated with this table.

- Internal ID of the permanent journal table.
- Table kind. Describes whether the table is permanent, temporary, volatile, or a join index. For temporary tables, volatile tables, and join indexes, also describes preserve-on-commit and transient journaling characteristics.
- Journal type:
  - After
  - Audit
  - Both
  - Before
  - None
- Protection type:
  - Fallback
  - Log
  - None
- User journal flag. Describes whether the table is a user-defined journal or not.
- Hash flag,. Describes whether table is hashed or not.
- Dropped flag. Set TRUE from the time the AMP receives a Drop Table step until the table is dropped in the End Transaction step. Otherwise set FALSE.
- DDL change flag.Prevents attempts to update the table after its DDL has changed. Set TRUE when table DDL is modified.  
Host utility table dump sets the flag FALSE.
- Byte count of the number of USIs defined on the table.
- Host character set at the time the table was created.
- Number of parent tables referenced by this table.
- Number of child tables referencing this table.
- Merge block ratio for the table.
- Merge block ratio validity. Indicates whether the specified merge block ratio for the table is valid or unspecified.
- Data block size for the table in bytes.
- Data block size validity. Indicates whether the specified data block size for the table is valid or unspecified.
- Percent free space for the table.
- Percent free space validity. Indicates whether the specified percent free space for the table is valid or unspecified.
- Disk I/O integrity checksum. Used to verify the integrity of user tables, hash indexes, join indexes, and secondary indexes.
- Message class of primary step.
- Message kind of primary step.
- Message class of secondary step.

- Message kind of secondary step.
- Host ID.
- Session number.
- Request number.
- Transaction number.
- Table ID of base temporary table. Used only for materialized temporary tables.
- Internal ID for primary key index.
- Restart flag. Tracks restart and non-restart cases of restore jobs during build phase.
- Row format, which specifies whether current environment is packed64 format or aligned row format.
- Dummy space.
- List of index descriptors for the table in Index ID order, one per primary and secondary index defined on the table.
- Row 1 length. Duplicate specification of the length of row 1.

### Field 2

Variable length.

Maximum length: 47,784 bytes.

Contains the primary index descriptor and all secondary index descriptors for the table.

### Field 3

Not used.

### Field 4

Variable length.

Maximum length: 423 bytes.

Contains MultiLoad, FastLoad, Archive/Recovery, and table rebuild information.

Field 4 is always present for permanent journal tables, but is context-dependent for non-journal tables.

### Field 5

Variable length.

Maximum length: 56,350 bytes.

Contains the following table column descriptors:

- Internal ID of the first column in the table.
- Number of varying length columns in the table.
- Number of presence bits in each row. The upper limit for the number of presence and compress bits per row is 89,991.

- Flag to indicate whether the table header is thin or fat. Used to determine whether the compressed value offsets list contains actual addresses or must be left-shifted to extract actual addresses.
- Compression flag. Specifies whether the table has compressible columns or not.
- Offset in the row to the presence bit array. Presence octet locations are determined by dividing the presence bit position by 8.
- Offset in the row to the first byte past the presence bit array.
- Number of columns in the table.
- Index into the field descriptor array to the first compressible column. If no columns are compressed, then the value points to the first varying length column in the row.
- Offset from the beginning of the row to the first optional (varying or compressible) data.
- Index of the field descriptor for the first physical column in the row.
- Field 5 type:
  - Table descriptor with row hash and unique rowID.
  - Index descriptor with row hash and unique rowID in the index.
  - Table descriptor with a PPI rowID.
  - Index descriptor with a PPI rowID in the index.
- Row format. Specifies whether current environment is packed64 format or aligned row format.
- Duplicate rows flag:
  - Dictionary and non-ANSI/ISO tables.
  - ANSI/ISO tables without unique indexes.
  - ANSI/ISO tables with unique indexes.
- Offset to system code to build rows and to calculate PPI internal partition numbers.
- Field descriptors array.
- Compressed values and UDT contexts. Compressed values and a UDT context are stored just beyond the Field 5 descriptors and PPI-related system code. The stored values are sorted first in ascending order of their binary values, then in descending order of their field size, and aligned on 2-byte boundaries. The stored UDT context is its autogenerated UDF constructor context. The area contains a 76-byte UDT context for every column typed with a UDT. This places a practical upper limit of approximately 1,600 UDT columns per table.
- System code for building rows, including code to calculate partition numbers for the rows of row-partitioned tables. The row-partition-related system code is stored after the Field 5 descriptors, and just before the compressed values, if any have been defined.
- UDT name stored as a variable length string of up to 128 Unicode characters.

## Field 6

Variable length. Usually null.

Maximum length: 118 bytes.

Contains restartable sort and ALTER TABLE information.

### Field 7

Variable length.

Maximum length: 94,592 bytes.

Contains:

- Up to 128 reference index descriptors: 64 from a parent table to child tables and 64 from child tables to a parent table. See “[Sizing a Reference Index Subtable](#)” on page 866 for a description of reference indexes.
- List of the names of unresolved child tables from referential integrity constraint specifications as a variable length string of up to 128 Unicode characters.

### Field 8

Variable length.

Maximum length: 520 bytes.

Contains BLOB, CLOB, and XML descriptors.

### Field 9

Variable length.

Maximum length: 518 bytes.

Contains:

- Length of, and offset to, the database name.
- Length of, and offset to, the table name.
- Database name (up to 128 Unicode characters).
- Table name (up to 128 Unicode characters).

The database and table names are in the format *databasename.tablename*.

## Column Sizing Guidelines

You should read the material on multi-value compression carefully, beginning with “[Compression Types Supported by Teradata Database](#)” on page 695 and ending with “[Calculating the Efficiency of Multi-Value Compression](#)” on page 725, to understand how to implement optimally most of the guidelines described in this topic.

See “[System-Derived and System-Generated Columns](#)” on page 800 for additional important column sizing information.

## Multi-Value Compression Guideline

Compress column values only when compression does not have the net effect of reducing storage capacity. For information about how to make this determination, see “[Calculating the Efficiency of Multi-Value Compression](#)” on page 725.

Multi-value compression produces shorter rows, which in turn generally produce more high-performing applications.

Note that column multi-value compression, once defined for base table columns, is inherited by join indexes defined on those value-compressed columns as well retained as a property of spool files containing data from the table as well, so compression is not just a property of data stored on disk.

## Miscellaneous Column Sizing Considerations and Guidelines

- Adding a column expands the capacity required for every row in a table if the column is not value compressible.
- Dropping a column reduces every row in a table when there was uncompressed data in the dropped column.
- Specify VARCHAR for any character column for which the space savings offsets the two bytes per row overhead and if compression on a fixed length CHARACTER column would save less than 5% additional capacity.

## Sizing Structured UDT Columns

For a distinct UDT, there is always a 1:1 correspondence between the type and its underlying predefined data type. As a result, to determine the size of a distinct UDT column, just use the size of the underlying predefined data type. Because of alignment differences for packed64 and aligned row systems, be sure to consult the size differences for the various data types on the two system types before making your row size estimates (see “[Data Type Size Differences For Packed64 and Aligned Row Format Architectures](#)” on page 823).

Unlike predefined data types, the size of each structured UDT you create is different, and you must calculate that size to account for the amount of storage a column with that type requires.

**Note:** You cannot use either multi-value compression or algorithmic compression for structured UDT column data.

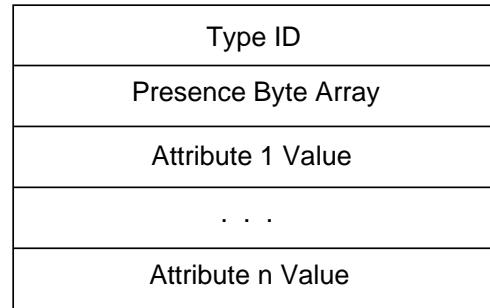
This topic first describes how a structured UDT is stored in a row, then describes how to calculate the number of bytes required to store a given structured UDT in persistent form.

See “[Geospatial Data Types](#)” on page 845 for information about sizing the system-provided geospatial UDTs.

The dynamic UDT data type is a structured UDT with a fixed data type name of VARIANT\_TYPE, so you can calculate the sizes of VARIANT\_TYPE instances in the same way you would any other structured UDT.

## Storage Structure of a Structured UDT Data Type

The following graphic illustrates the basic morphology of a structured UDT as it is stored in the row of a table in the Teradata Database:



1094A053

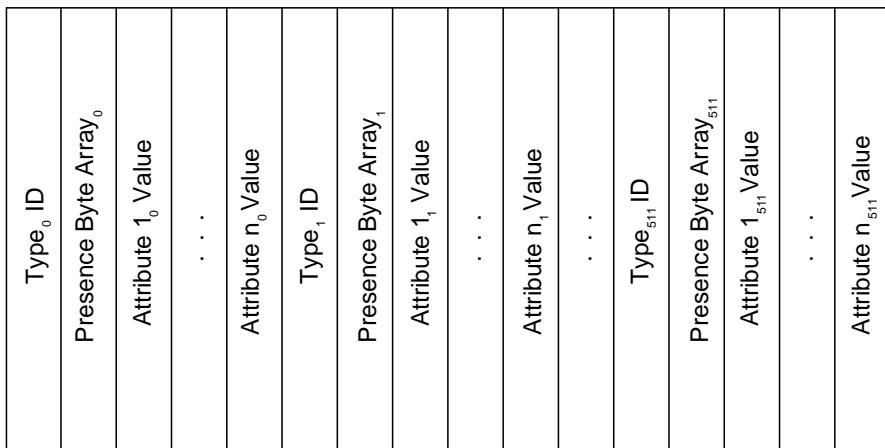
where:

UDT element ...	Specifies ...
Type ID	the TVMID type identifier for the structured UDT. Its data type is INTEGER (4 bytes).
Presence Bits Array	the octet array of presence bits for the UDT at level $m$ of the structured UDT.  This is a variably-sized bit array, rounded to the higher modulo(8) boundary, whose size depends on the number of attributes stored for a particular structured UDT. Its size increases in octet (8-bit byte) increments.
Attribute $n$ Value	attribute number $n$ for the structured UDT.  There is a variable number of attribute value fields. The exact number depends on two factors: <ul style="list-style-type: none"> <li>• The number of attributes a particular structured type has.</li> <li>• Whether or not an attribute is null.</li> </ul> The exact format of each attribute depends on the attribute type and is defined by the designer of the given structured type.

## Storage Structure of a Nested Structured Data Type

A structured UDT can be built from attributes that are themselves structured UDTs. This topic explains the persistent storage format for a column having such a type.

The following graphic illustrates the basic morphology of a structured UDT that contains nested structured UDT attributes as it is stored in the row of a table in Teradata Database:



1094A052

where:

UDT element ...	Specifies ...
Type <sub>m</sub> ID	<p>the TVMID type identifier for the level <math>m</math> structured UDT.</p> <p>The value of <math>m</math> is 0 for the highest level attribute set in a structured UDT, ranging to 511 for the lowest possible level attribute set in a multilevel nested structured UDT.</p> <p>Structured UDT attributes can be nested through multiple levels. The lowest level is numbered 511 because there can be 512 attribute nesting levels in a structured UDT, ranging from 0 - 511.</p>
Presence Bits Array <sub>m</sub>	<p>the octet array of presence bits for the UDT at level <math>m</math> of the structured UDT.</p> <p>This is a variably-sized bit array, rounded to the higher modulo(8) boundary, whose size depends on the number of attributes stored for a particular structured UDT. Its size increases in octet (8-bit byte) increments.</p>
Attribute n <sub>m</sub> Value	<p>attribute number <math>n</math> for the level <math>m</math> structured UDT component.</p> <p>The number of attribute value columns is variable. The exact number depends on two factors:</p> <ul style="list-style-type: none"> <li>• The number of attributes a particular structured type has.</li> <li>• Whether or not an attribute is null.</li> </ul> <p>The exact format of each attribute depends on the attribute type and is defined by the designer of the given structured type.</p>

## Equation For Sizing a Structured UDT

Because the size of a structured UDT value is not an arithmetic sum of the size of its individual attributes, it presents a special problem for capacity planning.

The storage footprint for a structured UDT is composed of the following components in the order given:

- 1 The TVMId Type Identifier for the UDT.
- 2 A variably sized attribute presence bit array with one bit per attribute, rounded up to the nearest 8-bit boundary.
- 3 A serialized list of the non-null attribute value sizes.

### **Equation: Structured UDT Size for Packed64 Format System**

The equation for determining the size of a structured UDT column for a packed64 format system is as follows:

$$UDT\_size = TVMID\_size + \left( 8 \times \left( \frac{\text{number\_of\_attributes}}{8} + 1 \right) \right) + \sum \text{fixed\_size\_non-null\_attributes} \\ + \sum (\text{variable\_size\_non-null\_attributes} + 2) + \sum \text{non-null\_LOB\_attribute\_OIDs}$$

### **Equation: Structured UDT Size for an Aligned Row System**

The equation for determining the size of a structured UDT column for an aligned row system is as follows:

$$size = MOD(8) \left( TVMID\_size + \left( 8 \times \left( \frac{\text{number\_of\_attributes}}{8} + 1 \right) \right) + \sum \text{fixed\_size\_non-null\_attributes} \right. \\ \left. + \sum (\text{variable\_size\_non-null\_attributes} + 2) + \sum \text{non-null\_LOB\_attribute\_OIDs} \right)$$

where:

Equation element ...	Represents the ...
<i>UDT_size</i>	total size in bytes of the structured UDT.
<i>MOD(8)</i>	modulo(8) factor required to align the structured UDT on an 8-byte boundary.
<i>TVMID_size</i>	size in bytes of the <i>DBC.TVMID</i> for structured UDTs. This is fixed at 6 bytes. There is a <i>DBC.TVMID</i> size for each nested level of a structured UDT.

Equation element ...	Represents the ...
<i>number_of_attributes</i>	<p>number of attributes in the structured UDT.</p> <p>The value is rounded up to the next higher modulo(8) boundary to pad the Presence Bits Array to a full 8 bits if necessary.</p> <p>This factor, which accounts for the Presence Bits Array, applies to each nested level of a structured UDT.</p>
<i>fixed_size_non-null_attributes</i>	<p>total size in bytes of all the fixed-size non-null attribute values in the structured UDT.</p> <p>The set of fixed size attributes is composed of the following elements:</p> <ul style="list-style-type: none"> <li>• Fixed-size predefined data types such as INTEGER, DECIMAL, and CHARACTER.</li> <li>• Fixed-size UDT types based on fixed-size predefined data types.</li> </ul> <p>You must also account for nested attributes in this calculation, each level of which carries its own <i>DBC.TVMID_size</i> and Presence Bits Array.</p>
<i>variable_size_non-null_attributes</i>	<p>total size in bytes of all the variable-length non-null attribute values in the structured UDT.</p> <p>Note that the calculation includes a 2-byte length indicator for each variable-length data type attribute in the UDT.</p> <p>For example, suppose you store the following character string in an attribute defined with the VARCHAR(20) predefined data type:</p> <pre data-bbox="670 1036 931 1064">sorry about that</pre> <p>This character string is stored in the following format:</p> <pre data-bbox="670 1100 1318 1178">2-byte length indicator, attribute value length</pre> <p>Given this information, you can see that the example string ‘sorry about that’ would be stored as follows:</p> <pre data-bbox="670 1269 1078 1296">{ 16, 'sorry about that' }</pre> <p>where:</p> <ul style="list-style-type: none"> <li>• 16 is the length of the stored value for the variable length string in bytes: 14 lowercase Latin characters plus 2 pad characters.</li> <li>• <i>sorry about that</i> is the value of that string.</li> </ul> <p>The set of variable-size attributes is composed of the following elements:</p> <ul style="list-style-type: none"> <li>• Variably-sized predefined data types such as VARCHAR and VARBYTE.</li> <li>• Variably-sized UDT types based on variably-sized predefined data types.</li> </ul>

Equation element ...	Represents the ...
<i>non-null_LOB_attribute_OIDs</i>	<p>total size in bytes of all the non-null OID references to values stored in BLOB, CLOB, and XML subtables.</p> <p>BLOB, CLOB, and XML values are never stored in the row. Instead, the system stores a 40-byte pointer to each BLOB, CLOB, or XML column value called an object identifier (OID). The LOB value pointed to is stored within a BLOB, CLOB, or XML subtable associated with that column of the table. See <a href="#">“Object Identifier Columns” on page 820</a> and <a href="#">“Sizing a LOB or XML Subtable” on page 861</a> for more information about OIDs and BLOB, CLOB, and XML subtables.</p> <p>The set of BLOB, CLOB, and XML attribute OIDs is composed of the following elements:</p> <ul style="list-style-type: none"> <li>• OIDs for attribute values having BLOB, CLOB, or XML types.</li> <li>• OIDs for attribute values having UDT types based on BLOB, CLOB, or XML data types.</li> </ul> <p>Each OID for a BLOB, CLOB, or XML column has a size of 40 bytes.</p>

Because the metadata within a structured UDT is byte-packed (meaning not aligned on an 8-byte boundary. Teradata Database automatically copies structured UDT values into properly aligned storage locations in memory whenever they are used by a system with a 64-bit byte-aligned format architecture) on both packed64 and aligned row format systems, it has no effect on the stored size of a given structured UDT. The entire structured UDT, however, *is* aligned on an 8-byte boundary on aligned row format systems.

## Example 1

What follows is a simple example of calculating the storage requirements of a two-level structured UDT on a packed64 format system:

Suppose you have the following nested structured type.

Level	Number of Attributes At The Level	Data Types of the Attributes
0	5	<ul style="list-style-type: none"> <li>• INTEGER</li> <li>• INTEGER</li> <li>• INTEGER</li> <li>• INTEGER</li> <li>• Structured UDT</li> </ul>
1	3	<ul style="list-style-type: none"> <li>• INTEGER</li> <li>• INTEGER</li> <li>• INTEGER</li> </ul>

Here is what is stored for a value having this structured type, assuming there are no null attributes. Nothing is stored to represent a null attribute other than a bit in the Presence Bits

Array, which means that there is no difference in the storage of a compressed null and an uncompressed null. This means that with respect to compression, there is only one storage state for nulls, and that state is referred to as compressed. Because the representation of the states is identical, it is often said that nulls are compressed by default, but this is somewhat misleading.

- 6 bytes for the TVMID for level 0, stored in INTEGER format.
- 1 octet (8-bit byte) byte for the Presence Bits Array for level 0, which contains 5 presence bits for the 5 attributes at level 0 (all set to 1) and 3 unused presence bits (all set to 0).
- 4 bytes for INTEGER value 1 at level 0.
- 4 bytes for INTEGER value 2 at level 0.
- 4 bytes for INTEGER value 3 at level 0.
- 4 bytes for INTEGER value 4 at level 0.
- Size in bytes of the level 1 structured UDT, which is:
  - 6 bytes for the TVMID for level 1, stored in INTEGER format.
  - 1 byte for the Presence Bits Array for level 1, which contains 3 presence bits for the 3 attributes at level 1 (all set to 1) and 5 unused presence bits (all set to 0).
  - 4 bytes for INTEGER value 1 at level 1.
  - 4 bytes for INTEGER value 2 at level 1.
  - 4 bytes for INTEGER value 3 at level 1.

Note that there is overhead of a TVMID value and a Presence Bits Array for *each* nesting level in a structured UDT.

For an aligned row format system, you just take the calculated size for a packed64 format system modulo(8) to align the column on an 8-byte boundary.

## Example 2

Now consider the following more concrete example, again for a packed64 system.

Suppose you create two structured types, one of which is an attribute of the other, as follows:

```
CREATE TYPE name_udt AS (
    first_name VARCHAR(20),
    last_name  VARCHAR(20)) ;

CREATE TYPE address_udt AS (
    street     VARCHAR(20),
    city      VARCHAR(20),
    zipcode   INTEGER,
    name      NameUdt);
```

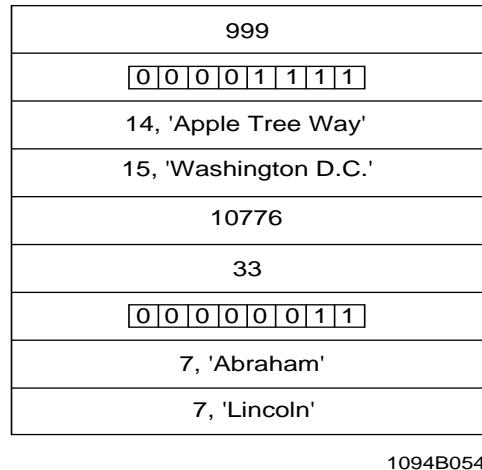
For the sake of this example, assume the TypeID for *name\_udt* is 33 and the TypeID for *address\_udt* is 999.

Suppose you insert the following data into a column typed as *address\_udt*:

```
INSERT INTO test_table
VALUES (NEW address_udt().street('Apple Tree Way')
        .city('Washington D.C.')
        .zipcode(10776))
```

```
(NEW name_udt() .name
      .first_name('Abraham')
      .last_name('Lincoln'));
```

The nested structured type *address\_udt*, which nests the UDT *name\_udt* as one of its attributes, is stored as indicated by the following graphic.



Following “[Equation: Structured UDT Size for Packed64 Format System](#)” on page 795, the size of any value having the *address\_udt* type is calculated as follows:

$$\begin{aligned}
 & \text{level} = 1 \\
 &= \sum_{\text{level} = 0}^{\text{level} = 1} \left( \text{TVMID\_size}_0 + \left( 8 \times \left( \frac{\text{number\_of\_attributes}}{8} + 1 \right) \right)_0 \right) + \left( \sum \text{fixed\_size\_non-null\_attributes} \right)_0 + \\
 & \quad \left( \sum (\text{variable\_size\_non-null\_attributes} + 2) \right)_0 + \left( \text{TVMID\_size}_1 + \left( 8 \times \left( \frac{\text{number\_of\_attributes}}{8} + 1 \right) \right)_1 + \right. \\
 & \quad \left. \left( \sum \text{fixed\_size\_non-null\_attributes} \right)_1 + \left( \sum (\text{variable\_size\_non-null\_attributes} + 2) \right)_1 \right. \\
 &= (6 + (8 \times \left( \frac{4}{8} + 1 \right)) + 4 + (20 + 2) + (20 + 2) + \left( 6 + \left( 8 \times \left( \frac{2}{8} \right) + 1 \right) \right) + 0 + (20 + 2) + 20 + 2) \\
 &= (6 + (8 \times 2.5) + 22 + 22 + 6 + (8 \times 1.5) + 0 + 22 + 22) \\
 &= 136 \text{ bytes}
 \end{aligned}$$

For an aligned row system, you just take the calculated size for a packed64 format system modulo(8) to align the column on an 8-byte boundary.

# System-Derived and System-Generated Columns

## About System-Derived Columns

As the name implies, system-derived columns are columns whose values are not created by users, but instead are derived dynamically by the system. Some system-derived columns contain values that Teradata Database creates and maintains for internal use, while others contain values that are critical for user applications.

The following system-derived column types are supported by Teradata Database:

- ROWID columns (see “[ROWID Columns](#)” on page 800)
- PARTITION and PARTITION#Ln columns (see “[PARTITION Columns](#)” on page 801)

ROWID, PARTITION, and PARTITION#Ln columns are always system-defined and their values are always system-generated.

## About System-Generated Columns

In some cases of system-generated columns the DBA specifies the name of the column and whether or not it is to be created for a table and in other cases, the existence, name, and contents of the column are all system-controlled.

The following system-generated column types are supported by Teradata Database:

- Identity columns (see “[Identity Columns](#)” on page 818)

Identity columns are user-defined, but Teradata Database defines the values inserted into them (in the case of GENERATED ALWAYS AS IDENTITY columns, all column values are system-generated. In the case of GENERATED BY DEFAULT AS IDENTITY columns, inserted column values can be user-generated or system-generated.).

- Object Identifier columns (see “[Object Identifier Columns](#)” on page 820)

Similarly, BLOB, CLOB, and XML columns are user-defined, but Teradata Database defines the OID values (see “[Object Identifier Columns](#)” on page 820) that point to them.

## ROWID Columns

Every Teradata Database base table, join index, and hash index has a system-generated column named ROWID. The fields in this column contain the RowID value for their rows.

FOR a partitioned table or join index that has ...	The data type for ROWID values is ...
65,535 or fewer combined partitions	BYTE(10)
> 65,535 combined partitions	BYTE(16)

The system-derived column ROWID contains the internal row identifier associated with a row of a base table or join index.

With one exception, there is nothing different for a column-partitioned table or join index. The exception is that the column partition is always 1 for the internal partition number in the ROWID of a column-partitioned table or join index. If you only specify column partitioning when you create a column-partitioned table or join index and do not specify the ADD option, the table or index always uses 2-byte partitioning.

As a user, you can only specify the ROWID keyword in a CREATE JOIN INDEX request to enable non-covering join indexes to join with base table columns to optimize query processing (see “[Partial Query Coverage](#)” on page 505, “[Restrictions on Partial Covering by Join Indexes](#)” on page 575, and “CREATE JOIN INDEX” in *SQL Data Definition Language*). You cannot specify ROWID in any other context at any time.

The rules for using the ROWID keyword in a CREATE JOIN INDEX request are as follows.

- You can optionally specify the ROWID for a base table in the select list of an nonpartitioned or PPI join index definition.

The select list for a column-partitioned join index *must* include the system-derived column ROWID of the base table, and it must be specified with an alias.

If you reference multiple tables in the join index definition, then you must fully qualify each ROWID specification.
- You can reference an alias for ROWID, or the keyword ROWID itself if no alias name has been specified for it, in the primary index definition or in a secondary index defined for the join index in its index clause.

This does *not* mean that you can reference a ROWID or its alias in the DDL you use to create a secondary index defined separately from CREATE TABLE using a CREATE INDEX request (see “CREATE INDEX” in *SQL Data Definition Language*) after the join index has been created.

- If you reference a ROWID alias in the select list of a join index definition, then you can also reference that correlation name in a CREATE INDEX request that creates a secondary index on the join index.
- Aliases are required to resolve any column name or ROWID ambiguities in the select list of a join index definition. An example is the situation where you specify ROWID for more than one base table in the index definition.
- Aliases are mandatory for a ROWID specification in a column-partitioned join index.

If you attempt to use the ROWID keyword in any other context, such as selecting or deleting from, updating, or inserting rows into base tables, views, or derived tables Teradata Database aborts the request and returns an error to the requestor.

These rules apply equally to a join index defined with a partitioned primary index and to a column-partitioned join index whether the partitioning is single-level or multilevel.

## PARTITION Columns

The system derives a PARTITION column for each Teradata Database base table, global temporary table, volatile table, and non-compressed join index whenever the words PARTITION BY are invoked for that base, global temporary, or volatile table in a CREATE TABLE or ALTER TABLE SQL statement (see *SQL Data Definition Language*).

System-Derived Column Name	Default Data Type	Default Title
PARTITION	INTEGER	PARTITION
PARTITION#Ln  <i>n</i> represents an INTEGER value ranging from 1 through 62, inclusive.	The data type for PARTITION and PARTITION#Ln columns can be cast to a different type (see “ <a href="#">Example 8: Selection of All Active Partitions From a Table</a> ” on page 807).	PARTITION#Ln

PARTITION provides the combined partition number for a row.

The system-derived columns PARTITION#Ln, where *n* ranges between 1 and 62, inclusive, provide the partition number of a row for the specified level.

If the partitioning expression for a table or join index defines between 1 and 65,535 combined partitions, the partition is stored as a 2-byte value in the row header.

If the partitioning expression for a table or join index defines more than 65,535 combined partitions, the partition is stored as an 8-byte value in the row header.

You cannot alter a 2-byte PPI table to become an 8-byte PPI table. A table is created with a 2-byte PPI if you define it with 65,535 or fewer combined partitions. To create an 8-byte PPI table that has 65,535 or fewer combined partitions in a way that it can later be altered to have more than 65,535 combined partitions as additional partitions are needed, first create the table with more than 65,535 combined partitions and then later alter it using an ALTER TABLE ... MODIFY PRIMARY INDEX request that specifies a DROP RANGE#L1 clause to create the desired initial number of combined partitions. This action retains the index as an 8-byte PPI.

The default values for PARTITION and PARTITION#Ln values have the following valid inclusive domain range values.

FOR this type of table ...	The valid range for PARTITION is ...
PPI	1 - 9,223,372,036,854,775,807  9,223,372,036,854,775,807 is the maximum number of partitions you can define for a PPI whether that PPI is single-level or multilevel.
nonpartitioned primary index	0

The partition number of the column-partitioning level for a row is always 1.

- If a table has column partitioning, but no row partitioning, PARTITION and PARTITION#L1 both return 1 and PARTITION#Ln, where *n* is between 2 and 62, inclusive, returns 0.
- If a table has both column partitioning and row partitioning, and the column partitioning is at level *m*, PARTITION#Lm is 1 and, for the column-level partitioning, Teradata

Database uses 1 as the column partition number in calculating the combined partition number value for PARTITION.

You should restrict the use of the system-derived columns PARTITION and PARTITION#*Ln* to database administrators. You can restrict usage of these system-derived columns by restricting access to PPI tables by means of views.

Note that PARTITION numbers can change if you alter the partitioning for a PPI table.

The principal end user application for the values stored in the system-derived PARTITION and PARTITION#*Ln* columns is to retrieve rows from specific primary index partitions, but you can specify PARTITION or PARTITION#*Ln* as a column name in any of the following SQL request types.

- Any DML request to determine the partition to which various rows in a table belong.
- A COLLECT STATISTICS request (both forms) to collect the external partition number for each row on which the statistics are collected for a base data table.  
See “COLLECT STATISTICS (Optimizer Form)” in *SQL Data Definition Language* and “COLLECT STATISTICS (QCD Form)” in *SQL Data Manipulation Language* for details.
- An ALTER TABLE request in the DROP RANGE WHERE clause.  
See “ALTER TABLE” in *SQL Data Definition Language* for details.
- A CREATE VIEW or CREATE RECURSIVE VIEW request.  
See “CREATE RECURSIVE VIEW” and “CREATE VIEW” in *SQL Data Definition Language* for details.

You *cannot* specify the system-derived PARTITION column in any of its forms in any of the following index definition statements. This means the restrictions that exist for the system-derived PARTITION column apply equally to the system-derived PARTITION#*Ln* column set.

- CREATE HASH INDEX (see “CREATE HASH INDEX” in *SQL Data Definition Language* for details).
- CREATE INDEX (see “CREATE INDEX” and “CREATE TABLE” in *SQL Data Definition Language* for details).
- CREATE JOIN INDEX (see “CREATE JOIN INDEX” in *SQL Data Definition Language* for details).

You *can* specify an ordinary user-created column named *partition*, but if you do so, you cannot reference the system-derived PARTITION column by that name in any of its forms. This rule applies equally to the PARTITION#L1 - PARTITION#L62 system-derived column set. See *SQL Data Definition Language* for details.

As with any other column, you must fully qualify all ambiguous PARTITION and PARTITION#*Ln* references. Like any other column, PARTITION and PARTITION#*Ln* can be qualified by a database name and table name, they can be aliased using the [AS] *column\_name* syntax, and they can be referenced wherever a column of the table can be referenced.

IF this many tables specified in a query have an explicit column named PARTITION ...	THEN ...
1	an unqualified reference to PARTITION refers to the explicitly defined column.
> 1	all references to PARTITION must be qualified.

If you reference multiple tables in a query, then all references to the system-derived column PARTITION or the system-derived PARTITION#Ln column set must be qualified appropriately.

Neither PARTITION nor any member of the PARTITION#Ln column set is included in the set of columns returned when you specify the ASTERISK (\*) character in a SELECT request. You must specify those columns explicitly in a select list to return them in a response set (see “[Example 4: PARTITION Values Not Returned Because PARTITION Not Specified in Select List](#)” on page 807, “[Example 1](#)” on page 813 - “[Example 4](#)” on page 815). In particular, the PARTITION and PARTITION#Ln columns cannot be accessed through a view on a table unless the view definition explicitly specifies the column in its select list (see “[Example 10: Using PARTITION In View Definitions](#)” on page 808, “[Example 7](#)” on page 815, “[Example 9](#)” on page 816, “[Example 10](#)” on page 816, “[Example 13](#)” on page 817).

Because neither PARTITION nor PARTITION#L1 through PARTITION#62 are reserved words, you can also assign their names to user-defined columns. If you do this, however, the system-defined PARTITION and PARTITION#Ln columns for any such table are not accessible from an SQL query because there is no way to distinguish them from the user-defined column with the same name.

Neither PARTITION nor PARTITION#L1 through PARTITION#62 are returned as a column by a HELP TABLE or HELP COLUMN request, nor do they appear in the data dictionary as columns of the referenced table. As a result, the create text returned by a SHOW TABLE statement does not report system-derived PARTITION and PARTITION#Ln columns.

Do *not* attempt to update partition values under any circumstances. This operation is not permitted, and the system aborts the request and returns an error if you attempt to perform it.

The value of PARTITION or PARTITION#Ln takes no additional space in the table. Teradata Database extracts the internal partition number from the partition field in the RowID of the row header and converts the value dynamically to its external partition number (see “[Base Table Row Format](#)” on page 740). The user-visible value in the PARTITION field of a row is the external number of the partition in which it is stored. Teradata Database returns an error to the requestor if there is no external partition number corresponding to the internal partition number requested.

PARTITION and PARTITION#Ln are equivalent to a value expression where that expression is identical to the partitioning expression defined for the primary index of the table with column references appropriately qualified as needed (or 0 if no partitioning expression for the primary index is defined).

This is true if, and only if, the row specifies the correct internal partition.

For example, a query of a PPI table that specifies the predicate `WHERE PARTITION <> partitioning_expression` should always return 0 rows. If any rows are returned, then they are not partitioned properly, and the table should be revalidated immediately. See “ALTER TABLE” in *SQL Data Definition Language* for more information about revalidating a PPI.

If the partitioning expression for a PPI table is changed, then the values of PARTITION and PARTITION#Ln for many rows in the table might also change.

You should always collect statistics on the single-column PARTITION and PARTITION#Ln columns for any PPI base table (You cannot collect PARTITION statistics on a global temporary table) to ensure that the Optimizer is able to use the most accurate cost estimates possible.

Multicolumn PARTITION and PARTITION#Ln statistics are used to do single table estimations when all the columns that include the PARTITION column have single-table equality conditions.

If the table is partitioned by a single-column expression, then its column statistics are inherited as PARTITION statistics. In this special situation, you need not also collect single-column PARTITION or PARTITION#Ln statistics.

See “COLLECT STATISTICS (Optimizer Form)” in *SQL Data Definition Language* for details about collecting PARTITION or PARTITION#Ln statistics.

The system-derived column PARTITION also provides the partition number of the combined partitioning expression associated with a row when a multilevel PPI has been defined on a table. The system-derived columns PARTITION#L1 through PARTITION#L62 provide the partition number associated with the corresponding level.

## Usage Examples for PARTITION

The following examples demonstrate usage of the system-derived PARTITION column for single-level partitioned primary index tables only. They do *not* cover usage considerations for the combined PARTITION expression of multilevel partitioned primary index tables.

For an example of PARTITION usage in that context, see “[Usage Examples for PARTITION#Ln](#)” on page 812 and “[Example 1](#)” on page 813 through “[Example 14](#)” on page 818.

The following table definitions are used for this set of examples.

```
CREATE TABLE orders (
    o_orderkey      INTEGER NOT NULL,
    o_custkey        INTEGER,
    o_orderstatus    CHARACTER(1) CASESPECIFIC,
    o_totalprice     DECIMAL(13,2) NOT NULL,
    o_orderdate      DATE FORMAT 'yyyy-mm-dd' NOT NULL,
    o_orderpriority  CHARACTER(21),
    o_clerk          CHARACTER(16),
    o_shipppriority  INTEGER,
    o_comment         VARCHAR(79))
PRIMARY INDEX (o_orderkey)
PARTITION BY RANGE_N(
```

```
    o_orderdate BETWEEN DATE '2001-01-01'
                    AND      DATE '2007-12-31'
                    EACH INTERVAL '1' MONTH)
UNIQUE INDEX (o_orderkey);

CREATE TABLE lineitem (
    l_orderkey      INTEGER NOT NULL,
    l_partkey       INTEGER NOT NULL,
    l_suppkey       INTEGER,
    l_linenumber    INTEGER,
    l_quantity      INTEGER NOT NULL,
    l_extendedprice DECIMAL(13,2) NOT NULL,
    l_discount      DECIMAL(13,2),
    l_tax           DECIMAL(13,2),
    l_returnflag    CHARACTER(1),
    l_linestatus    CHARACTER(1),
    l_shipdate      DATE FORMAT 'yyyy-mm-dd',
    l_commitdate    DATE FORMAT 'yyyy-mm-dd',
    l_receiptdate   DATE FORMAT 'yyyy-mm-dd',
    l_shipinstruct  VARCHAR(25),
    l_shipmode      VARCHAR(10),
    l_comment       VARCHAR(44))
PRIMARY INDEX (l_orderkey)
PARTITION BY RANGE_N(
    l_shipdate BETWEEN DATE '2001-01-01'
                  AND      DATE '2007-12-31'
                  EACH INTERVAL '1' MONTH);
```

## Example 1: Delete All Orders From a Specific Partition

Delete all orders in the *orders* table from partition 1.

```
DELETE
FROM orders
WHERE orders.PARTITION = 1;
```

## Example 2: Delete All Orders From a Range of Partitions

Delete all orders in the *orders* table from partitions 30 through 32, inclusive.

```
DELETE
FROM orders
WHERE orders.PARTITION BETWEEN 30 AND 32;
```

## Example 3: INSERT ... SELECT Rows Into a Target Table From Several Partitions, Then Delete Them From the Source Table

Use an INSERT ... SELECT statement to copy orders from partitions 1 and 2 of the *orders* table into the *old\_orders* table, then delete them from *orders*.

```
INSERT INTO old_orders
SELECT *
FROM orders
WHERE orders.PARTITION IN (1,2)
;DELETE FROM orders
WHERE orders.PARTITION IN (1, 2);
```

## Example 4: PARTITION Values Not Returned Because PARTITION Not Specified in Select List

In the following example, the value of PARTITION is not returned as one of the column values, even though it is specified in the WHERE clause, because it was not explicitly specified in the select list for the query:

```
SELECT *
FROM orders
WHERE orders.PARTITION = 10
AND   orders.o_totalprice > 19.99;
```

## Example 5: Qualification of PARTITION Not Necessary Because Specification Is Unambiguous

PARTITION does not have to be qualified in this example because its use is unambiguous:

```
SELECT orders.*, PARTITION
FROM orders
WHERE orders.PARTITION = 10
AND   orders.o_totalprice > 100;
```

## Example 6: Qualification of PARTITION Necessary Because of Ambiguity Otherwise

PARTITION must be qualified in the two following examples to distinguish between PARTITION values in the *orders* table and PARTITION values in the *lineitem* table:

```
SELECT *
FROM orders, lineitem
WHERE orders.PARTITION = 3
AND   lineitem.PARTITION = 5
AND   orders.o_orderkey = Lineitem.l_orderkey;

SELECT orders.*, lineitem.*, orders.PARTITION
FROM orders, lineitem
WHERE orders.PARTITION = 3
AND   lineitem.PARTITION = 5
AND   orders.o_orderkey = lineitem.l_orderkey;
```

## Example 7: Non-Valid Use of PARTITION In VALUES Clause of INSERT Statement

This example is *not* valid because PARTITION cannot be referenced in the VALUES clause:

```
INSERT INTO Orders VALUES (PARTITION, 10, 'A', 599.99,
                           DATE '2001-02-07', 'HIGH', 'Jack', 3, 'Invalid insert');
```

## Example 8: Selection of All Active Partitions From a Table

The following two examples provide a list of the populated partitions in the *orders* table:

```
SELECT DISTINCT PARTITION (FORMAT '999')
FROM Orders
ORDER BY PARTITION;
```

```
SELECT DISTINCT CAST (PARTITION AS BYTEINT)
```

```
FROM Orders
ORDER BY PARTITION;
```

## Example 9: Using PARTITION With An Aggregate Function

The following example counts the number of rows in each populated partition:

```
SELECT PARTITION, COUNT(*)
FROM Orders
GROUP BY PARTITION
ORDER BY PARTITION;
```

## Example 10: Using PARTITION In View Definitions

You cannot select PARTITION through this view because it was not specified explicitly in its select list:

```
CREATE VIEW ordersv AS
SELECT * FROM orders;
```

Because it is explicitly selected in the following view definition, PARTITION values are returned to the requestor if all columns are selected from the view using the \* literal:

```
CREATE VIEW ordersvp AS
SELECT orders.* , PARTITION
FROM orders;
```

## Example 11: A More Sophisticated Use of Partitioning

The following query finds orders for which at least one lineitem was shipped in the same month as the order was recorded:

```
SELECT o.o_orderkey
FROM orders AS o, lineitem AS l
WHERE o.PARTITION = l.PARTITION
AND o.o_orderkey = l.l_orderkey
GROUP BY o.o_orderkey;
```

Note that the primary indexes for both tables are partitioned in such a way that their rows are partitioned on the same date ranges, so that can be exploited as a WHERE clause search condition.

## Usage Considerations and Rules for PARTITION and PARTITION#Ln Columns

The usage considerations and rules for the system-derived PARTITION column for single-level PPIs apply equally to multilevel partitioned primary index tables except as noted in the following rules.

- The system-derived column PARTITION is equivalent to a value expression that is identical to combined partitioning expression derived for the primary index of the table, with appropriately qualified column references. This value is always 0 for a non-partitioned primary index.

The combined partitioning expression for a table defines how rows are ultimately partitioned on each AMP. The result of the combined partitioning expression for specific values of the partitioning column is referred to as the *combined partition number*.

Teradata Database derives a combined partitioning expression from the partitioning expressions defined for each individual row partitioning level, if any, and a column partition number of 1 for the column-partitioning level, if there is one.

For  $n$  partitioning levels, where  $n$  is the number of partitioning levels defined, the combined partitioning expression is defined as follows.

$$\text{Combined partitioning expression} = \sum_{i=1}^{n-1} ((p_i - 1) \times dd_i) + p_n$$

where:

Equation element ...	Specifies ...
$p_i$	the row partitioning expression at level $i$ , numbering from left to right, or 1 for a column partitioning level.
$dd_i$	a constant value equal to the following product. $\prod_{j=i+1}^n d_j$
$d_j$	the maximum partition number defined at level $j$ .

The parentheses delimiting the summation are not included in the combined partitioning expression. For example, this summation expands for one-, two-, three-, four-, and five-level partitioning as the following combined partitioning expressions, respectively, where  $p_i$  and  $dd_i$  are defined as they were in the preceding definition for a combined partitioning expression.

Partitioning Level	Expanded Summation
1	$p_1$
2	$(p_1-1)*dd_1+p_2$
3	$(p_1-1)*dd_1+(p_2-1)*dd_2+p_3$
4	$(p_1-1)*dd_1+(p_2-1)*dd_2+(p_3-1)*dd_3+p_4$
5	$(p_1-1)*dd_1+(p_2-1)*dd_2+(p_3-1)*dd_3+(p_4-1)*dd_4+p_5$

The combined partitioning expression reduces to  $p_1$  for single-level row partitioning, so there is no actual change in the usage rules for this case.

For column partitioning, the combined partition number for a specific column partition value of a table row can be derived from the combined partition number for this table row as follows.

$$\text{combined\_part\_number-specific\_col\_part\_value\_of\_row} = \text{cpn} + (c - 1) \times dd_i$$

where:

Equation element ...	Specifies ...
<i>combined_part_number-specific_col_part_value_of_row</i>	the combined partition number for a specific column partition value of a table row.
<i>cpn</i>	the combined partition for the row.
<i>c</i>	the column partition number for this column partition value within the row.
<i>dd<sub>i</sub></i>	a constant value equal to 1.

This is the same as computing the combined partitioning expression for a row-partitioned table row except instead of using 1 for the column-partitioning level, this equation uses the column partition number corresponding to the specific column partition value of the table row.

While column partitioning can be defined at any level, it is recommended in most cases to put the column-partitioning level first before any row partitioning. Some considerations that might lead to assigning the column partitioning to a lower level in the partitioning hierarchy are potential improvements for cylinder migration and temperature-based block compression effectiveness for hot and cold data.

- You cannot reference the system-derived PARTITION column in a CREATE JOIN INDEX, CREATE HASH INDEX, or CREATE INDEX requests (see *SQL Data Definition Language*).
- You can reference the system-derived columns PARTITION#L1 through PARTITION#L62 at any point in a DML request where a table column can be referenced. You can also reference the system-derived PARTITION#Ln columns in the DROP RANGE WHERE clause of an ALTER TABLE request (see *SQL Data Definition Language*).
- You can neither update these system-derived columns, nor can you assign a value or null to them with an insert operation. Both cases abort the request and return an error message to the requestor.
- As is the case with single-level PPI tables, if a multilevel PPI table definition explicitly specifies a column with the same name as any of the 62 system-derived PARTITION columns, then the system-derived column having that name cannot be accessed because the system interprets any reference to that column name for the table as a reference to the user-defined column rather than its homonymous system-derived column.
- You can qualify the system-derived PARTITION#Ln columns with a database name and table name just as you can any other table column.

The following list describes the details of qualifying system-derived PARTITION#*L<sub>n</sub>* columns:

- If only one table in a query has an explicitly named column of PARTITION#*L<sub>n</sub>* (where *n* is an integer in the range 1 - 62, inclusive), then an unqualified reference to PARTITION#*L<sub>n</sub>* refers to the user-specified column of that name in that table.
- If more than one table in the query has an explicit user-specified column named PARTITION#*L<sub>n</sub>*, then you must qualify any references to PARTITION#*L<sub>n</sub>*.
- If there are multiple tables in the query, then you must qualify all references to the system-derived column PARTITION#*L<sub>n</sub>*.
- If there is only one table in the query, and that table does not have a user-named column named PARTITION#*L<sub>n</sub>*, then an unqualified reference to PARTITION#*L<sub>n</sub>* refers to the system-derived column PARTITION#*L<sub>n</sub>* for that table.
- The data type for a PARTITION#*L<sub>n</sub>* column depends on whether its PPI is the 2-byte form or the 8-byte form.

FOR this form of PPI ...	THE data type for the system-derived PARTITION# <i>L<sub>n</sub></i> column is ...	AND the default format is ...
2-byte	INTEGER	INTEGER
8-byte	BIGINT	BIGINT

When invoked, each returns the value that is the partition number of the row in the table for the specified partitioning level. The value ranges between 1 and the number of partitions defined for the specified partitioning level. For a table without a partitioning expression at that level, or for a non-partitioned primary index, the value returned is 0. For single-level partitioning, the system-derived columns PARTITION and PARTITION#L1 are synonyms and have the same value, while the other system-derived PARTITION#*L<sub>n</sub>* columns have a value of 0.

- Also like the system-derived PARTITION column of single-level PPI tables, the values of the system-derived PARTITION#*L<sub>n</sub>* columns consume no space in the table.

When you reference a PARTITION#*L<sub>n</sub>* column, the system extracts the internal partition number for the combined partitioning expression from the row and converts it to the external partition number for the corresponding level of the system-derived column.

If no corresponding external partition number for the extracted internal partition number exists, the system aborts the request and returns an error message to the requestor.

- A system-derived PARTITION#*L<sub>n</sub>* column is equivalent to a value expression that is identical to the partitioning expression at the specified level defined for the primary index, or 0 if there is no partitioning expression for that level or if the primary index is not partitioned, of the table with appropriately qualified column references.
- Like the system-derived PARTITION column for single-level PPI tables, the system-derived PARTITION#*L<sub>n</sub>* columns are not included in the list of columns returned by specifying an ASTERISK character or `table_name.*` when you select rows from a

table. You can, however, explicitly select system-derived PARTITION#*Ln* column from the table.

- Similarly, you cannot access the system-derived PARTITION#*Ln* columns through a view based on an underlying multilevel partitioned primary index table unless that view explicitly includes the name of the system-derived column in its definition. In this case, you have explicitly defined the columns in the view, so selecting either \* or *view\_name*. \* from the view includes those explicit columns from the view.

If you attempt to select a system-derived PARTITION#*Ln* column from a view in which its name is not explicitly defined for the view, the request aborts and the system returns an error message to the requestor. Because the SQL semantics of views and derived tables are identical, the system aborts the request and reports the identical error under the same circumstances with a derived table.

- Like the system-derived PARTITION column for single-level PPI tables, the system does not return any system-derived PARTITION#*Ln* columns in response to a HELP TABLE or HELP COLUMN request because they are derived and are not stored in the dictionary as names of physical columns in the table.
- As is true of the system-derived PARTITION column for single-level PPI tables, you cannot reference a system-derived PARTITION#*Ln* column in a CREATE JOIN INDEX, CREATE HASH INDEX, or CREATE INDEX statement. If you attempt to do so, the system aborts the request and returns an error message to the requestor.

This restriction also implies that you cannot specify a system-derived PARTITION#*Ln* column as an index or partitioning column. Of course, you *can* specify a user-defined homonymous column name as an index or partitioning column.

- Note that if you use ALTER TABLE (see *SQL Data Definition Language*) to change one or more of the partitioning expressions for the primary index of an multilevel partitioned primary index table, the values of the system-derived PARTITION#*Ln* columns for rows in the altered table might change.

## Usage Examples for PARTITION#*Ln*

The following set of sixteen examples provides insight into how you can use the system-derived PARTITION#*Ln* set to acquire information about particular partitions in a table that has a multilevel partitioned primary index.

Assume that you have created the *orders* and *lineitem* tables defined by the following CREATE TABLE requests.

```
CREATE TABLE orders (
    o_orderkey      INTEGER NOT NULL,
    o_custkey        INTEGER,
    o_orderstatus    CHARACTER(1) CASESPECIFIC,
    o_totalprice     DECIMAL(13,2) NOT NULL,
    o_orderdate      DATE FORMAT 'yyyy-mm-dd' NOT NULL,
    o_orderpriority  CHARACTER(21),
    o_comment        VARCHAR(79))
PRIMARY INDEX (o_orderkey)
PARTITION BY (
    RANGE_N(o_custkey) BETWEEN 0
                            AND 49999
```

```

        EACH 100),
RANGE_N(o_orderdate BETWEEN DATE '2000-01-01'
                      AND   DATE '2006-12-31'
                      EACH INTERVAL '1' MONTH))
UNIQUE INDEX (o_orderkey);

CREATE TABLE lineitem (
    l_orderkey      INTEGER NOT NULL,
    l_partkey       INTEGER NOT NULL,
    l_suppkey       INTEGER,
    l_linenumber    INTEGER,
    l_quantity      INTEGER NOT NULL,
    l_extendedprice DECIMAL(13,2) NOT NULL,
    l_discount      DECIMAL(13,2),
    l_tax           DECIMAL(13,2),
    l_returnflag    CHARACTER(1),
    l_linestatus    CHARACTER(1),
    l_shipdate      DATE FORMAT 'yyyy-mm-dd',
    l_commitdate    DATE FORMAT 'yyyy-mm-dd',
    l_receiptdate   DATE FORMAT 'yyyy-mm-dd',
    l_shipinstruct  VARCHAR(25),
    l_shipmode      VARCHAR(10),
    l_comment       VARCHAR(44))
PRIMARY INDEX (l_orderkey)
PARTITION BY (
    RANGE_N(l_suppkey BETWEEN 0
                        AND 4999
                        EACH 10),
    RANGE_N(l_shipdate BETWEEN DATE '2000-01-01'
                           AND   DATE '2006-12-31'
                           EACH INTERVAL '1' MONTH));

```

## Example 1

The following example selects all rows from the *orders* table from January, 2000 (*orders.PARTITION#L2=1*).

```

SELECT *
FROM orders
WHERE orders.PARTITION#L2 = 1;

```

Because PARTITION#L2 is not referenced in the select list, its value is not returned.

Because only one table is referenced by the request, you need not fully qualify PARTITION#L2 in the WHERE clause, though this example does so.

## Example 2

The following example selects all rows from the *orders* table for customers that meet the following criteria:

- *custkey* values are between 100 and 199, specified by partition number 2 for level 1 (WHERE *orders.PARTITION#L1=2*)
- *orderdate* values are from January, 2001, specified by partition number 13 for level 2 (AND *orders.PARTITION#L2=13*)

```
SELECT *
```

```
FROM orders
WHERE orders.PARTITION#L1 = 2
AND   orders.PARTITION#L2 = 13;
```

Because neither PARTITION#L1 nor PARTITION#L2 is referenced in the select list, their values are not returned.

Because only one table is referenced by the request, you need not fully qualify PARTITION#L1 and PARTITION#L2 in the WHERE clause, though this example does so.

### Example 3

The following example selects all rows from the *orders* table for customers that meet the following criteria:

- *custkey* values between 100 and 199, specified by partition number 2 for level 1
- *orderdate* values from January, 2001, specified by partition number 13 for level 2

Note that the results set returned by this request is identical to that returned by the query in “[Example 2](#)” on page 813.

Teradata Database derives these predicates from the combined partitioning expression, which is expressed by the clause WHERE *orders*.PARTITION=97 in the request.

For this request, the partition number for the combined partitioning expression is derived from the following calculation:

$$\text{PARTITION} = ((\text{PARTITION}\#L1 - 1) \times 84) + \text{PARTITION}\#L2$$

where:

Equation element ...	Specifies ...
PARTITION	the partition number for the combined partitioning expression.
PARTITION#L1	the partition number for level 1 rows containing <i>custkey</i> values in the inclusive range 100 - 199.
84	the number of partitions defined for level 2 of the <i>orders</i> table by the following partitioning expression: <pre>RANGE_N(o_orderdate BETWEEN DATE '2000-01-01'           AND      DATE '2006-12-31'           EACH INTERVAL '1' MONTH))</pre> which defines six complete years, each having twelve months. $6 \times 12 = 84$
PARTITION#L2	the partition number for level 2 rows containing <i>orderdates</i> from January, 2001, which is 13.

The result is the partition number for the combined partitioning expression:

$$\text{PARTITION} = ((2 - 1) \times 84) + 13 = 97$$

```
SELECT *
FROM orders
WHERE orders.PARTITION = 97;
```

Because you do not specify PARTITION explicitly in the select list, the request does not return its value. Because only one table is referenced by the request, you need not fully qualify PARTITION in the WHERE clause, though this example does so.

## Example 4

In the following example, you *must* qualify PARTITION#L2 in the WHERE clause predicate specification because you reference it in both the *orders* and the *lineitem* tables:

```
SELECT *
FROM orders, lineitem
WHERE orders.PARTITION#L2 = 3
AND lineitem.PARTITION#L2 = 5
AND orders.o_orderkey = lineitem.l_orderkey;
```

Because you do not specify either orders.PARTITION#L2 or orders.PARTITION#L2 in the select list, the request does not return their values.

## Example 5

The following example aborts and returns an error message because you cannot specify the PARTITION#Ln keyword in a VALUES clause:

```
INSERT INTO orders
VALUES (PARTITION#L1, 10, 'A', 599.99, DATE '2006-02-07',
'HIGH', 'Invalid insert');
```

## Example 6

The following example returns a list of all the populated partitions for level 2 of the *orders* table:

```
SELECT DISTINCT CAST (PARTITION#L2 AS BYTEINT)
FROM orders
ORDER BY PARTITION#L2;
```

Teradata Database selects the partition number for level 2 from each row in the partition, but because of the DISTINCT operator, it reports only one row for the partition. The system does not return any rows for empty partitions.

Because only one table is referenced by the request, you need not fully qualify PARTITION#L2 in either the select list or the ORDER BY clause.

## Example 7

Consider the following view definition:

```
CREATE VIEW ordersv AS
SELECT *
FROM orders;
```

Because they are not explicitly selected in the view definition, you cannot select any of the system-derived PARTITION columns in the *orders* table through it.

For example, all three of the following SELECT requests fail with an Invalid Partition field error:

```
SELECT PARTITION
FROM ordersv; -- 5879 Invalid Partition field.
```

```
SELECT PARTITION#L1
FROM ordersv; -- 5879 Invalid Partition field.
```

```
SELECT PARTITION#L5
FROM ordersv; -- 5879 Invalid Partition field.
```

Now consider the following view definition that *does* explicitly specify the system-derived column PARTITION#L1:

```
CREATE VIEW ordersvp AS
SELECT orders.* , PARTITION#L1
FROM orders;
```

If you want to select from the *orders* table using this view, you can submit a `SELECT *` request, and it returns the value of the system-derived PARTITION#L1 column as well as the values for the physical columns defined for the table.

Similarly, you can select *only* the system-derived PARTITION#L1 column using this view.

## Example 8

The outcome of the following request depends on whether the relation defined by the view is populated or empty, as follows:

IF the <i>orders</i> table has ...	THEN the query returns ...
any rows that satisfy the <i>ordersvp</i> view definition	the maximum partition number in the view for level 1 of the <i>orders</i> table.
no rows that satisfy the <i>ordersvp</i> view definition	null.

```
SELECT MAX(PARTITION#L1)
FROM ordersvp;
```

## Example 9

Because it is not explicitly selected in the view definition, you cannot select the system-derived combined PARTITION column in the *orders* table using the *ordersvp* view.

The following SELECT request fails with an Invalid Partition field error:

```
SELECT PARTITION
FROM ordersvp; -- 5879 Invalid Partition field.
```

## Example 10

This example fails with the same error as the query in “[Example 9](#) on page 816” because the system-derived partition column PARTITION#L5 is not specified explicitly in the definition of the *ordersvp* view.

```
SELECT PARTITION#L5
FROM ordersvp; -- 5879 Invalid Partition field.
```

### Example 11

In the following view definition, the system-derived column PARTITION#L1 is explicitly selected and then redefined, or aliased, as P#1.

If you select all columns from the P#1 view by specifying an ASTERISK character as the only entry in the select list, then the system returns the value of PARTITION#L1 as P#1, but it does *not* return PARTITION#L1 as a column.

Given this view definition, you can also explicitly select P#1, but not PARTITION#L1, through it.

```
CREATE VIEW ordersvp1 AS
SELECT orders.* , PARTITION#L1 AS P#1
FROM orders;
```

### Example 12

The outcome of the following request depends on whether the relation defined by the view is populated or empty, as follows.

IF the orders table has ...	THEN the query returns ...
any rows that satisfy the <i>ordersvp1</i> view definition	the minimum partition number for level 1.
no rows that satisfy the <i>ordersvp1</i> view definition	null.

```
SELECT MIN(P#1)
FROM ordersvp1;
```

### Example 13

The following requests against the *ordersvp1* view all fail with an Invalid Partition field error because they specify a name for a system-derived partition column in their select list that is not defined explicitly in the view definition.

The following request fails because PARTITION is not defined explicitly in *ordersvp1*:

```
SELECT PARTITION
FROM ordersvp1; -- 5879 Invalid Partition field.
```

The following request fails because PARTITION#L1 is not defined explicitly in *ordersvp1*; it is aliased as P#1.

```
SELECT PARTITION#L1
FROM ordersvp1; -- 5879 Invalid Partition field.
```

To obtain the desired result of the request, you need to rewrite it as follows:

```
SELECT P#1
FROM ordersvp1;
```

The following request fails because PARTITION#L5 is not defined explicitly in *ordersvp1*:

```
SELECT PARTITION#L5
FROM ordersvp1; -- 5879 Invalid Partition field.
```

## Example 14

The outcome of the following request depends on whether the table is populated or empty, as follows:

IF the orders table is ...	THEN the query returns ...
populated	the number of partitioning levels defined for the table. Equivalently, the number of partitioning expressions defined for the table. These are just two different ways of saying the same thing.
not populated	null.

```
SELECT MAX(CASE_N(PARTITION#L1=0,      PARTITION#L2=0,      PARTITION#L3=0,
                  PARTITION#L4=0,      PARTITION#L5=0,      PARTITION#L6=0,
                  PARTITION#L7=0,      PARTITION#L8=0,      PARTITION#L9=0,
                  PARTITION#L10=0,     PARTITION#L11=0,
                  PARTITION#L12=0,     PARTITION#L13=0,
                  PARTITION#L14=0,     PARTITION#L15=0,     NO CASE)) - 1
FROM orders;
```

Note that if *orders* had more than 65,535 combined partitions, you would need to specify 62 partition numbers (PARTITION#L1 through PARTITION#L62) rather than 15.

## Identity Columns

To create a table with an identity column, you must explicitly specify one column in the table definition as either GENERATED ALWAYS AS IDENTITY or GENERATED BY DEFAULT AS IDENTITY.

You can only define one identity column per table.

Identity columns have many applications, including the automatic generation of unique primary indexes, unique secondary indexes, and primary keys. Values generated for GENERATED ALWAYS AS IDENTITY columns are always unique, but those generated for GENERATED BY DEFAULT AS IDENTITY are only unique if you also specify a UNIQUE constraint on the column. Note that if you load the same row twice into an identity column SET table, it is not rejected as a duplicate because it is made unique as soon as an identity column value is generated for it. This means that some preprocessing must still be performed on rows to be loaded into identity column tables if real world uniqueness is a concern.

Do *not* attempt to update GENERATE ALWAYS identity column values under any circumstances. This operation is not permitted and the system returns an error if you attempt to perform it.

The data type for an identity column is user-defined, but must be an exact numeric type drawn from the following list.

- BIGINT
- BYTEINT
- DECIMAL( $n,0$ )
- INTEGER
- NUMBER( $n,0$ )
- NUMERIC( $n,0$ )
- SMALLINT

In general, you should define the column to be DECIMAL, NUMBER, or NUMERIC, with the value for precision,  $n$ , being the largest number available on your system.

The largest size identity column Teradata Database produces is that stipulated by a DECIMAL(18,0), a NUMBER(18,0), or a NUMERIC(18,0) specification. This is true even when the DBS Control parameter MaxDecimal is set to 38 (see *SQL Data Types and Literals and Utilities: Volume 1 (A-K)*). You can define an identity column with more than 18 digits of precision, or even as a BIGINT type, without the CREATE TABLE or ALTER TABLE request aborting, but the values generated by the identity column feature remain limited to the DECIMAL(18,0) type and size.

Rows are processed in different orders on different systems under Unity Director, and there is no guarantee that the same data row will be assigned the same identity column value on all systems. Furthermore, it is not possible to assign Unity Director-determined values to specific rows across systems, especially when the identity column is the primary index for a table.

If you must drop the IDENTITY attribute from an identity column to use the Unity Director product with bulk data loads, you can use an ALTER TABLE request to do so. See *SQL Data Definition Language* for more information about ALTER TABLE and identity columns.

## Using Identity Columns

The most frequent use of identity columns is as surrogate keys. An identity column is an optional column attribute. When you associate an identity column attribute with a column, Teradata Database generates a table-level unique number for the column for every inserted row.

Identity column values are returned as part of the response, if an AGKR (Auto Generated Key Retrieval) option flag in the request-level Options parcel is set, when an INSERT or INSERT ... SELECT request is executed.

## Advantages of Identity Columns

The main advantage of an identity column is its ease of use in defining a unique row identity value. An identity column guarantees row uniqueness in a table when the column is defined as a GENERATED ALWAYS column with NO CYCLE allowed.

It might be difficult with some tables to find a combination of columns that make a row unique. If a composite index is not desirable for the table, you can define an identity column as the primary index for the table.

An identity column is also suited for generating unique primary key values used as employee numbers, order numbers, item numbers, and the like. In this way, you can get a uniqueness guarantee without the performance overhead of specifying a UNIQUE constraint.

## Disadvantages of Identity Columns

One disadvantage of an identity column is that the generated values have identity gaps whenever an INSERT request into a table having an identity column is aborted or rows are deleted from the table.

The sequence of identity column values is not guaranteed, and identity column values necessarily reflect the approximate chronological order of the rows inserted, though you cannot rely on a strict chronological order for the identity column values in a table because of the massive parallelism of Teradata Database.

Also keep in mind that once a table with an identity column is populated with data, deleting all of the rows in the table and reinserting new rows does *not* necessarily cause the numbering to restart from 1. Numbering continues from the last generated number of the table.

To restart numbering from 1, drop the table and recreate it before reloading the rows. Do not use identity columns for tables that are accessed by applications that do not tolerate gaps in their numbering. Identity gaps are more of an issue with applications using identity column for auto-numbering employees, orders, and so on.

**Note:** Unity Director does not support tables that have identity columns. To deal with this restriction for existing identity column tables, you can use an ALTER TABLE request to remove the identity column attribute from a table column without having to drop that column.

## Performance Considerations for Identity Columns

Identity columns have minimal cost with respect to system performance. However, the initial load of an identity column table might create a performance decrement because every AMP that has rows for an identity column table must reserve a range of numbers at about the same time.

When the table to be updated has a primary index, there is a performance degradation for INSERT and UPDATE operations if the primary index is an identity column. When an identity column is defined on a table column other than the primary index, the performance cost is negligible.

If you write applications that access tables with identity columns, coding application SQL requests to return values from the identity column improves the performance of open access products.

## Object Identifier Columns

For each BLOB, CLOB, or XML column created for a table, Teradata Database automatically stores a 40-byte or 45-byte multiple field pointer in the row to the subtable that stores the actual BLOB, CLOB, or XML data for that column (see “[Sizing a LOB or XML Subtable](#)” on page 861). This value, referred to as an OID, is stored in VARBYTE format.

If a BLOB, CLOB, or XML column is null, then so is its OID.

The column sizes reported here are for OIDs stored on disk in rows. They do not include, and are not the same as, the sizes of OIDs passed as inline or deferred host parameters as part of a USING request modifier row.

## System-Derived and System-Generated Column Data Types

The following table lists the data types for several system-derived and system-generated column data types:

Derived or Generated Column	Data Type	Default Title
OID	VARBYTE	None
ROWID	<ul style="list-style-type: none"> <li>• BYTE(10) for 2-byte partitioning</li> <li>• BYTE(16) for 8-byte partitioning</li> </ul>	<p>None.</p> <p>There is no default title for the ROWID keyword because you cannot specify it in the select list of any DML request. You can only specify ROWID in a CREATE JOIN INDEX DDL request.</p>

Derived or Generated Column	Data Type	Default Title
Identity column	<p>Any of the following.</p> <ul style="list-style-type: none"> <li>• BYTEINT</li> <li>• DECIMAL(<math>n,0</math>) The scale for a DECIMAL identity column must be 0.</li> <li>• INTEGER</li> <li>• NUMBER(<math>n,0</math>) Only a fixed NUMBER type is permitted for identity columns, and its scale must be 0.</li> <li>• NUMERIC(<math>n,0</math>)</li> <li>• SMALLINT</li> <li>• BIGINT</li> </ul> <p>The upper limits for DECIMAL and NUMERIC types are the following.</p> <ul style="list-style-type: none"> <li>• DECIMAL(18,0)</li> <li>• NUMERIC(18,0)</li> </ul> <p>This is true even when the DBS Control flag MaxDecimal is set to 38 (see <i>SQL Data Types and Literals</i>).</p> <p>You can define an identity column with more than 18 digits of precision, or even as a BIGINT or NUMBER(<math>n,0</math>) type, without the CREATE TABLE or ALTER TABLE request aborting, but the values generated by Teradata Database for the identity column remain limited to the DECIMAL(18,0) type and size.</p> <p>A table that is managed by Unity Director cannot have an identity column. Unity Director instead uses its own mechanism to generate “identity column” values. See the Unity Director documentation for details.</p>	Default title for the column designated as an identity column.
PARTITION	INTEGER	PARTITION
PARTITION#Ln	<ul style="list-style-type: none"> <li>• INTEGER for the 2-byte form of PPI.</li> <li>• BIGINT for the 8-byte form of PPI.</li> </ul>	PARTITION#Ln

## Data Type Considerations

Different column data types occupy different amounts of disk space. This topic examines the various Teradata Database data types and indicates their absolute sizes.

The information presented here is for sizing purposes only. For specific usage information about the various data types supported by Teradata, see *SQL Data Types and Literals*.

## Data Types And Hashing

The data types for the primary index column set also have an important effect on how rows hash. For example, a primary index value typed DECIMAL with one precision generally hashes to a different AMP than the same primary index value typed DECIMAL with a different precision (see “[Hashing and Data Types](#)” on page 250 for additional information).

## Data Type Size Differences For Packed64 and Aligned Row Format Architectures

Several data types have different sizes depending on whether the rows of a system are formatted using a packed64 or an aligned row format. The row size increases for aligned row formats are not all due to increased data type sizes. Byte alignment issues also play a significant role in this increase. Furthermore, neither study examined the effects of the increased data block header size introduced by WAL on storage (see “[Byte Alignment](#)” on page 770 for more information).

The following table lists the predefined data types that have different disk storage sizes for packed64 and aligned row formats. If a data type is not listed in the table, then its allocated size is identical for packed64 format and aligned row format systems.

Data Type	Packed64 Format Size (bytes)	Aligned Row Format Size (bytes)	Allocated Aligned Format Size (bytes)
TIME	6	4	8
TIMESTAMP	10	4	12
INTERVAL DAY TO SECOND	10	4	12
INTERVAL MINUTE TO SECOND	6	4	8
INTERVAL SECOND	6	4	8

## Numeric Data Types

### Differences Between Exact and Approximate Predefined Numeric Data Types

The ANSI/ISO SQL:2011 standard defines two families of predefined numeric data types.

- Exact
- Approximate

An exact predefined numeric data type is one that can represent a value exactly. Exact numeric types are divided between the true integer types, which specify a precision but not a scale, and the fractional types, which can specify both a precision and a scale.

Teradata Database also supports various forms of the NUMBER data type, which can be used to represent both exact numeric values and floating point numeric values, depending on the syntax used to define the value. By the definitions the ANSI/ISO SQL:2011 standard uses for exact and approximate predefined numeric data types, the NUMBER data type as implemented by Teradata is neither an exact nor an approximate numeric type.

## Exact Numeric Data Types

The definition of an exact numeric data type used by the ANSI/ISO SQL:2011 standard states that to be an exact numeric, the type must have a precision and a scale expressed either in base 2 or base 10.

According to the ANSI/ISO SQL:2011 standard, the following types are members of the exact predefined numeric set.

The following types are members of the set of true integers.

- BIGINT
- INTEGER
- SMALLINT

The following types are members of the set of fractional exact numeric types.

- DECIMAL
- NUMERIC

Teradata also supports the following exact predefined numeric data type extensions to the ANSI/ISO SQL:2011 standard.

Data Type	Type Family
BYTEINT	true integer
NUMBER( $p$ )	fractional exact numeric

**Note:** You can use the NUMBER type to represent both fixed point and floating point values, depending on the syntax you use to specify the value.

See *SQL Data Types and Literals* for more detailed information about the various exact numeric data types supported by Teradata Database.

## Approximate Numeric Data Types

The following types are members of the ANSI/ISO SQL:2011 approximate predefined numeric set.

- DOUBLE PRECISION
- FLOAT
- REAL

Although FLOAT, REAL, and DOUBLE PRECISION are distinct types as defined by the ANSI/ISO SQL:2011 standard, Teradata Database treats them as if they are equivalent to one another. The approximate numeric data types represent floating point numbers.

See *SQL Data Types and Literals* for more detailed information about the various approximate numeric data types supported by Teradata Database.

## Floating Point NUMBER Types

The requirements of an exact NUMBER include that for any addition, subtraction, multiplication, and division operations, the result must be an exact numeric value.

The NUMBER( $p,s$ ) syntax is not exact because the results of the previously mentioned dyadic operations do not produce an exact numeric result value.

Teradata supports the following predefined floating point numeric data type extensions to the ANSI/ISO SQL:2011 standard.

Data Type	Reason the Type is Not Exact	Type Family
NUMBER( $p,s$ )	The result of the dyadic operations +, -, *, and / on NUMBER( $p,s$ ) values do not produce an exact numeric value when they operate on the NUMBER( $p,s$ ) type.	fractional floating point numeric
NUMBER	NUMBER and NUMBER(*) do not specify a scale.	The exact floating point NUMBER types can be used anywhere an approximate numeric type is used, but NUMBER values are stored as exact values in the same way as the exact NUMBER types and with the same accuracy.
NUMBER(*)		
NUMBER(*, $s$ )	NUMBER(*, $s$ ) has a precision that cannot be expressed as an integer for either base 2 or base 10 arithmetic.	Floating point NUMBER types are not approximate types.

**Note:** You can use the NUMBER type to represent both fixed point and floating point values, depending on the syntax you use to specify the value.

See *SQL Data Types and Literals* for more detailed information about the NUMBER data type.

## Integer Data Types

BYTEINT: 1 Byte (All Platforms)	
Bit 0	Bits 1 - 7
Sign	-128 — +127

**SMALLINT: 2 Bytes (All Platforms)**

Bit 0	Bits 1 - 15
Sign	-32,768 — +32,767

**INTEGER: 4 Bytes (All Platforms)**

Bit 0	Bits 1 - 31
Sign	-2,147,483,648 — +2,147,483,647

**BIGINT: 8 Bytes (All Platforms)**

Bit 0	Bits 1 - 63
Sign	-9,223,372,036,854,775,808 — +9,223,372,036,854,775,807

The following table indicates the alignment requirements for each of the true integer numeric data types and their respective sizes on packed64 and aligned row format systems:

Data Type	Alignment	Packed64 Size (bytes)	Aligned Row Format Size (bytes)
BYTEINT	1	1	1
SMALLINT	2	2	2
INTEGER	4	4	4
BIGINT	8	8	8

## Non-INTEGER Numeric Data Types

The following table indicates the alignment requirements for each of the non-true integer numeric data types and their respective sizes on platforms that use packed64 versus aligned row storage formats:

Data Type	Range of $n$	Packed64 Size (bytes)	Aligned Row Size (bytes)	Allocated Size for Aligned Row Format (bytes)
DECIMAL( $n$ ) NUMERIC( $n$ )	$1 \leq n \leq 2$	1	1	1
	$3 \leq n \leq 4$	2	2	2
	$5 \leq n \leq 9$	4	4	4
	$10 \leq n \leq 18$	8	8	8
	$19 \leq n \leq 38$		4 Alignment is 4 because decimal numbers in this range are stored internally as four 32-bit integers on all systems.	16
NUMBER( $n$ ) NUMBER( $n,s$ )	$0 \leq n \leq 38$ and 0	0 - 18	0 - 18	0 - 18
FLOAT REAL DOUBLE PRECISION	Not applicable	8	8	8
NUMBER NUMBER(*) NUMBER(*,s)	Not applicable	0 - 18	0 - 18	0 - 18

#### FLOAT/REAL/DYNAMIC PRECISION: 8 Bytes (All Platforms)

Bit 0	Bits 1 - 10	Bits 11 - 63
	Exponent	Mantissa
Sign	$2 * 10^{-307} — 2 * 10^{308}$	

#### NUMBER: 0- 18 Bytes (All Platforms)

Byte 0	Bytes 1 - 17
Exponent	Mantissa



## Byte Data Types

BYTE(n): 1 - 64 000 Bytes Fixed (n bytes) (All Platforms)		
Byte 1	...	Byte 64,000
1 — 64,000		

VARBYTE(n): 1 - 64,000 Bytes Variable ( $\leq n$ bytes) (All Platforms)		
Byte 1	...	Byte 64,000
1 — 64,000		

LONG VARBYTE(n): 1 - 64,000 Bytes Fixed (equivalent to VARBYTE (64000) (All Platforms)		
Byte 1	...	Byte 64,000
1 — 64,000		

BLOB(n): 8 bytes - 2,097,088,000 Bytes Variable (All Platforms)			
Chunk	Bytes 1 - 8	...	Byte 64,000
1	length	1 — 64,000	
...	length	1 — 64,000	
m	length	1 — m	

The following table shows the alignment and size for byte types on both platforms.

Data Type	Packed64 Size	Aligned Row Size	Allocated Aligned Row Size
BYTE(n)	n	1	n
VARBYTE(n)	$\leq n$	$\leq n$	$\leq n$
LONGVARBYTE(n)	$\leq n$	$\leq n$	$\leq n$
BLOB(n)	n	1	n

## Date**Time** Data Types

DATE: 4 Bytes (All Platforms)

Year	Month	Day
AD January 1, 1 — AD December 31, 9999		

TIME: 6 Bytes (Packed64 Platforms)  
8 Bytes (Aligned Row Platforms)

Hour	Minute	Seconds
hh:mm:ss[.ssssss]		

TIMESTAMP: 10 Bytes (Packed64 Platforms)  
12 Bytes (Aligned Row Platforms)

Year	Month	Day	Hour	Minute	Second
yyyy-mm-dd hh:mm:ss					

TIME WITH TIME ZONE: 8 Bytes (All Platforms)

Hour	Minute	Second	Timezone_Hour	Timezone_Minute
hh:mm:ss.ssssss±hh:mm				

TIMESTAMP WITH TIME ZONE: 12 Bytes (All Platforms)

Year	Month	Day	Hour	Minute	Second	Timezone_Hour	Timezone_Minute
yyyy-mm-dd hh:mm:ss±hh:mm							

The following table indicates the alignment requirements for each of the ANSI/ISO Date**Time** data types and their respective sizes on packed64 and aligned row platforms. When the size of a stored value for a type differs for packed64 and aligned row platforms, the differing table cells are shaded.

Data Type	Packed64 Size (bytes)	Aligned Row Size (bytes)	Allocated Aligned Row Size (bytes)
DATE	4	4	4
TIME	6	4	8
TIMESTAMP	10	4	12
TIME WITH TIME ZONE	8	4	8
TIMESTAMP WITH TIME ZONE	12	4	12

## Interval Data Types

INTERVAL YEAR: 2 Bytes (All Platforms)			
Years Precision			
1	2	3	4
Range			
-9 — 9	Not used		
-99 — 99 (default)	Not used		
-999 — 999	Not used		
-9999 — 9999			

INTERVAL YEAR TO MONTH: 4 Bytes (All Platforms)				
Years Precision				Months Precision
1	2	3	4	
Range				Range
-9 — 9	Not used			
-99 — 99	Not used			
-999 — 999	Not used			

INTERVAL YEAR TO MONTH: 4 Bytes (All Platforms)				
Years Precision				Months Precision
1	2	3	4	
Range				Range
-9999 — 9999				00 — 11

INTERVAL MONTH: 2 Bytes (All Platforms)			
Months Precision			
1	2	3	4
Range			
-9 — 9	Not used		
-99 — 99	Not used		
-999 — 999	Not used		
-9999 — 9999			

INTERVAL DAY: 2 Bytes (All Platforms)			
Days Precision			
1	2	3	4
Range			
-9 — 9	Not used		
-99 — 99	Not used		
-999 — 999	Not used		
-9999 — 9999			

Days Precision				Hours Precision
1	2	3	4	
Range				
-9 — 9	Not used			
-99 — 99 (default)	Not used			
-999 — 999	Not used			
-9999 — 9999	00 — 23			

Days Precision				Minutes Precision
1	2	3	4	
Range				
-9 — 9	Not used			00 — 59
-99 — 99 (default)	Not used			
-999 — 999	Not used			
-9999 — 9999				

INTERVAL DAY TO SECOND: 10 Bytes (Packed64 Platforms) 12 Bytes (Aligned Row Platforms)									
Days Precision				Seconds Precision					
1	2	3	4	1	2	3	4	5	6
Range				Range					
-9 — 9	Not used			00.0 — 59.9	Not used				
-99 — 99 (default)	Not used			00.00 — 59.99	Not used				

INTERVAL DAY TO SECOND: 10 Bytes (Packed64 Platforms) 12 Bytes (Aligned Row Platforms)												
Days Precision				Seconds Precision								
1	2	3	4	1	2	3	4	5	6			
Range		Range										
-999 — 999		Not used		00.000 — 59.999			Not used					
-9999 — 9999				00.0000 — 59.9999					Not used			
				00.00000 — 59.99999						Not used		
				00.000000 — 59.999999 (default)								

INTERVAL HOUR: 2 Bytes (All Platforms)				
Hours Precision				
1	2	3	4	
Range				
-9 — 9		Not used		
-99 — 99 (default)			Not used	
-999 — 999				Not used
-9999 — 9999				

Hours Precision				Minutes Precision
1	2	3	4	
Range				
-9 — 9	Not used			00 — 59
-99 — 99 (default)		Not used		
-999 — 999			Not used	
-9999 — 9999				

Hours Precision				Seconds Precision							
1	2	3	4	1	2	3	4	5	6		
Range				Range							
-9 — 9	Not used			00.0 — 59.9	Not used						
-99 — 99 (default)		Not used		00.00 — 59.99		Not used					
-999 — 999			Not used	00.000 — 59.999			Not used				
-9999 — 9999				00.0000 — 59.9999				Not used			
				00.00000 — 59.99999					Not used		
				00.000000 — 59.999999						Not used	

Minutes Precision			
1	2	3	4
Range			
-9 — 9	Not used		

INTERVAL MINUTE: 2 Bytes (All Platforms)			
Minutes Precision			
1	2	3	4
Range			
-99 — 99		Not used	
-999 — 999			Not used
-9999 — 9999			

INTERVAL MINUTE TO SECOND: 6 Bytes (Packed64 Platforms) 8 Bytes (Aligned Row Platforms)									
Minutes Precision				Seconds Precision					
1	2	3	4	1	2	3	4	5	6
Range				Range					
-9 — 9	Not used			0.0 — 59.9		Not used			
-99 — 99 (default)		Not used		0.00 — 59.99			Not used		
-999 — 999			Not used	0.000 — 59.999				Not used	
-9999 — 9999				0.0000 — 59.9999					Not used
				0.00000 — 59.99999					
				0.000000 — 59.999999					

INTERVAL SECOND: 6 Bytes (Packed64 Platforms) 8 Bytes (Aligned Row Platforms)									
Seconds Precision				Fractional Seconds Precision					
1	2	3	4	1	2	3	4	5	6
Range				Range					
-9 — 9	Not used			0.0 — 0.9		Not used			

INTERVAL SECOND: 6 Bytes (Packed64 Platforms) 8 Bytes (Aligned Row Platforms)												
Seconds Precision				Fractional Seconds Precision								
1	2	3	4	1	2	3	4	5	6			
Range				Range								
-99 — 99 (default)	Not used		0.00 — 0.99			Not used						
-999 — 999			Not used	0.000 — 0.999				Not used				
-9999 — 9999				0.0000 — 0.9999					Not used			
				0.00000 — 0.99999						Not used		
				0.000000 — 0.999999 (default)								

This table shows the alignments for interval types and their respective sizes on both platforms. When the size of the stored value for a type differs for the platforms, the cells are shaded.

Data Type	Packed 64 Size (bytes)	64-Bit Aligned Row Size (bytes)	Allocated Aligned Row Size (bytes)
INTERVAL YEAR	2	2	2
INTERVAL YEAR TO MONTH	4	2	4
INTERVAL MONTH	2	2	2
INTERVAL DAY	2	2	2
INTERVAL DAY TO HOUR	4	2	4
INTERVAL DAY TO MINUTE	8	2	8
INTERVAL DAY TO SECOND	10	4	12
INTERVAL HOUR	2	2	2
INTERVAL HOUR TO MINUTE	4	2	4
INTERVAL HOUR TO SECOND	8	4	8
INTERVAL MINUTE	2	2	2
INTERVAL MINUTE TO SECOND	6	4	8
INTERVAL SECOND	6	4	8

## Period Data Types

The following table indicates the alignment requirements for each of the Period data types and their respective sizes on aligned row format platforms. When the size of the stored value for a type differs for packed64 and aligned row format platforms, the differing table cells are shaded.

Data Type	Packed64 Size (bytes)	64-Bit Aligned Row Size (bytes)	Allocated Aligned Row Size (bytes)
PERIOD (DATE)	8	2	8
PERIOD (TIME(n))	12	2	16
PERIOD (TIME(n) WITH TIME ZONE)	16	2	16
PERIOD (TIMESTAMP(n))	20	2	24
PERIOD (TIMESTAMP(n) WITH TIME ZONE)	24	2	24

### PERIOD (DATE): 8 Bytes (All Platforms)

Beginning Date	Ending Date
AD January 1, 1 — AD December 30, 9999	AD January 2, 1 — AD December 31, 9999

### PERIOD (TIME (Precision)): 12 Bytes (Packed64 Platforms) 16 Bytes (Aligned Row Platforms)

Beginning Time (6 bytes - Packed64 Platforms) 8 bytes - Aligned Row Platforms)			Ending Time (6 bytes - Packed64 Platforms) 8 bytes - Aligned Row Platforms)		
Hour	Minute	Seconds	Hour	Minute	Seconds
hh:mm:ss[.ssssss]			hh:mm:ss[.ssssss]		

**PERIOD (TIME (Precision) WITH TIME ZONE): 16 Bytes (All Platforms)**

Beginning Time With Time Zone (8 bytes)					Ending Time With Time Zone (8 bytes)				
Hour	Minute	Second	Timezone_Hour	Timezone_Minute	Hour	Minute	Second	Timezone_Hour	Timezone_Minute
hh:mm:ss.ssssss±hh:mi					hh:mm:ss.ssssss±hh:mi				

**PERIOD (TIMESTAMP (Precision) ):20 Bytes (All Platforms)**

Beginning Time Stamp (10 bytes)						Ending Time Stamp (10 bytes)					
Year	Month	Day	Hour	Minute	Second	Year	Month	Day	Hour	Minute	Second
yyy-mm-dd hh:mm:ss						yyy-mm-dd hh:mm:ss					

**PERIOD (TIMESTAMP (Precision) ) When Ending Element Value is UNTIL\_CHANGED:11 Bytes (All Platforms)**

Beginning Time Stamp (10 bytes)						Ending Time Stamp (1 byte)	
Year	Month	Day	Hour	Minute	Second	UNTIL_CHANGED	
yyy-mm-dd hh:mm:ss						uc	

**PERIOD (TIMESTAMP (Precision) WITH TIME ZONE): 24 Bytes (All Platforms)**

Beginning Timestamp With Time Zone (12 bytes)								Ending Timestamp With Time Zone (12 bytes)							
Year	Month	Day	Hour	Minute	Second	Timezone_Hour	Timezone_Minute	Year	Month	Day	Hour	Minute	Second	Timezone_Hour	Timezone_Minute
yyyy-mm-dd hh:mi:ss±hh:mi								yyyy-mm-dd hh:mi:ss±hh:mi							

**PERIOD (TIMESTAMP (Precision) WITH TIME ZONE) When Ending Element Value is UNTIL\_CHANGED: 13 Bytes (All Platforms)**

Beginning Timestamp With Time Zone (12 bytes)								Ending Timestamp With Time Zone (1 byte)
Year	Month	Day	Hour	Minute	Second	Timezone_Hour	Timezone_Minute	UNTIL_CHANGED
yyyy-mm-dd hh:mi:ss±hh:mi								uc

## Character Data Types

**CHARACTER(n): 1 - 64,000 Bytes Fixed (All Platforms)**

Byte 1	...	Byte 64,000
1 — 64,000		

**VARCHAR(n): 1 - 64 000 Bytes Variable (All Platforms)**

Byte 1	Byte 2	...	Byte 64,000
Column offset			1 — 64,000

**LONG VARCHAR: 1 - 64,000 Bytes Fixed (equivalent to VARCHAR(64000) (All Platforms)**

Byte 1	Byte 2	...	Byte 64,000
Column offset			1 — 64,000

**CLOB(n): 8 bytes - 2 097 088 000 bytes Variable (All Platforms)**

Chunk	Bytes 1 - 8	...	Byte 64,000
1	length	1 — 64,000	
...	length	1 — 64,000	

CLOB(n): 8 bytes - 2 097 088 000 bytes Variable (All Platforms)			
Chunk	Bytes 1 - 8	...	Byte 64,000
m	length	1 — n  If a CLOB is composed of Unicode characters, its upper limit is 1,048,544,000 bytes.  In both cases, the maximum supported size is slightly less than 2 GB or 1 GB, respectively.	

CHARACTER(n) CHARACTER SET GRAPHIC : 1 - 32,000 Bytes Fixed (All Platforms)			
Byte 1	...	...	Byte 32,000
1 — 32,000			

VARCHAR(n) CHARACTER SET GRAPHIC: 1 - 32,000 Bytes Variable (All Platforms)			
Byte 1	Byte 2	...	Byte 32,000
Column Offset			1 — 32,000

LONG VARCHAR CHARACTER SET GRAPHIC: 32,000 Bytes Fixed (equivalent to VARCHAR(n) CHARACTER SET GRAPHIC (32000) (All Platforms))			
Byte 1	Byte 2	...	Byte 32,000
Column Offset			1 — 32,000

The following table indicates the alignment requirements for the character data types and their respective sizes on packed64 and aligned row format platforms.

Data Type	Packed64 Size (bytes)	Aligned Row Size (bytes)	Allocated Aligned Row Size (bytes)
CHARACTER(n) CHARACTER SET LATIN	n	1	n
CHARACTER(n) CHARACTER SET KANJI			
CHARACTER(n) CHARACTER SET UNICODE	2n	2	2n
CHARACTER(n) CHARACTER SET GRAPHIC			
VARCHAR(n) CHARACTER SET LATIN	n	1	n

Data Type	Packed64 Size (bytes)	Aligned Row Size (bytes)	Allocated Aligned Row Size (bytes)
VARCHAR( $n$ ) CHARACTER SET UNICODE	$\leq 2n$	1	$\leq 2n$
VARCHAR( $n$ ) CHARACTER SET GRAPHIC			
VARCHAR( $n$ ) CHARACTER SET KANJI	$\leq n$	1	$n$
LONG VARCHAR CHARACTER SET LATIN	64,000	1	64,000
LONG VARCHAR CHARACTER SET KANJI1			
LONG VARCHAR CHARACTER SET UNICODE	32,000		32,000
LONG VARCHAR CHARACTER SET GRAPHIC			
LONG VARCHAR CHARACTER SET KANJISJIS			
CLOB( $n$ ) CHARACTER SET LATIN	$n$	1	$n$
CLOB( $n$ ) CHARACTER SET UNICODE	$\leq 2n$	2	$\leq 2n$

## XML/XMLTYPE Data Type

XML or XMLTYPE: 8 bytes - 2,097,088,000 Bytes Variable (All Platforms)

Chunk	Bytes 1 - 8	...	Byte 64,000
1	length	1 — 64,000	
...	length	1 — 64,000	
m	length	1 — m	

XML and XMLTYPE are synonyms.

The XML type has the same storage properties and limitations as the CLOB data type.

[“Character Data Types” on page 840](#) documents the properties of the CLOB data type.

## JSON Data Type

The JSON type stores data in an optimized format, depending on data size:

Column Size	Storage Location
≤64 KB (64,000 LATIN characters or 32,000 UNICODE characters)	All rows are stored in the table.
≥ 64 KB (64,000 LATIN characters or 32,000 UNICODE characters)	<p>Storage depends on the JSON data size:</p> <ul style="list-style-type: none"> <li>• JSON data &lt; 4 KB (4,096 bytes) is stored in the table.</li> <li>• JSON data &gt; 4 KB (4,096 bytes) is stored in a LOB subtable.</li> </ul> <p>A table may have some JSON data rows stored in the base table and other JSON data rows stored in a LOB subtable.</p>

If there is no column length specified, the default length is the maximum for the character set in use.

The JSON type has the same storage properties and limitations as the CLOB data type, except the maximum total size is 16 MB (16,776,192 LATIN characters and 8,388,096 UNICODE characters). “[Character Data Types](#)” on page 840 documents the properties of the CLOB data type.

## User-Defined Data Types

The required byte alignment and storage size of a UDT depends on how it is defined.

FOR this category of UDT ...	THE byte alignment and storage size are ...
Distinct	the same as that of the predefined data type on which it is defined.
Structured	<p>different, depending on whether they are defined on the following type of platform:</p> <ul style="list-style-type: none"> <li>• Packed64 format systems</li> <li>• Aligned row format systems</li> </ul> <p>See “<a href="#">Sizing Structured UDT Columns</a>” on page 792 for information about how to determine the storage size of a given UDT column.</p>

## Array Data Types

The storage size of an ARRAY/VARRAY depends on how it is defined.

Field	Description	Size (bytes)
Flag bits	<p>Used to configure the layout of the field.</p> <p>Bit 0: variable offset size.</p> <ul style="list-style-type: none"> <li>If 0, the variable offsets are 2 bytes each.</li> <li>If 1, the variable offsets are 4 bytes each.</li> </ul> <p>Bits 1 - 15: unused</p>	2
Last present element	<p>Index of the last present element in the array. This consists of 1 to <i>number_of_dimensions</i> integer values, one for each dimension.</p> <p>The index values are 0-based.</p> <p>If the array has uninitialized elements, the index values are equal to -1.</p>	$4 \times \text{number\_of\_dimensions}$
Presence bit array	<p>Presence bit array for the array elements.</p> <p>A variable length byte array with 1 bit per element laid out in row major order.</p> <p>0 means the element is null.</p> <p>1 means the element is not null.</p>	$\frac{\text{number\_of\_elements} + 7}{8}$
Variable offset array	<ul style="list-style-type: none"> <li>If the element type is a variable length type, an offset array is used to index into the element data.</li> </ul> <p>The offset array is the same as the offset array in the table row.</p> <ul style="list-style-type: none"> <li>If the element type is fixed length, the variable offset array is omitted.</li> </ul>	$2 \times \text{number\_of\_elements}$ or $4 \times \text{number\_of\_elements}$
Element value array	<p>All elements are stored in row major order.</p> <p>Variable length elements are stored in their actual size.</p>	$\text{number\_of\_elements} \times \text{element\_size}$

# Geospatial Data Types

Teradata Database provides these geospatial data types, which are stored in the SYSUDTLIB database:

- MBR
- MBB
- ST\_Geometry

For a full description of the geospatial data types, see *SQL Geospatial Types*.

## MBR

MBR returns the minimum bounding rectangle of a geometry. It is based on four DOUBLE PRECISION data types that store attributes of the MBR. MBR types require  $(4 \times 8) = 32$  bytes of storage.

The MBR data type is a Teradata extension to the ANSI/ISO SQL:2011 standard.

## MBB

The MBB type is the 3-D equivalent to the MBR type. MBB returns the 3-dimensional minimum bounding box around a 3-D spatial object. It is based on six DOUBLE PRECISION data types that store attributes of the MBB. MBB types require  $(6 \times 8) = 48$  bytes of storage.

## ST\_GEOMETRY

ST\_GEOMETRY can represent any of the following geospatial types that are defined in the ANSI/ISO SQL Multimedia and Application Packages standard. A column of type ST\_Geometry can contain any of these geospatial types:

- GeometryCollection
- GeoSequence
- LineString
- MultiLineString
- MultiPoint
- MultiPolygon
- Point
- Polygon

Note that the GeoSequence type is a Teradata extension to the ANSI/ISO standard.

Because ST\_GEOMETRY is based on the BLOB type, it is defined with a maximum size, as measured by its well-known binary representation, of 16 MB.

IF the geometry is ...	THEN the data is stored ...
≤10,000 bytes	within the row.
> 10,000 bytes	outside the row as a BLOB, with an in-row OID pointing to it (see “Object Identifier Columns” on page 820).

Teradata Database reserves space within the row for an additional 44 bytes of metadata whether the spatial object is stored within or outside the row.

## Related Topics

For complete information about the geospatial data types supported by Teradata Database, see [SQL Geospatial Types](#).

# Row Size Calculation

This topic describes a procedure for calculating the physical size of a row for a typical table stored on a system with a packed64 format architecture. The procedure does not account for BLOB, CLOB, or XML data, which are stored in 64 KB pieces in a subtable outside the base table row (see “Sizing Base Tables, LOB Subtables, XML Subtables, and Index Subtables” on page 858 and “Sizing a LOB or XML Subtable” on page 861), nor does it account for spool file rows, which can be approximately 1MB long.

Use the Row Size Calculation form (see [Appendix E: “Sample Worksheet Forms”](#) to record your calculations.

The following *employee* table row definition is used as a sample for this procedure:

Employee

EmpNum	SupEmpNum	DeptNum	JobCode	LName	FName	HireDate	BDate	SalAmt
PK, SA	FK	FK	FK	NN	NN	NN	NN	NN
INTEGER	INTEGER	INTEGER	SMALLINT	CHAR(20)	VARCHAR(30)	DATE	DATE	DECIMAL(10,2)

## Procedure for Packed64 Systems

Note the following points about sizing UDT columns.

- The size of a distinct UDT column is identical to the size of the underlying predefined data type for the UDT.
- The size of a structured UDT column is more difficult to determine and is not equivalent to the sums of the sizes of its individual underlying predefined data types. The calculation

is further complicated by the fact that structured types can be nested to a maximum of 512 levels.

See “[Sizing Structured UDT Columns](#)” on page 792 for details about the composition and sizing of structured UDTs.

Use “[Equation: Structured UDT Size for Packed64 Format System](#)” on page 795 to determine the sizes of each individual structured UDT.

This table used for this example has no LOB or UDT columns, so you would skip steps 14, 15, and 16 of the procedure, jumping from step 13 directly to step 17.

Follow this procedure to determine the physical size of a typical row for any given non-LOB table:

- 1 List all the varying length columns in the four columns labeled Variable Data Detail.
- 2 For each variable length column, estimate the average number of bytes expected.
- 3 Add the total number of bytes in varying length columns and record the figure in the column labeled  $SUM(a)$ .

In the example table, there is only one varying length column, FName, which is typed VARCHAR(30). Its average number of bytes is estimated to be 14, so the value for  $SUM(a)$  is 14.

- 4 Determine how many columns in the table there are for each fixed byte data type.

The example table has the following number of each fixed byte data type:

Data Type	Number of Columns
DATE	2
DECIMAL	1
INTEGER	3
SMALLINT	1

- 5 Enter these figures in the Number of Columns column next to each relevant data type. Remember that all row lengths must be an even number of bytes (see “[Byte Alignment](#)” on page 770), so be sure to take this into account.
- 6 Multiply the counts by the sizing factor provided for the type and enter the results under the Total column.
- 7 Enter the total byte counts for the fixed length character types in the  $SUM(n)$  column.
  - CHARACTER
  - BYTE
  - GRAPHIC

For the example, only one CHARACTER column with a byte count of 20 is found, so enter 20 as the result for  $SUM(n)$ .

- 8 Add the values for SUM(n) and SUM(a) and record them in the column labeled Logical Size.

For the example, the logical size is 64 bytes.

- 9 The overhead for any nonpartitioned primary index row is 14 bytes. For a PPI row, the overhead is 18 bytes. Those numbers are prerecorded for you in the column labeled Overhead. Use the appropriate column for the primary index type of the table.

- 10 Multiply the number of variable length columns by 2 to account for the 2-byte variable column offset pointers determined in step 1 of this procedure.

Write the value in the column labeled Variable Column Offsets.

For the example, there is only one variable column, so write the number 2 here ( $1 \times 2 = 2$ ).

- 11 Record the number of columns compressed on a non-null value.

- 12 Record the number of nullable columns.

- 13 Divide the sum of step 11 and step 12 by 8 and record the quotient of the operation.

For aligned row format systems, you will use this value again at step 20a.

The purpose of this step is to account for any required additional presence bits that might be required.

For the example, the calculation is  $3/8$ , so the quotient is 0.

- 14 Determine how many BLOB, CLOB, or XML columns are in the table and record the number under Number of Columns on the Row Size Calculation Form, Page 1 of 2. Multiply that number by 40 to determine the total row size taken up by BLOB, CLOB, and XML object IDs (OIDs).

- 15 Compute and record the sizes of any UDT columns on page 2 of 2 of the Row Size Calculation Form as follows:

a Column 1 records the name of the UDT being recorded.

b Column 2 records how many columns in the table have that type.

c Column 3 records the sizing factor for the UDT.

This is the physical size of the column, which is one of the following.

FOR this UDT type ...	The physical size is the ...
distinct	size of its underlying predefined data type.
structured	calculated size of the column as determined by “ <a href="#">Equation: Structured UDT Size for Packed64 Format System</a> ” on page 795.
ARRAY/VARRAY	<ul style="list-style-type: none"><li>size of its underlying predefined data type if it is a one-dimensional array.</li><li>calculated size of the column as determined by “<a href="#">Equation: Structured UDT Size for Packed64 Format System</a>” on page 795 if it is a multidimensional array.</li></ul>

- d Column 4 records the product of the number of columns and the sizing factor for the UDT.

- 16 Sum the UDT totals and record them on Page 1 of 2 of the Row Size Calculation Form in the cell labelled UDTs.
- 17 Add the integers recorded in the Total column and record the sum in the column labeled Physical Size.
- 18 If the sum is an uneven number, round it up to the next even number.  
For the example, the sum is 82, so no rounding is necessary.
- 19 Whether you continue or not depends on the addressing used by your system.

IF you are calculating the row size for this type of system ...	THEN ...
packed64	stop.
aligned row	continue to step 20.

- 20 Determine the byte alignment overhead for the row.
  - a Set the value of *64-bit\_byte\_alignment\_bit\_overhead* to 0 and consult the number of additional presence bits determined in Step 13 of this procedure.

IF the value recorded in Step 13 is an ...	THEN set <i>64-Bit_Byte_Alignment_Bit_Overhead</i> to this value ...
odd number AND the number of variable length columns in the row is $\geq 1$	1
even number	0

- b Determine the maximum alignment among all the fixed length columns in the row.  
Call this variable FA.
- c Determine the maximum alignment among all compressible columns in the row.  
Call this variable CA.
- d If there are no fixed length or value compressible columns in the row, set the value of FA or CA to 1.
- 21 Increment the value of *Byte\_Alignment\_Bit\_Overhead* by  $\text{MAX(FA,CA)} - 1$ .
- 22 Determine the maximum alignment of all variable length columns.  
Call this variable VA.  
If there are no variable length columns, set VA to 1.
- 23 Increment the value of *64-Bit\_Byte\_Alignment\_Bit\_Overhead* by  $(VA - 1)$ .
- 24 Add the value of *64-Bit\_Byte\_Alignment\_Bit\_Overhead* to the aligned row size determined by Steps 14 - 17 in this procedure.

25 Round up the value determined in Step 24 to the nearest multiple of 8.

This value is the row size in bytes for an aligned row format system.

Note that steps 20 - 25 are a conservative approximation of the row size for an aligned row format system.

## Procedure To Determine the Exact Row Size for Aligned Row Systems

The following procedure determines the exact row size for aligned row format systems.

- 1 Set the initial value for *RowSize* to 12 bytes and the initial value for *64-Bit\_Byte\_Alignment\_Bit\_Overhead* to 0 bytes.
- 2 Record the number of columns compressed on a non-null value.
- 3 Record the number of nullable columns.
- 4 Divide the sum of step 2 and step 3 by 8 and record the quotient of the operation. Call this variable *PB*.

The purpose of this step is to account for any required additional presence bits that might be required.

For example, if the ratio is  $\frac{3}{8}$ , then the quotient is 0.

- 5 Overwrite the value for *RowSize* as follows:

$$\text{RowSize} = \left( \frac{(\text{PB} + 7)}{8} \right) - 1$$

- 6 Record the number of variable length columns. Call this variable *VC*.
- 7 Determine the space required for the variable length columns offset array using the following pseudocode procedure:

```
IF VC > 0
  THEN
    IF (RowSize is odd)
      THEN
        RowSize += 1
        64BitOverhead += 1
      ENDIF
    RowSize = RowSize + 2 * (VC + 1)
  END IF
```

where  $+=$  is the assignment operator.

- 8 Determine the maximum alignment required for all fixed length columns. Call this variable *FA*.
- 9 Determine the size of all fixed length columns.

Call this variable *FS*.

- 10 Determine the maximum alignment required for all compressible columns.

Call this variable *CA*.

- 11 Determine the size of all compressible columns.

Call this variable *CS*.

- 12 Determine the maximum alignment required for all variable length columns.

This includes BLOB, CLOB, and XML columns, VARCHAR columns, and VARBYTE columns. Because BLOB, CLOB and XML values are stored in subtables outside of the row, this calculation should more correctly be referred to as a table size determination rather than a row size determination.

Call this variable *VA*.

- 13 Determine the size of all variable length columns.

Call this variable *VS*.

- 14 Calculate the actual size value (call it *FixedActual*) for fixed length columns using the following equation, where FP represents row size:

$$\text{FixedActual} = (\text{FP} + \text{FA} - 1) - (\text{FP} + \text{FA} - 1)\text{MOD}(\text{FA})$$

The modulo(*FA*) adjustment aligns the value of *FixedActual* to the nearest multiple of *FA*.

- 15 Calculate the fixed length column overhead (call it *B*) using the following equation:

$$B = \text{FixedActual} - \text{FP}$$

- 16 Set *RowSize* = *FixedActual* + *FS*.

Call this variable *CP*.

- 17 Calculate the actual size value (call it *CompressActual*) for compressible length columns using the following equation:

$$\text{CompressActual} = (\text{CP} + \text{CA} - 1) - (\text{CP} + \text{CA} - 1)\text{MOD}(\text{CA})$$

The modulo(*CA*) adjustment aligns the value of *CompressActual* to the nearest multiple of *CA*.

- 18 Calculate the value compressible column overhead (call it *C*) using the following equation:

$$C = \text{CompressActual} - \text{CP}$$

- 19 Set *RowSize* = *CompressActual* + *CS*.

- 20 Set *64-Bit\_Byte\_Alignment\_Bit\_Overhead* = *64-Bit\_Byte\_Alignment\_Bit\_Overhead* + *B* - *C*.

- 21 Adjust the values of *RowSize* and *64-Bit\_Byte\_Alignment\_Bit\_Overhead* if *B* + *C* > MAX(*FA*,*CA*) using the following pseudocode procedure:

```

IF( ( B+C ) > MAX( FA, CA ) )
THEN
    RowSize = RowSize - CA
    Byte_Alignment_Bit_Overhead = Byte_Alignment_Bit_Overhead - CA
END IF

```

- 22 Assign *RowSize* to *VP*.
- 23 Calculate the actual maximum alignment required for variable length columns using the following equation:

$$\text{VA\_Actual} = (\text{VP} + \text{VA} - 1) - (\text{VP} + \text{VA} - 1)\text{MOD}(\text{VA})$$

The modulo(*VA*) adjustment aligns the value of *VA\_Actual* to the nearest multiple of *VA*.

- 24 Set *Byte\_Alignment\_Bit\_Overhead* = *Byte\_Alignment\_Bit\_Overhead* + *VA\_Actual* - *VP*.
- 25 Set *RowSize* = *VA\_Actual* + *VS*.
- 26 Calculate the actual value for *RowSize* using the following equation:

$$\text{RowSize} = (\text{RowSize} + 7) - (\text{RowSize} + 7)\text{MOD}(8)$$

The modulo(8) adjustment rounds the value for *RowSize* upward to the nearest multiple of 8.

- 27 Calculate the actual value for 64-Bit\_Byte\_Alignment\_Bit\_Overhead using the following equation:

$$64\text{-Bit\_Byte\_Alignment\_Bit\_Overhead} = 64\text{-Bit\_Overhead} + (\text{RowSize} + 7) - (\text{RowSize} + 7)\text{MOD}(8)$$

The modulo(8) adjustment rounds the value for *64-Bit\_Byte\_Alignment\_Bit\_Overhead* upward to the nearest multiple of 8.

- 28 Calculate the total aligned row format size by adding the values of *RowSize* and *64-Bit\_Byte\_Alignment\_Bit\_Overhead* using the following equation:

$$64\text{-Bit\_Byte\_Alignment\_Bit\_Overhead} = \text{RowSize} + 64 - 64\text{-Bit\_Byte\_Alignment\_Bit\_Overhead}$$

## Sizing Databases, Users, and Profiles

Underestimating space requirements for databases, users, and profiles leads to performance problems. This is because Teradata Database requires free disk cylinders to enable the growth of permanent, temporary, and spool space as well as to enable the growth of permanent journal tables.

Remember that none of these tables can share cylinders when you make your permanent, temporary, and spool disk space assignments.

The best practice for the initial sizing of the disk space required by a database, user, or profile is to make a good estimate of the space these will require when a database, user, or profile is created, and then to modify those assignments at a later time using MODIFY DATABASE, MODIFY USER, or MODIFY PROFILE requests as appropriate.

You should always consider the following disk space requirements when making disk space assignments for databases, users, and profiles.

Teradata Database requires this type of disk space ...	For ...
spool	materializing volatile tables.
temporary	materializing global temporary tables. To materialize global temporary tables, temporary space must have enough empty disk cylinders to contain their rows.

## Reserving Disk Space for Spool

Teradata Database uses spool space as temporary storage for result rows that are returned for user requests. To ensure that space is always available for spool, you should consider reserving about 15% to 20% of the total available disk space as spool space. For example, you can create a database with a name like *spool\_reserve*, that is not be used to contain tables:

```
CREATE DATABASE spool_reserve AS
PERM = 2000000*(HASHAMP( )+1)
```

where the specified value for PERM space is roughly 15% to 20% of the total available disk space for the system, which is based on the multiplier of 2,000,000 bytes.

This specification uses the constant expression  $2,000,000 * (\text{HASHAMP}() + 1)$  to calculate the number of AMPs in the current system and then scales the PERM space for the *spool\_reserve* database to that size.

The application of the  $(\text{HASHAMP}() + 1)$  constant expression to specifying disk space requirements is particularly useful for spool space, because the more AMPs in a configuration, the more thinly spread the data, which means that more spool space is required per AMP.

You should always consider defining the spool space for a database, user, or profile using a constant expression to scale the amount of spool space assigned based on the number of AMPs on the system.

For example, you might specify the amount of spool space for database *nivv* as in the following CREATE DATABASE request.

```
CREATE DATABASE nivv AS SPOOL = 2e5 * (HASHAMP( ) + 1);
```

**Note:** After creating the database, the spool space does not change if you add more AMPs to your system because the size is based on the number of AMPs in the system at the time the database, user, or profile is created, not on the current configuration.

Also be aware that spool space is the only disk space requirement for profiles that you can specify using a constant expression. You can use constant expressions to specify permanent, temporary, and spool space for databases and users.

This guarantees that data tables never occupy more than roughly 80% to 85% of the total disk space. Because no data should be stored in the *spool\_reserve* database, Teradata Database can use its permanent space as spool space when necessary.

Data warehousing applications should consider reserving still more of the total disk space as reserved spool space because their SQL requests tend to generate larger spool files. Tactical

and OLTP applications can reserve less reserved spool space because their requests tend to generate smaller spool files.

## Modifying the Disk Space Currently Assigned to Databases, Users, or Profiles

Modifying the permanent, temporary, and spool space assignments for databases, user, and profiles is a simple operation using MODIFY DATABASE, MODIFY USER, and MODIFY PROFILE requests as appropriate.

You can use the system views listed here to help determine new values to for permanent, temporary, and spool space assignments. For more information about these views, see *Data Dictionary*.

- DBC.AllSpaceV[X]

Use this view to report disk space usage, including spool space, for any account, database, table, or user.

The following request reports how the space currently used by the department table is distributed on each AMP.

```
SELECT DatabaseName, TableName, AMP, CurrentPerm
FROM DBC.AllSpaceV
WHERE TableName = 'department'
ORDER BY 1,2,3;
```

DatabaseName	TableName	AMP	CurrentPerm
test	department	1-0	1,024
test	department	1-1	512
test	department	1-2	1,024
test	department	1-3	512
personnel	department	1-0	2,048
personnel	department	1-1	1,536
personnel	department	1-2	1,536
personnel	department	1-3	1,536
user1	department	1-0	2,048
user1	department	1-1	1,536
user1	department	1-2	1,536
user1	department	1-3	1,536

The following request reports the values for the *MaxPerm* and *CurrentPerm* columns for each table contained by *user*. Because *user* only contains one table, *employee*, the request only returns information about *employee* and *All*, where the *All* “table” represents all of the tables contained by the specified database or user. In this case, *All* represents all of the tables contained by the user named *user*.

The *MaxPerm* value for *All* is the amount of permanent space defined for *user*. Because *user* contains only one table, the number of bytes on each AMP is the same for both *All* and *employee*. All of the reported values represent the size of the *employee* table.

Note that *employee* returns 0 bytes in the *MaxPerm* column because tables do not have MaxPerm space. Only databases and users have MaxPerm space, represented by *All*.

```
SELECT Vproc, TableName (FORMAT 'X(20)'), MaxPerm, CurrentPerm
FROM DBC.AllSpaceV
WHERE DatabaseName = user
ORDER BY TableName, Vproc;
```

Vproc	TableName	MaxPerm	CurrentPerm
0	All	2,621,440	64,000
1	All	2,621,440	64,000
2	All	2,621,440	112,640
3	All	2,621,440	112,640
...	...	...	...
0	employee	0	41,472
1	employee	0	41,472
2	employee	0	40,960
3	employee	0	40,960
...	...	...	...

- DBC.DiskSpaceV[X]

Use this view to report disk space usage, including spool space, for any account, database, or user.

The following request reports the permanent disk space across all AMPs.

```
SELECT AMP, DatabaseName, CurrentPerm, MaxPerm
FROM DBC.DiskSpaceV;
```

AMP	DatabaseName	CurrentPerm	MaxPerm
.	.	.	.
.	.	.	.
0-0	stst14	0	125,000
0-0	ud12	0	125,000
1-0	atest	1,536	125,000
1-0	a1	0	247,500
1-0	btest	3,584	5,000
1-0	b2test	49,664	250,000
.	.	.	.
.	.	.	.
1-1	atest	1,536	125,000
1-1	a1	0	247,500
1-1	btest	3,584	5,000
1-1	b2test	50,688	250,000
.	.	.	.
.	.	.	.
1-2	atest	1,536	125,000

Similarly, you can submit a request like the following to calculate the percentage of space used by a particular database. Note that the request uses a NULLIFZERO specification to avoid a divide by zero exception.

```
SELECT DatabaseName, SUM(MaxPerm), SUM(CurrentPerm),
((SUM (CurrentPerm))/NULLIFZERO (SUM(MaxPerm)) * 100)
(FORMAT 'zz9.99%', TITLE 'Percent // Used')
```

```
FROM DBC.DiskSpaceV
GROUP BY 1
ORDER BY 4 DESC;
```

This request reports the following information from *DBC.DiskSpaceV*.

DataBaseName	Sum(MaxPerm)	Sum(CurrentPerm)	Percent Used
Finance	1,824,999,996	1,796,817,408	98.46%
DBC	12,000,000,006	8,887,606,400	73.98%
Spool_Reserve	2,067,640,026	321,806,848	15.56%
CrashDumps	300,000,000	38,161,408	12.72%
SystemFE	1,000,002	70,656	7.07%

- *DBC.TableSizeV[X]*

Use this view to report disk space usage excluding spool space, for any account or table.

The following request reports the total disk space currently used by the *employee* table with its peak space usage.

```
SELECT SUM(PeakPerm), SUM(CurrentPerm)
FROM DBC.TableSizeV
WHERE TableName = 'employee';

Sum(PeakPerm)      Sum(CurrentPerm)
-----          -----
260,608           260,608
```

In this case, the 2 sums match, indicating that the disk space currently used by the *employee* table is also its peak space usage.

The following request reports poorly distributed tables by returning the CurrentPerm values for *table\_2* and *table\_2\_nusi*, the only tables contained by *user* across all AMPs.

```
SELECT Vproc, TableName (FORMAT 'X(20)'), CurrentPerm, PeakPerm
FROM DBC.TableSizeV
WHERE DatabaseName = user
ORDER BY TableName, Vproc;
```

Vproc	TableName	CurrentPerm	PeakPerm
0	table_2	41,472	53,760
1	table_2	41,472	53,760
2	table_2	40,960	52,736
3	table_2	40,960	52,736
4	table_2	40,960	52,760
5	table_2	40,960	52,760
6	table_2	40,960	52,272
7	table_2	40,960	52,272
0	table_2_nupi	22,528	22,528
1	table_2_nupi	22,528	22,528
2	table_2_nupi	71,680	71,680
3	table_2_nupi	71,680	71,680
4	table_2_nupi	9,216	9,216
5	table_2_nupi	9,216	9,216
6	table_2_nupi	59,392	59,392

7	<i>table_2_nupi</i>	59,392	59,392
---	---------------------	--------	--------

The report indicates that *table\_2* is evenly distributed across the AMPs, but that *table\_2\_nusi* is not, with its current permanent space clumped into spaces of 9,216 bytes, 22,528 bytes, 59,392 bytes, and 71,680 bytes across 2 AMPs each.

## Different Disk Space Views Return Different Results

It is important to understand that because the *DBC.AllSpaceV*, *DBC.DiskSpaceV*, and *DBC.TableSizeV* system views report different information, using superficially identical requests to return information often returns conflicting results. For example, while *DBC.AllSpaceV* and *DBC.DiskSpaceV* include information about spool space in their reports, *DBC.TableSizeV* does not.

Similarly, selecting SUM(CurrentPerm) from each of these views returns different results. For example, requesting SUM(CurrentPerm) from *DBC.AllSpaceV* returns a sum of *All* “tables,” representing the database or user total *plus* the sum of each table in the database or user. Submitting the same request using *DBC.TableSizeV* returns sums for all tables *except* for *All*, and submitting the query using *DBC.DiskSpaceV* returns sums *only* for *All*.

The following requests illustrate the different results the same request returns for the same information, but using different system views to access the data.

- *DBC.AllSpaceV*

```
SELECT MAX(CurrentPerm), SUM(CurrentPerm)
FROM DBC.AllSpaceV
WHERE DatabaseName = user;
```

Maximum(CurrentPerm)	Sum(CurrentPerm)
-----	-----
112,640	1,308,672

The reported values represent the disk space for all of the tables contained by *user* plus the disk space for the *All* table.

- *DBC.DiskSpaceV*

```
SELECT MAX(CurrentPerm), SUM(CurrentPerm)
FROM DBC.DiskSpaceV
WHERE DatabaseName = user;
```

Maximum(CurrentPerm)	Sum(CurrentPerm)
-----	-----
112,640	654,336

The reported values represent the disk space for the *All* table.

- *DBC.TableSizeV*

```
SELECT MAX(CurrentPerm), SUM(CurrentPerm)
FROM DBC.TableSizeV
WHERE DatabaseName = user;
```

Maximum(CurrentPerm)	Sum(CurrentPerm)
71,680	654,336

The reported values represent the disk space for all of the tables contained by *user*, but *do not* include the *All* table.

Use the following table to determine which system views are appropriate for requesting disk space information for different database objects.

To report disk space information at this level ...	You should use this system view ...
table	DBC.TableSizeV
database, user, or profile	DBC.DiskSpaceV
tables plus database, user, or profile	DBC.AllSpaceV

## Sizing Base Tables, LOB Subtables, XML Subtables, and Index Subtables

Teradata Database data block sizes and alignments are somewhat variable. This permits a designer the flexibility of designing tables without having to spend time balancing data types, row sizes, and block sizes in order to optimize their storage.

### Critical Sizing Variables

The critical variables for sizing Teradata Database tables and indexes are the following:

- Row size
- Estimated cardinality
- Presence of fallback

Table headers are a fixed variable, but must be considered when you are evaluating the projected sizes of your tables. Ensure that you take multi-value compression into account when sizing table headers (see “[Compression Types Supported by Teradata Database](#)” on page 695).

### Sizing a Column-Partitioned Table

Column-partitioned tables have similar space usage as a table with row partitioning except for the impact of compression.

A column-partitioned table or join index has the potential to incur a large increase in size compared to table that is not column-partitioned if there are few physical rows in populated combined partitions or if there are many column partitions with ROW format and the subrows are narrow.

A container can only hold values of rows that have the same internal partition number and hash bucket value. The potentially increased size is because of the increased number of physical rows and the overhead of the row header for each physical row. It is far more common for a column-partitioned table to be smaller, and often significantly so, than a table that is not column-partitioned if COLUMN format is used for narrow column partitions and the table is not over-partitioned because of the reduced number of physical rows and autocompression.

Use the following general procedure to size a column-partitioned table.

- 1 Estimate the size of the table *without* column partitioning using the standard methods for calculating the size of a table.
- 2 Adjust your estimate based on the expected impact of autocompression, row header compression, and row header expansion.

You can estimate the necessary adjustment by measuring the impact of your proposed column partitioning scheme on some sample data.

## LOB and XML Sizing Variables

Each chunk of a BLOB, CLOB, or XML string stores an 8-byte length followed by as much as a 64 KB data fragment. LOBs are stored outside its base table row in a subtable. See “[Sizing a LOB or XML Subtable](#)” on page 861 for more information.

# Sizing Base Tables, Hash Indexes, and Join Indexes

Estimate the size of your base tables and hash and join indexes using the equations provided in “[Table Sizing Equations](#)” on page 860.

These equations assume a constant block size. Because block sizes can vary widely within a table, you can obtain more finely tuned estimates by computing values over a range of block sizes.

## Calculating a Typical Block Size

Depending on how you define your tables and the boundary conditions on their rows, the maximum size in bytes for a typical block can be estimated by the following equation:

$$\text{Typical block size} = \text{ROUNDUP}(\text{MAX} \times \frac{3}{4}, 512)$$

where:

This variable ...	Specifies ...
ROUNDUP	the ROUNDUP function.

This variable ...	Specifies ...
$\text{MAX} \times \frac{3}{4}$	three quarters of the maximum block size in bytes. The upper limit on this value is 255 sectors, or 130,560 bytes.
512	the rounded up sector boundary, which is the number of bytes per sector for the table.

## Sizing a Column-Partitioned Join Index

See “[Sizing a Column-Partitioned Table](#)” on page 858 for information about sizing a column-partitioned join index. The methodology for column-partitioned join indexes is identical to that used to size a column-partitioned table.

## Table Sizing Equations

The following parameter definitions are used with this equation set:

Parameter	Definition
Block Overhead	Block Header + Block Trailer = 72 bytes + 2 bytes = 74 bytes
Minimum Table Header Size	512 bytes
Typical Block Size	Number of bytes per block (see “ <a href="#">Calculating a Typical Block Size</a> ” on page 859).
NumAmps	Number of AMPs in the configuration.
RowCount	Estimated cardinality for the table.
Average RowSize	Physical row size for the table (see “ <a href="#">Row Size Calculation</a> ” on page 846).  Remember that all row lengths must be an even number of bytes (see “ <a href="#">Byte Alignment</a> ” on page 770), so be sure to take this into account.

$$\text{Rows per block (rounded down)} = \frac{\text{Typical block size} - 38}{\text{Row size}}$$

$$\text{Number of blocks (rounded up)} = \frac{\text{Row count}}{\text{Rows per block}}$$

$$\text{Number of table header bytes} = (\text{Number of AMPs})(512)$$

$$\text{Number of base table bytes (without fallback)} = (\text{Number of blocks} \times \text{Typical block size}) + \text{Number of header bytes}$$

$$\text{Number of base table bytes (with fallback)} = 2(\text{Number of blocks})(\text{Typical block size}) - \text{Number of header bytes}$$

## Example: Table, Hash, or Join Index on a Packed64 Format System

You have collected the following information.

Parameter	Value
Block size	5,120 bytes <b>Note:</b> Typical block size used.
Number of AMPs	20
Row count	1,000,000
Row size	100 bytes
Rows per block are rounded down. Number of blocks are rounded up.	

Calculate the size of this object when it is defined with fallback.

$$\text{Number of rows per block} = \frac{5,120 - 16}{100} = 51$$

$$\text{Number of blocks} = \frac{1,000,000}{51} = 19,608$$

$$\text{Number of table header bytes} = 20 \times 512 = 10,240$$

$$\text{Number of bytes in base table with fallback} = ((19,608 \times 5,120) \times 2) + 10,240$$

## Sizing a LOB or XML Subtable

One LOB or XML subtable is required for each column in a base table defined with a BLOB, CLOB, or XML data type. Depending on the size of each LOB or XML string, the cardinality of the subtable can exceed the cardinality of the base table. Estimate the size of your LOB and XML subtables using the equations provided in “[LOB and XML Subtable Sizing Equation](#)” on page 862.

Because LOBs and XML strings are variable length entities, there are no alignment issues for aligned row format systems.

### LOB and XML Subtable Rows and Skew

Each LOB or XML string is stored in 64 KB sections in a subtable, except for the row that stores the last section (because the number of bytes in a LOB or XML string is typically not an exact multiple of 64 KB). This means that for any LOB or XML string greater than 64 KB, there is not a 1:1 equivalence between the number of base table rows and the number of LOB or XML subtable rows supporting those base table rows. As a result, unless every LOB or XML string in a table has the identical size, it is likely that storage skew will occur.

IF the primary index for a base table is a ...	THEN ...
UPI	this skew does not affect performance in any way, but it is likely to produce an asymmetric storage effect on storage requirements.
NUPI	both of the following assertions are true: <ul style="list-style-type: none"> <li>Performance costs accrued by a skewed distribution of base table rows are likely to be further magnified.</li> <li>Storage skew is even more likely to occur than for tables with a UPI.</li> </ul>

## LOB and XML Subtable Sizing Equation

The following parameter definitions are used with this equation.

Parameter	Definition
BaseRowCount	Estimated cardinality for the base table.
OIDSize	<ul style="list-style-type: none"> <li>40 bytes for nonpartitioned primary index and 2-byte PPI tables.</li> <li>45 bytes for 8-byte PPI tables.</li> </ul>
RowOverhead	<p>The number of bytes devoted to subtable row overhead.</p> <ul style="list-style-type: none"> <li>If the base table has a non-partitioned primary index, the subtable row overhead is 12 bytes.</li> <li>If the base table has a partitioned primary index, the subtable row overhead is 16 bytes.</li> </ul>
AvgLOBSize	<p>Average LOB or XML string size for the table.</p> <p>Remember that all row lengths must be an even number of bytes (see “<a href="#">Byte Alignment</a>” on page 770), so be sure to take this into account.</p>

$$\text{LOB subtable size} = (\text{BaseRowCount}) \left( \text{OIDSize} \left( (\text{RowOverhead} + 64000) \left( \frac{\text{AvgLOBSize}}{64000} \right) \right) \right)$$

## LOB and XML Subtables and Fallback

The LOB or XML subtable size must be doubled if the base table for which the LOB or XML column is defined is fallback protected.

## Sizing a Unique Secondary Index Subtable

Estimate the size of your unique secondary indexes using the equation provided in “[USI Sizing Equation](#)” on page 863.

For details on USI subtable row layouts, see “[USI Subtable Row Structure](#)” on page 463.

Because the index is unique, there is one USI subtable row for each row in the base table, so the row counts are identical.

## USI Sizing Equation

The following parameter definitions are used with this equation:

Parameter	Definition
Index Value Size	<p>Size of the column set on which the USI is defined.</p> <p>This must include any trailing pad bytes added for alignment purposes.</p> <p>All row lengths must be an even number of bytes (see “<a href="#">Byte Alignment</a>” on page 770), so be sure to take this into account.</p>
Block Overhead	<p>Sum of the following factors.</p> <ul style="list-style-type: none"> <li>• Block headers and trailers</li> <li>• Row headers and trailers</li> <li>• USI rowID</li> <li>• Spare byte</li> <li>• Presence bit arrays</li> <li>• Base table rowID</li> </ul> <p>= 28 bytes for an nonpartitioned primary index table</p> <p>= 30 bytes for a PPI table</p>

## Packed64 Format Sizing Equations

$$\text{NPPI USI subtable size} = ((\text{Row Count}) \times (\text{Index Value Size} + 28))$$

$$\text{PPI USI subtable size} = ((\text{Row Count}) \times (\text{Index Value Size} + 30))$$

## Aligned Row Format Sizing Equations

$$\text{NPPI USI subtable size} = ((\text{Row Count}) \times (\text{Index Value Size} + 28))$$

$$\text{PPI USI subtable size} = ((\text{Row Count}) \times (\text{Index Value Size} + 30))$$

If fallback is defined for the base table, then double the calculated result.

Teradata Database implicitly creates a unique secondary index on any column set specified as PRIMARY KEY or UNIQUE, so you must take these indexes into consideration for your capacity planning as well. PRIMARY KEY and UNIQUE constraints are generally implemented as single-table join indexes for temporal tables (see *ANSI Temporal Table Support* and *Temporal Table Support* for details). You must also take any of these system-defined join indexes into account when doing capacity planning.

## Sizing a Non-Unique Secondary Index Subtable

Estimate the size of your non-unique secondary indexes using the equation provided in “[NUSI Sizing Equation](#)” on page 864.

For details on NUSI subtable row layouts, see “[NUSI Row Structure](#)” on page 468.

The number of base tables that can be referenced is limited by the maximum row size for the system and the length of the secondary index value. See “[NUSI Sizing Equation](#)” on page 864 for more information about sizing NUSI subtables.

### Special Considerations for NUSI Size Estimates

The number of AMPS in the configuration are an important factor in estimating the total size of any NUSIs defined on a base table, as summarized in the following table:

IF the number of AMPS is ...	THEN at least ...	AND the result is that ...
less than the number of rows per value	one row from each NUSI value is probably distributed to each AMP.	<ul style="list-style-type: none"><li>Every AMP has every value.</li><li>Every AMP has a subtable row for every value.</li></ul>
greater than the number of rows per value	some AMPS are missing some NUSI values.	<ul style="list-style-type: none"><li>Not every AMP has every value.</li><li>Not every AMP has a subtable row for every value.</li></ul>

### NUSI Sizing Equation

The following parameter definitions are used with this equation.

Parameter	Definition
Cardinality x 8	Each base table rowID is stored in a NUSI subtable.
Cardinality x 10	<ul style="list-style-type: none"><li>For a nonpartitioned primary index table, the row ID is 8 bytes long.</li><li>For a PPI table, the row ID is 10 bytes long.</li></ul> <p>This means that you must use the Cardinality * 8 factor for NUSI subtables for nonpartitioned primary index base tables and the Cardinality * 10 factor for NUSI subtables for PPI base tables.</p> <p>See “<a href="#">NUSI Sizing Equation for Nonpartitioned Primary Index Base Table</a>” on page 865 and “<a href="#">NUSI Sizing Equation for PPI Base Table</a>” on page 865.</p>
NumDistinct	The value is an estimate of the number of distinct NUSI subtable values and is based on each NUSI subtable having at least one index row per AMP for each distinct index value of a base table row stored on that AMP.
IndexValueSize	The number of index data bytes.

Parameter	Definition
NUSI Row Overhead	Sum of the following factors. <ul style="list-style-type: none"> <li>• Row headers and trailers</li> <li>• NUSI row rowID</li> <li>• Spare byte</li> <li>• Presence octets</li> </ul> = 18 bytes
MIN (NumAmps   Rows per value)	The lesser of the two parameters.
6	The number of bytes consumed by the 3 VARCHAR Offset fields that follow the Additional Overhead field.

If fallback is defined for the base table, then double the calculated result.

### NUSI Sizing Equation for Nonpartitioned Primary Index Base Table

$$\text{NUSI subtable size}_{\text{NPPI}} = 8 \times (\text{Cardinality}) + ((\text{NumDistinct}) \times (\text{IndexValueSize} + 18)) \times \text{MIN}(\text{NumAMPs} | \text{Rows per value})) + 6$$

### NUSI Sizing Equation for PPI Base Table

$$\text{NUSI subtable size}_{\text{PPI}} = 10 \times (\text{Cardinality}) + ((\text{NumDistinct}) \times (\text{IndexValueSize} + 18)) \times \text{MIN}(\text{NumAMPs} | \text{Rows per value})) + 6$$

## NUSI Space Considerations

The PERM space required during the creation of a NUSI might temporarily be much greater than the space occupied by the finished index as described by the following table.

IF you create this sort of table ...	THEN the following peak temporary PERM space usage factors apply ...
non-fallback or fallback with small AMP cluster size	Estimate the temporary PERM space required when building a NUSI using the following worst case estimation equation.  $\text{TemporarySpaceNF} = \text{Cardinality} \times (\text{LengthOfKey} + 30)$ where: <ul style="list-style-type: none"> <li>• <i>TemporarySpaceNF</i> is the size of the temporary PERM space required without fallback.</li> <li>• <i>Cardinality</i> is the number of rows in the base table.</li> <li>• <i>LengthOfKey</i> is the combined byte length of the key.</li> </ul>

IF you create this sort of table ...	THEN the following peak temporary PERM space usage factors apply ...
fallback	<p>Estimate the temporary PERM space required, assuming typical AMP cluster size, using the following equation.</p> $\text{TemporarySpaceFB} = \text{NUSISubtableSize} + \text{TemporarySpaceNF}$ <p>This is a very conservative space estimate. For typical AMP cluster sizes, peak usage exceeds the prediction made by the model.</p>

AMP cluster size, which determines the relative size of the backup subtables, is an important factor. For information about AMP clusters, see *Introduction to Teradata*. A typical AMP cluster size is 4 AMPs, but the valid range varies from 2 to 8 AMPs per cluster.

NUSIs do not use spool space and are built one subtable at a time.

## Sizing User-Defined Routines

You must consider the size of any user-defined functions, methods, and stored procedures stored within Teradata Database when you are doing capacity planning for your system. This includes any of the UDFs you create to enforce algorithmic compression and row-level security privileges.

There are no special sizing considerations for these routines beyond planning for the space they occupy on your system.

## Sizing a Reference Index Subtable

A reference index is an internal structure that the system creates whenever a referential integrity constraint is defined between tables using a PRIMARY KEY or UNIQUE constraint on the parent table in the relationship and a REFERENCES constraint on a foreign key in the child table.

The index subtable row contains a count of the number of references in the child, or foreign key, table to the PRIMARY KEY or UNIQUE constraint in the parent table.

A maximum of 64 referential constraints can be defined for a table.

Similarly, a maximum of 64 *other* tables can reference a *single* table. Therefore, there is a maximum of 128 reference indexes that can be stored *in the table header* per table.

The limit on reference indexes in the table header includes both references to and from the table and is derived from 64 references to other tables plus 64 references from other tables to the current table = 128 reference index descriptors.

However, the maximum number of reference indexes stored in the reference index subtable for a table is limited to 64, defining only the relationships between the table as a parent with its child tables.

Estimate the size of your reference indexes using the equation provided in [Reference Index Sizing Equation](#).

## Reference Index Sizing Equation

$$\text{RI Subtable Size} = (\text{NumDistinct}) \times (\text{FKLength} + \text{PresenceBitsOverhead} + \text{VarLengthOverhead} + 25)$$

If fallback is defined for the child table in the relationship, then double the calculated result.

The following parameter definitions are used with this equation.

Parameter	Definition
Row Count * 4	A count of the number of foreign key row references is stored in a reference index subtable. Each foreign key row count is 4 bytes long.
FKLength	<p>The length of a fixed length foreign key value in bytes.</p> <p>Use one of these parameters depending on the reference index in question.</p> <ul style="list-style-type: none"> <li>• If the FK column value is fixed, then use the length of the value.</li> <li>• If the FK column value is variable, then use the average length of the variable length Foreign Key values.</li> </ul>
NumDistinct	<p>The value is an estimate of the number of distinct foreign key subtable values.</p> <p>Exclude null foreign keys from this estimate.</p>
Presence Bits Overhead	<p>Use the following equation to calculate this parameter:</p> $\text{Overhead} = \frac{1 + \text{Number nullable FK fields}}{8}$ <p>If there are no presence bits, then the value for this parameter is 0.</p>
VarLength Overhead	<p>Use the following equation to calculate this parameter:</p> $\text{Overhead} = 2 \times (\text{Number variable length FK fields} + 1)$ <p>If there are no variable length foreign keys, then the value for this parameter is 0.</p>
RI Block Overhead	<p>Sum of the following factors.</p> <ul style="list-style-type: none"> <li>• Row length</li> <li>• RI row rowID</li> <li>• Spare byte</li> <li>• Presence octets</li> <li>• Offsets</li> <li>• Valid flag</li> <li>• Foreign key count</li> <li>• Reference array = 25 bytes</li> </ul>

## Sizing Spool Space

Spool space, which is a form of temporary space, is frequently overlooked in capacity planning, yet it is critical to the operations of the database. Spool space needs vary from table to table, user to user, application to application, and with frequency of use. Very large systems use even more spool than smaller systems. Unlike all other row types, spool file rows have a maximum length of approximately 1MB rather than 64KB.

Spool falls into these categories:

- Intermediate
- Output
- Persistent
- Volatile

### Intermediate Spool Space

Intermediate spool results are retained until no longer needed. You can determine when intermediate spool is flushed by examining the output of an EXPLAIN. The first step performed after intermediate spool has been flushed is designated “Last Use.”

### Output Spool Space

Output spool results are the final information returned for a query or the rows updated within, inserted into, or deleted from a base table. The length of time output spool is retained depends on the subsystem and various system conditions, as described in this table:

Subsystem/Condition	When Output Spool Is Released
BTEQ	Last spool response.
Embedded SQL	The open cursor is closed.
CLIV2	<ul style="list-style-type: none"><li>• ERQ received</li><li>• Function terminated</li></ul>
Session terminates asynchronously due to any number of conditions, including the following. <ul style="list-style-type: none"><li>• Job abort</li><li>• Timeout</li><li>• Logoff</li></ul>	At the time the termination occurs.
System restart	At the time the restart occurs.

### Persistent Spool Space

When Redrive protection is enabled, Teradata Database stores responses for sessions that participate in Redrive in non-fallback persistent spool tables. Persistent spools are not deleted following a Teradata restart or node failure. Persistent spools are retained until the SQL

request completes and the application has fully received the response. For details about Redrive protection, see *Database Administration*.

## Volatile Spool Space

The system uses volatile spool space for volatile tables. This is necessary because volatile tables do not have a persistent stored definition.

## Sources of Spool Space

Spool space is taken only from disk cylinders that are not being used for data. Data blocks and spool blocks cannot coexist on the same cylinder.

When spool is released, the file system returns the cylinders it was using to the free cylinder list.

## Spool Limits

If you find that queries do not run because they run out of spool space, then increase the spool assignment for the user or database having the problem using the MODIFY USER or MODIFY DATABASE statements, respectively. For the syntax and usage notes for these statements, see *SQL Data Definition Language*.

The maximum size for a spool row is approximately 1MB. This larger row size enhances DML operations that are limited by small spool rows.

The amount of spool space allocated to each user and database is assigned at CREATE USER or CREATE DATABASE time.

## Guidelines for Allocating Spool Space

Unless you reserve a pool of spool space, the space that is available for spool tends to disappear quickly. When value compressed data is spooled, the compression follows it into the spool, thus saving as much space as was saved by compressing the data on disk. When applications consume all the spool space allocated for a system, processing halts.

The following method is used as a guideline for allocating spool space by a large Teradata customer in the retail business:

- 1 Create a special database to act as a spool space reservoir.  
Allocate 2% of the total user space in the system for this database.
- 2 Assign roughly 0.25% of the total space to each user as an upper limit, ensuring that each receives at least as much space as the size of the largest table they access concurrently.

Consider the following factors to perform finer tuning of database, user, or profile spool allotment.

- Query size

The smaller the query, the less spool space required. If a particular user only performs small queries, then allocate less spool space to that user.

If a user performs many large queries, then allocate more spool to that user. With the exception of runaway queries, allocating more spool space to a user is never harmful as long as system resources are not wasted.

- Database size

The more AMPs in the configuration, the more thinly spread the data, so the more spool required per AMP.

Because of this, you should consider defining the spool space for a database or user using a constant expression to scale the amount of spool space assigned based on the number of AMPs on the system.

For example, you might specify the amount of spool space for a database as in the following CREATE DATABASE request.

```
CREATE DATABASE nivv AS SPOOL = 2e5 * (HASHAMP( ) + 1);
```

**Note:** After creating the database, the spool space does not change if you add more AMPs to your system because the size is based on the number of AMPs in the system at the time the database, user, or profile is created, not on the current configuration.

- Average spool use per user
- Query workload types

Decision support queries generally require more spool than OLTP queries.

- Number of concurrent users
- Number of concurrent queries permitted for any one user

Spool space is cumulative per user.

## Sizing a Query Capture Database

Query Capture Databases are used to analyze SQL queries for ways in which they can be better optimized. A QCD can occupy significant storage space, so it is important to understand how much capacity is required to perform query optimization analyses.

### Capacity Planning for a Query Capture Database

The following physical limits constrain the QCD functionality:

- Because QCD tables are ordinary Teradata relational tables, they are limited to a row length of 64 KB.

Remember that all row lengths must be an even number of bytes (see “[Byte Alignment](#)” on [page 770](#)), so be sure to take this into account.

- QCDs draw their disk space from PERM space just like any other persistent Teradata relational tables. Consider defining the spool space for a QCD using a constant expression to scale the amount of spool space assigned based on the number of AMPs on the system.

For example, you might specify the amount of spool space for a QCD as in the following CREATE DATABASE request.

```
CREATE DATABASE qcd AS SPOOL = 2e5 * (HASHAMP( ) + 1);
```

**Note:** After creating the QCD, the spool space does not change if you add more AMPs to your system because the size is based on the number of AMPs in the system at the time the QCD is created, not on the current configuration.

- The row length for tables in an aligned row format QCD is larger in order to align the rows on appropriate modulo(8) boundaries, just as it is for all other base table rows on an aligned row format system (see “[Row Structure for Aligned Row Format Systems](#)” on page 755).

You can estimate the magnitude of the raw data (without indexes) generated by a single INSERT EXPLAIN or DUMP EXPLAIN statement from the following equation:

$$\text{Raw data size} \equiv (2n + m + (t \times d))$$

where:

Equation element ...	Specifies the ...
$n$	size in bytes of the EXPLAIN modifier output for the same query. The approximation assumes that all captured data demographics can be accommodated within $2n$ bytes.
$m$	size in bytes of the SQL query text.
$t$	average size in bytes of the DDL text for the object.
$d$	number of tables and views in the query.

The approximation is slightly more complex when statistics are captured using INSERT EXPLAIN WITH STATISTICS. This equation applies only to INSERT EXPLAIN WITH STATISTICS. DUMP EXPLAIN does not capture statistics.

$$\text{Raw data size} \equiv (2n + m + (t \times d)) + 5000c$$

where:

Equation element ...	Specifies the ...
$n$	size in bytes of the EXPLAIN request modifier output for the same query. The approximation assumes that all captured data demographics can be accommodated within $2n$ bytes.
$m$	size in bytes of the SQL query text.
$t$	average size in bytes of the DDL text for the object.
$d$	number of tables and views in the query.
$c$	number of columns involved in range and explicit equality conditions in the query WHERE clause. If you do not specify WITH STATISTICS when you perform INSERT EXPLAIN, the value for $c$ is 0.

The coefficient 5,000 is used as the average size of statistics based on the following information:

- The 101 interval histograms used to store the statistics occupy 4,900 bytes per column. This count includes interval 0 in addition to intervals 1 through 100. Interval 0 contains column- or index-global statistics, while intervals 1 through 200 contain interval-specific column statistics. See *SQL Request and Transaction Processing* for further information.
- 100 bytes are added to account for an upward bound on miscellaneous bytes per column.

Storage overhead in bytes per column due to statistics  $\cong 4900 + 100 \cong 5000$

## Related Topics

For more information about query capture databases and query optimization analysis, see *SQL Request and Transaction Processing*.

# Sizing Table Space Empirically

The most accurate method for sizing table space for a new configuration is to create a table with a definition as similar as possible to the legacy table you want to port to Teradata Database. You cannot just extrapolate the space information from your legacy system and expect the result to accurately estimate your Teradata Database table space needs because commercial database vendors all store their data in different ways.

## General Procedure

Use one of the load utilities to load the empty table with a representative sample of rows that constitutes a known percentage of the entire cardinality of the legacy table.

For the first iteration of testing, do not define any secondary indexes on the table. Analyze the space characteristics for this test table, add a secondary index, and repeat the task. Continue this process until you are reasonably certain that you have captured the characteristics of the base table and all the secondary indexes you initially want to define on it.

Extrapolate the total spool space required from an examination of the smaller temporary table. The scaled up data should provide a reliable picture of the initial space requirements for the table and its indexes.

## Explicit Procedure

The following table provides a more explicit procedure:

- 1 Create a new base table on the Teradata platform with a definition that matches the definition for the legacy table it is replacing as closely as possible.  
Do not define any secondary indexes on this base table at this stage of the process.
- 2 Use one of the Teradata load utilities to load the newly defined base table with a representative sample of rows from the legacy table. The loaded sample should not only

reflect the demographics of the entire population of the legacy table, but also be a fairly precise percentage of the cardinality for that table.

- 3 Query the system view *DBC.TablesizeV* for the space occupied by the new base table. “[Querying DBC.TableSizeV](#)” on page 873 provides a representative SQL query for you to use for this purpose.
- 4 Record the number of bytes returned by the query.
- 5 Add a secondary index to the table.
- 6 Query the system view *DBC.TablesizeV* again for the space occupied by the new base table and the new secondary index.
- 7 Record the number of bytes returned by the query.
- 8 The arithmetic difference between the numbers you recorded in step 4 and step 7 is the size of the newly defined secondary index.
- 9 Iterate step 5 and step 6 until you have finished adding all the secondary indexes you anticipate defining for this table.
- 10 Repeat the procedure with another legacy table until you have estimated the initial size of your entire Teradata database.
- 11 You should also evaluate the sizes of any hash or join indexes (and any secondary indexes defined on join indexes) you anticipate using with the new database. Because hash and join index tables are structurally virtually identical to base tables, you can use the same procedure and the same queries documented by this procedure to determine their size and the size of their indexes.

## **Querying *DBC.TableSizeV***

Use the following query to evaluate the sizing of your legacy tables. Assume the database in which the table was created is named *employee\_largetable* and the name of the table is *employee*.

```
SELECT SUM(CurrentPerm)
FROM DBC.TableSizeV
WHERE DatabaseName = 'Employee_LargeTable'
AND TableName = 'Employee';
```

The information returned might look something like this:

```
Sum(CurrentPerm)
2,527,232
```

## **Table Sizing Summary**

The following list reviews the principal points to remember about estimating the table space requirements for your new database.

- Accurate estimates require accurate base data. The minimum base data required for these estimates are the following.
  - Cardinalities (or anticipated cardinalities) of your tables
  - Row size estimates

Remember that all row lengths must be an even number of bytes (see “[Byte Alignment](#)” on page 770), so be sure to take this into account.

- Estimate sizes for all the following database objects.
  - Base tables
  - Base table fallback tables (when defined)
  - LOB and XML subtables
  - Secondary indexes
  - Secondary index fallback tables (when defined)
  - Hash indexes
  - Hash index fallback tables (when defined)
  - Join indexes (including any secondary indexes defined on them)
  - Join index fallback tables (when defined)
  - Spool space
  - Stored procedures

# CHAPTER 15 System-Level Capacity Planning Considerations

---

This chapter examines capacity planning for system and table space.

The majority of the material focuses on system space planning, including planning for disk space. Database sizing issues such as allocating permanent space and estimating database size requirements are also described.

## Database Size Considerations

The sizes of the Teradata systems managing relational databases range up to 2,048 CPUs and 2,048 GB of memory supporting 128 TB or larger databases. The BYNET interconnect supports up to 1,084 nodes.

Database sizing considerations must take the following issues into account:

- System disk contents
- Data disk contents
- Data disk space allocation
- Determination of usable data space

[Appendix B: “Teradata System Limits”](#) contains information about the various minimum and maximum sizes of database objects.

## System Disk Contents

System disks Disk 0 and Disk 1, are cabled to a controller not associated with the disk arrays.

Disk 0 is the boot disk. The slices on Disk 0 contain file systems under the control of the operating system root directory.

Disk 1 provides additional space for dumps and memory swapping.

## Boot Disk Contents

The following table lists the contents of the boot Disk 0.

Content	Use
AMP identifiers file	Stores identifiers of all AMPs.
Configuration maps	Defines vprocs.  The configuration map lists the pdisks allocated to a clique, but does not assign them to vdisks or AMPs.
Open PDE and operating system	Stores software under which Teradata Database is running.
Executable Teradata software	Includes any optional Teradata client software packages.
UDF libraries	Support user-defined functions and external stored procedures.
Copy of Teradata GDOs	Includes DBS Control Record (DBSCONTROLGDO).
Values	Initialize hash buckets.
space	<ul style="list-style-type: none"><li>• Diagnostic reports</li><li>• Memory swap space</li><li>• Operating system and Open PDE dumps</li></ul>

## Data Disk Contents

The virtual data disks controlled by the AMPs include reserved space and permanent space.

### Reserved Space

Reserved space is used for the TVS map, DeviceTag, and statistics areas that are located at the beginning of each pdisk.

### Permanent Space

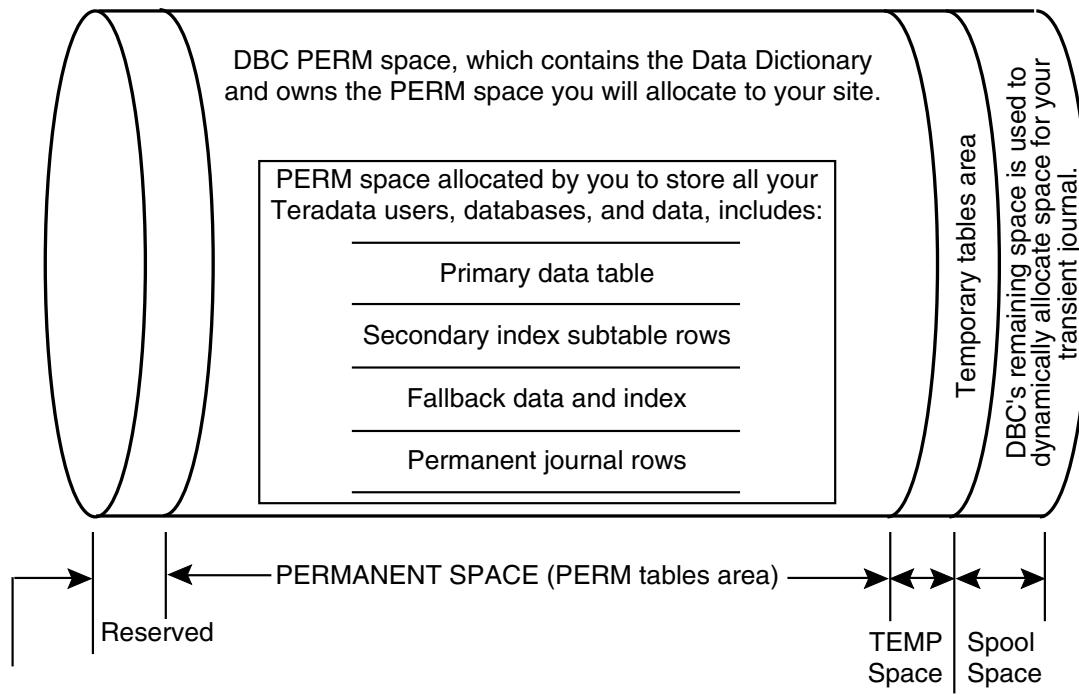
Content	Use
CRASHDUMPS user	Temporary storage of diagnostic information and crash dumps generated by the Teradata database and Open PDE.
SYSTEMFE user	Special macros and objects used by Teradata field support personnel.
SYSADMIN user	Special views and macros used by Teradata support personnel.
TDStats	Contains all of the tables, views, macros, UDFs, and external stored procedures used by the Teradata Database statistics management system.  TDStats is the repository for statistics management data.

Content	Use
Data dictionary	Maintains Teradata Database user, database, table, index, trigger, permanent journal, view, stored procedure, and macro definitions.
WAL log (transient journal) space	Stores Redo and Undo records that enable data to be brought to a consistent state as well as before-image records that enable uncommitted transactions to be rolled back.
Depot area space	Stores cylinders reserved by the File System as a staging area to temporarily contain modified-in-place data blocks before they are written to their home disk destinations.
Spool files	Stores intermediate and response results of SQL requests.
Temporary space	<p>Stores data inserted into materialized global temporary tables.</p> <p>Note that each materialized global temporary table also requires a minimum of 512 bytes from the PERM space of its containing database or user for its table header.</p>
Top-level hierarchical ownership of PERM and TEMP spaces	<ul style="list-style-type: none"> <li>• Teradata databases</li> <li>• Base data tables</li> <li>• Join index subtables</li> <li>• Hash index subtables</li> <li>• Fallback data and join index subtables</li> <li>• Secondary index subtables</li> <li>• Permanent journal tables</li> <li>• Stored procedures</li> <li>• Table headers</li> </ul>

## Data Disk Space

Data disk space is reserved for system use, PERM space, or TEMP space owned by user DBC, as illustrated in the following graphic. For purposes of this graphic, consider join index and hash index subtables to be identical to primary data tables.

The size considerations for each type of space are described in “[Permanent Space Allocations](#)” on page 878.



## Permanent Space Allocations

Permanent data disk space includes database and user fixed PERM allocation. The Teradata database initialization routine creates these system users:

- DBC (at the top of the hierarchy)
- SYSTEMFE (owned by DBC)
- SYSADMIN (owned by DBC)

All permanent space not consumed by SYSTEMFE and SYSADMIN is allocated to user DBC. The ramifications of the space allocation for each of the three system users are discussed in the subsections that follow.

### System User DBC

The PERM space allocation of system user DBC is used for the following:

This DBC space...	Performs this service ...	For this function ...
PERM	provides ownership	<p>Teradata production database space, which must accommodate the maximum PERM space allocated to each user and database by its creator.</p> <p>The allocated space is used to store the rows in the data tables created by each user, including the following items.</p> <ul style="list-style-type: none"> <li>• Primary data table rows</li> <li>• Join index table rows</li> <li>• Hash index table rows</li> <li>• LOB and XML subtable rows</li> <li>• Secondary index subtable rows</li> <li>• Optional permanent journal image rows</li> <li>• Optional fallback rows, which replicate the rows of primary data tables, index subtables, hash and join index tables, and permanent journals</li> </ul>
	creates and maintains the WAL log/transient journal	storing a before-image copy of every row affected by data modifications in all current sessions and Redo and Undo WAL records.
	stores system table, index, macro, trigger, and view definitions	setting up the data dictionary.
	provides permanent space	<ul style="list-style-type: none"> <li>• Table headers. Note that each materialized global temporary table also requires a minimum of 512 bytes from the PERM space of its containing database or user for its table header.</li> <li>• CRASHDUMPS database.</li> </ul>
TEMP	provides temporary space	data inserted into materialized global temporary tables.
SPOOL	provides maximum defined SPOOL space per creator-defined allocation to each user and database	<ul style="list-style-type: none"> <li>• Dynamic allocation and release of intermediate and final result sets during query processing.</li> <li>• Volatile temporary tables.</li> </ul>

## System User SYSTEMFE

System user SYSTEMFE is owned by user *DBC.SystemFE* contains special macros and objects. The quantity of PERM space allocated to SYSTEMFE is minimal.

Only Teradata field support personnel normally log on under this user name.

## System User SYSADMIN

System user SYSADMIN is owned by *DBC.SysAdmin* contains special views and macros. The quantity of PERM space allocated to SYSADMIN is minimal.

Only Teradata field support personnel normally log on under this user name.

# Estimating Database Size Requirements

## Allocating Space for External Stored Procedure and User-Defined Function Bodies

The code for both stored procedures and user-defined functions is stored outside Teradata Database on system disk, or, in the case of UDFs, possibly on client disk, so you do not need to account for it when you undertake database capacity planning.

## Allocating PERM, TEMP, and Spool Space

Use the CREATE USER statement to allocate all the PERM space required by the applications that support your enterprise to user *SysAdmin*. To determine these space requirements, use the equations provided in “[Calculating Total PERM Space Requirements](#)” on page 889.

**Note:** If the Data Dictionary and the *Crashdumps* user have not yet been created, run the DIP utility and then check the remaining PERM allocation before completing space calculations.

The quantity of space specified is allocated from the current PERM space of user *DBC*. *DBC* becomes the owner of the *SysAdmin* user and of all users and databases subsequently created. Be sure to leave enough space in *DBC* to accommodate the growth of system tables and logs and the WAL log.

Some guidelines for estimating temporary spool and WAL log space requirements are explained in the following subsections.

Note that the containing database or user of a global temporary table must also have a minimum of 512 bytes of PERM space free for the GTT table header.

## Estimating Administrative Spool Space Requirements

As with PERM space, spool space is allocated from the available space of the owning user. The SPOOL specification in the CREATE request is only a maximum limit.

During query processing, spool space is dynamically allocated from any free space, including the free PERM space of the user who submitted the query.

In general, the following guidelines apply.

- Reserve 25% to 35% of total space for spool space and spool growth buffer.  
When you create user *SysAdmin*, you can leave the SPOOL parameter unspecified, so it defaults to the maximum allocation of the owning user, user *DBC*.
- Allow an extra 5% of PERM space in user *DBC*.
- Each time a new user or database is created, you can specify the maximum amount of spool space that a query submitted by that user can consume.

Details and formulas for estimating user spool space requirements are provided in “[Tables Area: User Spool Space Requirements](#)” on page 887, “[Rules for Using the Spool Space Equations](#)” on page 887, “[Field Mode Spool Space Sizing Equation](#)” on page 887, and “[Record and Indicator Mode Spool Space Sizing Equation](#)” on page 888.

## Estimating Administrative TEMP Space Requirements

Allocate the TEMP space required by the applications that support your enterprise to user *SysAdmin*.

Like PERM space, TEMP space is allocated from the available space of the owning user. Make an estimate of temporary space requirements.

The TEMPORARY=n clause in the CREATE DATABASE and CREATE USER statements permits you to define how many bytes are to be allocated to a database or user for creating global temporary tables. Note that temporary space is reserved prior to spool space for any user defined to have this attribute.

Disk usage for materialized global temporary tables is charged to the temporary space allocation of the user who referenced the table.

Global temporary tables also require a minimum of 512 bytes from the PERM space of the containing database or user. This space is used for the GTT table header.

If no temporary space is defined for a user, then the space allocated for any global temporary tables referenced by that user is set to the maximum temporary space allocated for the immediate owner.

## Estimating WAL Log Space Requirements

Teradata Database creates and manages a WAL log that includes the transient journal to store the before-change image of every data row involved in every SQL transaction. This log is used to recover data tables when transactions are aborted.

As transactions are processed, the WAL log grows and shrinks. While transactions are in progress, the WAL log grows according to the total number of data rows being updated or deleted.

The applicable journal rows are purged at intervals (but not before the transaction is completed or, if it was aborted, not before the affected rows are recovered).

Space for the WAL log is acquired from the current PERM space of user *DBC*. Therefore, it is important that you leave enough PERM space in *DBC* to accommodate the growth of the largest foreseeable WAL log.

To estimate the maximum size of the WAL log, follow this procedure:

- 1 Determine the length of the longest row in your production database. Use this figure as your maximum row length.
- 2 Multiply the maximum row length by the total number of rows in your application programs, batch jobs, and ad-hoc queries that users are likely to update and delete in *concurrent* transactions.
- 3 Double the resulting value.

This is the space needed for the WAL log on your system.

## WAL Log Data Block Size

When you know the maximum row length, double the value to calculate the size of your multirow data block definition in the JournalDBSize parameter of the DBS Control Record. For example, the JournalDBSize value should be greater or equal to twice the maximum row length.

The file system allocates any row that exceeds the current size of a multirow data block to a WAL log data block of its own.

Depending on the configuration of your system, Teradata Database builds its data blocks using either native 512 byte sectors or native 4KB sectors. Systems built on 512 byte sectors are referred to as unaligned configurations, while systems built on 4KB sectors are referred to as aligned configurations.

For details on the JournalDBSize parameter, see the DBS Control utility chapter in *Utilities: Volume 1 (A-K)*.

## Determining Available User Table Data Space

To determine available user table data space, you must take several *nonuser* table data space allotments into account.

First you must determine how much space these allotments account for and then you must subtract them from the total available table space.

The nonuser table space allotments that must be accounted for are those in the following list.

- Overhead space, including space reserved for allocation maps and related statistics.
- Depot area, which includes cylinders reserved as a staging area for modified-in-place data blocks before they are written to their home disk destinations.

**Note:** Data blocks that are *not* modified in place are not written to the depot area.

- Tables area, which includes the following.
  - Data Dictionary (for the WAL log and system tables)
  - *Crashdumps* user
  - User temporary space
  - User spool space

The following topics describe specific space requirements and provide equations with which to determine variable space requirements.

### Depot Area Overhead

The Depot consists of two types of slots:

- Large Depot slots
- Small Depot slots

The Large Depot slots are used by aging routines to write multiple blocks to the Depot area with a single I/O.

The Small Depot slots are used when individual blocks that require Depot protection are written to the Depot area by foreground tasks.

The number of cylinders allocated to the Depot area is fixed at startup. Consult your Teradata support representative if you want to change this value.

The number of Depot area cylinders allocated is per pdisk, so their total number depends on the number of pdisks in your system. Sets of pdisks belong to a subpool, and the system assigns individual AMPs to those subpools.

Because it does not assign pdisks to AMPs, the system calculates the average number of pdisks per AMP in the entire subpool from the vconfig GDO when it allocates Depot cylinders, rounding up the calculated value if necessary. The result is then multiplied by the specified values to obtain the total number of depot cylinders for each AMP. Using this method, each AMP is assigned the same number of Depot cylinders.

The concept is to disperse the Depot cylinders fairly evenly across the system. This prevents one pdisk from becoming overwhelmed by all the Depot writes for your system.

## **Data Table Overhead**

The system uses some amount of data space, which includes permanent, spool, and temporary space, as overhead for various purposes.

The percentage of data space required for cylinder indexes is calculated based on the following information:

- Each cylinder has 2 cylinder indexes.
- Each cylinder index is 32 KB (64 sectors) in size.

Therefore, 128 sectors are reserved for cylinder indexes per cylinder.

- Each cylinder is 23,232 sectors in size.

The percentage of data space reserved for cylinder indexes is determined by the following calculation:

$$\text{Percent data space reserved for cylinder indexes} = \frac{2 \times \text{size of one cylinder index}}{\text{size of one cylinder}} \times 100$$

If the cylinder or cylinder index size changes for a release, use the same equation to determine the estimated percentage of data space required for cylinder indexes, but adjust the values accordingly.

For the current release, the equation evaluates to the following percentage of data space:

$$\text{Percent data space reserved for cylinder indexes} = \frac{128 \text{ sectors/cylinder}}{23,232 \text{ sectors/cylinder}} \times 100 = 0.55\%$$

The quantity of permanent data table space, temporary space, and spool space available on a system is also limited by the amount of disk space required by WAL to process update transactions. The amount of disk required by WAL is a function of the number of update

operations being undertaken by the workloads running on the system at any given time, and waxes and wanes as a result. The more updates that are done at a time, the greater the quantity of WAL data that is produced. The amount of WAL data produced is roughly equivalent to twice the amount of transient journal information that is required for updates.

Allow space for data table overhead as listed in the following table:

Number of Cylinders	Purpose
13	<p>General</p> <p>One cylinder each for the following purposes:</p> <ul style="list-style-type: none"> <li>• Permanent space sentinel</li> <li>• Permanent journal space sentinel</li> </ul> <p>System journals are stored either in user DBC, which is permanent space, or in the WAL log, which is not part of journal space.</p> <ul style="list-style-type: none"> <li>• Global temporary spool space sentinel</li> <li>• Spool space</li> </ul> <p>Nine cylinders for the following purpose:</p> <ul style="list-style-type: none"> <li>• Write Ahead Logging</li> </ul> <p>The additional cylinders for WAL are required to ensure that MiniCylPack is able to run in low disk situations to free space.</p>
Approximately 1.25% of the available data space (minimum)	Cylinder indexes
A minimum of 7% free space per cylinder	<p>Fragmentation</p> <p>You must be careful not to allow too much free space for fragmentation, but you must be equally careful not to allow too little.</p> <p>The most practical way to establish an optimal number of cylinders to reserve for fragmentation on your system is trial and error.</p>
Ranges between 1 and 20 per pdisk.	<p>Depot</p> <p>The number of cylinders assigned to Depot is set at startup. The value for your system can only be changed by Teradata support personnel.</p>

## Allocation Map and Statistics Overhead

The TVS layer requires space on each pdisk for its allocation map, device tag, and statistics areas. The number of cylinders required depends on the pdisk size as specified in the vconfig GDO. The following table lists the overhead for disk sizes ranging from 60 GB through 1,024 GB, using a value for 1 GB as  $1024 * 1024 * 1024$  bytes.

Be aware that the size of this file can change from release to release.

You can use the following equation to determine the number of cylinders required:

$$\text{cylinders} = \frac{\left( (2 \times t) + \frac{\text{roundup}(4 \times E, p)}{512} + \text{roundup}\left(\left(\frac{\left(\frac{\text{roundup}(E, d)}{d} \times 1.1 + A\right)}{512}\right), \frac{p}{512}\right) + \text{roundup}\left(\frac{\text{roundup}(E, 32)}{32} + A, \frac{p}{512}\right) \right) \times 3872}{3872}$$

where:

Equation Element ...	Represents the ...
E	number of cylinders on the device, which is determined as follows: $E = \frac{N}{3872}$ where N represents the number of 512-byte sectors in the partition on the pdisk, as indicated by the PDISK entry in the vconfig.txt file. Devices with a sector size of 4096 bytes are also expressed in units of 512 bytes for the purposes of this equation (so, for example, a device with 1000 4096-byte sectors would have N=8000).
A	number of TVS vprocs that share the device.
roundup	function used to round up the number of cylinders. The function is defined as follows: $\text{roundup}(a, b) = \text{int}\left(\frac{(a + b - 1)}{b}\right) \times b$
m	TVS map block size: <ul style="list-style-type: none"><li>• 512-aligned systems: 1024</li><li>• 4096-aligned systems: 4096</li></ul>
d	Map entries per map block: <ul style="list-style-type: none"><li>• 512-aligned systems: 33</li><li>• 4096-aligned systems: 135</li></ul>
t	TVS device tag size (in units of 512): <ul style="list-style-type: none"><li>• 512-aligned systems: 1</li><li>• 4096-aligned systems: 8</li></ul>
p	Alignment: <ul style="list-style-type: none"><li>• 512-aligned systems: 512</li><li>• 4096-aligned systems: 4096</li></ul>

The following table shows a few sample results for various disk sizes:

Disk Size (GB)	Allocation Map and Statistics Space Required (cylinders)
60	1
203	4
474	8
744	12
1,014	16

Space is also required on system disk in the *tpi-data* directory for the allocator memory-mapped file. Note that *tpi-data* is stored on system disk in a node, not on the user data space on the disk array. The amount of space required depends on the size and number of pdisks and how many allocators there are. For most system configurations, the size of this file ranges between 5 and 40 MB.

## Tables Area: Dictionary and Spool Space Requirements

After you determine how much space remains when overhead is accounted for, determine how much space is required for the data dictionary and spool files.

You should reserve approximately 80 MB for growth of system tables and the WAL log. However, optimum space for the WAL log should be based on how many rows are dropped or updated by concurrently running transactions during your peak workload.

## Tables Area: User TEMP Space Requirements

The TEMPORARY clause in the CREATE DATABASE and CREATE USER statements permits you to allocate default space within this database or user for inserting data into global temporary tables materialized by users. Note that Teradata Database always reserves temporary space *prior* to spool space for any user defined with this attribute.

Disk usage for a materialized global temporary table is charged against the temporary space allocation of the user who referenced the table.

Global temporary tables also require a minimum of 512 bytes from the PERM space of the containing database or user. This space is used for the GTT table header.

If no default temporary space is defined for a database, then the space allocated for any global temporary tables created in that database is set to the maximum temporary space allocated for its immediate owner.

Subtract the TEMP amount allocated to this database.

## Tables Area: CRASHDUMPS User Space Requirements

Subtract the PERM amount allocated to this user. The DIP utility creates this database with a default allocation of 1 GB.

If you modified the default amount when you set up the *Crashdumps* user, subtract the actual current amount.

## Tables Area: User Spool Space Requirements

Whenever you create a new user, reserve a minimum of 20% of user permanent space for spool space. It is better to determine the optimum amount of space according to your application environment. This takes into account such factors as average table size, protection choices, number of rows involved in commonly used transactions, and so forth.

The optimum allocation depends on your application environment. If your requirement is to perform the calculations according to the formulas provided in this section, and the calculated amount exceeds 20% of permanent space, use the actual figure plus 5%. To determine an allowance based on your applications, use the formulas in this chapter.

## Rules for Using the Spool Space Equations

When using the spool space equations, keep the following rules in mind:

- The equations are based on the following variables.
  - Number of rows in the spool file
  - Amount of data in each row
  - Mode of select operation (Field or Record)
- The equations only determine the space needed to return rows to the user.
- Response rows are limited to approximately 64,000 bytes.  
If the row is longer, a row length error is reported.
- Even if an ORDER BY clause is not included, the system creates a minimum length sort key of 8 bytes for both Record mode and Field mode select operations.
- All descriptions of CHARACTER columns also apply to BYTE columns.
- Be sure the NF, CF, SCF, and RDS values represent the sums of all selected or sorted columns. For example, a FORMAT phrase can cause blanks to be added, which must be figured into the total column length.

## Field Mode Spool Space Sizing Equation

Use the following equation to determine the amount of usable data space for Field mode:

$$\text{Usable Data Space} = a(RO + RP + n(PH + NF + CF)) + b(SNF + SCF)$$

The following parameter definitions are used with this equation:

Parameter	Definition
a	Number of rows being selected.
RO	Row overhead (22 bytes per row).
RP	Row parcel indicators; 8 bytes per row (rec start, rec end).

Parameter	Definition
n	Number of columns being selected.
PH	Parcel headers (4 bytes per field).
NF	Formatted size of each numeric column (spaces, dollar signs, and commas should be included).
CF	Formatted size of each character column.  For example, if a selected column is defined as VARCHAR(200), it takes up 200 characters even if it contains only 3 nonblank characters.  The value for CF can be less than the CREATE TABLE definition of the string only if there is a FORMAT phrase.
b	Number of numeric columns in the ORDER BY clause (sort key).
SNF	Sorted numeric fields (8 bytes each).
SCF	Formatted size of each character column in the ORDER BY clause.  If an order operation is done on a column defined as VARCHAR(200), 200 characters are allowed, even if there are only 3 nonblank characters.  The value for CF can be less than the CREATE TABLE definition of the string only if there is a FORMAT phrase. The formatted length of the character string should be rounded up to the next even value, then 2 more bytes should be added to the number.

## Record and Indicator Mode Spool Space Sizing Equation

Use the following equation to determine the amount of usable data space for Record and Indicator modes:

$$\text{Usable Data Space} = a(\text{RO} + \text{RP} + \text{RDS} + (\text{b}(\text{SNF} + \text{SCF})) + \frac{n}{(\text{IF} + 1)})$$

The following parameter definitions are used with this equation:

Parameter	Definition
a	Number of rows being selected.
RO	Row overhead (12 bytes per row).
RP	Row parcel indicators; 8 bytes per row (rec start, rec end).
RDS	Raw data size of each field returned to the user.
b	Number of numeric columns in the ORDER BY clause (sort key).
SNF	Sorted numeric fields (8 bytes each).

Parameter	Definition
SCF	<p>Formatted size of each character column in the ORDER BY clause.</p> <p>If an order operation is done on a column defined as VARCHAR(200), 200 characters are allowed, even if the value contains only 3 nonblank characters.</p> <p>The value for CF can be less than the CREATE TABLE definition of the string only if there is a FORMAT phrase. The formatted length of the character string should be rounded up to the next even value, then 2 more bytes should be added to the number.</p>
n	Number of columns being selected.
IF	Record mode selects use indicator variables to represent nulls. IF is the constant 8, meaning that for each 8 columns selected, 1 byte is needed to represent a null column.

## Calculating Total PERM Space Requirements

Use the following procedure to determine how much PERM space to allocate to user *SysAdmin*.

- 1 Estimate the size of each database, as follows.
  - a Estimate the size of the primary data table, including fallback (see “[Sizing Base Tables, LOB Subtables, XML Subtables, and Index Subtables](#)” on page 858 for instructions for how to do this) and LOB and XML subtables.  
LOB and XML values are also stored in the permanent journal, so LOB and XML column space must account for additional permanent journal append and storage costs in addition to LOB and XML subtable storage costs.
  - b Estimate the size of unique and nonunique secondary index subtables, join indexes, and hash indexes.
  - c Add the primary data table estimate and index and LOB and XML subtable estimates (including fallback) to obtain the estimated total table size.
- 2 Estimate the total table storage by adding together the space estimates of all table sizes (Substeps a, b, and c). Use this sum for step 3.
- 3 Estimate the space requirement for the application.
 

**Note:** If the calculated sum is greater than the remainder calculated in step 4, contact your Teradata sales representative to discuss a system expansion.

  - a Add 7% for fragmentation.
  - b Add extra space for spooling, TEMP space, and for variability in hashing. Although this space requirement varies according to the applications, adding 20% to 30% is usually adequate if you do not use permanent journaling.
  - c Estimate the size of each journal table per database.
  - d Note that if journaling is used, the extra space requirement depends on two factors:
    - Size of the data tables that write to that journal
    - Number of changes applied to those tables before the journal table is dropped

- e Add the sums calculated for steps 1 through 4 and use their sum for the calculation in step 5.
  - f For each disk, add 3 cylinders per disk as spaces and allow 2% of the formatted space for cylinder indexes.
- 4 Subtract the amount of space needed to accommodate user *DBC* contents plus your maximum WAL log from the user *DBC* PERM space.  
Specify about 80 MB for the growth of system tables and the WAL log plus another 5% of the current PERM space for spool growth.  
Use the remainder for step 5.
- 5 If possible, subtract the sum of step 3 from the result of step 4.
- 6 Deduct the number of cylinders that make up the default free space specified by the DBS Control record for operations that use the percent freespace (FSP) value.  
For example, if your FSP is set to 15, deduct an additional 15%.  
If you plan to create large tables with a freespace percentage greater than the GDO default, then use the larger percentage.  
For example, if you define your largest tables with an FSP of 30, then deduct an additional 30%.  
The remainder is the amount of PERM space you should allocate to the *SysAdmin* user.

**Note:** Table sizing is normally the responsibility of Teradata field support personnel.

## Designing for Backups

There was a time not so very long ago when backups were a minor issue in database management. You set up a batch job to perform the backup overnight, while the system was not being used by anybody else, and that was that. Several issues have complicated that once common scenario.

First is the tremendous increase in the size of databases that can be supported by a relational database management system like Teradata. Where at one time a “very large database” might be on the order of several GB, it is increasingly common to see multiterabyte databases supporting large data warehouses and support for petabyte and even yottabyte databases is on the near horizon. Magnetic tape is a serial medium. Even if you archive to multiple tape drives simultaneously, there is still a considerable time issue involved with backing up multiterabyte databases.

Second is the end of the era of the batch window. Enterprise data warehouses now typically support worldwide operations, and that means that the system must be available to users 24 hours per day, seven days a week. Without the old batch window, how do you back up your business-critical data without having a significant negative effect on the capability of your data warehouse to support its worldwide user base?

Many approaches to solving these problems fall outside the scope of database design, but at least one method for minimizing the impact of backing up very large databases can be designed into your database. That method is the subject of this topic.

## Terminology

For purposes of this topic, *large* and *small* tables and databases are defined as follows:

Term	Definition
Small table	Less than 100 GB.
Small database	
Large table	Greater than 100 GB.
Large database	

## Product Issues

The issue described in this topic and its suggested solution apply to the large system environment, primarily when using the NetVault product to backup to and recover from different machines. The REEL product makes it possible to restore to or recover from *the same machine* using its fast path option, but even that option does not relieve the problem of backing up to a second machine.

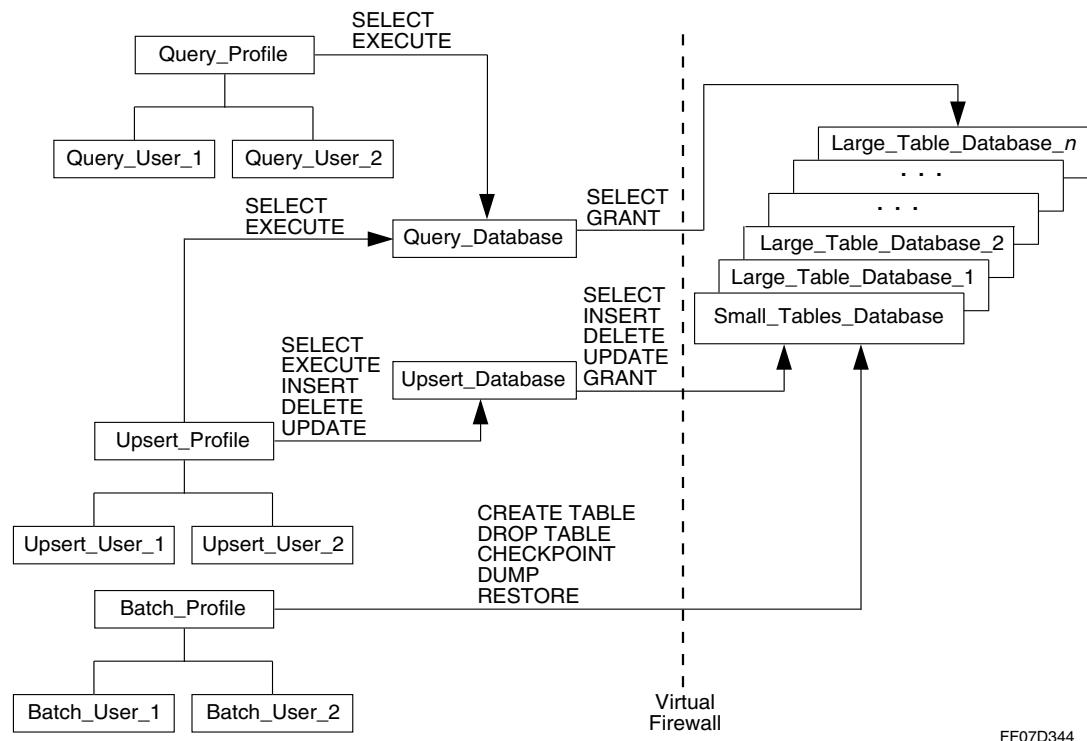
## The Scenario

Database and table dumps and restores are a time consuming process. To minimize this problem, you can isolate large tables in their own databases in order to isolate their backup and restoration. Otherwise, when tables of this magnitude are mixed with much smaller tables, you have two choices:

- Back up the large table in one pass and then backup the remaining tables in a second pass.
- Back up the entire database at once, which takes a very long time to complete (and even longer to restore)

## A Solution

Design your database in such a way that large tables are isolated within their own, separate databases, then build views for your users to use to access the data. Carefully constructed views can mask this separation and permit users to access data across multiple databases transparently within a single query. The following graphic illustrates one possible design that uses this method.



# CHAPTER 16 Design Issues for Tactical Queries

---

The majority of the topics covered by this manual apply to designing for strategic, or decision support, queries. Because you can also run your operational data store on Teradata Database, this chapter highlights some of the design issues you must consider when implementing a physical design to support mixed workloads of both strategic and tactical queries.

The manual discusses other material related to more specific design issues for tactical queries in “[Chapter 9 Primary Indexes and NoPI Objects](#)” on page 261 and “[Chapter 11 Join and Hash Indexes](#)” on page 499.

The chapter does not make suggestions about scheduling job priorities to achieve an optimal mix of strategic and tactical workloads for your system, nor does it describe the various locking issues associated with tactical queries.

See *Utilities: Volume 2 (L-Z)* for a description of the Priority Scheduler utility and *SQL Request and Transaction Processing* for information about lock management for tactical queries.

## Tactical Queries Defined

Traditionally, data warehouse applications have been based on drawing strategic advantage from the data. Strategic queries are often complex, sometimes long-running, usually broad in scope. The parallel architecture of Teradata Database supports these types of queries by spreading the work across all of the parallel units and nodes in the configuration.

As data warehouse users require more versatility out of their data, they are supplementing their strategic focus with a tactical component. Today, increasing numbers of Teradata Database users are introducing their quick turnaround tactical queries alongside the classic strategic queries in their Teradata Database system.

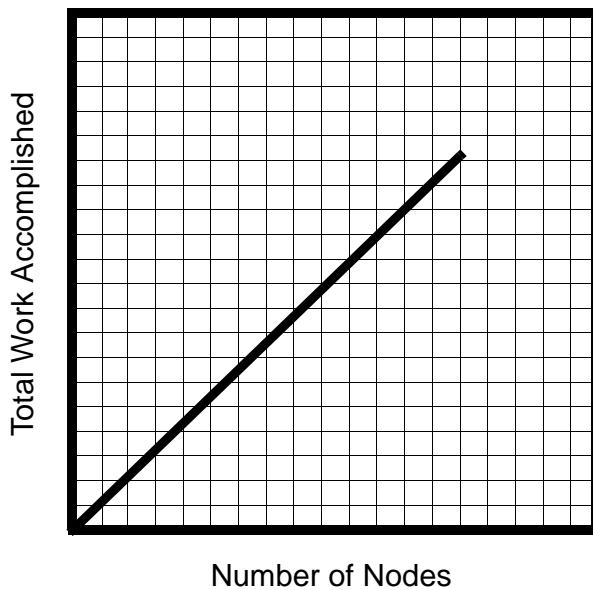
The name tactical query is given to a certain category of work that is short, highly tuned, and focused on more operational decision-making. This class of queries is sometimes categorized under the initialism OLCP, which stands for Online Complex Processing. Tactical queries are similar to classical online transaction processing (OLTP), in that both are composed of short, often direct-access queries, coming with defined response time requirements.

OLTP, however, is more focused on bookkeeping activities and recording operational events, while tactical queries are more focused on decision-making, recording selections or choices made. OLTP generally comes with a higher ratio of writes compared to reads, while tactical queries tend to be more read intensive. OLTP usually has a departmental view and a very narrow context, while tactical queries are often more enterprise-wide. With OLTP queries, short latency is always critical. Tactical queries, on the other hand, are designed for response time requirements that can vary from subsecond up to 10 or 20 seconds or more.

# Scalability Considerations for Tactical Queries

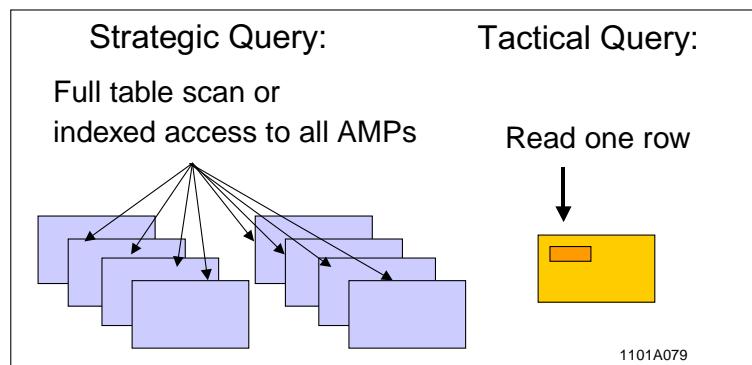
## Scalability Is a Relative Concept

In traditional decision support applications, setting the stage for scalable performance with Teradata Database involves distributing the work equally across all the AMPs in the system. As more nodes are added to a system, the number of AMPs increases proportionally, and the effort to process a complex query is spread across more parallel units, reducing response times. The following plot indicates linear scalability, graphing the total workload accomplished as a function of the number of nodes in the system. Strategic queries run proportionally faster as more nodes are added to a system.



1101A078

Spreading the data evenly across all AMPs remains the goal of designing the database to support tactical queries. But to achieve the type of scalability that best supports tactical queries, you need to localize the work in such a way that it accesses only a few rows using the least resources possible. This means accessing the smallest number of AMPs possible to perform an operation; optimally only one. The following figure indicates that when tactical queries access only a few AMPs, more of them can run concurrently.



1101A079

## Effect of Data Volume Growth on Tactical Query Response Times

Query times for complex decision support are directly impacted by increases in the data volume. Because a large portion, if not all, of a table might need to be read, if that table doubles in size, then the effort required to process the work can also double. On the other hand, a query that accesses one or a few rows in the database is not impacted by changes to the data volume. In most cases, a tactical query accesses one or few rows whether the table those rows are in is 1 GB or 10 TB in size.

## Effect of Growth in Concurrent Users on Tactical Query Response Times

Raising the level of concurrent users doing the same work in traditional decision support tends to slow response times for the current users because more demands are now being made on the same resources, and all are spread out across all nodes and AMPs in the system. In contrast, increasing the number of users performing tactical queries that are very localized and limited in their resource use boosts overall throughput up to the point of system saturation.

## Effect of Configuration Expansion on Tactical Query Response Times

For a complex query, adding nodes translates to a proportional decrease in response time because the work is distributed across a larger number of AMPs. At the same time, the response time for a query that accesses a single AMP, as many tactical queries do, is not affected by the number of AMPs in the configuration.

There is a one benefit gained by single-AMP tactical queries when nodes are added to a configuration: more of them can be performed at the same time and still deliver short turnaround times.

Consider the following contrived example: Assume you have a table with a cardinality of 1 million rows. The read capacity of each node in your configuration is 100 rows per second. If you are doing a complex strategic query that involves a full table scan of this table, then the response time for the query diminishes proportionally with the increase in nodes, as illustrated by the data in the following table:

Number of Nodes	Number of Rows per Node	Response Time (seconds)
1	1,000,000	10,000
10	100,000	1,000
100	10,000	100
200	5,000	50

This performance enhancement occurs because each node has fewer rows as more nodes are added, so each node can perform its portion of the scan faster.

On the other hand, if your application is performing single- or few-AMP tactical queries, adding nodes does not shorten the response time. However, it does increase the number of

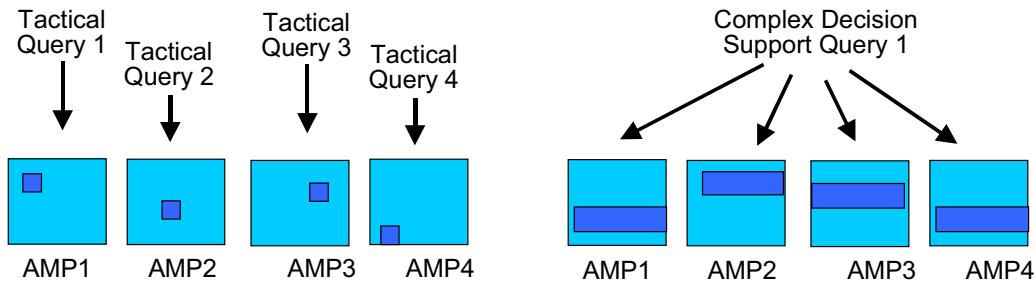
tactical queries that can be performed in a given interval of time, as indicated by the data in the following table:

Number of Nodes	Response Time (seconds)	Throughput (requests/second)
1	0.01	100
10		1,000
100		10,000
200		20,000

## Localizing the Work

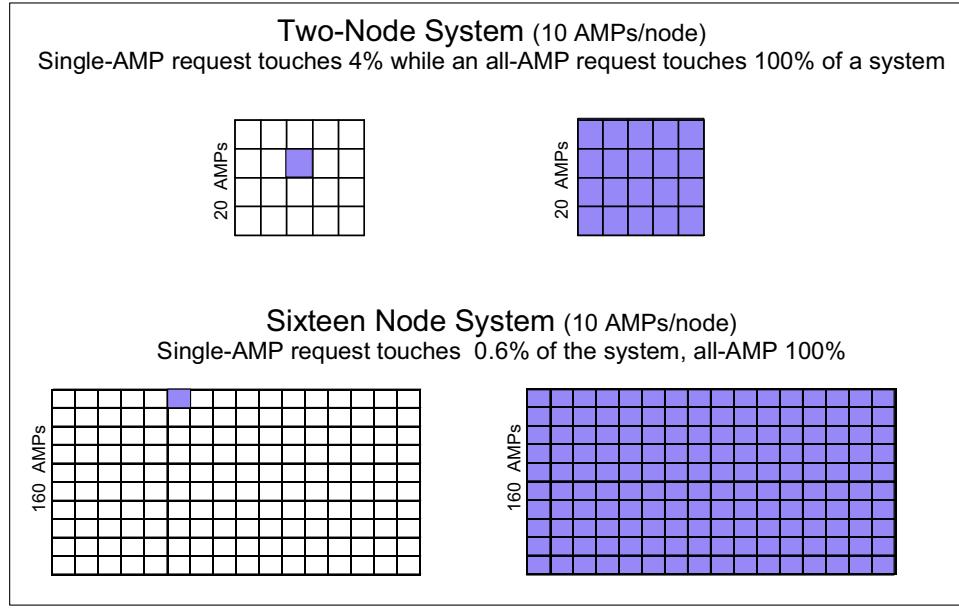
After an SQL statement is optimized, the Dispatcher sends the individual steps that make up that query, one at a time, to some number of AMPs in the system. Steps are usually dispatched, and processes started, on all AMPs in the system. Under some conditions, the Optimizer might decide that this query work can be accomplished by accessing only a single AMP in the configuration. This is referred to as a single-AMP operation, or a single-AMP step.

Teradata Database uses single-AMP steps to localize work to the fewest resources necessary when it is beneficial to do so. This is frequently the case for tactical queries. Single-AMP work frees the other AMPs in the system to do other work, thereby increasing the potential for overall system throughput. This is diagrammed in the following figure:



1094A023

Tactical queries are most efficient and most scalable when designed around single- or few-AMP operations. This is the single most important way to increase throughput and preserve response times for very short tactical queries. This is diagrammed in the following figure:

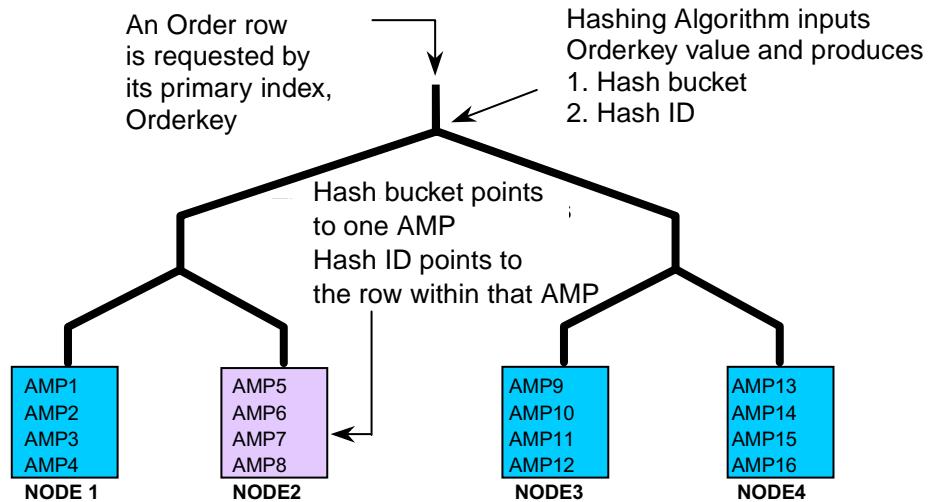


1094A041

Single- or few-AMP queries scale well because they engage the same small level of database resources even when the system doubles or triples in size (see “[Effect of Configuration Expansion on Tactical Query Response Times](#)” on page 895).

## Single-AMP Operations

Single-AMP operations use a primary index value to locate a row.



1094A024

Single-AMP operations can be achieved by any of the following data manipulation operations:

- Simple single-row inserts
- Simple selects, updates and deletes qualified with a primary index value
- Joins between two tables that share one of the following characteristics:

- the same primary index domain
- a primary index making up the join constraint
- a single primary index value is specified for the join

The following EXPLAIN report is for a query that accesses the supplier table using the single primary index value `s_suppkey = 583`. Only a single AMP is engaged, as demonstrated by the single-AMP RETRIEVE step reported in step 1 of the EXPLAIN text:

```
EXPLAIN
SELECT s_name, s_acctbal
FROM supplier
WHERE s_suppkey = 583;

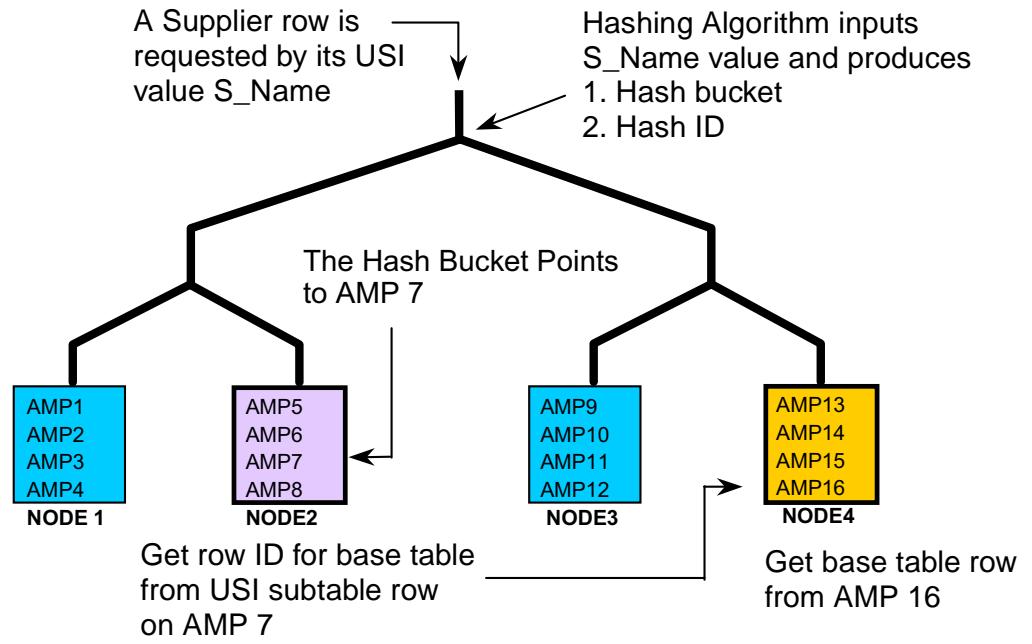
Explanation
-----
1) First, we do a single-AMP RETRIEVE step from TPCD50G.supplier by
way of the unique primary index "TPCD50G.supplier.S_SUPPKEY = 583"
with no residual conditions. The estimated time for this step is 0.03 seconds.
```

Notice that there are no references to locking in the EXPLAIN report for this query. That is because the Optimizer has folded the locking activity (in this case a single row hash READ lock) into the same step that retrieves the row. This sort of lock folding is done only with row hash locks.

This special shortcut for handling row hash locks eliminates the need for the Dispatcher to dispatch a separate locking step when only one AMP and one row are involved. This reduces the PE-to-AMP communication effort.

## Two-AMP Operations: USI Access and Tactical Queries

When an application specifies a value that can be used to access a table using its USI, a 2-AMP operation results. Note that USI access *can* be a single-AMP operation if the USI value for a row happens to hash to a subtable on the same AMP as the primary index for the same row, but is *never* more than a 2-AMP operation.



1094A025

In the following query, the column s\_name is defined as a USI on the original supplier table having s\_suppkey as its UPI:

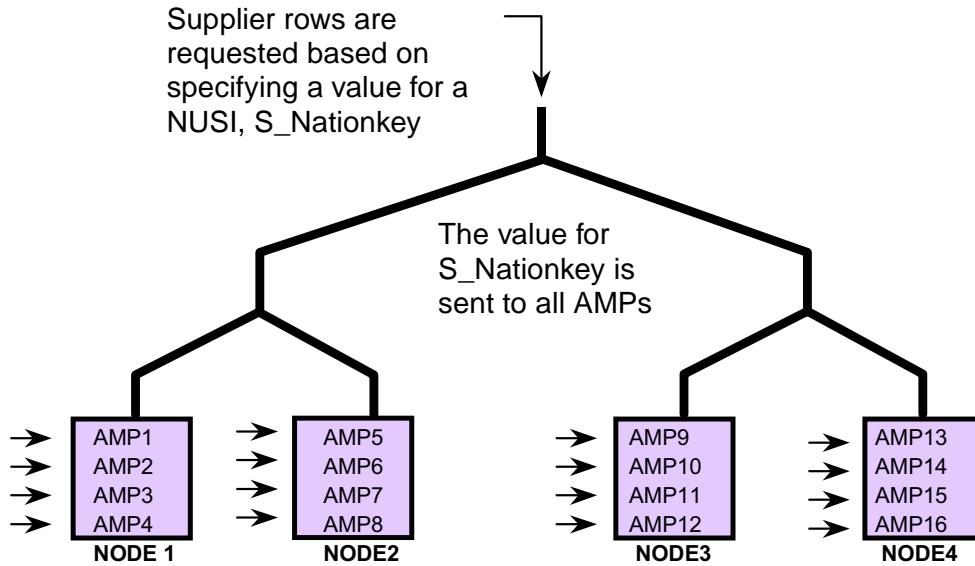
```
EXPLAIN
SELECT s_suppkey, s_acctbal
FROM supplier
WHERE s_name = 'Supplier#000038729';
```

#### Explanation

- 1) First, we do a **two-AMP RETRIEVE step** from CAB.supplier by way of unique index # 8 "CAB.supplier.S\_NAME = 'Supplier#000038729'" with no residual conditions. The estimated time for this step is 0.07 seconds.

## NUSI Access and Tactical Queries

Compare the previous example of USI access to access using a NUSI. With NUSIs, each AMP contains an index structure for the base table rows that it owns. Even if only a single row contains the specified NUSI value, the Optimizer cannot know that a priori, so the NUSI subtables for all AMPs are always searched, making NUSI accesses all-AMPs operations.



All AMPs search their NUSI subtables and get rowIDs of base table rows that match. Only those base table rows are returned.

1094A026

The following example assumes the *s\_name* column is defined as a NUSI on supplier. Notice the difference between this EXPLAIN report and the previous one for the USI (see “[Two-AMP Operations: USI Access and Tactical Queries](#)” on page 898):

```
EXPLAIN
SELECT s_suppkey, s_acctbal
FROM supplier
WHERE s_name = 'SUPPLIER#000000647';
```

#### Explanation

- 1) First, we lock TPCD50G for read on a RowHash (proxy lock) to prevent global deadlock for TPCD50G.supplier.
- 2) Next, we lock TPCD50G.supplier for read.
- 3) We do an all-AMPS RETRIEVE step from TPCD50G.supplier by way of index # 8 "TPCD50G.supplier.S\_NAME = 'SUPPLIER#000000647'" with no residual conditions into Spool 1, which is built locally on the AMPs. The size of Spool 1 is estimated with high confidence to be 1 row. The estimated time for this step is 0.20 seconds.
- 4) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.

With NUSI access, table-level locks are applied and an all-AMPs operation is performed. Value-ordered indexes and covering indexes are variants of a NUSI. Because of that, they also require all-AMPs operations that are similar to NUSI access.

## Group AMP Operations

Sometimes less parallelism is best. Tactical queries, for example, are more efficient when they engage fewer resources. On the other hand, short queries often require more than one AMP and cannot be accommodated with single-AMP processing alone. The Optimizer looks for opportunities to transform what would otherwise be all-AMP query plan into a few-AMP plan. This few-AMPs approach is called Group AMP.

The Group AMP approach not only reduces the number of AMPs active in supporting a query, it also reduces the locking level from a table-level lock to several partition and rowhash locks. Removing the need for table-level locks eliminates two all-AMP steps from the query plan:

- A step to place the table-level lock on all AMPs.
- A step to remove the table-level lock from all AMPs when the query completes.

## Determining When the Optimizer Will Consider Group AMP Processing

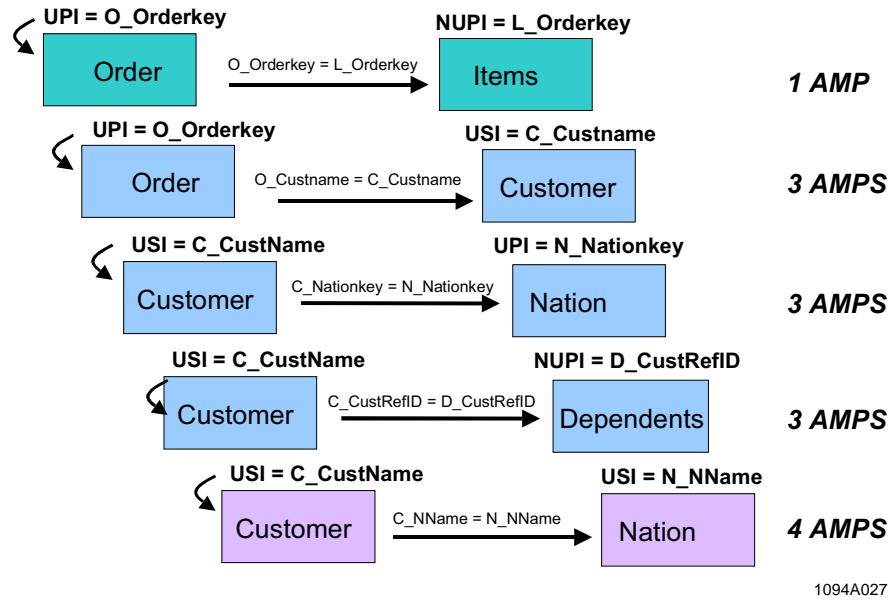
The Optimizer considers several criteria when it determines whether a query is a candidate for Group AMP processing or not:

- How many AMPs needed to satisfy the request?  
For most systems this number of participating AMPs must be 50% or fewer of the total number of AMPs configured in the system. Statistics collected on the selection and join columns being referenced in the query help the Optimizer make the correct assessment. If no statistics for these columns exist, the Group AMP option is less likely to be chosen. This makes the regular collection of statistics a critical prerequisite for gaining the advantages of the Group AMP feature.
- Can the query be driven from a single-AMP or Group AMP step as the first meaningful database access step?  
Because NUSI access is always an all-AMP activity, accessing rows by means of a NUSI as the first step in the query plan eliminates Group AMP considerations later in the plan, even if the NUSI access returns only one row from one AMP.
- Does a product join based on table duplication appear in any step in the query plan?  
If so, the Group AMP option is not considered. Under this condition, a product join is always an all-AMP operation. The existence of a single all-AMP operation in the plan negates the possibility of using a Group AMP later.

## Few-AMP Joins and Tactical Queries

Even when joins must be made to process a tactical query, it is possible to make the join with few-AMP operations. The following picture illustrates the combinations of accessing and joining that can be performed engaging one or few AMPs. In all of these few-AMP examples, the first table is accessed using either a UPI or a USI.

The first join in the following example is made using a merge join. Because the primary index columns of both tables share the same domain, their associated rows reside on the same AMP, with the second table being joined based on a NUPI. The remaining joins use combinations of primary index and USI access and perform few-AMP nested joins between the tables.



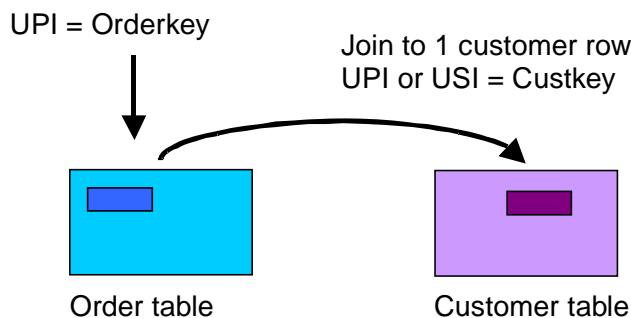
One idiosyncrasy of EXPLAIN text is that it sometimes says "we do a two-AMP join step" when a USI is defined on one half of the nested join. Three AMPs are engaged for most such joins, not two.

## Tactical Queries Benefit From Nested Joins

All AMPs are engaged for product and hash joins, and also for most merge joins. For this reason these join methods are less desirable for more localized queries. Tactical queries benefit from the few-AMP joins described in “[All-AMP Queries](#)” on page 908.

Of these join methods, the most beneficial for tactical queries is the nested join, which under specific conditions involves only a few AMPs.

A few-AMP nested join is possible when the first table can be accessed by a UPI or USI specified by the request. Following that, a foreign key within that first table must be available to drive a nested join into a second table, which contains its associated primary key. For this nested join to involve only one or a few AMPs, the second table must have either a primary index or a USI defined on the column set mapped to by the foreign key in the first table, as demonstrated by the following graphic:



1094A028

# Database Design Techniques to Support Localized Work

Several approaches to physical database design can make one-AMP or few-AMP query plans more likely. Most of these choices need to be made carefully, because they also influence how other work in the system is optimized. All applications running in the database need to be considered, particularly when selecting or changing primary index columns:

- Use the Same Primary Index Definitions

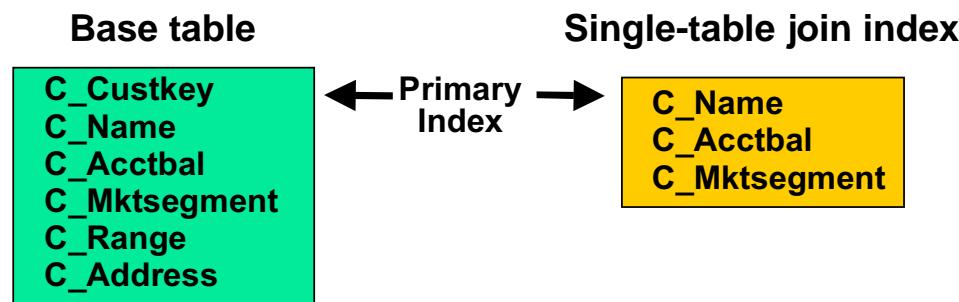
When you expect frequent joins between two associated tables, consider using identical column sets for the primary index definitions of both tables. In other words, define the primary indexes and the join columns are the columns. This technique can enhance both decision support and tactical queries, depending on the frequency of the join and the demographics of the data.

- Increase the Likelihood of Few-AMP Nested Joins

Consider placing USIs on the join constraint of one of the tables. From the perspective of logical database design, you are placing the USI on the primary key of the primary table in the primary key-foreign key relationship.

- Consider a Join Index

You can create a single-table join index or hash index with a different primary index than the base table. For example, you could define a primary index for the join or hash index composed of the column that corresponds to values frequently specified by the application. The example in the following diagram indicates how a query that only has a value for customer name could use the join index for single-AMP access.



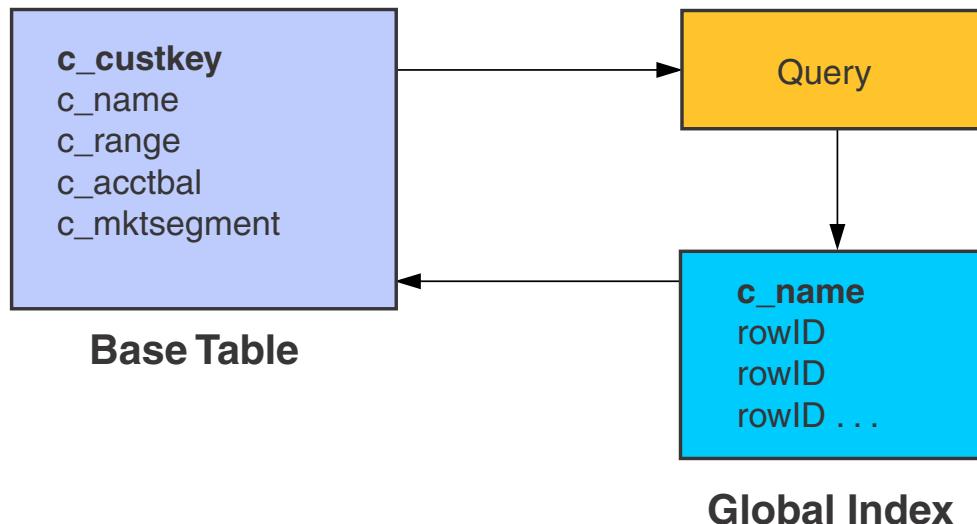
1094A031

- Consider a Global Join Index

A global join index is a single-table join index, similar to the one illustrated above, except for one important difference: each global join index row contains a pointer to its base table that the Optimizer can use as an alternate way to access base table rows. Using a join index for base table access is referred to as partial covering because such a join index only partially covers the query (see “[Partial Query Coverage](#)” on page 505 and “[Restrictions on Partial Covering by Join Indexes](#)” on page 575).

Global join indexes offer the combined advantages of a NUSI (by supporting duplicate rows per value) and a USI (hashed index rows), and the Optimizer can often take advantage of these capabilities for Group AMP operations.

The following graphic indicates how the query first accesses the global join index, then uses the rowID information from the index to access the base table rows:



1094A044

Suppose you create the following base table:

```
CREATE MULTISET TABLE adw.customer (
    c_custkey      DECIMAL(18,0) NOT NULL,
    c_name         VARCHAR(30) CHARACTER SET LATIN CASESPECIFIC
                  NOT NULL,
    c_address       VARCHAR(40) CHARACTER SET LATIN CASESPECIFIC
                  NOT NULL,
    c_nationkey    DECIMAL(18,0) NOT NULL,
    c_phone        CHARACTER(10) CHARACTER SET LATIN CASESPECIFIC
                  NOT NULL,
    c_mktsegment   CHARACTER(10) CHARACTER SET LATIN CASESPECIFIC
                  NOT NULL,
    c_comment       CHARACTER(100) CHARACTER SET LATIN CASESPECIFIC
                  NOT NULL)
UNIQUE PRIMARY INDEX ( c_custkey )
INDEX ( c_nationkey );
```

To support group AMP access to the data in this table, you create the following global join index:

```
CREATE JOIN INDEX adw_ji AS
    SELECT (c_phone), (ROWID)
    FROM customer
PRIMARY INDEX(c_phone);
```

A typical query against the customer table might be something like the following:

```
SELECT c_name, c_mktsegment
FROM customer
WHERE c_phone = '5363333428';
```

To make the Optimizer aware of the opportunity for using group AMP access to respond to this query, you must first collect statistics on the *c\_phone* column of the base table. Otherwise, the Optimizer still uses the global join index, but generates an all-AMP plan instead of the more cost effective group AMP plan.

The main advantages of global join indexes are scalability and throughput. Whether a global join index supports faster processing than a NUSI largely depends on how busy the system is at the time the query is submitted. On a system with a light load, a query supported by a NUSI might run slightly faster, because a NUSI scan is one step that all AMPs perform in parallel. To use a global join index, the plan requires two steps to perform the same operation.

When a system is heavily loaded, a global join index is likely to provide a performance advantage, all things being equal, because it does not impel the overhead of an all-AMP operation. The higher the number of AMPs involved in satisfying a request, the higher the likelihood of experiencing resource contention and experiencing response delays, and the difference is more pronounced as a configuration grows in size. As a result, the practical benefits accrued from using a global join index rather than a NUSI also increase.

Because global join indexes can partially cover a query, some, or even most, of the columns requested by a query need not be defined in the join index itself. Multitable join indexes also support partial covering, though aggregate join indexes do not. Frequently only the primary index of the global join index is carried, along with the unique identifier of the base table it supports.

You can specify any of the following unique identifiers in the definition of a global join index:

- The row ID of the base table (expressed as the keyword ROWID).
- The primary index of the base table.
- A unique secondary index on the base table.

See “[Restrictions on Partial Covering by Join Indexes](#)” on page 575 for details.

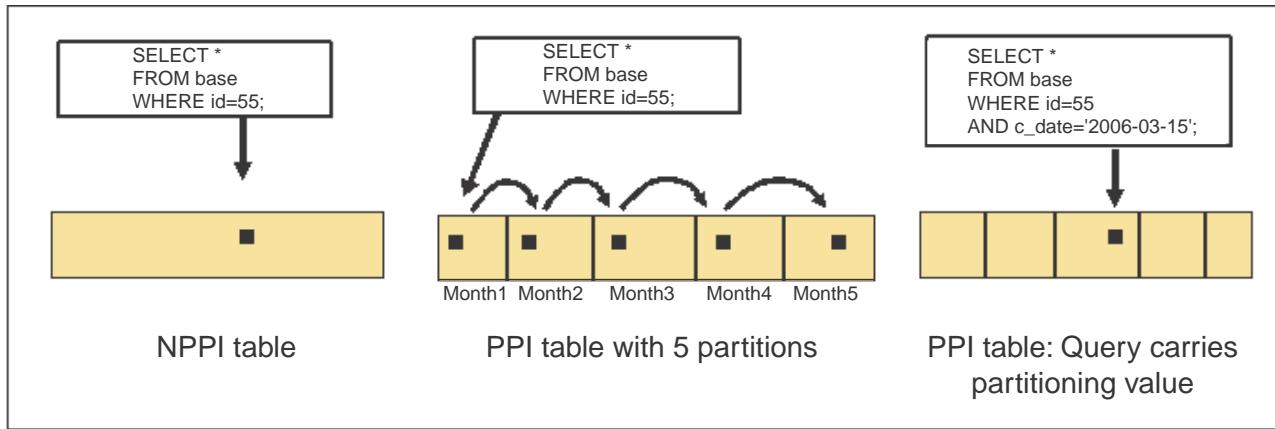
The Optimizer can also use multitable join indexes for partial query coverage capability (see “[Partial Query Coverage](#)” on page 505 and “[Restrictions on Partial Covering by Join Indexes](#)” on page 575).

## Single-AMP Queries and Partitioned Tables

Primary index partitioning refers to the physical ordering of rows within the table. Partitioned rows are grouped on each AMP based on a partitioning key, which can be one or multiple columns (see “[Row-partitioned Primary Indexes](#)” on page 267 for details). The partitioning columns need not be part of the primary index; however, when the partitioning column is different from the primary index, the performance of tactical queries that pass only a primary index value might be affected.

When the system performs a primary index access to a partitioned table and the query does not provide the partitioning key in a condition, each primary index partition might need to be probed individually to determine if it contains rows reflecting that value. As seen in the following graphic, if you specify a partitioning key value in a WHERE condition (in this case,

the condition is `c_date = '2006-03-15'`, the Optimizer can use row partition elimination to probe only the partition that contains the sought after value.



1094B045

Probing always occurs when the partitioning is non-unique because of the possibility that duplicate primary index rows with different partitioning key values might be spread across the table. The primary index of a partitioned table must be defined as non-unique unless its partitioning key is also included as part of the primary index definition (see “[Row-partitioned Primary Indexes](#)” on page 267). When the partitioning key is included as part of the primary index definition, the system does not need to probe each individual index partition.

## Recommendations for Tactical Queries and Partitioned Tables

When executing tactical queries based on primary index access to partitioned tables, consider the followings suggestions for enhancing query performance, listed in order of greatest general benefit:

- 1 The best performance is achieved when both of the following conditions are true:
  - The partitioned table is defined with a unique primary index.
  - The partitioned definition also contains the partitioning key for the table.

All UPIs on partitioned tables must contain the partitioning key. Because of this requirement, any primary index access returns a single row, and only a single partition must be probed to access that row.
- 2 If you cannot include the partitioning key in the primary index definition for a partitioned table, primary index access will also perform well if you specify a value for the partitioning key WHERE clause as an additional constraint. This specification eliminates the need to probe all of the partitions.
- 3 You can achieve good performance by defining a NUSI on the NUPI column set for the partitioned table.

- 4 If for some reason you cannot specify a partitioning key as a condition in a query against a partitioned table, you should consider the following alternative methods, where appropriate:

IF you cannot provide the value for the partitioning key in the WHERE clause, the PPI is a NUSI, and its values are ...		THEN consider creating this type of index on the primary index columns of the PPI ...
unique		USI.
not unique		global join index.

Either design strategy avoids the necessity of probing each partition because the indexes point directly to each relevant physical row.

- 5 If none of these methods is appropriate for your workloads, consider defining the partitioned table with fewer partitions. By making the partition granularity more coarse, you reduce the level of probing required for a primary index access.

## Sparse Join Indexes and Tactical Queries

Join indexes are particularly useful for tactical query applications. [Chapter 11: “Join and Hash Indexes,”](#) discusses join indexes more fully, but one interesting join index approach that is worth highlighting here for its relevance to tactical queries is the sparse join index.

Sparse join indexes include only a subset of a base table rows in their definition, using a WHERE clause to determine which base tables rows are retained and which are not (see [“Sparse Join Indexes” on page 556](#)). Sparse join indexes are quicker to build, faster to scan, and take up less disk space, depending on the degree of sparseness. Like all join indexes, sparse join indexes support single-AMP access based on their primary index definition.

### Sparse Join Index Defined on One Partition

A partitioned primary index can be defined on a join index as long as the index is not row compressed. You can also define a sparse join index on only one partition of a partitioned base table by expressing sparseness-defining criteria that match the borders of the partition.

Building sparse join indexes on partitions of partitioned tables that support single-AMP access is frequently useful for situations in which tactical queries always have both the sparse-defining column (in this example, a date range that matches one partition) and the primary index value (in this example, the store identifier) of the sparse join index. Of course, the tactical queries also need to be accessing a similar subset of columns from the base table: the ones carried in the sparse join index.

A different sparse join index could be built independently on several different partitions of the same partitioned table. As long as each query specifies a constraint that matches the sparse-defining columns for one of those sparse join indexes, the Optimizer can choose the appropriate one to use for the query.

The appropriate primary index for a sparse join index depends on what values the tactical queries specify when they are submitted.

## Considerations For Using Sparse Join Indexes With Dense NUPIs

Selecting the primary index for a sparse join index that has thousands of rows per value, with each AMP controlling some percentage of these values, provides several benefits and carries few of the negatives associated with a high number of duplicate primary index values.

For example:

- During join index creation there is no duplicate row checking as there is with a base table, so one of the principal reasons to avoid such high numbers of duplicates on a primary index does not apply to the case of creating a join index.
- The join itself can be more efficient with a higher numbers of NUPI duplicates because when so many rows carry the same NUPI row-hash value, the physical I/O involved in storing them can be less.
- While balanced processing is always important in selecting a primary index for a base table, the dense NUPI approach is appropriate for join indexes when it enables fast query execution and replaces an all-AMP alternative that would process only a few rows from each AMP.

On the other hand, it is *not* desirable to overload one AMP unduly, whether the access is single- or all-AMP. If an inordinate number of data blocks would have to be processed by one AMP using the dense NUPI approach, then parallelizing the work across all AMPs by selecting an alternative primary index is probably a better choice for enhancing performance.

- Designing a sparse join index to ensure that the number of distinct values in its index primary index is greater than the number of AMPs in the system is a good strategy to protect against too many queries being concentrated on too few AMPs. However, if the queries are very short and are infrequent, that concern is less important.

## All-AMP Queries

Some tactical queries require all-AMPs steps. All-AMPs queries are likely to have a more relaxed response time expectation than single-AMP queries even when they are capable, at times of low concurrency, of subsecond response.

### Coding Suggestions for All-AMP Tactical Queries

All-AMP tactical queries behave differently than few-AMP queries as the level of concurrency increases. Some suggestions for coding tactical queries that involve all-AMPs operations follow:

- Review the queries for unnecessary database access. If found, eliminate those accesses from the query.
- Tune partitioned tables so queries against them can avoid unnecessary probing.

- Rely on NUSI access, where possible, to avoid scanning a large table.
- Define a hash or join index that partitions the base table vertically where it makes sense to do so because scanning a join index that contains only a subset of the columns from the base table is faster than scanning the base table.
- Look for possibilities to break complex tactical queries into several smaller statements and encapsulating them within a stored procedure, particularly when all-AMP operations can be replaced by multiple single-AMP operations.
- Specify explicit ACCESS locks wherever possible (see *SQL Request and Transaction Processing* for further information) or set the default session read lock to READ UNCOMMITTED using the SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL statement (see *SQL Data Definition Language*).  
ACCESS locks are more important for all-AMP queries than for single-AMP queries because an all-AMP query can require access to more data over a longer period than a single-AMP query.

## All-AMP Tactical Queries and Partitioned Tables

The major benefit that can be realized for all-AMP tactical queries running against partitioned primary index tables is row partition elimination. Rather than scanning an entire table, it is possible to scan a smaller subset of rows if the query specifies the partitioning key value for the table.

Avoid adjusting the granularity of partitions more finely than is warranted by the majority of the queries that run against the table. For example, if users never access data with a granularity smaller than a monthly range, there is no additional performance advantage to be gained by defining partitions that have a smaller range than one month.

For all-AMP tactical queries, you should include the partitioning columns as a constraint in the WHERE clause of your queries. If a query joins partitioned tables that have identical partitioning, you can enhance join performance still further by including an additional equality constraint on the partitioning columns of the two tables.

### Group AMP Check in Final Query Step

Whether a query is single-AMP or all-AMP, simple or very complex, there is always an opportunity in the final step of the plan to reduce an all-AMP operation to a Group AMP operation.

Group AMP logic restricts which AMPs participate in the BYNET merge activity that is part of response processing. This action eliminates unnecessary all-AMP activities at the end of each query. While this feature is likely to have a greater impact on short, tactical queries, any time all-AMP activities can be eliminated anywhere in any query plan, the potential system throughput increases because resources are freed for other work.

To see how this works, examine the final step in a complex decision support query that returns millions of rows. In this case, because of the number of rows returned, each AMP is active in

response processing and the Group AMP group includes them all. For other queries, the group might be a subset of the AMPs that is determined when the step executes.

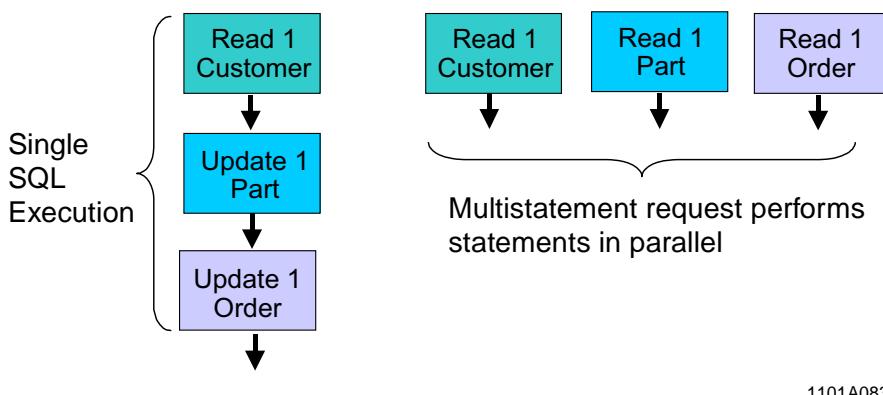
- 4) We do an all-AMPS RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1 (**group\_amps**), which is built locally on the AMPs. Then we do a SORT to order SORT to order Spool 1 by the sort key in spool field1. The result spool file will not be cached in memory. The size of Spool 1 is estimated with no confidence to be 379,305,309 rows. The estimated time for this step is 32 minutes and 32 seconds.

## Application Opportunities for Tactical Queries

In addition to physical database design choices, there are several application opportunities that benefit tactical query performance. The main areas of interest are multistatement requests and macros.

### Multistatement Requests

Multistatement requests are a Teradata Database feature in which multiple SQL statements are bundled together and treated as a single parsing and recovery unit, as illustrated by the following graphic:



1101A083

Because they are run in parallel, these single-AMP multistatement requests are processed with great efficiency. Multistatement requests are application-independent and can improve performance in a variety of ways. They are particularly useful in improving response times for the combined statements.

The benefits of multistatement requests include the following:

- Communication overhead is reduced
- One parser-optimization process is performed instead of several
- Greater inter-request parallelism is possible

### Coding Multistatement Requests

Multistatement requests are application-independent, but their behavior can be different depending on whether they are specified as an implicit transaction or as part of an explicit transaction and whether they are submitted in Teradata or ANSI/ISO session mode.

Teradata Database treats multistatement requests as single implicit transactions in Teradata session mode when no open BEGIN TRANSACTION statement precedes them.

With the exception of multistatement INSERT requests, Teradata Database treats multistatement requests as single recovery units only if they are executed either as implicit transactions in Teradata session mode, or in ANSI/ISO session mode. Depending on any errors generated by the statements in the transaction, either the entire transaction or only the erring request is rolled back. For example, in Teradata session mode within an explicit transaction, Teradata Database rolls the entire transaction back to the BEGIN TRANSACTION statement, so in this case, the complete transaction is the recovery unit.

If several UPDATE requests are specified within one multistatement request inside an explicit transaction in Teradata session mode, and one of them fails, then all are rolled back. The fewer UPDATE statements in the multistatement request, the lower the impact of the rollback. However, the more statements included in the request, the higher the degree of parallelism among them.

The rollback issue for UPDATE requests is *not* true for multistatement INSERT requests, where the statement independence feature can frequently enable multistatement INSERT requests to roll back only the statements that fail within an explicit transaction or multistatement request and not the entire transaction or request.

Statement independence supports the following multistatement INSERT data error types:

- Column-level CHECK constraint violations
- Data translation errors
- Duplicate row errors for SET tables
- Primary index uniqueness violations
- Referential integrity violations
- Secondary index uniqueness violations

Statement independence is *not* enabled for multistatement INSERT requests into tables defined with the following options:

- Triggers
- Hash indexes
- Join indexes

See “*INSERT/INSERT ... SELECT*” in *SQL Data Manipulation Language* for more information about statement independence. Note that various client data loading utilities also support statement independence. Consult the appropriate Teradata Tools and Utilities documentation for information about which load utilities and APIs support statement independence and what level of support they offer for the feature.

## ACCESS Locking and Multistatement Requests

If you want each statement in a multistatement request to use row hash-level ACCESS locking, specify the LOCKING ROW FOR ACCESS modifier preceding each individual SELECT statement in the multistatement request. See “[Coding Multistatement Requests](#)” on page 910

for additional details about how Teradata Database handles transactions, particularly those containing multistatement requests.

## Macros and Tactical Queries

You automatically get a multistatement request without coding it if you write multiple SQL statements within a macro because macros are always performed as a single request. The macro code can also specify parameters that are replaced by data each time the macro is performed. The most common way of substituting for the parameters is to specify a USING request modifier (see “USING Request Modifier” in *SQL Data Manipulation Language*).

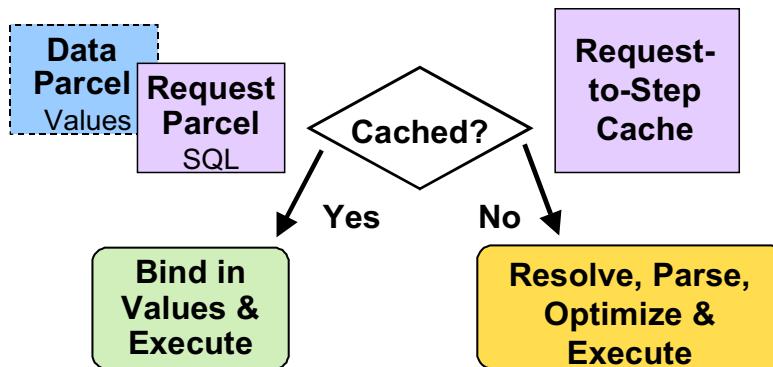
Macros are usually more efficient for repetitive queries than single DML statements. Unlike stored procedures, macros are not designed to return parameters to the requestor: they return only the answer set for the SQL statements they contain.

Macros have the following advantages:

- Network and channel traffic are reduced.
- Execution plans can be cached, reducing parsing engine overhead.
- They ensure the efficient performance of their component SQL statements.
- Reusable database components save client resources.
- They can be used to force data integrity and locking modifiers.
- They can be used to enforce security.

## Cached Plans

Caching of repeatable requests in the Request-to-Step Cache reduces performance time because parsing and optimizing do not need to be done when cached requests are repeated. For subsecond queries, cached plans significantly reduce the query time and enhance throughput. The following figure is a high-level flow chart for the request-to-steps cache that resides in each parsing engine on a Teradata system.



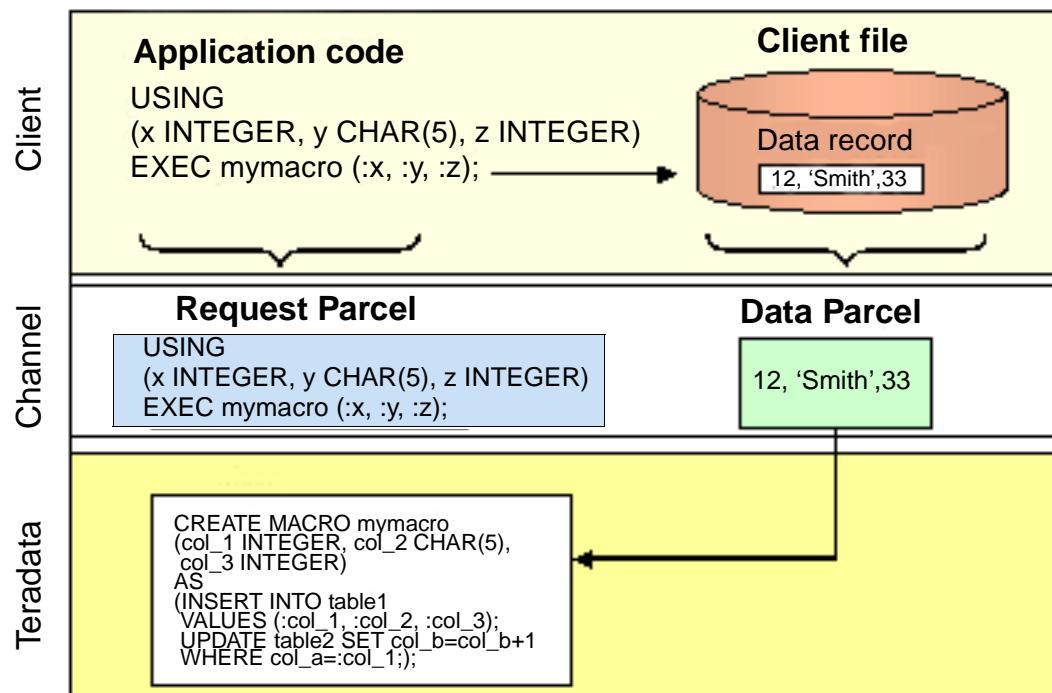
1101A084

For macros to be considered for caching, Teradata Database must know that the SQL text they contain is repeatable. All of the following attributes must be identical each time the request is sent to the Teradata platform:

- Host, workstation, or LAN type
- Default database name
- National character set
- Request text

Parameter variables in the macro support repeatability. These parameters must be specified in both the SQL code itself and in the USING request modifier that precedes the EXEC statement that invokes the macro.

The following picture illustrates the conditions required for caching the SQL statements for a macro. In this example, the request is made using BTEQ from a mainframe client across a block multiplexor channel. Note that the SQL code is in a request parcel, while the data values are transmitted in a data parcel. Both a request and a data parcel must be transmitted for the SQL to be cached immediately.



1101A090

The data values are passed from the application to the macro code where they are then processed. Their format depends on the programming conventions for the application. For example, in BTEQ you must use the .IMPORT FILE = command when you perform the macro, and the import file specified in the command contains records with the data values to be passed to the macro. The values in the file are positionally associated, from left to right, with the parameters in the USING request modifier.

## Other Tools Useful for Monitoring and Managing Tactical Queries

The following recommendations refer to monitoring and management facilities other than Database Query Logging or Teradata Viewpoint that might be active in a Teradata data warehouse. These recommendations are made with tactical query applications in mind; however, their value should be understood in light of the entire Teradata Database environment.

- Avoid using the account string expansion &T variable for tactical queries, because it is likely to write one row in AmpUsage for every request issued. This extra processing overhead can impact response time and reduces the throughput of single- or few-AMP tactical queries. See *Database Administration* for more information about account string expansion variables.
- Stop using Access Logging to log queries. Database Query Logging provides most of the same functionality and offers much more. DBQL also has considerably less impact on the performance of tactical query applications. See *Database Administration* and *SQL Data Definition Language* for more information about Access Logging and Query Logging.
- Taking frequent snapshots using Priority Scheduler monitor (every 5 or 10 seconds) incurs no meaningful overhead and does not affect tactical query performance. Information from this monitor output can be useful in understanding how priorities are functioning and which groups are using what extent of CPU activity.
- A ten-minute collection interval for ResUsage has proven sufficient in many diverse environments, and minimizes any impact on tactical queries. ResUsage data is critical to tracking overall system-level health and usage patterns, and for capacity planning. See *Resource Usage Macros and Tables* for more information about ResUsage.

## Monitoring Active Work

Workload management is critical to the success of tactical query applications. Improved workload management starts with understanding what is active on the platform and what the resource consumption profiles of your different applications look like.

### Recommended Monitoring Activities

A quick list of monitoring activities that offer a foundation for workload management follows:

- Use Teradata Viewpoint to monitor and manage the workload.
- Collect user resource usage detail data.

The *DBC.Acctg* table is the underlying table for the *AMPUsage* view, and captures data about user usage of CPU and I/O for each AMP. Heavy resource consumers over time, skewed work, and application usage trends can be identified.

- Collect ResUsage data.

The ResUsage tables report on the aggregate effect of all user requests on various system components over time, and can identify bottlenecks and capacity issues. See *Resource Usage Macros and Tables* for details on all ResUsage data and macros.

- Use the Lock Viewer Viewpoint portlet.

Lock Viewer is essential for identifying locking conflicts.

- Use Database query logging (DBQL).

The Database Query Log records details on queries run, including arrival rates, response times, objects accessed, and SQL statements performed. See *Database Administration* and *SQL Data Definition Language* for more information about Query Logging.

- Use the Priority Scheduler utility.

Priority Scheduler monitor output information shows comparative CPU usage across Priority Scheduler groups. This monitor output should be collected daily, minimally at a 5 or 10-minute interval, then summarized or charted. See *Utilities: Volume 2 (L-Z)* for more information about the Priority Scheduler utility.

- Enable canary queries.

You should use canary queries with tactical query applications. See “[Using Canary Queries](#)” on page 915 for more information about this monitoring technique.

## Using Canary Queries

Canary queries are SQL statements that represent the characteristics of a particular application. These queries are introduced into the work stream entering your data warehouse in order to monitor system responsiveness. Some sites run canary queries once every 1 to 5 minutes, while others run them only every 30 minutes. Canary queries provide a quick health check for tactical applications specifically and for your Teradata Database system in general. By monitoring the response times when the canary queries run, you can identify delays or congestion states early on.

Some users run canary queries in each Priority Scheduler Performance Group that is active, others only use them for their tactical query applications or in groups that have a defined service level.

Many sites make charts or graphs of canary query behavior during the previous 24 hours, with particular attention paid to any outliers. When the response time for a canary query is out of line with expectations, you can attempt to match that occurrence to system conditions at that time. It is a common practice for canary queries that exceed a specified response time threshold to send an e-mail alert to the DBA staff.

Take care to avoid over-scheduling canary queries. If the information your canary queries produce is too large to be easily processed and analyzed, reduce the scope of their execution.

Teradata Viewpoint is designed to automate the submission of canary queries. See the Teradata Viewpoint online HELP to learn how to do this.

The term *canary query* comes from the 19th century coal mining practice of lowering a caged canary into a mine to detect the presence of odorless, toxic gases. If the canary was dead when the cage was pulled back to the surface, the miners knew not to descend into the shaft until it could be properly ventilated. Because these small birds are so sensitive to toxic fumes, they

were also an excellent early warning system for the detection of toxic gases that might build up after the miners had already begun work in the mine. If your canary died, you knew it was time to ascend to the surface until the workplace could once again be made safe.

# APPENDIX A Notation Conventions

---

This appendix describes the notation conventions used in this book.

Throughout this manual, two conventions are used to describe the SQL syntax and code:

- Syntax diagrams, used to describe SQL syntax form, including options.
- Square braces in the text, used to represent options. The indicated parentheses are required when you specify options.

For example:

- DECIMAL [(n[,m])] means the decimal data type can be defined optionally:
  - without specifying the precision value *n* or scale value *m*
  - specifying precision (*n*) only
  - specifying both values (*n,m*)
  - you cannot specify scale without first defining precision.
- CHARACTER [(n)] means that use of (n) is optional.

The values for *n* and *m* are integers in all cases

Symbols from the predicate calculus, set theory, and dependency theory are also used occasionally to describe logical operations. See “[Predicate Calculus and Set Theory Notation Used in This Manual](#)” on page 919 and “[Dependency Theory Notation Used in This Manual](#)” on page 920.

## Table Column Definition and Constraint Abbreviations

The following set of abbreviations is used throughout this manual to denote various characteristics of, or constraints on, table columns.

Abbreviation	Definition
FK	Foreign key
NC	No changes allowed
ND	No duplicates allowed
NN	No nulls allowed
PK	Primary key
SA	System-assigned value

Abbreviation	Definition
UA	User-assigned value

## Character Symbols

The symbols, along with character sets with which they are used, are defined in the following table.

Symbol	Encoding	Meaning
a–z A–Z 0–9	Any	Any single byte Latin letter or digit.
ⓐ–ⓩ ⓐ–ⓩ ⓪–⓫	Any	Any fullwidth Latin letter or digit.
<	KanjiEBCDIC	Shift Out [SO] (0x0E). Indicates transition from single to multibyte character in KanjiEBCDIC.
>	KanjiEBCDIC	Shift In [SI] (0x0F). Indicates transition from multibyte to single byte KanjiEBCDIC.
T	Any	Any multibyte character. The encoding depends on the current character set. For KanjiEUC, code set 3 characters are always preceded by “ss <sub>3</sub> ”.
I	Any	Any single byte Hankaku Katakana character. In KanjiEUC, it must be preceded by “ss <sub>2</sub> ”, forming an individual multibyte character.
△	Any	Represents the graphic pad character.
Δ	Any	Represents a single or multibyte pad character, depending on context.
ss <sub>2</sub>	KanjiEUC	Represents the EUC code set 2 introducer (0x8E).
ss <sub>3</sub>	KanjiEUC	Represents the EUC code set 3 introducer (0x8F).

For example, string “TEST”, where each letter is intended to be a fullwidth character, is written as TEST. Occasionally, when encoding is important, hexadecimal representation is used.

For example, the following mixed single byte/multibyte character data in KanjiEBCDIC character set

`LMN<TEST>QRS`

is represented as:

`D3 D4 D5 0E 42E3 42C5 42E2 42E3 0F D8 D9 E2`

## Predicate Calculus and Set Theory Notation Used in This Manual

Relational databases are based on the theory of relations as developed in set theory. Predicate calculus is often the most unambiguous way to express certain relational concepts.

Occasionally this manual uses the following predicate calculus and set theory notation to explain concepts.

This symbol ...	Represents this phrase ...
<b>Predicate Calculus Notation</b>	
<code>iff</code>	If and only if
$\forall$	For all
$\exists$	There exists
<b>Set Theory Notation</b>	
$\emptyset$	Empty set
$\cup$	Union
$\cap$	Intersection
$\aleph$	Infinitely large set Pronounced <i>aleph</i>

## Dependency Theory Notation Used in This Manual

The decomposition of relations during normalization relies on the notation of dependency theory.

Occasionally this manual uses the following dependency theory notation to explain concepts:

This symbol ...	Represents this phrase ...
$\rightarrow$	Functionally determines.
$\overline{\rightarrow}$	Does <i>not</i> functionally determine.
$\Rightarrow$	Multivalued determines.

## APPENDIX B Teradata System Limits

This appendix provides the following Teradata Database limits.

- System limits
- Database limits
- Session limits
- System-derived and system-generated column data types

The reported limits apply only to the platform software. Platform-dependent client limits are not documented in this appendix.

## System Limits

The system specifications in the following tables apply to an entire Teradata Database *configuration*.

### Miscellaneous System Limits

Parameter	Value
Maximum number of combined databases and users.	$4.2 \times 10^9$
Maximum number of database objects per system lifetime.  A <i>database object</i> is any object whose definition is recorded in <i>DBC.TVM</i> .  Because the system does not reuse <i>DBC.TVM</i> IDs, this means that a maximum of 1,073,741,824 such objects can be created over the lifetime of any given system. At the rate of creating one new database object per minute, it would take 2,042 years to use 1,073,741,824 unique IDs.	1,073,741,824
Maximum size for a table header.	1 MB
Maximum size of table header cache.	$8 \times 10^6$ bytes
Maximum size of a response spool file row.	64KB
Maximum number of change logs that can be specified for a system at any point in time.	1,000,000
Maximum number of locks that can be placed on aggregate online archive logging tables, databases, or both per request.  The maximum number of locks that can be placed per LOGGING ONLINE ARCHIVE ON request is fixed at 25,000 and cannot be altered.	25,000

## Appendix B: Teradata System Limits

### System Limits

Parameter	Value
Maximum number of 64KB parse tree segments allocated for parsing requests.	12,000
Maximum number of nodes per system configuration. Limits on vprocs of each type restrict systems with a large number of nodes to fewer vprocs per node. Systems with the maximum vprocs per node cannot approach the maximum number of nodes.	1,024

## Message Limits

Parameter	Value
Maximum number of CLIV2 parcels per message	256
Maximum message size	~ 65,000 bytes This limit applies to messages to and from client systems and to some internal Teradata Database messages.
Maximum error message text size in a failure parcel	255 bytes

## Storage Limits

Parameter	Value
Total data capacity	~ 12 TB/AMP
Minimum data block size with small cylinders	~9KB 18 512-byte sectors
Default data block size with small cylinders	~127KB 254 512-byte sectors
Maximum data block size with small cylinders	~256KB 512 512-byte sectors
Minimum data block size with large cylinders	~21KB 42 512-byte sectors
Default data block size with large cylinders	~127KB 254 512-byte sectors
Maximum data block size for a large cylinder system that does not use 4KB alignment	1,023.5KB 2,047 sectors

Parameter	Value
Maximum data block size for a large cylinder system that uses 4KB alignment	1,024KB 2,048 sectors
Maximum number of data blocks that can be merged per data block merge	8
Maximum merge block ratio	100% of the maximum multirow block size for a table.

## Gateway and Vproc Limits

Parameter	Value
Maximum number of sessions per PE.	120
Maximum number of gateways per node.	Multiple.  This is true because the gateway runs in its own vproc on each node. See <i>Utilities</i> for details.
Maximum number of sessions per gateway.	Tunable: 1 - 2,147,483,647.  1,200 maximum certified.  The default is 600.  See <i>Utilities</i> for details.
Maximum number of vprocs per system.	30,720  This includes the sum of all of the following types of vproc for a configuration: <ul style="list-style-type: none"> <li>• AMP Access Module Processor vprocs</li> <li>• GTW Gateway Control vprocs</li> <li>• PE Parsing Engine vprocs</li> <li>• RSG Relay Services Gateway vprocs</li> <li>• TVS Teradata Virtual Storage allocator vprocs</li> </ul>
Maximum number of AMP vprocs per system.	16,200

## Appendix B: Teradata System Limits

### System Limits

Parameter	Value
<p>Maximum number of GTW vprocs per system.</p> <p>This is a soft limit that Teradata technical support personnel can reconfigure for you.</p> <p>Each GTW vproc on a node must be in a different host group. If two GTW vprocs are in the same host group, they must be on different nodes.</p>	<p>The default is one GTW vproc per node with all of them assigned to the same host group.</p> <p>The maximum depends on:</p> <ul style="list-style-type: none"> <li>Number of IP addresses assigned to each node.</li> <li>Number of host groups configured in the database.</li> </ul> <p>Each gateway on a node must be assigned to a different host group from any other gateway <i>on the same node</i> and each gateway needs to be assigned a disjoint set of IP addresses to service.</p> <p>This does <i>not</i> mean that all gateways <i>in a system</i> must be assigned to a different host group. It means that each gateway <i>on the same node</i> must be assigned to a different host group.</p> <p>Gateways on <i>different</i> nodes can be assigned to the same host group.</p>
Maximum number of PE vprocs per system.	2,048
This is a soft limit that Teradata technical support personnel can reconfigure for you.	
Maximum number of TVS allocator vprocs per system.	2,048
This is a soft limit that Teradata technical support personnel can reconfigure for you.	
Maximum number of vprocs, in any combination, per node.	127
Maximum number of AMP vprocs per cluster.	8
<p>Maximum number of external routine protected mode platform tasks per PE or AMP.</p> <p>This value is derived by subtracting 1 from the maximum total of PE and AMP vprocs per system (because each system must have at least one PE), which is 16,384. This is obviously not a practical configuration.</p> <p>The valid range is 0 to 20, inclusive. The limit is 20 platform tasks for <i>each</i> platform type, not 20 combined for both. See <i>Utilities</i> for details.</p>	20
<p>Maximum number of external routine secure mode platform tasks per PE or AMP.</p> <p>This value is derived by subtracting 1 from the maximum total of PE and AMP vprocs per system (because each system must have at least one PE), which is 16,384. This is obviously not a practical configuration.</p>	20
Size of a request control block	~ 40 bytes
<p>Default number of lock segments per AMP vproc.</p> <p>This is controlled by the NumLokSegs parameter in DBS Control.</p>	2

Parameter	Value
Maximum number of lock segments per AMP vproc.  This is controlled by the NumLokSegs parameter in DBS Control.	8
Default size of a lock segment.  This is controlled by the LockLogSegmentSize parameter in DBS Control.	64 KB
Maximum size of a lock segment.  This is controlled by the LockLogSegmentSize parameter in DBS Control.	1 MB
Default number of locks per AMP	3,200
Maximum number of locks per AMP.	209.000
Maximum size of the lock table per AMP.  The AMP lock table size is fixed at 2 MB and cannot be altered.	2 MB
Maximum size of the queue table FIFO runtime cache per PE.	<ul style="list-style-type: none"> <li>• 100 queue table entries</li> <li>• 1 MB</li> </ul>
Maximum number of SELECT AND CONSUME requests that can be in a delayed state per PE.	24
Amount of private disk swap space required per protected or secure mode server for C/C++ external routines per PE or AMP vproc.	256 KB
Amount of private disk swap space required per protected or secure mode server for Java external routines per node.	30 MB

## Hash Bucket Limits

Parameter	Value
Number of hash buckets per system.  This value is user-selectable. See the topic “Hash Maps” in Chapter 8 of <i>Database Design</i> for details.	<p>The number is user-selectable per system.</p> <p>The choices are:</p> <ul style="list-style-type: none"> <li>• 65,536</li> <li>• 1,048,576</li> </ul> <p>Bucket numbers range from 0 to the system maximum.</p>

Parameter	Value
Size of a hash bucket	<p>The size depends on the number of hash buckets on the system. If the system has:</p> <ul style="list-style-type: none"> <li>• 65,536 hash buckets, the size of a hash bucket is 16 bits.</li> <li>• 1,048,576 hash buckets, the size of a hash bucket is 20 bits.</li> </ul> <p>Set the default hash bucket size for your system using the DBS Control utility (see <i>Utilities: Volume 1 (A-K)</i> for details).</p>

## Interval Histogram Limits

Parameter	Value
Number of hash values.	$4.2 \times 10^9$
Maximum number of intervals per index or column set histogram.  The system-wide maximum number of interval histograms is set using the MaxStatsInterval parameter of the DBS Control record or your cost profile. For descriptions of the other parameters listed, see <i>SQL Request and Transaction Processing</i> .	500
Default number of intervals per index or column set histogram.	250
Maximum number of equal-height intervals per interval histogram.	500

# Database Limits

The database specifications in the following tables apply to a single Teradata database. The values presented are for their respective parameters individually and not in combination.

## Name and Title Size Limits

Parameter	Value
Maximum name size for database objects, for example, account, attribute, authorization, column, constraint, database, function, GLOP, index, macro, method, parameter, password, plan directive, procedure, profile, proxy user, query, role, stored procedure, table, transform group, trigger, UDF, UDM, UDT, using variable name, view.  See <i>SQL Fundamentals</i> for other applicable naming rules.	Depends on the setting of the DBS Control parameter EnableEON: <ul style="list-style-type: none"> <li>• 30 bytes, if EnableEON = no</li> <li>• 128 UNICODE characters, if EnableEON = yes</li> </ul>
Maximum system name size  Used in SQL statements for target level emulation. See <i>SQL Data Manipulation Language</i> .	Depends on the setting of the DBS Control parameter EnableEON: <ul style="list-style-type: none"> <li>• 30 characters, if EnableEON = no</li> <li>• 63 characters, if EnableEON = yes</li> </ul>
Maximum SQL text title size.	Depends on the setting of the DBS Control EnableEON field: <ul style="list-style-type: none"> <li>• 60 characters, if EnableEON = no</li> <li>• 256 characters, if EnableEON = yes</li> </ul>

## Table and View Limits

Parameter	Value
Maximum number of journal tables per database.	1
Maximum number of error tables per base data table.	1
Maximum number of columns per base data table or view.	2,048
Maximum number of columns per error table.  This limit includes 2,048 data table columns plus 13 error table columns.	2,061
Maximum number of UDT columns per base data table.  The same limit is true for both distinct and structured UDTs.  The absolute limit is 2,048, and the realizable number varies as a function of the number of other features declared for a table that occupy table header space.	~1,600

Parameter	Value
Maximum number of LOB and XML columns per base data table.  This limit includes predefined type LOB columns and UDT LOB columns.  A LOB UDT or XML UDT column counts as one column even if the UDT is a structured type that has multiple LOB attributes.	32
Maximum number of columns created over the life of a base data table.	2,560
Maximum number of rows per base data table.	Limited only by disk capacity.
Maximum number of bytes per table header per AMP.  A table header that is large enough to require more than ~64,000 bytes uses multiple 64KB rows per AMP up to 1 MB.  A table header that requires 64,000 or fewer bytes uses only a single row and does not use the additional rows that are required to contain a larger table header.  The <i>maximum</i> size for a table header is 1 MB.	1 MB
Maximum number of characters per SQL index constraint.	16,000
Maximum non-spool row size.	64,256 bytes
Maximum internal spool row size.	~ 1MB
Maximum size of the queue table FIFO runtime cache per table.	2,211 row entries
Maximum logical row size.  In this case, a logical row is defined as a base table row plus the sum of the bytes stored in a LOB or XML subtable for that row.  This value is derived by multiplying the maximum number of LOB or XML columns per base table (32) times the maximum size of a LOB or XML column, both of which are 2,097,088,000 8-bit bytes.  Remember that each LOB or XML column consumes 40 bytes of Object ID from the base table, so 1,248 of those 67,106,816,000 bytes cannot be used for data.	67,106,816,000 bytes
Maximum non-LOB column size for a nonpartitioned table.  This limit is based on subtracting the minimum row overhead value for a nonpartitioned table row (12 bytes) from the system-defined maximum row length (64,256 bytes).	64,244 bytes
Maximum non-LOB column size for a partitioned table.  This limit is based on subtracting the minimum row overhead value for a partitioned table row (16 bytes) from the system-defined maximum row length (64,256 bytes).	64,240 bytes
Maximum number of values (excluding NULLs) that can be multivalue compressed per base table column.	255
Maximum amount of BYTE data per column that can be multivalue compressed	4,093 bytes
Maximum amount of data per column that can be multivalue compressed for GRAPHIC, LATIN, KanjiSJIS, and UNICODE server character sets	8,188 characters

Parameter	Value
<p>Maximum number of columns that can be compressed per primary-indexed table using multivalue compression or algorithmic compression.</p> <p>This assumes that the object is not a non-partitioned NoPI table, a column-partitioned table or join index, or a global temporary trace table. All other tables, hash indexes, and join indexes must have a primary index, and primary indexes cannot be either multivalue compressed or algorithmically compressed. Because of this, the limit is the maximum number of columns that can be defined for a table, which is 2,048, minus 1.</p> <p>The limit for multivalue compression is far more likely to be reached because of table header overflow, but the amount of table header space that is available for multivalue compressed values is limited by a number of different factors. See <i>Database Design</i> for details.</p> <p>Join index columns inherit their compression characteristics from their parent tables.</p>	2,047
<p>Maximum number of columns that can be compressed per nonpartitioned NoPI table or column-partitioned table using multi-value compression or algorithmic compression</p> <p>Column-partitioned join index columns inherit their compression characteristics from their parent tables.</p>	2,048
Maximum number of algorithmically compressed values per base table column.	Unlimited
Maximum width of data that can be multivalue compressed for BYTE, BYTEINT, CHARACTER, GRAPHIC, VARCHAR, and VARGRAPHIC data types	Unlimited
Maximum width of data that can be multivalue compressed for data types other than BYTE, BYTEINT, CHARACTER, DATE, GRAPHIC, VARCHAR, and VARGRAPHIC	Unlimited
<p>Maximum width of data that can be algorithmically compressed.</p> <p>The maximum data width is unlimited if you specify only algorithmic compression for a column.</p> <p>If you specify a mix of multivalue and algorithmic compression for a column, then the limits for multivalue compression also apply for algorithmic compression.</p>	Unlimited
Maximum number of table-level CHECK constraints that can be defined per table.	100
Maximum number of primary indexes per table, hash index, or join index that is not a NoPI or column-partitioned database object.	1
Minimum number of primary indexes per primary-indexed table, hash index, or join index.	1
<p>Maximum number of primary indexes per non-partitioned NoPI table, column-partitioned table or join index, or global temporary trace table.</p> <p>This maximum applies <i>only</i> to nonpartitioned NoPI tables, column-partitioned tables and join indexes, and global temporary trace tables. All other tables, hash indexes, and join indexes <i>must</i> have a primary index.</p>	0
Maximum number of columns per primary index.	64
Maximum number of column partitions per table, including two columns partitions reserved for internal use).	2,050

Parameter	Value
Minimum number of column partition numbers that must be available for use by an ALTER TABLE request to alter a column partition.	1
Maximum partition number for a column partitioning level.	Maximum number of partitions for that level + 1
Maximum combined partition number for a single-level column-partitioned table or column-partitioned join index.	The same as the maximum partition number for the single partitioning level.
Maximum number of rows per hash bucket for a 44-bit uniqueness value.	17,592,186,044,415
Maximum combined partition number for a multilevel partitioning for 2-byte partitioning.	65,535
Maximum combined partition number for a multilevel partitioning join index for 8-byte partitioning.	9,223,372,036,854,775,807
Maximum number of ranges, including the NO RANGE, UNKNOWN, and NO RANGE OR UNKNOWN and UNKNOWN partitions, for a RANGE_N partitioning expression for 2-byte partitioning.  This value is limited by the largest possible INTEGER value.	65,533
Maximum number of ranges, including the NO RANGE, UNKNOWN, and NO RANGE OR UNKNOWN and UNKNOWN partitions, for a RANGE_N partitioning expression for 8-byte partitioning.  This value is limited by the largest possible BIGINT value.	9,223,372,036,854,775,807
Minimum value for $n$ in a RANGE#Ln expression.	1
Maximum value for $n$ in a RANGE#Ln expression for 2-byte partitioning	15
Maximum value for $n$ in a RANGE#Ln expression for 8-byte partitioning.	62
Maximum number of partitions, including the NO RANGE, UNKNOWN, and NO RANGE OR UNKNOWN partitions, for a single-level partitioning expression composed of a single RANGE_N function with INTEGER data type	2,147,483,647
Maximum number of ranges for a single-level partitioning expression composed of a single RANGE_N function with INTEGER data type that is used as a partitioning expression if the NO RANGE and UNKNOWN partitions are not specified.	65,533
Maximum number of ranges for a single-level partitioning expression composed of a single RANGE_N function with BIGINT data type that is used as a partitioning expression if the NO RANGE and UNKNOWN partitions are not specified.	9,223,372,036,854,775,807
Maximum number of ranges for a single-level partitioning expression composed of a single RANGE_N function with BIGINT data type that is used as a partitioning expression if both the NO RANGE and UNKNOWN partitions are specified.	9,223,372,036,854,775,807
Maximum value for a partitioning expression that is not based on a RANGE_N or CASE_N function.  This is allowed only for single-level partitioning.	65,535

Parameter	Value
Maximum number of defined partitions for a column partitioning level	The number of column partitions specified + 2
	The 2 additional partitions are reserved for internal use.
Maximum number of defined partitions for a row partitioning level if the row partitions specify the RANGE_N or CASE_N function.	The number of row partitions specified.
Maximum number of defined partitions for a row partitioning level if the row partitions do not specify the RANGE_N or CASE_N function.	65,535
Maximum number of partitions for a partitioning level when you specify an ADD clause.  This value is computed by adding the number of defined partitions for the level plus the value of the integer constant specified in the ADD clause.	9,223,372,036,854,775,807
Maximum number of partitions for a column partitioning level when you do not specify an ADD clause and at least one row partitioning level does not specify an ADD clause	The number of column partitions defined + 10.
Maximum number of column partitions for a column partitioning level when you do not specify an ADD clause, you also specify row partitioning, and each of the row partitions specifies an ADD clause.	The largest number for the column partitioning level that does not cause the partitioning to be 8-byte partitioning.
Maximum number of partitions for each row partitioning level without an ADD clause in level order, if using the number of row partitions defined as the maximum for this and any lower row partitioning level without an ADD clause.	The largest number for the column partitioning level that does not cause the partitioning to be 8-byte partitioning.
Maximum partition number for a row-partitioning level.	The same as the maximum number of partitions for the level.
Minimum number of partitions for a row-partitioning level.	2
Maximum number of partitions for a CASE_N partitioning expression.  This value is limited by the largest possible INTEGER value.	2,147,483,647
Maximum value for a RANGE_N function with an INTEGER data type.	2,147,483,647
Maximum value for a RANGE_N function with a BIGINT data type that is part of a partitioning expression.	9,223,372,036,854,775,805
Maximum value for a CASE_N function for both 2-byte and 8-byte partitioning.	2,147,483,647
Maximum number of partitioning levels for 2-byte partitioning.  Other limits can further restrict the number of levels for a specific partitioning.	15
Maximum number of partitioning levels for 8-byte partitioning.  Other limits can further restrict the number of levels for a specific partitioning.	62
Maximum value for $n$ for the system-derived column PARTITION#Ln.	62
Minimum number of partitions per row-partitioning level for a multilevel partitioning primary index.	1

Parameter	Value
Minimum number of partitions defined for a row-partitioning level.	2 or greater
Maximum number of partition number ranges from each level that are not eliminated for static row partition elimination for an 8-byte row-partitioned table or join index.	8,000
Maximum number of table-level constraints per base data table.	100
Maximum size of the SQL text for a table-level index CHECK constraint definition.	16,000 characters
Maximum number of referential integrity constraints per base data table.	64
Maximum number of columns per foreign and parent keys in a referential integrity relationship.	64
Maximum number of characters per string constant.	31,000
Maximum number of row-level security constraints per table, user, or profile.	5
Maximum number of row-level security statement-action UDFs that can be defined per table.	4
Maximum number of non-set row-level security constraint encodings that can be defined per constraint.  The valid range is from 1 to 10,000.  0 is not a valid non-set constraint encoding.	10,000
Maximum number of set row-level security constraint encodings that can be defined per constraint.	256

## Spool Space Limits

Parameter	Value
Maximum internal spool row size	~ 1MB

## BLOB, CLOB, XML, and Related Limits

Parameter	Value
Maximum BLOB object size	2,097,088,000 8-bit bytes
Maximum CLOB object size	<ul style="list-style-type: none"> <li>• 2,097,088,000 single-byte characters</li> <li>• 1,048,544,000 double-byte characters</li> </ul>
Maximum XML object size	2,097,088,000 8-bit bytes
Maximum number of LOB rows per rowkey per AMP for NoPI LOB or XML tables	<p>~ 256M</p> <p>The exact number is 268,435,455 LOB or XML rows per rowkey per AMP.</p>

Parameter	Value
Maximum size of the file name passed to the AS DEFERRED BY NAME option in a USING request modifier	VARCHAR(1024)

## User-Defined Data Type, ARRAY Data Type, and VARRAY Data Type Limits

Parameter	Value
<p>Maximum structured UDT size.</p> <p>This value is based on a table having a 1 byte (BYTEINT) primary index. Because a UDT column cannot be part of any index definition, there must be at least one non-UDT column in the table for its primary index.</p> <p>Row header overhead consumes 14 bytes in an NPPI table and 16 bytes in a PPI table, so the maximum structured UDT size is derived by subtracting 15 bytes (for an NPPI table) or 17 bytes (for a PPI table) from the row maximum of 64,256 bytes.</p>	<ul style="list-style-type: none"> <li>• 64,242 bytes (NoPI or column-partitioned table)</li> <li>• 64,241 bytes (NPPI table)</li> <li>• 64,239 bytes (PPI table)</li> </ul>
<p>Maximum number of UDT columns per base data table.</p> <p>The absolute limit is 2,048, and the realizable number varies as a function of the number of other features declared for a table that occupy table header space.</p> <p>The figure of 1,600 UDT columns assumes a FAT table header.</p> <p>This limit is true whether the UDT is a distinct or a structured type.</p>	~1,600
<p>Maximum database, user, base table, view, macro, index, trigger, stored procedure, UDF, UDM, UDT, constraint, or column name size.</p> <p>Other rules apply for Japanese character sets, which might restrict names to fewer than 30 bytes. See <i>SQL Fundamentals</i> for the applicable rules.</p>	30 bytes in Latin or Kanji internal representation
<p>Maximum number of attributes that can be specified for a structured UDT per CREATE TYPE or ALTER TYPE request.</p> <p>The maximum is platform-dependent, not absolute.</p>	300 - 512
<p>Maximum number of attributes that can be defined for a structured UDT.</p> <p>While you can specify no more than 300 to 512 attributes for a structured UDT per CREATE TYPE or ALTER TYPE request, you can submit any number of ALTER TYPE requests with the ADD ATTRIBUTE option specified as necessary to add additional attributes to the type up to the upper limit of approximately 4,000.</p>	~4,000
<p>Maximum number of levels of nesting of attributes that can be specified for a structured UDT.</p>	512
<p>Maximum number of methods associated with a UDT.</p> <p>There is no absolute limit on the number of methods that can be associated with a given UDT.</p> <p>Methods can have a variable number of parameters, and the number of parameters directly affects the limit, which is due to Parser memory restrictions.</p> <p>There is a workaround for this issue. See “ALTER TYPE” in <i>SQL Data Definition Language Detailed Topics</i> for details.</p>	~500

Parameter	Value
Maximum number of input parameters with a UDT data type of VARIANT_TYPE that can be declared for a UDF definition.	8
Minimum number of dimensions that can be specified for a multidimensional ARRAY or VARRAY data type.	2
Maximum number of dimensions that can be specified for a multidimensional ARRAY or VARRAY data type.	5

## Macro, UDF, SQL Stored Procedure, and External Routine Limits

Parameter	Value
Maximum number of parameters specified in a macro.	2,048
Maximum expanded text size for macros and views.	2 MB
Maximum number of open cursors per stored procedure.	15
Maximum number of result sets a stored procedure can return	15
Maximum number of columns returned by a dynamic result table function. The valid range is from 1 to 2,048. There is no default.	2,048
Maximum number of dynamic SQL requests per stored procedure	15
Maximum length of a dynamic SQL request in a stored procedure This includes its SQL text, the USING data (if any), and the CLIV2 parcel overhead.	Approximately 1 MB
Maximum combined size of the parameters for a stored procedure	1 MB for input parameters 64 KB for output (and input/output) parameters
Maximum size of condition names and UDF names specified in a stored procedure.	30 bytes, regardless of the DBS Control EnableEON setting.  <b>Note:</b> Other names specified in a stored procedure follow the EnableEON setting. See <i>SQL Fundamentals</i> for more information.
Maximum number of parameters specified in a UDF defined without dynamic UDT parameters.	128
Maximum number of parameters that can be defined for a constructor method for all types except ARRAY/VARRAY	128
Maximum number of parameters that can be defined for a constructor method of an ARRAY/VARRAY type	$n$ where $n$ is the number of elements defined for the type

Parameter	Value
Maximum number of combined return values and local variables that can be declared in a single UDF.	Unlimited
Maximum number of combined external routine return values and local variables that can be instantiated at the same time per session.	1,000
Maximum combined size of the parameters defined for a UDF.	1 MB for input parameters 64 KB for output parameters
Maximum number of parameters specified in a UDF defined with dynamic UDT parameters.  The valid range is from 0 to 15. The default is 0.	1,144
Maximum number of parameters specified in a method.	128
Maximum number of parameters specified in an SQL stored procedure.	256
Maximum number of parameters specified in an external stored procedure written in C or C++.	256
Maximum number of parameters specified in an external stored procedure written in Java.	255
Maximum size of an ARRAY or VARRAY UDT.  This limit does not include the number of bytes used by the row header and the primary index of a table.	64 KB
Maximum length of external name string for an external routine.  An external routine is the portion of a UDF, external stored procedure, or method that is written in C, C++, or Java (only external stored procedures can be written in Java). This is the code that defines the semantics for the UDF, procedure, or method.	1,000 characters
Maximum package path length for an external routine.	256 characters
Maximum number of nested CALL statements in a stored procedure.	15
Maximum SQL text size in a stored procedure.	64 KB
Maximum number of Statement Areas per SQL stored procedure diagnostics area. See <i>SQL Stored Procedures and Embedded SQL</i> and <i>SQL External Routine Programming</i> .	1
Maximum number of Condition Areas per SQL stored procedure diagnostics area. See <i>SQL Stored Procedures and Embedded SQL</i> and <i>SQL External Routine Programming</i> .	16

## Query and Workload Analysis Limits

Parameter	Value
Maximum size of the Index Wizard workload cache.  The default is 48 MB and the minimum is 32 MB.	187 MB

Parameter	Value
Maximum number of columns and indexes on which statistics can be collected or recollected at one time. 512 or limited by available parse tree memory and number of spool files.	512
Maximum number of pseudo indexes on which multicolumn statistics can be collected and maintained at one time.  A pseudo index is a file structure that allows you to collect statistics on a composite, or multicolumn, column set in the same way you collect statistics on a composite index.  This limit is independent of the number of indexes on which statistics can be collected and maintained.	32
Maximum number of sets of multicolumn statistics that can be collected on a table or join index if single-column PARTITION statistics are <i>not</i> collected on the table or index.	32
Maximum number of sets of multicolumn statistics that can be collected on a table or join index if single-column PARTITION statistics <i>are</i> collected on the table or index.	31
Maximum size of SQL query text overflow stored in QCD table <i>QryRelX</i> that can be read by the Teradata Index Wizard.	1 MB

## Secondary, Hash, and Join Index Limits

Parameter	Value
Number of tables that can be referenced in a join.	128
Minimum number of secondary, hash, and join indexes, in any combination, per base data table.  The only index required for any Teradata Database table or index is a primary index.  A primary index is <i>not</i> required or allowed for: <ul style="list-style-type: none"><li>• Global temporary trace tables</li><li>• Non-partitioned NoPI tables</li><li>• Column-partitioned tables and join indexes</li><li>• Secondary indexes</li></ul>	0
Maximum number of secondary, hash, and join indexes, in any combination, per base data table.  Each composite NUSI defined with an ORDER BY clause counts as 2 consecutive indexes in this calculation.  The number of system-defined secondary and single-table join indexes contributed by PRIMARY KEY and UNIQUE constraints counts against the combined limit of 32 secondary, hash, and join indexes per base data table.	32
Maximum number of columns referenced per secondary index.	64
Maximum number of columns referenced per single table in a hash or join index.	64
Maximum number of rows per secondary, hash, or join index.	Limited only by disk capacity.

Parameter	Value
Maximum number of columns referenced in the fixed part of a compressed join index.  Teradata Database implements two very different types of user-visible compression in the system. When describing compression of hash and join indexes, compression refers to a logical row compression in which multiple sets of nonrepeating column values are appended to a single set of repeating column values. This allows the system to store the repeating value set only once, while any nonrepeating column values are stored as logical segmental extensions of the base repeating set. Compression of column values refers one of the following: <ul style="list-style-type: none"><li>• Multi-value compression, in which Teradata Database stores the compressed values for a column one time only in the table header, not in the row itself.</li><li>• Algorithmic compression, in which you specify scalar UDFs to compress and decompress specified byte, character, or graphic data values. The compression and decompression algorithms used by algorithmic compression are determined by you and follow the rules that are defined for a given pair of compression and decompression algorithms.</li></ul>	64
Maximum number of columns referenced in the repeating part of a compressed join index.	64
Maximum number of columns in an uncompressed join index.	2,048
Maximum number of columns in a compressed join index.	128

## Reference Index Limits

Parameter	Value
Maximum number of reference indexes per base data table.  There is a maximum of 128 Reference Indexes in a table header, 64 from a parent table to child tables and 64 from child tables to a parent table.	64

## SQL Request and Response Limits

Parameter	Value
Maximum SQL text size per request.  This includes SQL request text, USING data, and parcel overhead.	1 MB
Maximum number of entries in an IN list.  There is no fixed limit on the number of entries in an IN list; however, other limits such as the maximum SQL text size, place a request-specific upper bound on this number.	Unlimited
Maximum SQL activity count size.	8 bytes
Maximum number of tables and single-table views that can be joined per query block.  This limit is controlled by the MaxJoinTables DBS Control parameter and the Cost Profile flags.	128
Maximum number of partitions for a hash join operation.	50
Maximum number of subquery nesting levels per query.	64

Parameter	Value
Maximum number of tables or single-table views that can be referenced per subquery. This limit is controlled by the MaxJoinTables DBS Control parameter and the Cost Profile flags.	128
Maximum number of fields in a USING row descriptor.	2,543
Maximum number of open cursors per embedded SQL program	16
Maximum SQL text response size.	1 MB
Maximum number of columns per DML request ORDER BY clause.	64
Maximum number of columns per DML request GROUP BY clause.	64
Maximum number of ORed conditions or IN list values per request	1,048,576
Maximum number of fields in a CONSTANT row	32,768

## Row-Level Security Constraint Limits

Parameter	Value
Maximum number of row-level security constraints per table.	5
Maximum number of hierarchical row-level security constraints per user or profile.	6
Maximum number of values per hierarchical row-level security constraint.	10,000
Maximum number of non-hierarchical row-level security constraints per user or profile.	2
Maximum number of values per non-hierarchical row-level security constraint.	256

## Session Limits

The session specifications in the following table apply to a single *session*:

Parameter	Value
Maximum number of sessions per PE.	120
Maximum number of sessions per gateway vproc. See <i>Utilities</i> for details.	Tunable: 1 - 2,147,483,647. 1,200 maximum certified. The default is 600.
Maximum number of active request result spool files per session.	16
Maximum number of parallel steps per request. Parallel steps can be used to process a request submitted within a transaction (which can be either explicit or implicit).	20

Parameter	Value
<p>Maximum number of channels required for various parallel step operations. The value per request is determined as follows:</p> <ul style="list-style-type: none"> <li>• Primary index request with an equality constraint.</li> <li>• Requests that do not involve row distribution.</li> <li>• Requests that involves redistribution of rows to other AMPs, such as a join or an INSERT ... SELECT operation.</li> </ul>	0 channels 2 channels 4 channels
Maximum number of materialized global temporary tables that can be materialized simultaneously per session.	2,000
Maximum number of volatile tables that can be instantiated simultaneously per session.	1,000
Maximum number of SQL stored procedure Diagnostic Areas that can be active per session.	1

## Appendix B: Teradata System Limits

### Session Limits

# APPENDIX C Designing With Task-Oriented Profiles

---

This appendix demonstrates how you can use task-oriented views to isolate users from direct contact with the database.

## Concepts, Policies, User Profiles, and Rules

### Isolating Users from the Database Using Views

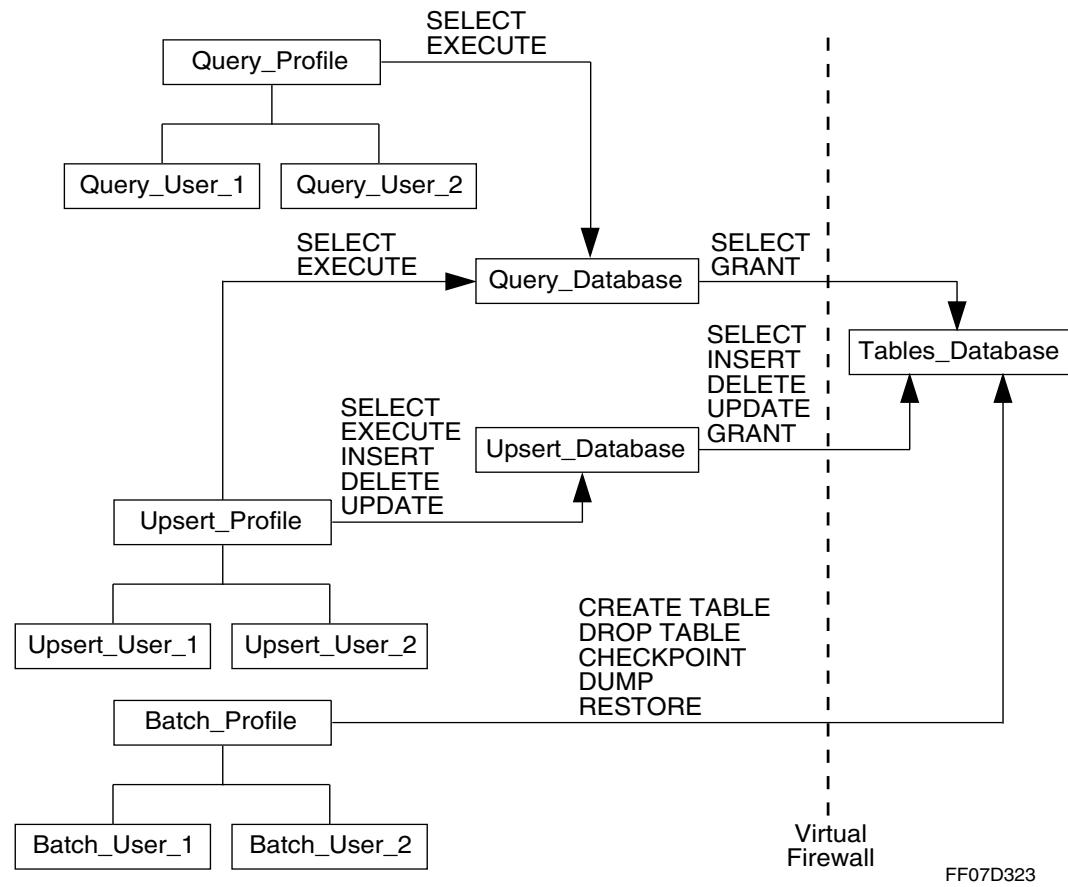
This topic presents a well-designed database structure based on a structure formulated and used by a Teradata customer.

The design incorporates different task-oriented profiles that access only a series of views and macros defined within several task-oriented databases.

Only these task-oriented access definition databases can access the base tables defined for the database. This design structure builds a virtual firewall between all users and the base tables they access.

### Flow Diagram of the Database Structure

The following graphic illustrates the work flows and individual task-oriented profiles and database objects defined for this database structure.



Note that *Tables\_Database* contains only base tables and their associated index subtables.

The example database defines three types of end user profile. The following table describes those types.

## End User Profiles

Profile	Description	Permitted SQL Statements and Utility Commands
Query_Profile	Defines privileges to the views, macros, and stored procedures defined in <i>Query_Database</i> .	<ul style="list-style-type: none"> <li>EXECUTE</li> <li>SELECT</li> </ul>
Upsert_Profile	Defines privileges to the views, macros, and stored procedures defined in <i>Upsert_Database</i> .	<ul style="list-style-type: none"> <li>DELETE</li> <li>EXECUTE</li> <li>INSERT</li> <li>SELECT</li> <li>UPDATE</li> </ul>

Profile	Description	Permitted SQL Statements and Utility Commands
Batch_Profile	Defines privileges to the tables defined in <i>Tables_Database</i> .	<ul style="list-style-type: none"> <li>• CHECKPOINT</li> <li>• CREATE TABLE</li> <li>• DROP TABLE</li> <li>• DUMP</li> <li>• RESTORE</li> </ul>

## Rules for End User Profiles

The following rules apply to all user profiles except *Batch\_Profile*:

- All users belong to one or more task profiles.
- Users inherit their privileges from the task profiles to which they belong.
- A user can belong to more than one task profile.
- Privileges are granted at DATABASE or USER levels *only*.
- *Query\_Profile* and *Upsert\_Profile* only have access to databases that contain macros, stored procedure definitions, and views exclusively.
- *Batch\_Profile* is the only profile that permits the base tables in *Tables\_Database* to be accessed directly.

## Rules for *Batch\_Profile*

- Users inherit their privileges from *Batch\_Profile*.
- Privileges are granted at Database or User levels *only*.
- *Batch\_Profile* has direct access to the base tables in *Tables\_Database* because of the functions performed using it.
- *Batch\_Profile* users can perform the following functions against *Tables\_Database*
  - CHECKPOINT transactions
  - CREATE TABLEs
  - DROP TABLEs
  - DUMP databases
  - RESTORE databases

## Rules for *Query\_Database*

Note that *Query\_Database* objects are organized in such a way that the entire database can be archived in one operation. You can also archive *Upsert\_Database* in the same operation if you choose to do so.

The following rules apply to *Query\_Database*.

- Both *Query\_Profile* and *Upsert\_Profile* users can perform the following functions against *Query\_Database*.
  - CALL permitted stored procedures

- EXECUTE permitted macros
- SELECT rows from *Tables\_Database* objects through views
- The database contains only views, stored procedure definitions, and macros that permit an end user to query or grant privileges on *Tables\_Database* objects indirectly.

## Rules for *Upsert\_Database*

Note that *Upsert\_Database* objects are organized in such a way that the entire database can be archived in one operation. You can also archive *Query\_Database* in the same operation if you choose to do so.

The word *upsert* derives from *update-insert* and implicitly refers to delete as well.

Only *Upsert\_Profile* users exclusively can perform the following functions against *Upsert\_Database*:

- CALL permitted stored procedures
- DELETE rows in *Tables\_Database* objects through views
- EXECUTE permitted macros
- INSERT rows into *Tables\_Database* objects through views
- SELECT rows from *Tables\_Database* objects through views
- UPDATE rows in *Tables\_Database* objects through views

The database contains only views, stored procedure definitions, and macros that permit an end user to perform the following actions on *Tables\_Database* objects indirectly.

- DELETE rows
- INSERT rows
- SELECT rows
- UPDATE rows
- GRANT privileges on database objects

# APPENDIX D Summary Physical Design Scenario

---

This appendix presents an overview of the physical design process.

The following scenario assumes that you are pursuing a 3NF logical model for your database schema.

## Prerequisites for the Process Review

Use the following checklist to ensure you are ready to enter the physical design phase of your project.

√	Physical Design Phase Entry Criteria
	Fully documented, fully normalized logical model
	Fully documented column and table demographics
	Good understanding of Teradata architecture-specific physical design principles

## Process Review

The following are the high-level stages in the physical design process:

- 1 Identify all index and partitioning candidates.
- 2 Review table demographics, then verify or modify index and partitioning choices.
- 3 Evaluate denormalization possibilities, including dimensional views, then verify or modify index and partitioning choices.
- 4 Evaluate derived data, then verify or modify index and partitioning choices.
- 5 Implement the system and initiate production processing.
- 6 Track changing demographics.
- 7 Review indexes and partitioning periodically to ensure they are still the best choices.
- 8 Perform appropriate EXPLAIN request modifiers periodically to track whether defined indexes are being selected for query processing by the Optimizer and whether the current partitioning is still facilitating row and column partition elimination.
- 9 Reevaluate table use.

- 10 Document any changes.
- 11 Archive older versions of your findings for use in analyzing trends.

## APPENDIX E Sample Worksheet Forms

---

This appendix contains a complete set of forms that you can print or photocopy and use for your database design projects.

The following forms are included:

- Domains Form
- Constraints Form
- System Form
- Report/Query Analysis Form
- Table Access Summary By Column Form
- Table Form
- Row Size Calculation Form for Byte-Packed Format Systems, parts 1 and 2
- Row Size Calculation Form for Byte-Aligned Format Systems, parts 1 and 2

Because several data types have different storage sizes on byte-packed format and 64-bit byte-aligned format systems (see [Chapter 14: “Database-Level Capacity Planning Considerations”](#) for details), separate forms are provided for the two architectures to minimize the possibility of mistakenly entering column values that are either too small or too large.

## Data Types

Bigint	I8	Interval Month	IMO	PERIOD(Time)	PT
BLOB	BL(n)	Interval Day	ID	PERIOD(Time With Time Zone)	PTTZ
Byte	B(n)	Interval Day To Hours	IDH	PERIOD(Timestamp)	PTS
Byteint	II	Interval Day To Minute	IDM	PERIOD(Timestamp <UNTIL_CHANGED>)	PTS_UC
Character	C(n)	Interval Day To Second	IDS	PERIOD(Timestamp With Time Zone)	PTSTZ
Character Varying	CV(n)	Interval Hours	IHS	PERIOD(Timestamp With Time Zone <UNTIL_CHANGED>)	PTSTZ_UC
CLOB	CL(n)	Interval Hour To Minute	IHM	Real	R
Date	D	Interval Hour To Second	IHS	Smallint	I2
Decimal	DEC(n,m)	Interval Minute	IM	Time	T
Double Precision	DP	Interval Minute To Second	IMS	Timestamp	TS
Float	F(n)	Interval Second	IS	Time With TimeZone	TTZ
Graphic	G(n)	Long Character Varying	LVC	Timestamp With TimeZone	TSTZ
Integer	I	Long Graphic Varying	LVG	Varbyte	VB
Interval Year	IY	Numeric	NUM(n,m)	Varchar	VC(n)
Interval Year to Month	IYM	PERIOD(Date)	PD	Vargraphic	VG(n)

1094H332

## Constraints

Page: \_\_\_\_\_ Of \_\_\_\_\_

ELDM Page: \_\_\_\_\_

System: \_\_\_\_\_

Constraint Number	Constraint Description

FF07D333

## System

Page: \_\_\_\_\_ Of \_\_\_\_\_

ELDM Page: \_\_\_\_\_

## System:

**System Name:** \_\_\_\_\_

## **System Description:**

FF07D335

## Report/Query Analysis Form

**Frequency or Importance Ranking:** \_\_\_\_\_

**Report/Query Description:** \_\_\_\_\_  
\_\_\_\_\_

**Table:**

**Number Of Output Rows:**

Input Value Or Source	
Access Column(s)	

**Table:**

**Number Of Output Rows:**

Input Value Or Source	
Access Column(s)	

**Table:**

**Number Of Output Rows:**

Input Value Or Source	
Access Column(s)	

**Table:**

**Number Of Output Rows:**

Input Value Or Source	
Access Column(s)	

FF07D337

## Table Access Summary By Column

Page: \_\_\_\_\_ Of \_\_\_\_\_

ELDM Page: \_\_\_\_\_

System: \_\_\_\_\_

**Table Name:**

**Column Names:**

FF07D334

<b>Table</b> Page: _____ Of _____				
Table Name		Table Type	Cardinality	Data Protection
Column Name				
PK/FK/ID				
Constraint Number				
Value Access Frequency				
Joint Access Frequency				
Joint Access Rows				
Distinct Values				
Maximum Rows/Value				
Maximum Rows Null				
Typical Rows/Value				
Change Rating				
PI/SI				
Sample Data				

1094E336

## Row Size Calculation Form for Packed64 Systems - Page 1 of 2

Table Name:

Variable Data Detail

Column Name	Type	Max	Avg		Data Type	Number of Columns	Sizing Factor	Total
					Byteint	*	1	=
					Smallint	*	2	=
					BIGINT	*	8	=
					Integer	*	4	=
					Date	*	4	=
					Time(ANSI)	*	6	=
					Time With Time Zone	*	8	=
					Timestamp	*	10	=
					Timestamp With Time Zone	*	12	=
					Interval Year	*	2	=
					Interval Year to Month	*	4	=
					Interval Month	*	2	=
					Interval Day	*	2	=
					Interval Day to Hour	*	4	=
					Interval Day to Minute	*	8	=
					Interval Day to Second	*	10	=
					Interval Hour	*	2	=
					Interval Hour to Minute	*	4	=
					Interval Hour to Second	*	8	=
					Interval Minute	*	2	=
					Interval Minute to Second	*	6	=
					Interval Second	*	6	=
					Period (Date)	*	8	=
					Period (Time)	*	12	=
					Period (Time With Time Zone)	*	16	=
					Period (Timestamp)	*	16	=
					Period (Timestamp <UNTIL_CHANGED>)	*	11	=
					Period (Timestamp With Time Zone)	*	24	=
					Period (Timestamp With Time Zone <UNTIL_CHANGED>)	*	13	=
					Decimal or Number 1-2	*	1	=
					3-4	*	2	=
					5-9	*	4	=
					10-18	*	8	=
					19-38	*	16	=
					LOB	*	39	=
					Float	*	8	=
					Fixed	SUM(n)	=	
					Variable	SUM(a)	=	
						Logical Size	=	
						UDTs	=	
						NPPI Overhead	=	14
						PPI Overhead	=	18
					Variable Column Offsets		=	
					_____ Compress Values			
					_____ Nullable Columns			
					_____ /8 (Quotient only)		=	
						Physical Size**	=	
				SUM(a)				

SUM(a)=SUM of the AVERAGE number of bytes expected for the variable column.  
 SUM(n)=SUM of the CHAR and GRAPHIC column bytes.

\*\*Round up to an even number of bytes

1094L347

## Row Size Calculation Form for Packed64 Systems - Page 2 of 2

Table Name:

Variable Data Detail

UDT Name	Number of Columns	Sizing Factor	Total
		* 1 =	
		* 2 =	
		* 4 =	
		* 4 =	
		* 6 =	
		* 8 =	
		* 10 =	
		* 12 =	
		* 2 =	
		* 4 =	
		* 2 =	
		* 2 =	
		* 4 =	
		* 8 =	
		* 10 =	
		* 2 =	
		* 4 =	
		* 8 =	
		* 2 =	
		* 6 =	
		* 6 =	
		* 1 =	
		* 2 =	
		* 4 =	
		* 8 =	
		* 39 =	
		* 8 =	
	SUM(n)	=	

UDT Name	Number of Columns	Sizing Factor	Total
		* 1 =	
		* 2 =	
		* 4 =	
		* 4 =	
		* 6 =	
		* 8 =	
		* 10 =	
		* 12 =	
		* 2 =	
		* 4 =	
		* 2 =	
		* 2 =	
		* 4 =	
		* 8 =	
		* 10 =	
		* 2 =	
		* 4 =	
		* 8 =	
		* 2 =	
		* 6 =	
		* 6 =	
		* 1 =	
		* 2 =	
		* 4 =	
		* 8 =	
		* 39 =	
		* 8 =	
	SUM(n)	=	

1094D055

Row Size Calculation Form for Aligned Row Format Systems - Page 1 of 2

Table Name:

## Variable Data Detail

Data Type	Number of Columns	Sizing Factor	Total
Byteint	*	1	=
Smallint	*	2	=
BIGINT	*	8	=
Integer	*	4	=
Date	*	4	=
Time(ANSI)	*	8	=
Time With Time Zone	*	8	=
Timestamp	*	12	=
Timestamp With Time Zone	*	12	=
Interval Year	*	2	=
Interval Year to Month	*	4	=
Interval Month	*	2	=
Interval Day	*	2	=
Interval Day to Hour	*	4	=
Interval Day to Minute	*	8	=
Interval Day to Second	*	12	=
Interval Hour	*	2	=
Interval Hour to Minute	*	4	=
Interval Hour to Second	*	8	=
Interval Minute	*	2	=
Interval Minute to Second	*	8	=
Interval Second	*	8	=
Period (Date)	*	8	=
Period (Time)	*	16	=
Period (Time With Time Zone)	*	16	=
Period (Timestamp)	*	20	=
Period (Timestamp <UNTIL_CHANGED>)	*	11	=
Period (Timestamp With Time Zone)	*	24	=
Period (Timestamp With Time Zone <UNTIL_CHANGED>)	*	13	=
Decimal or Number 1-2	*	1	=
3-4	*	2	=
5-9	*	4	=
10-18	*	8	=
19-38	*	16	=
LOB	*	39	=
Float	*	8	=
Fixed	SUM(n)		=
Variable	SUM(a)		=
	Logical Size		=
	UDTs		=
	NPPI Overhead	=	14
	PPI Overhead	=	18
Variable Column Offsets			=
_____ Compress Values			
_____ Nullable Columns			
_____ /8 (Quotient only)			=
	Physical Size**		=

$\text{SUM}(a) = \text{SUM}$  of the  $\text{AVERAGE}$  number of bytes expected for the variable column.

**SUM(a)=SUM of the AVERAGE number of bytes expected**  
**SUM(n)=SUM of the CHAR and GRAPHIC column bytes.**

\*\*Round up to an even number of bytes

1094E064

## Row Size Calculation Form for Aligned Row Format Systems - Page 2 of 2

Table Name:

Variable Data Detail

UDT Name	Number of Columns	Sizing Factor	Total
		* 1 =	
		* 2 =	
		* 4 =	
		* 4 =	
		* 6 =	
		* 8 =	
		* 10 =	
		* 12 =	
		* 2 =	
		* 4 =	
		* 2 =	
		* 2 =	
		* 4 =	
		* 8 =	
		* 10 =	
		* 2 =	
		* 4 =	
		* 8 =	
		* 2 =	
		* 6 =	
		* 6 =	
		* 1 =	
		* 2 =	
		* 4 =	
		* 8 =	
		* 39 =	
		* 8 =	
		SUM(n) =	

UDT Name	Number of Columns	Sizing Factor	Total
		* 1 =	
		* 2 =	
		* 4 =	
		* 4 =	
		* 6 =	
		* 8 =	
		* 10 =	
		* 12 =	
		* 2 =	
		* 4 =	
		* 2 =	
		* 2 =	
		* 4 =	
		* 8 =	
		* 10 =	
		* 2 =	
		* 4 =	
		* 8 =	
		* 2 =	
		* 6 =	
		* 6 =	
		* 1 =	
		* 2 =	
		* 4 =	
		* 8 =	
		* 39 =	
		* 8 =	
		SUM(n) =	

1094C065

## Appendix E: Sample Worksheet Forms

# APPENDIX F Designing Tables for Optimal Performance

---

You can optimize some of the CREATE TABLE and ALTER TABLE table parameters to improve efficiency in accessing and maintaining the table.

## Minimizing Table Size

Request performance is directly proportional to the size of the set of tables accessed. As table size increases, Teradata Database requires additional I/O operations to retrieve or update data.

For large tables that cannot be reduced in size, you can use the single-table join index, partitioned primary index, and column partitioning features to limit the portion of a table that must be scanned to access needed data.

## Reducing the Number of Table Columns

As the number and size of columns increases, the row size necessarily increases. A row cannot occupy more than one data block, so if a row exceeds the specified maximum data block size, Teradata Database returns an error message.

## Adjusting the DATABLOCKSIZE and MERGEBLOCKRATIO Table Parameters

You can control the default size for multirow data blocks on a table-by-table basis in a CREATE TABLE or ALTER TABLE request using the DATABLOCKSIZE and MERGEBLOCKRATIO parameters:

Disk arrays can scan at higher rates if the I/Os are larger, but larger I/Os can be less efficient for row-at-a-time access which requires that the entire data block be read for the relatively few bytes contained in a row. Cylinder reads enable smaller data blocks for row-at-a-time access and large reads for scans.

For average workloads, the benefits of using large data blocks outweighs the small penalty associated with row-at-a-time access up to 64 KB. Setting the data block size requires more judgment and analysis for 128KB and 1MB data blocks where the penalty for row-at-a-time access becomes measurable.

IF you specify...	THEN...
DATABLOCKSIZE in CREATE TABLE	<p>the data block can grow to the size specified in DATABLOCKSIZE instead of being limited to the global PermDBSize (see “PermDBSize” in <i>Utilities: Volume 1 (A-K)</i>).</p> <p>For any row, Teradata Database uses only the data block size required to contain the row.</p> <p><b>Note:</b> If you use block level compression, adjust DATABLOCKSIZE to the maximum, 255 sectors.</p>
DATABLOCKSIZE in ALTER TABLE	<p>the data blocks can grow to the size specified in DATABLOCKSIZE when the row size requires the growth.</p> <p>Whether data blocks are adjusted to that new size immediately or gradually over a long period of time depends on the use of the IMMEDIATE clause.</p>
MERGEBLOCKRATIO in CREATE TABLE	<p>Teradata Database limits attempts to combine blocks if the result is larger than the specified percent of the maximum multirow data block size.</p>
MERGEBLOCKRATIO in ALTER TABLE	<p>the size of the resulting block when multiple existing data blocks are being merged has an upper limit.</p> <p>The limit depends on whether Teradata Database determines that logically adjacent data blocks can be merged with the single data block being modified or not.</p> <p>Data blocks can still be initially loaded at the PermDBSize specification or the data block size specified with the DATABLOCKSIZE option. Merges occur only during full-table modifications.</p>
the IMMEDIATE clause	<p>the rows in all existing data blocks of the table are repacked into data blocks using the newly specified size. For large tables, this can be a time-consuming operation, requiring spool space to accommodate 2 copies of the table while it is being rebuilt.</p> <p>If you do not specify the IMMEDIATE clause, existing data blocks are not modified. When individual data blocks of the table are modified as a result of user transactions, the new value of DATABLOCKSIZE is used. Thus, the table changes over time to reflect the new data block size.</p>

To specify the global data block size, use PermDBSize (see “PermDBSize” in *Utilities: Volume 1 (A-K)*).

## Adjusting FREESPACE

You can specify the default value for free space left on a cylinder during certain operations on a table-by-table basis by specifying the FREESPACE parameter in the CREATE TABLE and ALTER TABLE requests.

This parameter enables you to select a different value for tables that are constantly modified versus tables that are only read after they are loaded. To specify the global free space value, use the FreeSpacePercent command of the DBS Control utility (see “FreeSpacePercent” in *Utilities: Volume 1 (A-K)*).

## Using Identity Columns, Compression, and Referential Integrity for Optimal Performance Design

The following chapters in this book contain information about designing tables optimally using referential integrity and compression, respectively.

- [“Chapter 9 Primary Indexes and NoPI Objects” on page 261](#)
- [“Chapter 12 Designing for Database Integrity” on page 617](#)
- [“Chapter 14 Database-Level Capacity Planning Considerations” on page 691](#)

## Using Indexes to Enhance Performance

The following chapters in this book contain information about designing tables optimally using indexes.

- [“Chapter 8 Teradata Database Indexes and Partitioning” on page 191](#)
- [“Chapter 9 Primary Indexes and NoPI Objects” on page 261](#)
- [“Chapter 10 Secondary Indexes” on page 455](#)
- [“Chapter 11 Join and Hash Indexes” on page 499](#)

## Understanding the Effects of Altering Tables

Using the ALTER TABLE statement can affect system performance and space requirements.

**Note:** Changes to the Data Dictionary resulting from these actions have minimal effect on performance.

Action	Performance Impact	Space Requirements
Add a column (COMPRESS, NULL)	All table rows are changed if a new presence byte is added.	Slight increase in required permanent space.
Add a column (NOT NULL, DEFAULT, and WITH DEFAULT)	All table rows are changed.	Increase in required permanent space.
Add a column (NULL, fixed-length)	All table rows are changed.	Increase in required permanent space.

Appendix F: Designing Tables for Optimal Performance  
Understanding the Effects of Altering Tables

Action	Performance Impact	Space Requirements
Add a column (NULL, variable length)	All table rows are changed.	Slight increase in required permanent space.
Add FALBACK	Entire table is accessed to create the fallback copy.  Long-term performance effects.	Approximately doubled the required permanent space.
Add CHECK constraints	Takes time to validate rows, which impacts performance.	Unchanged.
Add referential integrity	Takes time to check data.  Impacts performance long term. Similar to adding indexes.	Possible large increase in the following. <ul style="list-style-type: none"> <li>Spool space.</li> <li>Permanent space (for index if not soft batch).</li> </ul>
Change the format, title, default	No impact.	Unchanged.
Change the cylinder free space percent	<ul style="list-style-type: none"> <li>Raising the free space percent can make inserting new data less expensive by reducing migration and cylinder allocations.</li> <li>Lowering the free space percent has the opposite effect.</li> </ul>	Increase in required permanent space for operations such as default maximum, MultiLoad, restore.
Change the maximum multirow block size	<ul style="list-style-type: none"> <li>If the table is not updated after the change, then there is no impact.</li> <li>If the table is changed, there can be performance impact whether or not the IMMEDIATE clause is specified.</li> </ul>	<ul style="list-style-type: none"> <li>Slight increase in required permanent space for smaller values.</li> <li>Slight decrease in required permanent space for larger values.</li> </ul>
Delete the FALBACK option	FALBACK subtable is deleted.  Long-term performance effects.	Approximately half the required permanent space.
Drop a column	All table rows are changed.	Decrease in required permanent space.

## APPENDIX G References

---

This appendix provides a bibliography of Teradata publications related to database design.

Carrie Ballinger, *Implementing Tactical Queries: The Basics*, Teradata Database Orange Book 541-0005686A02, 2006.

Carrie Ballinger, *Techniques for Tuning Teradata to Support Tactical Query Consistency*, Teradata Database Orange Book 541-0005687A02, 2006.

Helen Fan and May Pederson, *Teradata Operational Data Store*, Teradata Database White Paper 541-0003976A04, 2003.

Jerry Klindt, Paul Sinclair, Steve Molini, Jeremy Davis, Rama Krishna Korlapati, Hong Gui, and Stephen Brobst, *Partitioned Primary Index Usage (Single-Level and Multilevel Partitioning)*, Teradata Database Orange Book 541-0003869E02, 2008.

Tam Ly, *No Primary Index (NoPI) Table User's Guide*, Teradata Database Orange Book 541-0007565B02, 2009.

Paul Sinclair, *Increased Partition Limit and Other Partitioning Enhancements*, Teradata Database Orange Book 541-0009027A02, 2011.

Paul Sinclair and Carrie Ballinger, *Teradata Columnar*, Teradata Database Orange Book 541-0009036A02, 2011.

Colin White, "In the Beginning: An RDBMS History," *Teradata Magazine*, 4(3):32-39, 2004.

## Appendix G: References

# Glossary

## Numerics

**2-Byte Partitioning** This term refers to the size of the [Internal Partition Number](#) for a partitioned table or join index.

The maximum number of column partitions for 2-byte partitioning is 65,534, including 2 partitions for internal use by Teradata Database.

The maximum partition number for 2-byte partitioning of a column-partitioned table or join index is 65,535 because Teradata Database always reserves at least one column partition number for altering a column partition. However, a column-partitioned table or join index can never have more than 2,050 column partitions because the maximum number of columns for a table or join index is 2,048.

The data type for the system-derived columns PARTITION[#Ln] is INTEGER for 2-byte partitioning. In this case, there is a maximum PARTITION level is 15.

**8-Byte Partitioning** This term refers to the size of the [Internal Partition Number](#) for a partitioned table or join index.

If the maximum combined partition number for a partitioned table or join index is greater than 65,535, it has 8-byte partitioning by default.

The maximum number of ranges for a partitioning expression composed from only RANGE\_N functions column partitions for 8-byte partitioning is 9,223,372,036,854,775,805, including 2 partitions for internal use by Teradata Database. This requires 8-byte partitioning.

The data type for the system-derived columns PARTITION[#Ln] is BIGINT for 8-byte partitioning. In this case, there is a maximum PARTITION level is 62.

**2VL** Two-Valued Logic.

**3GL** Third (3rd) Generation Language.

**3NF** Third (3rd) Normal Form. See [Third Normal Form](#).

**3VL** Three-Valued Logic.

**4VL** Four-Valued Logic.

## A

**ACM** Association for Computing Machinery.

**Aggregate Join Index** A join index that precomputes one or more aggregate columns. An aggregate join index can be built using built-in functions such as AVG, SUM, MIN, and MAX, or using UDFs.

**AK** See [Alternate Key](#).

**ALC** See [Algorithmic Compression](#).

**Aligned Configuration** A Teradata Database system with at least one 4KB sector data storage device using native 4KB sectors. Compare with [Unaligned Configuration](#).

**Algorithmic Compression** Any user-defined algorithm pair for compressing and decompressing column values is considered to be an algorithmic compression method. An algorithmic compression method must first be defined as a scalar UDF, then specified as part of the definition for any column to which it applies in a CREATE TABLE or ALTER TABLE request. Teradata Database then invokes the specified algorithm to compress or decompress data values when data is inserted into or retrieved from the table.

Also see [Compression](#) and [SQL External Routine Programming](#).

**Aligned Row Format** A system in which data fields are aligned on 8-byte boundaries. If a field does not align naturally on an 8-byte boundary, Teradata Database expanded with pad bytes to achieve proper alignment.

Pad bytes consume extra disk space and result in extra I/O operations, which impacts performance, so the [Packed64 Row Format](#) is generally preferred to Aligned Row Format.

**Alternate Key** Any [Candidate Key](#) that is not selected to be the [Primary Key](#) in a referential integrity relationship.

**AMP** Access Module Processor vproc.

The set of software services that controls the file system and data management components of Teradata Database.

**AMP Cluster** A hardware mechanism that groups small numbers of [AMPs](#) together in such a way that the primary rows of each group are duplicated in a [Fallback Table](#) on a different [AMP](#) in the same cluster. See *Introduction to Teradata* for further information.

**ANSI** American National Standards Institute.

**Arity** A property of relations. The arity of a relation is the count of its attributes. [Degree](#) is a synonym for arity or number of columns.

**ASF** Archive Storage Facility.

**ATM** Activity Transaction Modeling.

**Attribute** As the term is used in database design, attributes are the set of constructs that constitute the heading of a relation. Each attribute is made up of a *column\_name:domain\_name* pair, where *domain\_name* signifies the data type for the column. When attributes are transferred from the logical realm into the physical realm, they are usually referred to as *columns*.

**Autocompression** A method of compressing container data automatically and dynamically by Teradata Database as rows are inserted into a column-partitioned NoPI table or join index. Teradata Database determines the compression techniques to apply on a per container or

subrow basis. No applicable compression technique exists for some values, and when this occurs, Teradata Database determine not to compress the values for that container or subrow. Teradata Database also decompresses any autocompressed values when you retrieve them.

## B

**BCNF** See [Boyce-Codd Normal Form](#).

**Bidirectional Inheritance** The property of base tables and their underlying indexes being able to inherit and use existing statistics from one another when either database object in a pair has no existing interval histogram statistics.

If *both* database objects have existing interval histogram statistics, the Optimizer uses the set with the more recent collection timestamp. See *SQL Request and Transaction Processing* for details.

**Bijection** In function theory, a bijection is defined as a one-to-one *onto* relationship.

More formally, a bijective function is a function  $f$  from a set  $X$  to a set  $Y$  with the property that for every  $y$  in  $Y$ , there is exactly one  $x$  in  $X$  such that  $f(x)=y$ . A bijective function is said to be a one-to-one correspondence, which is different from a one-to-one *function*, which is an [Injection](#).

Compare with [Injection](#), which is a one-to-one *into* relationship, and [Surjection](#).

**BLC** See [Block Level Compression](#).

**BLOB** An initialism for Binary Large OBject. A BLOB is a data object, usually larger than 64 KB, that contains only binary data such as pictures, movies, or music. Teradata Database stores BLOB data in subtables rather than in the row.

**Block Level Compression** A feature that allows any normal data block to be stored in compressed form. Not all the data in a data block is compressed: the file system block header and trailer information are left uncompressed. This allows operations such as migration to move blocks without having to undergo a decompress:recompress cycle. It also enables easier recovery if data corruption occurs because it is then unnecessary to decompress the data blocks to access the basic information in the block header.

There are several ways that you can control compression of data at the block level:

- Using an appropriate query band when you load data into the table.
- Using the Ferret utility.
- Specifying several DBS Control flags.

**Body** The body of a relation is the composite value set assigned to its tuple variables. Each SQL relation must have a body.

**Boyce-Codd Normal Form** A [Relation](#) is said to be in Boyce-Codd Normal Form if and only if every [Determinant](#) in the [Relation](#) is a [Candidate Key](#).

**Broad Join Index** A [Covering](#) join index whose definition includes one or more tables that is not specified in the query it covers. To qualify for this kind of coverage, the join to an extra

table only adds more columns to the final results and does not change the rows in the join index itself.

A wide range of queries can make use of a broad join index, especially when there are foreign key-primary key relationships defined between the fact table and the dimension tables that enable the index to be used to cover queries over a subset of dimension tables.

**BTEQ** Basic TEradata Query facility.

**BYNET** BanYan NETwork (high-speed proprietary interconnect).

## C

**Candidate Key** In logical database design, any column set identified as a possible Primary Key is called a candidate key. Once a [Primary Key](#) has been selected for a table, any remaining candidate keys are referred to as [Alternate Keys](#).

**Cardinality** A cardinal number expresses quantity. Therefore, the word cardinality refers to the quantity of some thing.

In relational database management, cardinality typically refers to the number of rows in a table, whether estimated or actual, but it can also express other quantities, such as the number of partitions defined for a partitioned table or join index.

**CASE** Computer Aided Software Engineering.

**Character String Literal** See [String Literal](#).

**Checksum** A value computed from a block of data for the purpose of detecting errors that might have been introduced during its storage. The integrity of the data can be checked at any later time by recomputing the checksum and comparing it with the stored checksum value. If the checksum values do not match, the data has probably been altered.

In the specific case of the Teradata file system, disk I/O integrity checking is used to improve the detection of corruption in user data by adding an end-to-end checksum to the structures in the file system B\*-tree.

By computing and storing a full checksum of file system data in units of these structures when they are written to disk, disk I/O integrity checking verifies data integrity when a read completes for one of the file system structures. A full checksum means that all words in a file system structure are XORed together to form the checksum.

**Child Table** In a referential integrity relationship, the parent table in the relationship is the one that is the [Referenced Table](#), and it is referenced by means of a [Foreign Key](#). In other words, the [Parent Table](#) in the relationship is the one that contains the [Primary Key](#) or other [Candidate Key](#), and the child, or [Referencing Table](#) in the relationship is the one that contains the [Foreign Key](#).

**CI** Cylinder Index.

**CID** Cylinder Index Descriptor.

**CK** See [Candidate Key](#).

**CLI** Call-Level Interface.

**Clique** An optional feature of MPP systems that physically groups nodes together by multiported access to common disk array units. Cliques enable the migration of [vprocs](#) to another node when their node in a clique fails. This makes it possible for vprocs from a failed node to continue to operate while recovery occurs on their home node.

**CLOB** An initialism for Character Large OBject. A CLOB is a data object, usually larger than 64 KB, that contains only character data such as XML or other text files. Teradata Database stores CLOB data in subtables rather than in the row.

**COLUMN Format** The format of the physical row used as a [Container](#) in a column partition. The format includes an indication of the autocompression used, if any, an optional multi-value compression dictionary, a series of column partition values, and optional sets of autocompression bits: one set for each column partition value in the [Container](#). This format, along with column partitioning, provides a column-store in Teradata Database. See *Database Design* for more detailed information.

Compare with [ROW Format](#).

**Column-Level Constraint** A constraint that is defined to apply only to elements of a single column is called a column-level [Constraint](#).

Compare with [Table-Level Constraint](#).

**Column Partition** A set of one or multiple table columns grouped into a set of [Column Partition Values](#) or in a [Subrow](#) and stored in a [Physical Row](#) that is referred to as a [Container](#), which represents the values and nulls of a column partition.

See [ROW Format](#) and [COLUMN Format](#) for definitions of the various ways that a [Physical Row](#) can be specified for a column partition.

**Column Partition Context** A column partition context is a data structure that is allocated in memory to keep information about a column partition when it is being processed. There is a column partition context for each of the column partitions being accessed at the same time. Because the memory that is required for each column partition context can be fairly large, the number of column partitions, and, therefore the amount of memory used for column partition contexts, is limited.

Teradata Database uses the DBS Control parameter PPICacheThrP in the Performance group to specify the percentage value to use for calculating the amount of [FSG Cache](#) memory that can be used for keeping a set of data blocks in FSG cache for operations accessing multiple combined partitions at the same time, including both row and column partitions. The number of data blocks that can be kept in this amount of memory defines how many file contexts can be opened at the same time to access combined partitions. The permissible range for the number of data blocks that can be kept in memory is at least 8 and no more than 256 unless an internal override is set to a lower or higher value.

PPICacheThrP also specifies the percentage value to use for calculating the number of memory segments that can be allocated to buffer the appending of column partition values to column partitions. The sum of the sizes of these memory segments minus some overhead and

divided by the size of a column partition context determines the number of available column partition contexts.

If there are more column partitions in a target column-partitioned table than there are available column partition contexts, multiple passes over the source rows are required to process a set column partitions where the number of column partitions in each set is up to the number of available column partition contexts. In this case, there is only one file context open, but each column partition context allocates buffers in memory.

**Column Partition Value** The set of values of the columns in a [Column Partition](#) projected from a table row. A column partition value consists of values of one or more columns for a table row that are grouped in a column partition.

If the column partition is either user-specified or system-determined to have [COLUMN Format](#), a column partition value is just a column value if the column partition contains only a single column, or multiple column values if the column partition has more than one column. One or more column partition values can be represented in a [Container](#).

If the column partition is either user-specified or system-determined to have [ROW Format](#), it is stored using [Subrows](#) and the column partition value is the set of column values in the subrow. There is one column partition value represented per subrow.

**Column-Partitioned Join Index** A column-partitioned [Join Index](#) does not have a [Primary Index](#). A subset of the join index columns can be grouped into a column partition by delimiting their definitions in the select list of their CREATE JOIN INDEX request with parentheses; otherwise, each individual column is placed in a [Column Partition](#) by itself.

You can also specify column grouping using the COLUMN option in the PARTITION BY clause. You can optionally specify [COLUMN Format](#) or [ROW Format](#) for the [Physical Rows](#) for a column partition.

If you specify COLUMN format, Teradata Database places a series of column partition values in a physical row of a column partition, referred to as a [Container](#).

If you specify ROW format, Teradata Database places only a single column partition value in a physical row, referred to as a [Subrow](#).

If you specify neither COLUMN nor ROW, Teradata Database determines whether COLUMN or ROW format is used for a column partition based on the size of a column partition value and other factors. A column partition consists of either a series of containers or subrows.

**Column-Partitioned Object** A table or join index that has no primary index but is partitioned into rows with either [COLUMN Format](#) or a mix of COLUMN format and [ROW Format](#).

**Column-Partitioned Table** A column-partitioned [Table](#) does not have a [Primary Index](#) and must have a table type of [Multiset](#). A subset of the table columns can be grouped into a column partition by placing their definitions in parentheses; otherwise, a column for a column definition is placed in a [Column Partition](#) by itself.

You can also specify column grouping using the COLUMN option in the PARTITION BY clause. You can optionally specify [COLUMN Format](#) or [ROW Format](#) for [Physical Rows](#) for a column partition.

If you specify COLUMN format, Teradata Database places a series of [Column Partition Values](#) in a physical row of a column partition, referred to as a container.

If you specify ROW format, Teradata Database places only a single column partition value in a physical row, referred to as subrow.

If you specify neither COLUMN nor ROW, Teradata Database determines whether COLUMN or ROW format is used for a column partition based on the size of a column partition value and other factors. A column partition consists of either a series of containers or subrows.

**Column Partitioning** A method of vertically partitioning sets of columns of a table.

**Column Partitioning Level** The partitioning level at which a [Column Partition](#) is defined for a column-partitioned table or join index. The recommended placement of the column partition for a column-partitioned object, with rare exceptions, is the first level of the [Partitioning Expression](#).

You cannot specify more than one column partitioning level for the partitioning expression of a column-partitioned table or join index.

**Combined Partitioning Expression** An expression that combines all of the [Partitioning Expressions](#) of a PARTITION BY clause and the column partition number for a partitioned table, which is 1, into a single expression.

The combined partitioning expression for a non-partitioned table is 0.

The combined partitioning expression is such that ordering on the combined partitioning expression value for rows would have the same ordering for rows if they were ordered by the value of the first partitioning expression, the second partitioning expression, and so on.

**Combined Partition Number** The result of the [Combined Partitioning Expression](#) for a specific set of values of the partitioning columns.

If a table is column-partitioned, its combined partition number is the combined partitioned number for the table row using one as the column partition number for a combined partition number of a table row. The combined partition number is adjusted to the partition number of a specific column partition when determining the combined partition number for that column partition. For a non-partitioned table, the combined partition number is 0.

**Composite Secondary Index** A secondary index defined on anywhere from 2 to 64 columns. Compare with [Simple Secondary Index](#).

**Composite Key** A key defined on more than one attribute. Compare with [Simple Key](#).

**Compression** The term *compression* means different things for the built-in compression methods support by Teradata Database. All forms (except some cases of user-defined algorithmic compression) are [Lossless](#), meaning that the original data can be reconstructed exactly from their compressed forms. Teradata Database also supports user-defined compression and decompression methods, which are referred to collectively as [Algorithmic](#)

**Compression.** Teradata Database also uses various lossless compression methods, collectively referred to as [Autocompression](#), for column-partitioned data that is stored in [Containers](#).

For hash and join indexes, compression refers to a logical row compression in which multiple sets of nonrepeating column values are appended to a single set of repeating column values. This allows the system to store the repeating value set only once, while any nonrepeating column values are stored as logical segmental extensions of the base repeating set.

For column values, compression refers to the storage of those values one time only in the table header, not in the row itself, and pointing to them by means of an array of presence bits in the row header (see [Multi-Value Compression](#)).

For data blocks, compression refers to the storage of primary table data, [Join Index](#) subtable data, or [Hash Index](#) subtable data. [Secondary Index \(SI\)](#) subtable data cannot be compressed.

For a definition of user-defined compression methods that can be implemented for Teradata Database, see *SQL External Routine Programming*.

**Consolidation Path** In OLAP terminology, a consolidation path is a set of hierarchically related data. Consolidation itself is the process of aggregating the hierarchical data to form subtotals. The highest level in the consolidation path is the dimension for the data.

**Constant Expression** An SQL expression that does not make any column references.

**Constraint** A conditional expression that must not evaluate to FALSE.

When a constraint is simple, or defined on only one column, it is referred to as a [Column-Level Constraint](#).

When a constraint is composite, or defined on multiple columns or across multiple tables, it is referred to as a [Table-Level Constraint](#).

**Container** A physical row that has COLUMN format.

A container contains a representation of a series of column partition values for a column partition that has COLUMN format. A container can be a single-column container or it can be a multicolumn container. A series of containers, with increasing rowid values, represent a table column or set of table columns. See also [Subrow](#).

**Correlation Name** An aliased name assigned to a table name that is specified in an SQL request. Correlation names are also sometimes called range variables.

Correlation names are associated with a table or view only within the context in which they are defined. Once you specify a correlation name for a table or view name, you must specify that correlation name in any qualified reference to a column of that instance of the table or view instead of specifying the actual table or view name.

**Cover** A condition in which all the column data requested by a query can be obtained by index-only access.

**CWA** Closed World Assumption.

**Cylinder** A disk cylinder is a division of data in a disk drive. Logically, a cylinder is a concentric, hollow, cylindrical slice through the physical disks, collecting the respective circular tracks aligned through the stack of disk platters.

The cylinder is the allocation unit for disk space on Teradata systems. The number of cylinders a disk drive has is equal to the number of tracks on a single surface in the drive. Each cylinder contains one or more [Data Blocks](#). Each cylinder has 3,872 sectors.

A cylinder can contain data blocks for multiple tables. Cylinders are sometimes referred to as disk extents.

In the Teradata file system, a structure called the master index is a memory-resident file system data structure that contains pointers to every cylinder index on a given AMP in rows referred to as Cylinder Index Descriptors, or CIDs. When a cylinder does not contain data, it is not listed in the master index.

The Teradata Database file system cylinder index is a disk-resident file system file structure that contains rows referred to as Data Block Descriptors, or DBDs. Each disk cylinder has its own cylinder index. The structure contains pointers to all the data blocks and free sectors on the cylinder it represents.

The size of a cylinder, as the term is used in the documentation, need not match the exact physical size of a cylinder on a particular disk drive on a particular system.

## D

**Data Block** The data block is the physical I/O unit for the Teradata file system. Block sizes range between 512 and 130,560 bytes (1 to 255 sectors). The data block is a disk-resident file system file structure that contains one or more rows from the same table. Row storage is atomic: either the entire row fits within a data block or none of it does (in which case it is stored in a different block). In other words, rows cannot be split between blocks. Neither can rows from different tables be mixed within the same data block.

**Data Model** Date (2004, pp. 15-16) provides a very precise definition of a data model: “A **data model** is an abstract, self-contained, logical definition of the objects, operators, and so forth, that together constitute the *abstract machine* with which users interact. The objects allow us to model the *structure* of data. The operators allow us to model its *behavior* ... In a nutshell: The model is what users have to know about; the implementation is what users do not have to know about ... A data model in ...[this]... sense is like a *programming language*—albeit one that is somewhat abstract—whose constructs can be used to solve a wide variety of specific problems, but in and of themselves have no direct connection with any such specific problem” (emphasis in original).

**DB** Data Block.

**DBA** DataBase Administrator.

**DBD** Data Block Descriptor.

**DCL** Data Control Language. A subset of the SQL language that consists of all the SQL statements that support the definition of security authorization for accessing database objects. This includes statements such as GIVE, GRANT, and REVOKE.

**DDL** Data Definition Language. A subset of the SQL language that consists of all the SQL statements that support the definition and destruction of database objects, user-defined data types, and other objects that require dictionary definitions. This includes statements such as CREATE TABLE, DROP TABLE, CREATE INDEX, DROP INDEX, and so on.

**Defined Partition (Column Partitioning)** A user-specified number of [Column Partitions](#) for a column partitioning level.

The number of defined partitions for a column partitioning level is the number of user-specified column partitions plus 2 for column partitions reserved for internal use.

**Defined Partition (Row Partitioning)** A user-specified number of row partitions for a column partitioning level.

The number of defined partitions for a row partitioning level is one of three possible things.

- The number of row partitions specified by a RANGE\_N function.
- The number of row partitions specified by a CASE\_N function.
- 65,535 row partitions if neither a RANGE\_N nor a CASE\_N function is used to define a partitioning level.

**Degree** Degree is a property of relations. The degree of a relation is the count of its attributes. [Arity](#) and number of columns are synonyms for degree.

**Delayed Row Partition Elimination** Row partition elimination can occur with conditions comparing a partitioning column to a USING variable or built-in function. This cannot be done when building a plan that is cached, because a cached plan needs to be general enough to handle changes in these values in subsequent executions.

In certain cases, row partition elimination can be delayed until the finalized plan is built from a cached plan using the values for this specific execution of the plan, and this deferred row partition elimination is called delayed row partition elimination.

Delayed row partition elimination is similar to [Static Partition Elimination](#) in that it is performed by the Optimizer, but it is done at a later stage of the optimization process.

Delayed row partition elimination is unlike [Dynamic Row Partition Elimination](#) because it is done as part of the optimization process undertaken by the Optimizer, not done post optimization by the AMP-based database software.

Also see [Partition Elimination](#).

**Depot** An area of disk used together with [WAL](#) to ensure that in-place modifications cannot destroy existing data because of a failure that interrupts a write operation. The Depot area includes cylinders reserved by the File System as a staging area to temporarily contain modified-in-place data blocks before they are written to their home disk destinations.

**Derived Statistics** Statistics that are transformed from various constraint sources, including query predicates, and then adjusted dynamically at each stage of the query optimization

process. Derived Statistics are propagated from optimization stage to optimization stage by means of a data structure that contains both the relevant static interval histogram statistics and the dynamically adjusted derived statistics. See *SQL Request and Transaction Processing* for details.

**Derived Table** A derived table is a transitory table that is created dynamically from one or more other tables by evaluating a query expression or table expression. The semantics of derived tables are identical to the semantics of views.

Derived tables must always be defined with a correlation name.

**Determinant** For any **Relation R**, the set of **Attributes X** in the **Functional Dependency**  $X \rightarrow Y$  is the determinant group for the dependency.

**DevX** Teradata Developer Exchange. A community-oriented technical web site that connects Teradata engineers with developers and customers who are building and deploying solutions on Teradata platforms. The site features downloads, articles, blogs, forums, and best practices for a wide variety of topics.

The URL for DevX is <http://developer.teradata.com>.

**DIP** Database Initialization Program.

**Dispatcher** Software that manages the AMP steps it receives from the Generator.

Each AMP step is dispatched to the appropriate AMP set over the BYNET. The next step in sequence is not dispatched until its predecessor has completed its task.

**DK/NF** Domain/Key Normal Form.

**DM** Dimensional Model or Dimensional Modeling.

**DML** Data Manipulation Language. A subset of the SQL language that consists of all the statements that support read and update access to existing database objects. This includes statements such as DELETE, INSERT, MERGE, SELECT, and UPDATE.

**Domain** The set of all possible values that can be specified for a given **Attribute**.

The physical representation of a domain is a data type, and the ideal representation of a domain is a distinct user-defined data type (see *SQL Data Definition Language* and *SQL External Routine Programming* for more information about UDTs).

**DSS** Decision Support System. Decision support is a deprecated term for what is now called data warehousing.

**Dynamic Row Partition Elimination** A form of dynamic re-optimization performed by the AMP software when query conditions reference values in other tables that permit row partition elimination, but which cannot be determined at the time a query is initially optimized.

See [Partition Elimination](#), [Delayed Row Partition Elimination](#), and [Static Partition Elimination](#).

## E

**ED** Existence Dependency.

**ELDM** Extended Logical Data Model.

**Eliminated Partition** A partition that is skipped for a particular query because the Optimizer has determined that it contains no qualifying rows for row partitioning or no columns are referenced in the partition for column partitioning.

See [Partition Elimination](#).

**Entity** According to Peter Pin-Shan Chen, “an entity is a “thing” which can be distinctly identified. A specific person, company, or event is an example of an entity.”

**Entity Subtype** A synonym for [Minor Entity](#).

**Entity Supertype** A synonym for [Major Entity](#).

**E-R or ER** Entity-Relationship.

A somewhat vague data model that is based on the concept of an [Entity](#) and the

**ERQ** The CLIV2 End Request function call. See *Teradata Call-Level Interface Version 2 Reference for Mainframe-Attached Systems* or *Teradata Call-Level Interface Version 2 Reference for Workstation-Attached Systems* for details.

**ETL** Extraction, Transforming, and Loading.

**External Partition Number** The value computed by the partitioning expression for a row in a table with row partitioning. Compare with [Internal Partition Number](#) and [Combined Partition Number](#).

## F

**Fallback** A feature that enables Teradata Database to create and maintain a secondary copy of the rows of a base table. Fallback is used to prevent a disk failure from rendering a data warehouse unusable. Fallback is best implemented using [AMP Clusters](#). See *Introduction to Teradata* for further information.

**Fallback Table** A duplicate of a [Primary Table](#) used to prevent a disk failure from rendering a data warehouse unusable.

**FD** Functional Dependency.

**FIB** File Information Block.

**Field** The intersection of a tuple and an attribute. Columns in a table are often referred to as fields, but strictly speaking, that is incorrect.

When speaking of a field in the row header, the term refers to a bit array that determines some property of that row such as its internal partition number or whether particular values are multi-value compressed.

**FIFO** First In First Out.

**FK** See [Foreign Key](#).

**Foreign Key** A means of establishing referential integrity between tables in a relational database. A foreign key in a [Child Table](#) is typically the logical primary key of its [Parent Table](#). If it is not the primary key for the parent table, then it is one of its [Alternate Keys](#).

A foreign key is a column set in the [Child Table](#) in a referential integrity relationship that references the [Primary Key](#) or an [Alternate Key](#) in the [Parent Table](#), or [Referenced Table](#), of the relationship.

Compare with [Primary Key](#).

See *Database Design* for details.

**FSG Cache** A system cache in the Teradata [Open PDE](#) system that buffers disk I/O operations.

FSG is an abbreviation for File SeGment, the Open PDE subsystem that manages memory segments.

**FSP** Free Space Percent.

**FTS** Full Table Scan.

**Full Functional Dependency** An attribute set X is said to have full functional dependence on another attribute set Y if X is functionally dependent on the whole set of Y, but not on any subset of Y.

**Functional Dependency** For any relation R, attribute X is said to have a functional dependency on attribute Y if for every valid instance, the value of Y determines the value of X.

## G

**GDO** Global Database Object.

**Generator** Software that takes the access plan produced by the Optimizer and converts it into a set of discrete tasks for the Database Manager software to perform. These tasks are referred to as AMP steps.

The hashing algorithm is also a component of the Generator.

See *SQL Request and Transaction Processing* for details.

**Geospatial Index** A [NUSI](#) defined on a single column that has a Geospatial data type. Geospatial indexes are implemented using a [Hilbert R-Tree](#) rather than the Teradata file system. The R-tree is implemented on top of the Teradata file system and uses row-level API calls to interact with the file system.

**Global Index** A [Join Index](#) defined with the ROWID keyword to reference the corresponding base table rows.

**Global Temporary Table** A table defined in the dictionary that produces a materialized instance of itself for each session that references it. The rows of a Global Temporary Table are not persistent across sessions.

## H

**Hash Index** A special form of single-table join index that provides access to its base table rows by default.

Although hash indexes are essentially a type of data table, they cannot be accessed directly using SQL. Their use is reserved for the Optimizer and query planning.

**Heading** Each [Attribute](#) of a [Relation](#) must have a heading. Each such [Attribute](#) has two required parts: a name and a [Domain](#), or data type. It is common practice not to call out the [Domain](#) of an [Attribute](#) unless it is germane to the problem at hand, but that does not render the typing of each [Attribute](#) in a relation any less necessary.

**HI** See [Hash Index](#).

**Hilbert R-Tree** An extension to the Teradata file system that is used to handle simple [NUSIs](#) defined on a Geospatial column. Like other NUSIs, geospatial indexes are AMP-local.

## I

**IDNF** Inclusion Dependency Normal Form.

**IEEE** Institute of Electrical and Electronics Engineers.

**Index** A physical mechanism used to store and access data in a table. Indexes are defined, maintained, and accessed on the basis of a set of one or more table columns.

Each individual column of an index column set is referred to as an index column. The index column set is collectively called either the key of the index or its index key.

[Secondary Index \(SI\)](#)es are stored in index [Subtables](#), while [Hash Indexes](#) and [Join Indexes](#) are stored in primary tables that are essentially identical to ordinary base tables except that users cannot access them using SQL requests.

A secondary index can either be simple or composite. A simple index is defined on a single column, while a composite index is defined on anywhere from 2 to 64 columns.

**Note:** Multitable join indexes are also called simple join indexes, but they are always defined on multiple columns.

**Index Key** The set of columns on which an [Index](#) is based.

**Index Value** The value of the [Index Key](#). For a specific data row, its index value is the set of column values and nulls of the column set that defines the [Index Key](#). If every index column value is not null, the index value is said to be wholly non-null. Also see [Index](#).

**Injection** In function theory, an injection is defined as a one-to-one *into* relationship.

More formally, an injective function is a function  $f$  from a set  $X$  to a set  $Y$  with the property that for any  $y$  in  $Y$  there is exactly one  $x$  in  $X$ . An injective function is said to be a one-to-one function, which is different from a one-to-one *correspondence*, which is a [Bijection](#).

Compare with [Bijection](#) and [Surjection](#).

**Instance** A [Tuple](#) drawn from the complete set of [Tuples](#) for a relation. The term is sometimes used to describe any selected set of [Tuples](#) from a [Relation](#).

**Intelligent Key** An overloaded [Simple Key](#) that encodes more than one fact. The principal implementation problem with intelligent keys is that if any of the components of the key change, then all applications that access the key are affected. This is why you should always select unchanging [Attributes](#) for your [Keys](#).

The classic example of an intelligent key is the International Standard Book Number (ISBN) used by publishers to identify individual books. Each ISBN is composed of a group identifier, a publisher identifier, a title identifier, and a check digit.

**Internal Partition Number** A value calculated from the [Combined Partition Number](#) of the [Combined Partitioning Expression](#) that is used to number partitions internally. Teradata Database places the internal partition number for each row of a PPI table or [Column-Partitioned Table](#) or join index in the [Row ID](#) of each [Physical Row](#).

This number might be the same as the [Combined Partition Number](#) if no modification is needed. Modification of the internal partition number for a row is required for a single-level partitioning expression that consists solely of a CASE\_N or RANGE\_N function. The partitions for the NO RANGE [OR UNKNOWN], NO CASE [OR UNKNOWN], and UNKNOWN options are placed at fixed internal partitions (internal partition numbers 2, 2, and 1, respectively), with the partitions for ranges and conditions following, beginning at internal partition number 3 initially.

Modification of an internal partition number might also occur for multilevel partitioning. For both single-level and multilevel partitioning, additional modification is required to retain existing internal partition numbers after an ALTER TABLE request that drops or adds ranges or partitions.

An internal partition number in a Row ID referring to a table row indicates a column partition number value of 1 by convention. This can be modified to indicate a specific column partition to access that column partition.

**I/O** Input/Output.

**ISO** International Organization for Standardization.

## J - K

**JD** Join Dependency.

**Join Dependency** A condition in which a relation variable (or, more commonly, a [Relation](#)) is equal in value to the join of two or more projections of its current value. Join dependency is key to the nonloss decomposition of relations that occurs during normalization of a database

schema. Note that every [Multivalued Dependency](#) is a join dependency, but some join dependencies are not multivalued dependencies.

**Join Index** An index that represents either a join result or a single table projection with (possibly) different distribution of rows *to* the AMPs and ordering *on* the AMPs. A join index is a vertical partition of a base table that can, depending on how it is defined, create various types of prejoins of tables, including sparse and aggregate forms. Join indexes cannot be queried directly by an SQL request; instead, they are used by the [Query Rewrite](#) subsystem and the [Optimizer](#) to enhance the performance of any queries they [Cover](#).

Teradata Database maintains join indexes in a subtable that is distinct from its underlying base table set. The system maintains join indexes automatically, so maintenance to a base table is automatically reflected in any join indexes defined on that base table.

A join index that only vertically partitions a base table is referred to as a single-table join index.

A join index that prejoins two or more base tables is referred to as a multitable join index.

Both types of join index can be created in sparse or aggregate forms and can have a subset of their columns compressed.

A join index can have a non-partitioned primary index, a partitioned primary index, or no primary index. A join index that is a [NoPI Object](#) must be column-partitioned, defined on a single-table, a nonaggregate index, and it must not be row-compressed.

Unlike the primary index, which is stored in-line with the row it indexes, join indexes are stored in separate subtables that must be maintained by the system. Join index subtables also consume disk space, so you should monitor your queries periodically using EXPLAIN modifiers to determine whether the Optimizer is using any of the join indexes you designed for them. If not, you should either drop those indexes or rewrite your queries in such a way that the Optimizer does use them.

Although join indexes are essentially a type of data table, they cannot be accessed directly using SQL DML requests. Their use is reserved for the [Query Rewrite](#) subsystem and the [Optimizer](#).

**Key** An [Attribute](#) set that uniquely identifies each [Tuple](#) in a [Relation](#). Implicitly synonymous with [Primary Key](#), though it applies equally well to any [Candidate Key](#) or [Foreign Key](#). See [Primary Key](#).

This term is sometimes used to describe other things. For example, the set of columns that defines an index is referred to as an index key, the set of values that Teradata Database sorts a set of rows on is referred to as a sort key, and so on.

## L

**LDM** Logical Data Model.

**Lexer** Component of the Parser that checks SQL statements for proper use of the SQL lexicon.

If lexical use is incorrect, an appropriate error message is returned to the requestor.

See *SQL Request and Transaction Processing* for details.

**LHS** The Left Hand Side of an equation or predicate condition.

**LOB** An abbreviation for Large OBject.

Teradata Database supports two types of LOB.

- [BLOBs](#)
- [CLOBs](#)

**Logical Row** For compressed [Hash Indexes](#) and [Join Indexes](#), a logical row is defined as one of multiple sets of nonrepeating column values appended to a single set of repeating column values. This allows the system to store the repeating value set only once, while any nonrepeating column values are stored as logical segmental extensions of the base repeating set.

**Lossless** In the context of [Compression](#), a compression method is lossless if the original data can be reconstructed exactly from its compressed form. Text data is stored using lossless methods. ZIP is a lossless method. Compare with [Lossy](#) methods.

**Lossy** In the context of [Compression](#), a compression method is lossy if the original data cannot be reconstructed exactly from its compressed form. Lossy methods discard data bits, so data stored using a lossy compression algorithm is different from the original data, but is similar enough to retain its usefulness. JPEG is a common method of lossy compression used to store video images, and MPEG is a common method of lossy compression used to store audio data. Compare with [Lossless](#) methods.

## M

**Major Entity** A synonym for [Entity Supertype](#).

Suppose you define an *Employee* entity in your logical E-R model. All employees are members of the major entity *Employee*. Those employees who are also eligible to receive commissions have all the attributes of the major entity *Employee* plus their own distinct attribute, Commission.

Given this definition, the entity *Commissioned\_Employee* would be a minor entity, or [Entity Subtype](#), of the *Employee* entity, and the *Employee* entity would be a major entity, or [Entity Supertype](#), of *Commissioned\_Employee*.

**Merge Block Ratio** A value that defines the percentage threshold for merging data blocks. A merged block cannot exceed this percentage size of the maximum multirow data block size for the associated table.

**MI** Master Index.

**Minor Entity** A synonym for [Entity Subtype](#). See the definition of [Major Entity](#) for an example of a major entity-minor entity pair.

Minor entities have the common attributes of their major entity plus their own distinct attributes.

### Multicolumn Container

A [Container](#) with a subset of two or more columns of a table row. This kind of [Container](#) contains a series of [Column Partition Value](#). Each [Column Partition Value](#) has a format similar to the [ROW Format](#) used for a regular row or [Subrow](#) but without a row header. A [Column Partition Value](#) in this case does not correspond to a [Physical Row](#) but is a structure that can repeat within a [Container](#).

**Multicolumn Partition** A [Column Partition](#) with two or more columns of a table row.

**Multicolumn Partition Value** The values and nulls of columns in a multicolumn partition projected from a table row.

If the multicolumn partition is specified or system-determined to have a [COLUMN Format](#), one or more multicolumn-partition values and nulls can be represented in a [Container](#).

If the multicolumn partition is specified or system-determined to have a [ROW Format](#), the column partition is represented using [Subrows](#) and the multicolumn-partition value is the set of column values and nulls in the [Subrow](#). There is one multicolumn-partition value represented per [Subrow](#).

Compare with [Single-Column Partition Value](#).

**Multilevel Partitioning** A partitioning scheme where partitions at any one level are subpartitioned. Multiple partitioning level specifications, specified as either a partitioning expression or a [COLUMN](#), are used to define the partitioning. No more than one level can be specified as [COLUMN](#).

**Multiset** An extension to set theory, the notion of multiset generalizes the concept of set in which elements can appear more than once. In traditional set theory, an element can only appear once in a set.

In terms of database management, the concept of multiset enables duplicate rows in a table or index. A multiset is sometimes referred to as a bag.

**Multitable Join Index** A join index that is defined on more than one table. Compare with [Single-Table Join Index](#).

**Multi-Value Compression** A form of value compression in which Teradata Database stores the compressed values for a column one time only in the table header, not in the row itself, and points to them by means of an array of presence bits in the row header.

**Multivalued Dependency** For any relation  $R$  with attribute set  $(X,Y,Z)$ , the set  $X$  multivalue determines the set  $Y$  if, for every pair of tuples containing duplicates in  $X$ , the instance also contains the pair of tuples obtained by interchanging the  $Y$  tuples in the original pair.

**MVD** MultiValued Dependency.

## N

**Natural Key** The representation of a real world [Tuple](#) identifier in a relational database. For example, a common identifier of employees in a corporation is a unique employee number.

An employee is assigned an employee number whether that information is stored within the database or not.

Natural keys are sometimes confused with [Intelligent Keys](#), but they are very different concepts.

**NC** No Changes.

**ND** No Duplicates.

**Nested Query** A SELECT request that is embedded within another SELECT request.

Depending on which clause of a SELECT request contains the nested query, it has a different name.

IF the embedded query is specified in this clause ...	THEN it is referred to as a ...
select list	<a href="#">Scalar Subquery</a> .
FROM	<a href="#">Derived Table</a> .
<ul style="list-style-type: none"> <li>• WHERE</li> <li>• HAVING</li> </ul>	<a href="#">Subquery</a> .

**NF<sup>2</sup>** Non First Normal Form. This term is deprecated because all tables in a relational schema are, by definition, minimally in First Normal Form.

**NFNF** Non First Normal Form.

**NN** No Nulls.

**Nonloss Decomposition** A state in which replacing a relation variable (or, more commonly, a [Relation](#))  $R$  by its projections  $R_1, R_2, \dots, R_n$ , is such that the join of  $R_1, R_2, \dots, R_n$  is guaranteed to be equal to  $r$ .

**Nontemporal Table** Any table that is not a temporal table is, by exclusion, a nontemporal table. See [ANSI Temporal Table Support](#) and [Temporal Table Support](#) for details.

**NoPI Object** A table or [Join Index](#) that has no [Primary Index](#).

A table or join index that has no primary index. All access to NoPI rows is by means of a full-table scan unless you define [Secondary Index \(SI\)](#)es on the table that are used in conditions for queries against it, or in the case of NoPI tables, [Join Indexes](#) that cover queries made against it.

A column-partitioned table or join index is a NoPI database object.

**Null** A construct used to represent missing information. Nulls are not values, so the term *null value* is incorrect usage. The truth value of nulls is usually, but not always, UNKNOWN.

**Null Partition Handler** The following optional clauses are referred to as null partition handlers for the ALTER TABLE and ALTER TABLE TO CURRENT statements because they

instruct Teradata Database how to handle the null partitions that might arise after the values of a table or join index partition have been altered.

- WITH DELETE
- WITH INSERT [INTO] *save\_table*

**NUPI** Non-Unique Primary Index. A primary index with no uniqueness constraint on the index values of the rows of its table, join index, or hash index. That is, any number of rows can have the same index value. The primary index of join and hash indexes are always NUPIs.

**NUPI Table** A table defined with a non-unique primary index, or [NUPI](#).

**NUPPI** Non-Unique Partitioned Primary Index. A NUPI that is also partitioned by means of a partitioning expression.

**NUSI** Non-Unique Secondary Index. A secondary index with no uniqueness constraint on the index values. That is, any number of rows in the base table or join index can have the same index value. Each NUSI subtable is constructed locally on each AMP. Its rows are not hash-distributed. This means that NUSI rows are present on an AMP if and only if corresponding data rows exist on that AMP.

A NUSI row consists of one or more RowID identifiers of data rows on this AMP that have the same index value. The system maintains RowIDs for a given index value in a NUSI row in ascending order of RowID. If more data rows exist on an AMP for a given index value than can be maintained in a single NUSI row, then more than one NUSI row exists on that AMP for that index value. Note that RowIDs are *not* kept in order across such NUSI rows. As a result, any number of NUSI rows can exist on any number of AMPs for a given index value.

Therefore, to retrieve all rows with a given NUSI index value, the system must check the NUSI values on all AMPs for their existence and then access of data rows for that index value. The RowID of a NUSI index row includes either a column value or the hash value of set of columns in addition to the uniqueness value.

## O

**OID** Object IDentifier. A 40-byte pointer to a LOB or XML subtable from an in-line row.

**OLAP** OnLine Analytical Processing.

**OLCP** OnLine Complex Processing.

**OLTP** OnLine Transaction Processing.

**Open PDE** Open Parallel Database Extensions.

A Teradata subsystem that acts as a virtual machine to interface the Teradata database management and file systems with the operating system of the machine on which they are running.

**Optimizer** Software that takes error-free output from the Parser and performs all the following tasks in order to optimize the cost of an SQL query.

- Restructures SQL statements to make them more efficient

- Determines an optimal cost method for retrieving data rows
- Determines an optimal cost method for joining data tables
- Creates an access plan for the query
- Creates a join plan for the query when it joins relations
- Compiles the access and join plans into machine code

See *SQL Request and Transaction Processing* for details.

## P - Q

**Packed64 Row Format** A system in which data fields need not be aligned on 8-byte boundaries, but are instead stored without pad bytes or other alignment details.

**Parent Table** In a referential integrity relationship, the parent table in the relationship is the one that is referenced by a [Foreign Key](#). In other words, the parent table in the relationship is the one that contains the [Primary Key](#),

**Parser** Software that checks SQL requests for the following properties.

- Proper SQL syntax
- Proper use of the SQL lexicon
- Access permissions for requested database objects
- Existence of requested database objects

See *SQL Request and Transaction Processing* for details.

**Partition** An AMP-based cluster of PPI table rows that share the same value, or partition number, for their evaluation by the [Partitioning Expression](#) for that table.

Each row is inserted into a partition based on the evaluation of the user-defined partitioning expression for the table. Rows are first hashed to an AMP and then stored within each partition on that AMP in the order of their hash value and uniqueness.

**PARTITION** A system-derived column created dynamically from the [Combined Partition Number](#) stored in the row ID field in the row header for each row in a table. The PARTITION column that is returned in the result set of a DML request contains the combined partition number for the row it represents. Teradata Database does not store the combined partition number for a row as an actual column in the row, but derives the number from the internal partition number that is stored or implied in the row ID field of the row header. For a non-partitioned table, the internal partition number is 0 and is implied by the flag bit settings in the row header.

**Partition Elimination** An automatic optimization in which the Optimizer determines, based on query conditions and a partitioning expression, that some partitions for that partitioning expression cannot contain qualifying rows, and causes those partitions to be skipped. Partitions that are skipped for a particular query are called eliminated partitions (see [Eliminated Partition](#)).

When multiple partitioning expressions are defined on a table or join index, the system can combine partition elimination at each of the levels to further reduce the subsets of data that

need to be scanned. Generally, the greatest benefit of a PPI is obtained from partition elimination.

Also see [Delayed Row Partition Elimination](#), [Dynamic Row Partition Elimination](#), [Static Partition Elimination](#), and [SQL Request and Transaction Processing](#).

**Partition Number** A value computed by the partitioning expression for each row in a PPI table. There are two kinds of partition number: [External Partition Numbers](#) and [Internal Partition Numbers](#).

**Partitioning Column** A column that is a member of the set of partitioning columns. A partitioning column is referenced in one or more of the partitioning expressions specified by a PARTITION BY clause. Note that a partitioning column can also be a primary index column.

**Partitioning Expression** An optional expression based on a column set from a base table or join index that can be specified in the PARTITION BY clause to determine the placement of rows from that table or join index in a particular [Partition](#) on an AMP once it has been hashed or otherwise distributed to that AMP.

**pdisk** A logical partition of physical disk that controls the allocation of storage to the AMPS. The cylinders belonging to a pdisk can be on any disk array within a given clique.

Note that there might not be an exact 1:1 relationship between disk drives and pdisks, depending on the disk vendor.

**PE** Parsing Engine.

**Physical Row** A row as seen by the file system and identified by a row ID. A physical row contains a row header that contains the row ID of the physical row, followed by a sequence of bytes. A physical row can have [ROW Format](#), [COLUMN Format](#), secondary index format, table header format, or compressed join index format, or one several special purpose formats.

**PI** Primary Index. A type of index that determines the distribution of rows across AMPS for a table, hash index, or join index. Primary indexes are based on row values: they are not stored in a separate subtable.

**PODS** Principles Of Database Systems. Derived from the annual ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems.

**Polyinstantiation** Polyinstantiation is a property that allows a relation to contain multiple rows with the same [Primary Key](#) value, where the multiple instances are distinguished by their security levels, where a security level is defined by a row-level security constraint column.

For this property not to violate the relational model, the security level instances would need to be defined as components of a composite primary key.

**PPI** Partitioned Primary Index. A type of primary index for a table or join index that defines the partitioning of data rows across the AMPS based on a partitioning expression that references columns in the table or join index. These columns, called partitioning columns, might or might not be part of the primary index column set depending on how the PPI is designed.

**Presence Bits** One or more arrays of bits in the row header that are used to indicate the status of each column with respect to its nullability and value compressibility. Compression presence bits are added to the row header of each row to specify how [Multi-Value Compression](#) is used for that row. Each row has at least one octet of presence bits and can have more, depending on the degree of the table and the cumulative number of values compressed, and each column can have a maximum of 256 presence bits.

**Primary Data Table** A base relational table that contains the primary rows for a given data table. Ordinary base tables, materialized [Global Temporary Tables](#), [Volatile Tables](#), and [Queue Tables](#), are all examples of primary data tables.

A primary data table is distinguished from any of the following table types because it is the first file structure Teradata Database accesses when it retrieves data from a table or inserts rows into a table, updates existing rows of a table, or deletes existing rows from a table.

- BLOB subtable
- CLOB subtable
- Fallback table
- Hash index subtable
- Join index subtable
- Reference index subtable
- Secondary index subtable

BLOB and CLOB subtables are functional components of primary data tables, but all BLOB and CLOB data is stored outside the primary data table row in Teradata Database.

**Primary Index** A column subset chosen from the columns of a table to be used for four purposes:

- To distribute its individual rows across the AMPs of a Teradata system.
- [NoPI Objects](#) do not have a primary index, so they are distributed to the AMPs using a different method.
- To access single rows or single row hashes more efficiently than with a full-table scan.
- To facilitate efficient join operations.
- To facilitate efficient aggregation operations.

Primary indexes have two orthogonal dimensions:

- Unique ([UPI](#)) vs. non-unique ([NUPI](#))
- Partitioned vs. nonpartitioned

Because the primary index of a table is an inline structure of each row, it is not stored in a separate subtable as other Teradata indexes are.

Teradata does *not* define the term primary index in the same way it is commonly used to describe a clustered index in an indexing system based on B+ trees.

**Primary Key** A column set used to uniquely identify the rows of tables in the logical data model of relational systems. When realized physically, primary keys are used to maintain Referential Integrity ([RI](#)) between related tables. Teradata Database implements primary keys

as [Unique Secondary Indexes](#) for [Nontemporal Tables](#) and as single-table [Join Indexes](#) for most temporal tables (see *ANSI Temporal Table Support* and *Temporal Table Support* for information about temporal tables).

**Primary Table** A base table that is neither a fallback table nor a [Subtable](#). Ordinary base tables, [Global Temporary Tables](#), volatile tables, and queue tables, are all examples of primary tables.

**Query Rewrite** Parser software that rewrites the submitted SQL text using various methods such as: pushing projections into views, converting outer joins to inner joins where possible, replacing views and derived tables with their underlying base tables and hash or join indexes, substituting hash or join indexes for base tables where appropriate, eliminating unnecessary joins, and the like. See *SQL Request and Transaction Processing* for details.

**Queue Table** A [Primary Data Table](#) that is used to maintain queue-oriented data, such as event processing and asynchronous data loading applications, with subsequent complex processing of the buffered data load. The properties of queue tables are similar to those of ordinary base tables, with the additional unique property of behaving like an asynchronous first-in-first-out (FIFO) queue.

## R

**RAID** Redundant Array of Independent Disks.

**RDBMS** Relational Database Management System.

**Referenced Table** See [Parent Table](#).

**Referencing Table** See [Child Table](#).

**Referential Integrity** A method of ensuring that no data is ever orphaned in a relational database. Referential integrity uses the parent-child relationships between a [Primary Key](#) and a [Foreign Key](#) to prevent [Child Table](#) rows from ever being orphaned from deleted [Parent Table](#) rows.

Teradata Database supports three different kinds of relational integrity constraints:

- Referential Integrity constraint

This is the standard RI constraint defined by the ANSI/ISO SQL standard.

- Batch Referential Integrity constraint

This is a special Teradata Database form of RI that is less expensive to enforce in terms of system resources than standard referential integrity because it is enforced as an all-or-nothing operation (the entire transaction must complete successfully) rather than on a row-by-row basis, as standard referential integrity is checked.

- Referential Constraint

This is a special Teradata Database form of RI, sometimes informally referred to as *soft RI*, that specifies constraints the Optimizer can use to optimize queries, but which are not enforced by the system.

Temporal relationship constraints are a special kind of Referential Constraint that can be defined between a valid time or bitemporal parent table and a nontemporal, transaction time, or system time child table. See *ANSI Temporal Table Support* and *Temporal Table Support* for details.

The Optimizer often uses [Referential Integrity Constraints](#) to enhance query performance.

**Referential Integrity Constraint** An explicitly defined constraint between the [Primary Key](#) or other [Candidate Key](#) of a [Parent Table](#) and the [Foreign Key](#) of a [Child Table](#). Referential integrity constraints are used to prevent parent table rows (specifically their primary key values) from being deleted and leaving orphaned child table foreign key values behind, which corrupts the integrity of the database.

**Regular Row** A table [Row](#) stored as a single [Physical Row](#) with [ROW Format](#).

**Relation** The combination of a [Heading](#) and [Tuple](#) body (in which the [Tuples](#) are distinct) defined over a common set of [Attributes](#).

A relation is sometimes said to  $n$ -ary, where  $n$  is the number of [Attributes](#) (also known as the [Arity](#) or [Degree](#) of the [Relation](#)).

Tables depict representations of  $n$ -ary [Relations](#); they are *not* the same thing as  $n$ -ary [Relations](#). Confusion over this distinction often leads people to say that [Relations](#) are flat, or 2-dimensional, when they are, in fact,  $n$ -dimensional. More formally, “if [Relation](#)  $r$  has  $n$  [Attributes](#), then each [Tuple](#) in  $r$  represents a point in a certain  $n$ -dimensional space (and the relation overall represents a set of such points)” Date (2005, p. 47).

When relations are transferred from the logical domain to the physical domain, they are usually called *tables*; however, in query optimization the term is often used to describe *any* table, view, hash index, join index, or spool file.

**Relation Body** See [Body](#).

**Relation Heading** See [Heading](#).

**Relation Variable** A relation variable, or time-varying relation, as opposed to a relation value, or the current value of a relvar.

**Relational Database** A database whose properties are based on the relational model of data developed by E.F. Codd. The relational model is based on the first-order predicate logic of symbolic logic, and many of the terms used with relational database management originate in the fields of symbolic logic and the related field of axiomatic set theory.

**Relational Database Management System** a database management system that implements the relational model.

**Relational Schema** A set of [Relations](#) in a logical relational model. In the physical model, a relational schema is manifested as a database.

**Relationship** According to Peter Pin-Shan Chen, “A relationship is an association among entities. For instance, “father-son” is a relationship between two “person” entities. A relationship between entities can be represented in a database management system by a referential integrity constraint. See [Entity](#).

**Relvar** See [Relation Variable](#).

**Repeating Group** A collection of logically related [Attributes](#) that occur more than once in a [Tuple](#).

**Request** One or more SQL statements submitted to Teradata Database as a single unit of work.

Requests are the semantic elements of Teradata SQL, in contrast to [Statements](#), which are its syntactic elements.

**Resolver** Component of the Parser that checks SQL requests for access permissions to, and existence of, requested database objects.

If the requestor does not have appropriate access permissions or if a requested database object does not exist, then an appropriate error message is returned to the requestor. See *SQL Request and Transaction Processing* for details.

**RI** [Referential Integrity](#).

**RHS** The Right Hand Side of an equation or predicate condition.

**RI** [Referential Integrity](#). Also see [Referential Integrity Constraint](#).

RI can also refer to a Reference Index.

**RM/T** Relational Model Tasmania.

**RM/V1** Relational Model Version 1.

**RM/V2** Relational Model Version 2.

**RM/V3** Relational Model Version 3. Codd died before he could develop RM/V3.

**ROLAP** Relational OnLine Analytical Processing.

**Row** A set of tuple values or nulls for the attributes (columns) of a table. The number of columns is fixed for a table. This is a logical concept.

A table row is represented as a [Regular Row](#) for an object that is not [Column Partitioned](#).

A table row is split across [Column Partitions](#) for a [Column Partitioned](#) object. The value set from the table row for a [Column Partition](#) are represented in a [Container](#) if the [Column Partition](#) has a [COLUMN Format](#) or a [Subrow](#) if the [Column Partition](#) has a [ROW Format](#).

**ROW Format** A format for a physical row, regular row, [Subrow](#), or multicolumn-partition value that consists of row length, rowid, flag byte, presence bits, and a fixed number of column values. This is the traditional Teradata row format. For a multicolumn partition value in a container, this form does not have a rowid or flag byte, the presence byte is omitted if there are no presence bits, and the row length is omitted if the multicolumn-partition value is fixed length. See also [COLUMN Format](#).

**Row ID** The row ID makes each row in a table uniquely identifiable even if it otherwise duplicates one or more other rows in the table.

A row ID includes an internal partition number and is 8-bytes logically, but can be stored as 0 bytes for a non-partitioned table or join index and as 2-bytes for 2-byte partitioning.

For NoPI, the 64-bits are split up as 20 bits of hash bucket and 44 bits of uniqueness instead 32 bits of hash value and 32 bit of uniqueness.

When a row is inserted into a primary-indexed table, the file system stores the 32-bit row hash value of its primary index in place with the column data for the row.

Because row hash values are not necessarily unique, the file system generates a unique 32-bit numeric value for each row called the [Uniqueness Value](#). Teradata Database appends each [Uniqueness Value](#) it generates to its partner row hash value, forming a unique row ID that enables rows having the same rowkey value to be distinguished from one another.

**Row Partition** An AMP-based cluster of table rows that share the same value, or row partition number, for their evaluation by the [Partitioning Expression](#) for that table.

Each row is inserted into a row partition based on the evaluation of the user-defined partitioning expression for the table. Rows are first hashed to an AMP and, if the table or join index has a primary index, they are then stored within each row partition on that AMP in the order of their hash value and uniqueness.

See *Database Design* for details.

**Row Partition Elimination** An automatic optimization in which the Optimizer determines, based on query conditions and a partitioning expression, that some row partitions for that partitioning expression cannot contain qualifying rows, and causes those row partitions to be skipped. Row partitions that are skipped for a particular query are called eliminated row partitions (see [Eliminated Partition](#)). When multiple partitioning expressions are defined on a table or join index, the system can combine row partition elimination at each of the levels to further reduce the subsets of data that need to be scanned.

If a table or join index also column partitioning, column partition elimination can further reduce the subset of data that needs to be scanned. Generally, the greatest benefit of partitioning is obtained from row partition elimination.

Also see the following glossary definitions: [Delayed Row Partition Elimination](#), [Dynamic Row Partition Elimination](#), and [Static Partition Elimination](#).

**Row Partition Number** An identifier computed by the partitioning expression for a partitioned table or join index and used to uniquely identify each row partition for that partitioning level of the partitioned table or join index.

**RVA** Relation-Valued Attribute.

## S

**SA** System-Assigned.

**Scalar Subquery** A subquery that can be specified wherever a value expression can be specified. There are two types of scalar subquery: correlated and noncorrelated.

- A noncorrelated scalar subquery returns a single value to the containing query.

- A correlated scalar subquery returns a single value for each row of the correlated outer table set.

**Secondary Index (SI)** An optional index for a table or join index that is not related to the distribution of the data rows for that database object. There are two types of secondary index: unique ([USI](#)) and non-unique ([NUSI](#)).

An SI is an index structure, called a subtable, that is physically separate from the data rows of its base table or join index subtable. Each index value present among the base object data rows is associated with one or more SI rows. A row of an SI consists of an index value along with one or more identifiers of rows having this index value. A table or join index may have zero or more secondary indexes. For a join index, the secondary index must be non-unique.

A secondary index can be either simple or composite. A simple secondary index is defined on a single base table or join index column, while a composite secondary index is defined on anywhere from 2 to 64 base table or join index columns.

Teradata does *not* define the term secondary index in the same way it is commonly used to describe a non-clustered index in an indexing system based on B+ trees.

**Select List** The list of columns, expressions, or both to be projected from a set of tables, views, or both in a SELECT request or [Subquery](#).

**Selectivity** A measure of the ability of a predicate or index to return a highly discriminating subset of rows from a table. The higher the selectivity, the fewer rows retrieved.

A predicate or index that retrieves many rows is said to have low selectivity. By definition, a predicate or index with low selectivity typically accesses more than one row per data block from the table on which it is defined. Older definitions of low selectivity stated that a predicate or index with low selectivity accesses more than 10% of the rows in the table on which it is defined, but the greatly increased size of newer data block configurations makes this definition obsolete, because access rates of 1.0% or even 0.1% frequently occur for predicates and indexes with low selectivity.

A predicate or index that retrieves few rows is said to be highly selective. A highly selective predicate or index is one that does not access all of the data blocks for the table on which it is defined. Older definitions stated that a highly selective predicate or index typically accesses fewer than 10% of the rows in the table on which it is specified or defined, but this definition is no longer accurate for the same reasons that the old definition of low selectivity is obsolete.

The more highly selective a predicate or index is, the more useful it is for enhancing performance.

**SI** Secondary Index. See [Secondary Index \(SI\)](#).

**SIGACT** ACM Special Interest Group for Algorithms and Computation Theory, the ACM SIG devoted to theoretical computer science (including theoretical issues of importance to database management).

**SIGART** ACM Special Interest Group for ARTificial Intelligence.

**SIGFIDET** ACM Special Interest Group for File DEscription and Translation, the predecessor organization to ACM SIGMOD.

**SIGKDD** ACM Special Interest Group for Knowledge Discovery and Data Mining.

**SIGMOD** ACM Special Interest Group for MODification of Data, the ACM SIG devoted to database management issues.

**Simple Join Index** See [Multitable Join Index](#).

**Simple Key** A key defined on a single attribute. Compare with [Composite Key](#).

**Simple Secondary Index** A secondary index that is defined on only one column. Compare with [Composite Secondary Index](#).

**Single-Column Partition** A [Column Partition](#) with only one column of a table row.

**Single-Column Partition Value** The value of the column in a [Single-Column Partition](#) projected from a table row.

If the [Single-Column Partition](#) is specified explicitly or is system-determined to have a [COLUMN Format](#), one or more single-column-partition values can be represented in a [Container](#).

If the [Single-Column Partition](#) is specified or system-determined to have a [ROW Format](#), the column partition is represented using [Subrows](#) and the single-column-partition value is the column value in the [Subrow](#). There is one single-column-partition value represented per [Subrow](#).

Compare with [Multicolumn Partition Value](#).

**Single-Level Partitioning** A partitioning scheme with one level of partitioning, which is defined by a single partitioning expression.

**Single-Table Join Index** A [Join Index](#) that is defined on a single table. You can think of single-table join indexes as being something like a hashed [NUSI](#).

**Sixth Normal Form** A [Relation Variable](#)  $R$  is said to be in sixth normal form iff it satisfies no nontrivial join dependencies, where nontrivial is defined to mean that at least one relevant [Join Dependency](#) is taken over the set of all the attributes for the relation of interest.

**SK** Surrogate Key.

An artificial key typically defined when no natural [Primary Key](#) exists for a table.

**SPARC** Standards Planning And Requirements Committee.

A committee convened by [ANSI](#) and [SIGMOD](#) as ANXI/X3/SPARC to develop a general architecture for database management systems.

**SQL** The programming language used to create relational database objects (Data Definition Language, or DDL), to manipulate their contents (Data Manipulation Language, or DML), and to define their security attributes (Data Control Language, or DCL).

Now preferably pronounced ess-kew-ell, the language was originally named SEQUEL 2 (Structured English QUEry Language) and was pronounced as it is spelled.

According to the ANSI/ISO SQL standard, SQL does *not* stand for Structured Query Language, as is commonly thought. The name is neither an acronym nor does it represent anything other than the characters *S*, *Q*, and *L*.

**SQO** Semantic Query Optimization.

A method that uses integrity constraints to optimize queries.

**Stale Statistics** Interval histogram statistics that no longer represent an accurate description of the column or index sets on which they were originally collected. See *SQL Request and Transaction Processing*.

**Statement** Statements are the syntactic elements of Teradata SQL, while [Requests](#) are its semantic elements.

**Static Partition Elimination** Partition elimination undertaken by the Optimizer during the primary optimization process, as opposed to being delayed until the time the finalized plan is built from a cached plan using built-in function values or USING request modifier variable values for the specific execution of the plan as is done in [Delayed Row Partition Elimination](#).

Also see [Dynamic Row Partition Elimination](#), [Partition Elimination](#), and *SQL Request and Transaction Processing*.

**String Literal** Zero or more alphanumeric characters enclosed by APOSTROPHE characters.

**Subquery** A SELECT ... FROM ... WHERE expression that is nested within one or more other SELECT ... FROM ... WHERE expressions. Also referred to as a [Nested Query](#).

**Subrow** A physical row formed from a subset of the column values of a table row. The column values are for a set of one or more columns corresponding to the set of columns defined for a column partition that is specified to have [ROW Format](#) or that has [ROW Format](#) by default. A subrow is a [Physical Row](#) with [ROW Format](#). A series of subrows with increasing row ID values represents a table column or set of table columns as a [Column Partition](#).

Compare with [Container](#).

**Subtable** A collection of rows stored in data blocks that belongs to a specific element of a [Primary Data Table](#). Common examples of subtables include hash, join, secondary, and reference index subtables, LOB subtables, XML subtables, and fallback subtables. A given primary data table or join index subtable can have multiple subtables associated with it.

**Superkey** Any set of [Attributes](#) that uniquely identifies a [Tuple](#), whether redundantly or not. The allowance of redundant [Attributes](#) within a super key distinguishes it from a simple [Candidate Key](#).

**Surjection** In function theory, a surjection is defined as a many-to-one *onto* relationship. A function  $f$  is said to be surjective if its values span its whole codomain, meaning that for every  $y$  in the codomain, there is at least one  $x$  in the [Domain](#) such that  $f(x) = y$ .

The codomain of a function  $f: X \rightarrow Y$  is the set  $Y$ .

Compare with [Bijection](#) and [Injection](#).

**Surrogate Key** An artificial [Simple Key](#) used to identify individual entities when there is no [Natural Key](#) or when the situation demands a non-composite key, but no natural non-composite key exists.

Surrogate keys do not identify individual entities in a meaningful way: they are simply an arbitrary method to distinguish among them. You should only resort to surrogate keys if there is no other way to uniquely identify the rows of a table.

Surrogate keys are typically arbitrary system-generated sequential integers. See “[Identity Columns](#)” on page 818 and “[CREATE TABLE](#)” in *SQL Data Definition Language*, for information about how to generate surrogate keys in Teradata.

**Syntaxer** Component of the Parser that checks SQL requests for proper syntax.

If syntax is incorrect, an appropriate error message is returned to the requestor.

See *SQL Request and Transaction Processing* for details.

## T

**Table** The physical analog of a [Relation](#), or more exactly, of a [Relation Variable](#), in a [Relational Database Management System](#).

**Table-Level Constraint** A constraint that is defined to apply to elements of more than one column within a base table or across multiple base tables is called a table-level [Constraint](#).

Compare with [Column-Level Constraint](#).

**TD** Transitive Dependency.

**Temperature** A measure of how frequently data is accessed.

The following set of temperature labels applies to [Block Level Compression](#) as it relates to data temperature.

- Cold  
The least accessed 20% of the data.
- Hot  
The most frequently accessed 20% of the data.
- Warm  
The 60% of the data whose access frequency is neither cold nor hot.

**Temporal Table** A table used to track temporal information. See *ANSI Temporal Table Support* and *Temporal Table Support* for details.

**Third Normal Form** A [Relation Variable](#)  $R$  is said to be in Third Normal Form when it is in Second Normal Form and one of the following statements is also true for every nontrivial [Functional Dependency](#) (assume the functional dependency  $X \rightarrow A$ ):

- The attribute set  $x$  is a [Superkey](#).
- Attribute A is part of some [Candidate Key](#).

**TJ** Transient Journal

**Transitive Dependency** Transitive dependencies are dependencies that exist among three or more attributes in a relation in such a way that one attribute determines a third by way of an intermediate attribute. For example, consider the relation  $R(X,Y,Z)$ , where X is the primary key.

Attribute Z is transitively dependent on attribute X if attribute Y satisfies the following dependencies, where the symbol  $\overline{\rightarrow}$  indicates *does not determine*:

- $X \overline{\rightarrow} Y$
- $Y \overline{\rightarrow} Z$
- $Y \overline{\rightarrow} X$

**Trusted Parallel Application** Frequently abbreviated TPA. An application that Teradata has certified to run safely on Teradata Database. The Teradata Database software itself is a TPA. Such applications are frequently referred to by the initialism TPA.

**Tuple** The set of all [Attribute](#) values associated with an [Instance](#) of a [Relation](#) is a tuple or, more properly, a tuple value. When tuples are transferred from the logical domain to the physical domain, they are usually called *rows*.

## U

**UA** User-Assigned.

**UDF** User-Defined Function.

**UDM** User-Defined Method.

**UDT** User-Defined Type.

**UML** Unified Modeling Language.

**Unaligned Configuration** A Teradata Database system where all data storage devices permit 512-byte read and write operations using native 512 byte sectors. Compare with [Aligned Configuration](#).

**UNIQUE Constraint** A constraint that ensures that all values in the column on which it is defined are unique. Examples include unique primary indexes, primary keys, unique secondary indexes, and UNIQUE constraints. Teradata Database implements UNIQUE constraints as [Unique Secondary Indexes](#) for [Nontemporal Tables](#) and as single-table [Join Indexes](#) for most temporal tables.

**Unique Primary Index** Unique Primary Index. A primary index that constrains the rows of its table such that no two rows are allowed to have the same primary index value. For purposes of uniqueness, nulls are considered equal. This differs from the treatment of nulls with comparison operators. The system returns an error for any attempt to insert a new row set or to update an existing row set such that the result would be two or more rows having the same index value for a UPI.

**Unique Secondary Index** Unique Secondary Index. A secondary index that constrains the rows of its table such that no two rows are allowed to have the same index value (that is, each index value is unique). For purposes of uniqueness, nulls are considered to be equal to one another. This differs from the treatment of nulls by comparison operators where comparisons are not allowed. The system returns an error for any attempt to insert a new row set or to update an existing row set such that the result would be two or more rows having the same index value for a USI.

The rows of a USI are hash-distributed, so USI rows might be present on an AMP on which no data rows exist and, conversely, data rows might exist on an AMP on which no USI rows exist.

Note that for a USI on the same columns as the PI of a nonunique PPI, USI rows are on the same AMP as their referenced data rows. Only one USI row can exist for a given index value. A row of a USI consists of exactly one row identifier for that index value.

**Uniqueness Value** A 32-bit unique value for a primary index or secondary index, or 44-bits for a NoPI object generated by the file system and appended to the rowkey for each row to make it distinguishable from other rows in the same table that have the same Rowkey value.

**UPI** See [Unique Primary Index](#).

**UPPI** Unique Partitioned Primary Index.

**USI** See [Unique Secondary Index](#).

**UV** Uniqueness Value.

## V

**vdisk** A vdisk is the collection of cylinders currently allocated to an AMP.

**VLDB** Very Large DataBase.

**Volatile Table** A type of temporary table whose definition and data are only accessible within their own session. Teradata Database does not retain volatile table definitions or data across sessions.

**vproc** Virtual Process

The Teradata architecture is based on a common node configuration. Each [Trusted Parallel Application](#) node can run one or more PE and AMP vprocs.

## W

**WAL** Write Ahead Logging.

A transaction logging scheme maintained by the File System in which a write cache for disk writes of permanent data is maintained using log records instead of writing the actual data blocks at the time a transaction is processed. Multiple log records representing transaction updates can then be batched together and written to disk with a single I/O thus achieving a large savings in I/O operations and enhancing system performance as a result.

## Glossary

# Index

## Numerics

1:1 relationships  
guidelines for placing the foreign key 68  
3NF  
definition of third normal form 86

## A

### Abbreviations

key to ANSI/SPARC database architecture flow diagram 52

Access column  
report/query analysis form term 147

Activity transaction modeling 54  
activity transaction modeling and referential integrity 95  
activity transaction modeling as the first step in the  
transition from logical design to physical design 123  
application form 145  
constraints form 142  
definition of an application for activity transaction  
modeling 144  
domains form 137, 138  
forms set 947  
goals of the activity transaction modeling process 123  
process activities 54, 124  
report/query analysis form 146  
system form 144

Activity transaction modeling terms  
definition of a column for the activity transaction  
modeling process 126  
definition of a constraint for the activity transaction  
modeling process 125  
definition of a domain for the activity transaction  
modeling process 124  
definition of a foreign key for the activity transaction  
modeling process 126  
definition of a primary key for the activity transaction  
modeling process 126  
definition of a row for the activity transaction modeling  
process 125  
definition of a table for the activity transaction modeling  
process 125  
definition of an identity column for the activity transaction  
modeling process 126  
definition of normalization for the activity transaction  
modeling process 127

Ad hoc join index  
definition 528

### Aggregate functions

restrictions on the use of aggregate functions in join index  
definitions 591

### Aggregate join indexes 552

aggregate join indexes defined using the EXTRACT  
function 556

multitable aggregate join indexes 217

single-table aggregate join indexes 217

tactical queries and aggregate join indexes 565

### Aggregate operations

how aggregate functions handle nulls 684

NUSIs and aggregate operations 483

### Alternate key

definition of an alternate key 75, 206

### ANSI DateTime data types

how DateTime data types work with nulls 684

### ANSI/SPARC architecture

abbreviations 52

conceptual level of the ANSI/SPARC architecture 50, 51

description of the ANSI/SPARC three-schema database  
architecture 49

detailed view of the ANSI/SPARC database architecture 51

external level of the ANSI/SPARC architecture 50, 51

high-level description of the ANSI/SPARC three-level  
database architecture 50

internal level of the ANSI/SPARC architecture 50, 51

### Application form 145

completing the application form 146

information on the application form 145

purpose of the application form 145

### Approximate numeric data types 823

DOUBLE PRECISION 824, 827

FLOAT 824, 827

REAL 824, 827

### Archive and Restore utility

Restore and block-level compression 694

Restore and reserved query bands 694

Restore and temperature level 694

restrictions with hash indexes 613, 614

restrictions with join indexes 613, 614

### Arithmetic functions

how SQL arithmetic functions handle nulls 683

### Arithmetic operators

how SQL arithmetic operators handle nulls 683

### Assignment Principle 664

ASSOCIATE STATISTICS. See COLLECT STATISTICS  
(Optimizer Form).

- Associative table 66
- Attributes**
- definition of a relation attribute 76
  - definition of an attribute as used in the Entity-relationship model 64
  - derivative attributes 65
  - foreign key attributes 64
  - non-key attributes 65
  - primary key attribute 64
- Augmentation rule**
- augmentation rule for functional dependencies 81
  - augmentation rule for multivalued dependencies 82
- Autocompression**
- autocompression and column-partitioned spool files 303
  - autocompression and user-defined compression methods 303
  - checking the effectiveness of autocompression 304
  - using the NO AUTO COMPRESS option 304
- Autocompression of column-partitioned tables** 281, 285, 288, 289, 292, 293, 294, 295, 296, 297, 298, 300, 303, 309, 310, 311, 312, 326, 327, 328, 329, 333, 334, 335
- B**
- Backups**
- designing the database to facilitate backups 891
- Batch referential integrity** 98, 99
- Batch\_Profile rules** 943
- Benefits of a join index** 534
- computing the benefits of a join index 534
  - definition of join index benefits 521
- BIGINT data type** 131, 824, 826
- alignment 826
  - size on aligned row format systems 826
  - size on packed64 systems 826
- Binary relationships** 67
- Bit mapping**
- complex conditional expressions with NUSIs 480
  - computing for NUSIs 481
  - NUSI bit mapping 479
  - performance advantages with NUSIs 479
  - restrictions for NUSIs 479
- BLOB data type** 130, 131, 829, 842
- alignment 829
  - size on aligned row format systems 829
  - size on packed64 systems 829
- Block level compression** 502, 608
- block level compression of hash indexes 608
  - block level compression of join indexes 502
- Boot disk**
- contents of the boot disk 876
  - disk 0 875
  - file systems 876
- Boyce-Codd normal form**
- definition of Boyce-Codd normal form 87
- Break-even percentage**
- definition of break-even percentage for compression 719
  - how to calculate the break-even percentage for compression 720
- Break-even point**
- definition of break-even point for compression 718
- BTEQ**
- BTEQ and output spool space 868
- Business rules**
- business rules and triggers 142
  - using triggers to define business rules 142
  - See also Constraints
- BYTE data type** 130, 131, 829
- alignment 829
  - size on aligned row format systems 829
  - size on packed64 systems 829
- Byte data types**
- BLOB 829, 842
  - BYTE 829
  - LONG VARBYTE 829
  - VARBYTE 829
- Byte versus octet**
- conceptual difference between bytes and octets 781, 987
  - presence bits 781, 987
- BYTEINT data type** 131, 825
- alignment 826
  - size on aligned row format systems 826
  - size on packed64 systems 826
- C**
- Canary queries**
- definition of canary queries 915
  - origin of the term canary query 915
- Candidate key**
- definition of a candidate key 76, 206
- Capacity planning considerations**
- BIGINT data type 825
  - BLOB data type 829
  - BYTE data type 829
  - BYTEINT data type 825
  - CHARACTER data type 840
  - CLOB data type 840
  - column sizing guidelines 791
  - data access frequency 691
  - data access frequency for cool data 693
  - data access frequency for hot data 693
  - data access frequency for icy data 694
  - data access frequency for warm data 693
  - DATE data type 830
  - DateTime data types 830
  - DECIMAL data type 826

DOUBLE PRECISION data type 826  
 FLOAT data type 826  
 INTEGER data type 825  
 INTERVAL data types 831  
 INTERVAL MONTH data type 831  
 INTERVAL YEAR data type 831  
 INTERVAL YEAR TO MONTH data type 831  
 LONG VARCHAR data type 840  
 NUMERIC data type 826  
 PERIOD(DATE) data type 838  
 PERIOD(TIME WITH TIME ZONE) data type 838  
 PERIOD(TIME) data type 838  
 PERIOD(TIMESTAMP WITH TIME ZONE) data type 838  
 PERIOD(TIMESTAMP) data type 838  
 REAL data type 826  
 reserved query bands for managing block-level compression and storage temperature of newly loaded data 694  
 sizing base tables 858, 859  
 sizing columns 791  
 sizing hash indexes 859  
 sizing index subtables 858  
 sizing join indexes 859  
 sizing LOB and XML subtables 861  
 sizing LOB subtables 858  
 sizing rows 846  
 sizing rows on a packed64 system 846  
 sizing rows on an aligned row format system 850  
 sizing user-defined routines 866  
 SMALLINT data type 825  
 TIME data type 830  
 TIME WITH TIME ZONE data type 830  
 TIMESTAMP WITH TIME ZONE 830  
 TVSTemperature reserved query bands for managing the storage temperature of newly loaded data 694  
 VARBYTE data type 829  
 VARCHAR data type 840  
 XML data type 842  
 XMLTYPE data type 842  
 Cardinality  
     definition of cardinality in an E-R relationship 67  
     definition of the cardinality of a relationship 67  
 CEILING function 782  
 Change ratings for columns 160  
 CHARACTER data type 131, 840  
 Character data types  
     CHARACTER 840  
     CLOB 840  
     LONG VARCHAR 840  
     VARCHAR 840  
 CHARACTER(n) CHARACTER SET GRAPHIC data type  
     alignment 841  
     size on aligned row format systems 841  
     size on packed64 systems 841  
 CHARACTER(n) CHARACTER SET KANJI data type  
     alignment 841, 842  
     size on aligned row format systems 841, 842  
     size on packed64 systems 841, 842  
 CHARACTER(n) CHARACTER SET LATIN data type  
     alignment 841  
     size on aligned row format systems 841  
     size on packed64 systems 841  
 CHARACTER(n) CHARACTER SET UNICODE data type  
     alignment 841  
     size on aligned row format systems 841  
     size on packed64 systems 841  
 CHECK constraints  
     definition of a CHECK constraint 642  
     effect of session collation on enforcing CHECK constraints for character values 642  
     rules for CHECK constraints 643  
 Child key  
     definition of a child key 206  
 CLOB data type 131, 840  
 CLOB(n) CHARACTER SET LATIN data type  
     alignment 842  
     size on aligned row format systems 842  
     size on packed64 systems 842  
 CLOB(n) CHARACTER SET UNICODE data type  
     alignment 842  
     size on aligned row format systems 842  
     size on packed64 systems 842  
 Closed World Assumption 630, 663, 677  
 COLLECT STATISTICS (Optimizer Form) statement  
     collecting statistics on a multitable join index 595  
     collecting statistics on a single-table join index 595  
     collecting statistics on PARTITION columns 805  
 Column attributes  
     COMPRESS 712  
 Column candidacy for covered access using a NUSI 484  
 Column partitioning  
     autocompression 281, 285, 288, 289, 292, 293, 294, 295, 296, 297, 298, 300, 303, 309, 310, 311, 312, 326, 327, 328, 329, 333, 334, 335  
     join indexes 285  
     performance issues 297  
     tables 285  
 Column sizing guidelines 791  
     column multi-value compression guidelines 791  
 Column-partitioned join indexes 285, 541, 547, 553  
     joins with a column-partitioned table, join index, or spool file 318  
 Column-partitioned spool files  
     joins with a column-partitioned table, join index, or spool file 318  
 Column-partitioned tables 285  
     anticipated workload characteristics 305

- autocompression and column-partitioned spool files 303
- bulk loading a column-partitioned table 318
- checking the effectiveness of autocompression 304
- comparing different table configurations for the number of I/O operations required to answer the same SELECT request 313
- deleting rows from a column-partitioned table 323
- effect of skewed data on column-partitioned tables 326
- inserting rows into a column-partitioned table using Teradata Parallel Data Pump 324
- interactions with user-specified compression methods files 303
- joins with a column-partitioned table, join index, or spool file 318
- locks and concurrency when accessing a column-partitioned table 316
- maintenance cost for column-partitioned tables and join indexes 322
- performance guidelines 306
- rules and limitations for column-partitioned tables 282
- selecting rows from a column-partitioned table 317
- setting the SERIALIZED option when inserting rows into a column-partitioned table using Teradata Parallel Data Pump 321
- source table in a data moving request 322
- updating rows from a column-partitioned table 323
- uses for column-partitioned tables 281
- using the NO AUTO COMPRESS option 304
- Columns**
  - change ratings for columns 160
  - choosing the column set for primary indexes 408
  - compressing column values 791
  - compressing hash index column values 599
  - compressing join index column values 599
  - definition of a column for the activity transaction modeling process 126
  - fixed length 773
  - guidelines for naming columns 134
  - miscellaneous column sizing guidelines 792
  - multi-value compression guidelines 712
  - naming conventions for columns 135
  - offsets 773
  - planning column size 791
  - purpose of presence bits 780
  - rules for comparing columns 132
  - storage structure for nested structured UDTs 793
  - storage structure for structured UDTs 792
  - types of derived columns 181
  - VARCHAR columns and base table row format 774
- Comparison operators**
  - how SQL comparison operators handle nulls 683
- Complement rule for multivalued dependencies 82
- Composite keys**
  - definition of a composite key 76
- COMPRESS attribute**
  - how column multi-value compression works 712
  - performance benefits 699
  - presence bits for compressed values 780
  - space saving benefits 732, 733, 735, 737, 738
- Compression**
  - autocompression 709
  - benefit of enhanced system performance 696
  - benefit of reduced storage costs 696
  - block level compression 502, 608
  - break-even percentage for compression 719
  - byte alignment considerations for multi-value compression 721
  - byte and character limits per column for multi-value compression 697
  - calculating the efficiency of multi-value compression 725
  - calculating the uncompressed table capacity 725
  - column multi-value compression of blanks 713
  - column multi-value compression of constants 713
  - column multi-value compression of nulls 713
  - column multi-value compression of zeros 713
  - compressing hash index rows 598, 607
  - compressing join index rows 598
  - compressing spool file values 697, 792
  - compression of spool file values 792
  - computing the break-even point for multi-value compression 718
  - definition of spool compression 697
  - determining how much compression can be realized with multi-value compression 721
  - determining how much table header space is used for column multi-value compression 711
  - guideline for determining the break-even point for compression 720
  - guidelines for column multi-value compression 712
  - how column multi-value compression works 711, 712
  - how presence bits map compressibility 782
  - how presence bits map nullability 782
  - how to calculate net capacity usage 727
  - how to calculate the compression ratio for multi-value compression 726
  - how to calculate the percent compression for multi-value compression 722
  - net capacity usage as a measure of the net savings realized from compressing a set of values for a column 727
  - number of presence bits required to represent compressed values 784
  - presence bits 780
  - presence bits octet 781, 987
  - relationship between percent compression and compression ratio for a given compression calculation 727
  - row compression 709
  - row header compression 710

- row header presence bits and nullability 781
  - row structure for algorithmically compressing variable length columns 743
  - row structure for multi-value compressing and algorithmically compression variable length columns 750
  - row structure for multi-value compressing variable length columns 747
  - row structures for compressing variable length columns 742
  - typical uses for the capacity reclaimed by multi-value compression 716
  - Compression ratio**
    - definition of the compression ratio for multi-value compression 726
    - relationship of the compression ratio to the percent compression 727
  - conditions that support read from fallback 672
  - Connectivity in a relationship** 67
  - Constraint codes**
    - definitions of constraint codes for the constraint form 143
  - Constraint form**
    - definitions of constraint codes for the constraint form 143
  - Constraints**
    - CHECK constraints** 642
    - CHECK constraints, character data, and session collation** 643
    - check time for constraints by Teradata Database 657
    - constraint names 642
    - constraints activity transaction modeling form 142
    - deferred integrity constraint checking 658
    - defining constraints using ALTER TABLE requests 142
    - defining constraints using CREATE TABLE requests 142
    - enforcement through views 657
    - FOREIGN KEY ... REFERENCES constraints** 644
    - immediate integrity constraint checking 657
    - join constraints and join index definitions 519
    - maintaining database integrity using constraints 138
    - named constraints 642
    - PRIMARY KEY constraints** 654
    - referential integrity constraints 138
    - UNIQUE constraints** 656
    - UNIQUE constraints and NOT NULL** 656
    - when constraints are enforced by Teradata Database 657
  - Constraints form** 142
    - information recorded on the constraints form 142
    - purpose of the constraints form 142
  - Corruption**
    - detection of physical data corruption 664, 667
  - Cost of a join index** 520, 534
    - cost of creating a join index 522
    - cost of maintaining a join index 525, 526, 528, 529, 531
    - disk resources 522
  - Cost/benefit analysis for a join index** 534
  - Covered access**
    - column candidacy for covered access 484
  - CRASHDUMPS user**
    - DIP utility 880
    - space requirements for the CRASHDUMPS user 886
    - usage 876
  - CREATE JOIN INDEX statement** 695
  - CurHashBucketSize DBS Control parameter** 229
  - Cursors**
    - database integrity issues 658
    - positioned cursors 658
    - updatable cursors 658
  - Cylinder indexes** 247
    - calculating the percentage of data space required for cylinder index overhead 883
    - overhead required for cylinder indexes 884
- D**
- Data**
    - data independence in database design 49
    - definition of data dependence 110
    - definition of data independence 49, 110
    - optimal data access 408
    - types of derived columns 181
    - uniform distribution 408
    - See also Derived data
  - Data block** 248
    - structure of a data block 770
  - Data corruption**
    - detection of physical data corruption 664, 667
  - Data demographics**
    - calculating data demographics for multicolumn tables 168
    - data demographic trends 167
    - information required for multicolumn objects for the table form 166
    - information required for single-column objects on the table form 158
  - Data dictionary**
    - estimating space requirements for the data dictionary 886
  - Data disks**
    - allocating space to primary indexes 450
    - contents of data disks 876
    - data disk space allocation 877
    - permanent space usage 876, 879
    - reserved space usage 876
    - spool space usage 879
    - temporary space usage 879
  - Data independence**
    - data independence in database design 49
    - definition of data independence 49
  - Data manipulation language.** See DML
  - Data marts**
    - common problems with data marts 23

- data mart-centrism 24
- data marts and data warehouses 19
  - definition of a data mart 19
  - definition of a dependent data mart 20
  - definition of a logical data mart 20
  - definition of an independent data mart 20
  - why you should avoid data mart-centrism 24
- Data modeling
  - enterprise data model 58
  - guidelines for data modeling 47
- Data models
  - applications of the entity-relationship model 63
  - contrasting definitions of the term data model 61
  - entity-relationship model 62
  - relational model, version RM/T 62
- Data quality
  - data quality problems compromise information security 621
  - detecting physical data corruption 667
  - disk I/O integrity checking and physical data quality 666
  - disk I/O integrity levels and table types 669
  - inclusion dependencies 632
  - logical database integrity 629
  - physical data integrity constraints 625
  - physical database integrity and data quality 664
  - semantic data integrity constraints 623
  - sources of data quality problems 618
  - the role of human error in producing data quality problems 622
  - using checksums to determine physical data quality 668
- Data space
  - calculating the amount of usable data space in Indicator Mode 888
  - calculating the amount of usable data space in Record Mode 888
- Data storage
  - storing CHARACTER data efficiently 716
  - storing data efficiently 713
  - storing DECIMAL/NUMERIC data efficiently 713
  - storing INTEGER data efficiently 714
  - storing VARCHAR and VARBYTE data efficiently 715
  - typical uses for the capacity reclaimed by multi-value compression 716
- Data types
  - approximate numeric data types 823
  - ARRAY 844
  - BIGINT 824, 825
  - BLOB 829
  - BYTE 829
  - BYTEINT 825
  - CHARACTER 840
  - choosing a data type to describe a domain 130
  - CLOB 840
  - DATE 830
  - DECIMAL 824, 826
  - DOUBLE PRECISION 824, 826
  - exact numeric data types 823
  - FLOAT 824, 826
  - hashing 250
  - INTEGER 824, 825
  - INTERVAL DAY 831
  - INTERVAL DAY TO HOUR 831
  - INTERVAL DAY TO MINUTE 831
  - INTERVAL DAY TO SECOND 831
  - INTERVAL HOUR 831
  - INTERVAL HOUR TO MINUTE 831
  - INTERVAL HOUR TO SECOND 831
  - INTERVAL MINUTE 831
  - INTERVAL MINUTE TO SECOND 831
  - INTERVAL MONTH 831
  - INTERVAL SECOND 831
  - INTERVAL YEAR 831
  - INTERVAL YEAR TO MONTH 831
  - JSON 843
  - LONG VARBYTE 829
  - LONG VARCHAR 840
  - MBR 845
  - NUMERIC 824, 826
  - PERIOD(DATE) 838
  - PERIOD(TIME(precision) WITH TIME ZONE) 838
  - PERIOD(TIME(precision)) 838
  - PERIOD(TIMESTAMP(precision) WITH TIME ZONE) 838
  - PERIOD(TIMESTAMP(precision)) 838
  - REAL 824, 826, 829
  - SMALLINT 824, 825
  - TIME 830
  - TIME WITH TIME ZONE 830
  - TIMESTAMP 830
  - TIMESTAMP WITH TIME ZONE 830
  - UDTs 843
  - VARBYTE 829
  - VARCHAR 840
  - VARRAY 844
  - XML 842
  - XMLETYPE 842
- Data warehouses
  - ad hoc queries 44
  - data marts and data warehouses 19
  - data placement to support parallel processing 26
  - detail data 41
  - how data warehousing support differs from OLTP 48
  - intelligent internodal communication in an MPP system 29
  - negative effects of storing only summary data 40
  - non-unique indexes and data warehousing support queries 192
  - OLTP 36

- proactive use of detail data 41
- query complexity as a function of the value of the result set
  - returned by queries 42
- request parallelism 32
- synchronization of parallel operations 34
- Teradata definition of a data warehouse 24
- the database is the heart of the data warehouse 19
- Database design**
  - activity transaction modeling 53, 54
  - data independence 49
  - data placement to support parallel processing 26
  - database design life cycle 47
  - designing for backups 891
  - designing for OLTP versus designing for data warehousing 48
  - designing the database to support data warehousing operations 48
  - designing the database to support decision support operations 48
  - designing the database to support OLTP operations 48, 49
  - designing to support ad hoc queries 44
  - designing to support tactical queries 45
  - differences between OLTP queries and decision support queries 48
  - isolating large tables for backups 891
  - isolating users from the database using views 941
  - logical database design phase 53
  - normalization in the logical database design phase 53
  - physical design phase 55
  - summary data has a low information content 41
  - usage considerations for OLTP 36
  - why detail data is better than summary data for supporting data warehouse requests 41
  - why you should not store only summary data in the database 40
  - See also Physical design
- Database design principles**
  - Assignment Principle 664
  - Closed World Assumption 663
  - Domain Closure Assumption 630
  - Entity Integrity Rule 663
  - Golden Rule 629, 664
  - Information Principle 663
  - Principle of Interchangeability 663
  - Principle of the Identity of Indiscernibles 664
  - principles of normalization 664
  - Referential Integrity Rule 663
  - Unique Name Assumption 630
- Database integrity**
  - logical database integrity 629
- Database maxima** 927
- Database structures**
  - Batch\_Profile end user profile 943
  - objects 941
  - profiles 941
  - Query\_Profile end user profile 942
  - rules for end user profiles 943
  - rules for Query\_Database 943
  - rules for the Batch\_Profile 943
  - rules for Upsert\_Database 944
  - Upsert\_Profile end user profile 942
  - work flows 941
- DATABLOCKSIZE**
  - performance and DATABLOCKSIZE 959
- DATE** data type 131, 830
  - alignment 831
  - size on aligned row format systems 831
  - size on packed64 systems 831
- DateTime** data types
  - DATE 830
  - TIME 830
  - TIME WITH TIME ZONE 830
  - TIMESTAMP 830
  - TIMESTAMP WITH TIME ZONE 830
- DBC** system user 878
- DBS** Control parameters
  - CurHashBucketSize parameter 229
  - NewHashBucketSize parameter 229
- DECIMAL** data type 131, 824, 828
  - alignment 827
  - size on aligned row format systems 827
  - size on packed64 systems 827
- Decomposing relations** 83, 89
  - horizontal decomposition 83
  - vertical decomposition 84
- Decomposition rule for dependencies**
  - decomposition rule for functional dependencies 81
  - decomposition rule for multivalued dependencies 82
- Default definitions for primary indexes when none is defined**
  - explicitly 263
- Deferred integrity constraint checking** 658
- Definition of join index cost** 520
- Definition of normalization terms**
  - alternate keys 75
  - attribute heading 76
  - candidate keys 76
  - composite keys 76
  - definition of relational schemas 78
  - domains 76
  - field 76
  - foreign keys 76
  - intelligent keys 77
  - natural keys 77
  - primary keys 78
  - relations 78
  - repeating groups 78
  - super key 78
  - surrogate keys 78

- tuple instance 76
- tuples 78
- Degree of a relationship 67
- Delayed partition elimination 402
- Denormalization
  - alternatives to denormalization 178
  - considerations for denormalization 176
  - costs of denormalization 119
  - denormalizing the physical schema for usability 112
  - denormalizing the virtual schema for usability 112
  - design using the logical model instead of denormalizing the physical database 180
  - how denormalization makes cross-functional analysis difficult 119
  - how denormalization makes data relationships ambiguous 113
  - how denormalization makes referential integrity difficult to enforce 115
  - negative and positive effects 176
  - optimizing performance with 178
  - problems for non-dimensional data 117
  - problems with denormalizing for performance 113
  - problems with unbalanced hierarchies 118
  - reasons for not using derived data to denormalize the physical database schema 182
  - reasons for using a join index to denormalize the physical database schema 181
  - reasons for using prejoins to denormalize the physical database schema 180
  - repeating groups 178
  - using a join index to denormalize the physical schema 181
  - using global temporary tables to denormalize the physical database 183
  - using views to provide the illusion of denormalizing the physical database schema 185
  - using views with aggregation to provide the illusion of denormalizing the physical database schema 185
- Denormalized views 112, 115
- Dependency theory
  - definition of a determinant 80
  - full functional dependencies 80
  - functional dependencies 81
  - multivalued dependencies 81
  - transitive dependencies 81
- Dependent data mart 20
- Depot area
  - overhead required for the Depot area 882, 884
- Derived columns
  - normalization and derived columns 181
- Derived data
  - denormalizing standalone derived data 182
  - handling guidelines for standalone derived data 182
  - reasons for not using derived data to denormalize the physical database schema 182
  - using global temporary tables instead of derived data 183
- Determinant
  - definition of a determinant in dependency theory 80
- DIFFERENCE relational operator 79
- Dimensional modeling
  - contrasting with entity-relationship modeling 190
  - definition of dimensional modeling 187
  - dimensional views 186
  - snowflake schema 189
  - star schema 188
- Dimensional views
  - definition of a dimensional view 186
  - dimensional views as a normalized alternative to physical dimensional model design 186
- DIP utility 880
- Disk 0 system boot disk 875
- Disk 1 system dump and memory swapping disk 875
- Disk I/O integrity checking 666
  - checksum levels and CPU utilization 671
  - DBS Control utility 670
  - detecting data corruption 667
  - effect on CPU utilization 667
  - folding 64-bit block data into 32-bit check data 668
  - levels of disk I/O integrity checking 666
  - setting system-level defaults 670
  - settings by table type 669
  - using checksums to determine disk I/O integrity 668
- Disk resources
  - disk resource cost of a join index 522
- Distinct UDT data type
  - alignment 843
  - size on aligned row format systems 843
  - size on packed64 systems 843
- DIVIDE relational operator 80
- DML
  - basic SQL DML statements 209
  - purpose of the WHERE clause in SQL DML 210
- Domain Closure Assumption 630
  - definition of the Domain Closure Assumption 630
- Domain data types
  - BIGINT 131
  - BLOB 130, 131
  - BYTE 130, 131
  - BYTEINT 131
  - CHARACTER 131
  - CLOB 131
  - DATE 131
  - DECIMAL 131
  - DOUBLE PRECISION 131
  - FLOAT 131
  - GRAPHIC 131
  - INTEGER 131
  - INTERVAL 131
  - LONG VARCHAR 131

- Period 131
  - REAL 131
  - SMALLINT 131
  - TIME 131
  - TIMESTAMP 131
  - VARBYTE 130, 131
  - VARCHAR 131
  - Domain type data codes 139
  - Domains
    - activity transaction modeling form 138
    - defining domains during the logical design phase 132
    - definition of a domain 76
    - definition of domains for the activity transaction modeling process 124
    - domain rules for foreign keys 102
    - domain rules for primary keys 101
    - domains activity transaction modeling form 137
    - domains and UDTs 124, 129, 130, 132, 134, 138, 139
    - domains and user-defined data types 129, 130, 131, 132, 134, 136
    - primary key-foreign key relationships in domains 129
    - storing definitions 137
    - the concept of domains in the activity transaction modeling process 128
  - Domains form 138
    - information in the domains form 138
    - purpose of the domains form 138
  - DOUBLE PRECISION data type 131, 824, 827
    - alignment 827
    - size on aligned row format systems 827
    - size on packed64 systems 827
  - Dynamic partition elimination 402
- E**
- Efficiency of compression
    - calculating the efficiency of multi-value compression 725
  - Enterprise data model 58
    - things to avoid when developing an enterprise data model 58
  - Entity
    - definition of an entity 62, 63
    - entity relationship ratios 64
  - Entity Integrity Rule 92, 96, 663
  - Entity-relationship model 62
    - applications of the entity-relationship model 63
    - associative tables 66
    - definition of a binary relationship 67
    - definition of a tertiary relationship 67
    - definition of a unary relationship 67
    - definition of an associative table 66
    - definition of an n-ary relationship 67
    - definition of connectivity 67
    - definition of degree 67
- definition of non-prime tables 66
  - definition of prime tables 65
  - definition of the cardinality of a relationship 67
  - entity-relationship modeling contrasted with dimensional modeling 190
  - existence dependencies 67
  - guidelines for drawing M:M relationship tables for users 70
  - guidelines for placing the foreign key in a 1:1 relationship 68
  - modeling 1:1 relationships 68
  - modeling 1:M relationships 69
  - modeling M:M relationships 70
  - non-prime tables 66
  - terms associated with relationships 67
  - Exact numeric data types 823
    - BIGINT 824, 826
    - BYTEINT 825
    - DECIMAL 824, 827
    - DOUBLE PRECISION 824
    - FLOAT 824
    - INTEGER 824, 826
    - NUMERIC 824, 827
    - REAL 824
    - SMALLINT 824, 826
  - Existence dependency in a relationship 67
  - EXPLAIN PLAN. See EXPLAIN request modifier.
  - EXPLAIN request modifier
    - flushing intermediate spool and Last Use in the EXPLAIN request modifier 868
    - information provided by the EXPLAIN request modifier 202
    - reasons for using the EXPLAIN request modifier 196
    - using the EXPLAIN request modifier to determine NUSI bit mapping 481
  - EXTRACT function
    - use in aggregate join index definitions 556
- F**
- Fallback
    - defining fallback for join indexes 508
    - hash indexes and fallback 607
    - hash indexes do not default to fallback if their base tables are defined with fallback 196
    - join indexes do not default to fallback if their base tables are defined with fallback 196
    - space considerations for fallback tables and index subtables 866
  - FastLoad utility
    - FastLoad and block-level compression 695
    - FastLoad and reserved query bands 695
    - FastLoad and temperature level 695
    - restrictions with hash indexes 461
    - restrictions with join indexes 461, 608, 613, 614

Few-AMP joins and tactical queries 901

Field

definition of a field 76

field contrasted with column 76

Flag byte

definition of flag byte 773

FLOAT data type 131, 824, 827

alignment 827

size on aligned row format systems 827

size on packed64 systems 827

FLOOR function 729

Foreign join index

definition of foreign join index 528

Foreign key

definition of a foreign key 76, 94, 206

definition of a foreign key for the activity transaction modeling process 126

domain rules and referential integrity 102

foreign key attributes 64

forming referential constraints 98

naming foreign key columns 136

nulls in foreign keys 96

placing guidelines 68

uses for foreign keys 94

using 94

FOREIGN KEY ... REFERENCES constraints 651, 652

CREATE TABLE option rules 649

FOREIGN KEY ... REFERENCES constraints 644

rules for FOREIGN KEY ... REFERENCES constraints 649

rules for FOREIGN KEY ... REFERENCES constraints for Referential Constraints 652

rules for table-level FOREIGN KEY ... REFERENCES constraints 652

Forms for ATM

application form 145

constraints form 142

domains form 138

report/query analysis form 146

system form 144

Fragmentation

overhead required for fragmentation 884

FREESPACE, performance and 960

Functional dependencies

conflicts with normalization 105

definition of functional dependencies 81

full functional dependencies 80

inference axioms for functional dependencies 81

preservation of functional dependencies 103

Fundamental relational database principles

Assignment Principle 664

Closed World Assumption 630, 663

Entity Integrity Rule 92, 663

Golden Rule 629, 664

Information Principle 663

Principle of Interchangeability 663

Principle of Orthogonal Design 664

Principle of the Identity of Indiscernibles 664

principles of normalization 664

Referential Integrity Rule 95, 663

## G

geospatial data type

MBB 845

Geospatial data types

MBR 845

ST\_GeOMETRY 845

Global indexes. See Global join indexes

Global join indexes 500, 506, 903, 907

Global temporary tables

using global temporary tables instead of derived data 183

using global temporary tables instead of prejoins 183

using global temporary tables to denormalize the physical database 183

using global temporary tables to enhance performance 184

Golden Rule 629, 664

GRAPHIC data type 131

Group AMP operations and tactical queries 900, 901

Grouping key definition of a grouping key 206

## H

Hash 243

Hash buckets

criteria for converting from 16-bit hash buckets to 20-bit

hash buckets 231

hash bucket number 225

how small numbers of hash buckets on a system deal

poorly with skew 230

how the number of hash buckets on a system is controlled by the CurHashBucketSize DBS Control parameter 229

how the number of hash buckets on a system is controlled by the NewHashBucketSize DBS Control parameter 229

number of hash buckets per system 229

size of hash buckets on Teradata Database systems 229

Hash collisions 228

Hash indexes 195, 217

applications of hash indexes 606

block level compression of hash indexes 608

collect statistics on base table columns instead of hash index columns 610

comparison of hash indexes with base tables 605

comparison of hash indexes with secondary indexes 606

comparison of hash indexes with single-table join indexes 603

compressing hash index column values 599

compressing hash index rows 599, 607

defining Fallback for hash indexes 607  
 definition of a hash index 191, 606  
 hash index row compression 598  
 hash indexes and partitioned primary indexes 615  
 hash indexes are not supported with FastLoad  
     FastLoad utility  
         restrictions with hash indexes 613  
 hash indexes are not supported with MultiLoad 613  
 hash indexes are not supported with permanent journal recovery 613  
 hash indexes are not supported with tables that have triggers 613  
 hash indexes are not supported with the Archive and Restore utility 613  
 how hash indexes are stored 219  
 maintenance considerations for tactical query support 566  
 mandatory use of hash indexes 218  
 organizing the storage of hash indexes 598  
 restrictions for hash indexes on tables with triggers 613  
 restrictions for permanent journal recovery 613  
 restrictions on data types of hash index columns 611  
 restrictions on hash index definitions 611  
 restrictions on hash indexes 218  
 restrictions on number of base tables columns referenced in a hash index definition 611  
 restrictions on number of hash indexes defined per base table 611  
 restrictions on the use of the system-derived PARTITION and PARTITION[#Ln] columns in a hash index definition 612  
 restrictions with the Archive and Restore utility 614  
 restrictions with the FastLoad utility 461  
 row structure of hash indexes on aligned row format systems 777  
 row structure of hash indexes on packed64 systems 775  
 row structure on aligned row format systems 777  
 row structure on packed64 systems 775  
 space overhead formula for hash indexes 599  
 tradeoff considerations for hash indexes 615  
     using to support specific queries and query workloads 605  
**Hash key**  
     definition of a hash key 206  
**Hash maps** 233  
     bit map hash map 233  
     consistency of hash maps across different configurations 230  
     current configuration fallback hash map 233  
     eliminating skew 230  
     hash bucket number 234  
     hash map lookup 234  
     hash maps are maintained by the BYNET 229  
     Open PDE hash map 233  
     primary hash map of the current configuration 233  
     primary reconfiguration hash map 233  
     reconfiguration fallback hash map 233  
     HASHAMP built-in function 243  
     HASHBAKAMP built-in function 243  
     Hash-based table partitioning 234  
     HASHBUCKET built-in function 244  
**Hashed row**  
     cylinder index 247  
     data block 248  
     effect of different data types on hashed row allocation for the hashed columns 250  
     master index 246  
**Hashed row access** 246  
     role of the cylinder index in hashed row access 247  
     role of the data block in hashed row access 248  
     role of the master index in hashed row access 246  
**Hashing**  
     advantages of hashing over indexing 222  
     algorithm 200, 225  
     description of the HASHAMP built-in function 243  
     description of the HASHBAKAMP built-in function 243  
     description of the HASHBUCKET built-in function 244  
     description of the HASHROW built-in function 244  
     distributing join index rows across the AMPs 602  
     hash mapping 250  
     hash mapping of indexes 200  
     hashing of primary indexes 221  
     how hashing works 220  
     minimizing hash collisions 228  
     row hash value 197, 198  
     tradeoffs between hashing and indexing 222  
     uniqueness value 197, 227  
     using the HASHROW function 244  
     USIs 462  
**Hashing algorithm** 200  
     definition 225  
**Destination Selection Word** 234  
     hash bucket number 225  
     input values 226  
     uniqueness value 227, 772  
**Hash-ordered non-unique secondary indexes** defined on a single column with no ALL option 194  
**Hash-ordered non-unique secondary indexes** defined on a single column with the ALL option 194  
**Hash-partitioned row allocation** to the AMPs for a PPI row 242  
**Hash-partitioned row allocation** to the AMPs for an unpartitioned primary index row 235  
**Hash-related built-in functions** 243  
     HASHAMP function 243  
     HASHBAKAMP function 243  
     HASHBUCKET function 244  
     HASHROW function 244  
**HASHROW built-in function** 244

**Heading**  
 definition of an attribute heading 76

**Horizontal partitioning** 234

**I**

**Identity columns** 818  
 definition for the activity transaction modeling process 126  
 performance and identity columns 819

**Immediate integrity constraint checking** 657

**Inclusion dependencies** 632

**Independent data mart** 20

**Indexed views.** See *Join indexes*.

**Indexes**  
 advantages of indexes 195  
 advantages of indexing over hashing 224  
 comparing types of indexes 252  
 comparison of the functions of Teradata Database indexes 252  
 definition of a hash index 606  
 definition of a join index 499  
 definition of an index 207  
 differences between indexes and keys 206, 208  
 disadvantages of indexes 195  
 hash index applications 606  
 hash index definition 191  
 hash indexes 195  
 hash mapping 200, 250  
 how indexes are used in WHERE clause predicates 210  
 how indexes work 220  
 join index applications 504  
 join index definition 191  
 join indexes described 194, 499  
 Multiple and composite NUSI access performance compared 476  
 multitable join indexes 194  
 non-unique indexes 192  
 non-unique multilevel partitioned primary indexes 193  
 non-unique primary indexes 192  
 non-unique secondary indexes 192  
 non-unique single-level partitioned primary indexes 193  
 non-unique unpartitioned primary indexes 193  
 partitioned primary indexes 192  
 primary indexes 211, 262  
 purpose of join indexes 500  
 purpose of primary indexes 211  
 selection factors for indexes 259  
 selectivity of indexes 197  
 single-table join indexes 194  
 sizing hash indexes 859  
 sizing join indexes 859  
 SQL and indexes 209  
 summary of differences between indexes and keys 208  
 tradeoffs between indexing and hashing 222

types of indexes supported by Teradata Database 191  
 unique indexes 191  
 unique multilevel partitioned primary indexes 193  
 unique primary indexes 191  
 unique primary indexes and nulls 675  
 unique secondary indexes 191  
 unique single-level partitioned primary indexes 193  
 unique unpartitioned primary indexes 193

**Indexing versus hashing** 220  
 advantages of indexing over hashing 224  
 how indexing and hashing differ 220  
 tradeoffs between indexing and hashing 222

**Indicator Mode** operation 888

**Inference axioms for dependencies** 82

**Inference axioms for functional dependencies**  
 augmentation rule 81  
 decomposition rule 81  
 pseudotransitivity rule 81  
 reflexive rule 81  
 transitivity rule 81  
 union rule 81

**Inference axioms for inclusion dependencies** 633  
 permutation rule 633  
 projection rule 633  
 reflexive rule 633  
 transitivity rule 633

**Inference axioms for multivalued dependencies** 82  
 augmentation rule 82  
 complement rule 82  
 decomposition rule 82  
 mixed inference rules 83  
 pseudotransitivity rule 82  
 reflexive rule 82  
 transitivity rule 82  
 union rule 82

**Information Principle** 663

**In-place join index definition** 527

**INSERT** 695

**INSERT ... SELECT** 695

**Instance**  
 definition of a tuple instance 76

**INTEGER** data type 131, 824, 826  
 alignment 826  
 size on aligned row format systems 826  
 size on packed64 systems 826

**Integer data types**  
 BIGINT 826  
 BYTEINT 825  
 INTEGER 826  
 SMALLINT 826

**Integrity constraints**  
 CHECK constraints 642  
 checksums for enforcing disk I/O integrity 668  
 enforcement of semantic constraints 657

- immediate integrity constraint checking 657, 658
- physical integrity constraints 664
- PRIMARY KEY constraints 654
- referential integrity constraints 644
- UNIQUE constraints 656
- uniqueness constraint 656
- Intelligent key
  - definition of an intelligent key 77
- INTERSECTION relational operator 80
- INTERVAL data type 831
  - Interval data types 131
    - INTERVAL DAY 832
    - INTERVAL DAY TO HOUR 833
    - INTERVAL DAY TO MINUTE 833
    - INTERVAL DAY TO SECOND 833
    - INTERVAL HOUR 834
    - INTERVAL HOUR TO MINUTE 835
    - INTERVAL HOUR TO SECOND 835
    - INTERVAL MINUTE 835
    - INTERVAL MINUTE TO SECOND 836
    - INTERVAL MONTH 832
    - INTERVAL SECOND 836
    - INTERVAL YEAR 831
    - INTERVAL YEAR TO MONTH 831
  - INTERVAL DAY data type 832
    - alignment 837
    - size on aligned row format systems 837
    - size on packed64 systems 837
  - INTERVAL DAY TO HOUR data type 833
    - alignment 837
    - size on aligned row format systems 837
    - size on packed64 systems 837
  - INTERVAL DAY TO MINUTE data type 833
    - alignment 837
    - size on aligned row format systems 837
    - size on packed64 systems 837
  - INTERVAL DAY TO SECOND data type 833
    - alignment 837
    - size on aligned row format systems 837
    - size on packed64 systems 837
  - INTERVAL HOUR data type 834
    - alignment 837
    - size on 64-bit aligned row format systems 837
    - size on packed64 systems 837
  - INTERVAL HOUR TO MINUTE data type 835
    - alignment 837
    - size on aligned row format systems 837
    - size on packed64 systems 837
  - INTERVAL HOUR TO SECOND data type 835
    - alignment 837
    - size on aligned row format systems 837
    - size on packed64 systems 837
  - INTERVAL MINUTE data type 835
    - alignment 837
- size on aligned row format systems 837
- size on packed64 systems 837
- INTERVAL MINUTE TO SECOND data type 836
  - alignment 837
  - size on aligned row format systems 837
  - size on packed64 systems 837
- INTERVAL MONTH data type 832
  - alignment 837
  - size on aligned row format systems 837
  - size on packed64 systems 837
- INTERVAL SECOND data type 836
  - alignment 837
  - size on 64-bit aligned row format systems 837
  - size on packed64 systems 837
- INTERVAL YEAR data type 831
  - alignment 837
  - size on aligned row format systems 837
  - size on packed64 systems 837
- INTERVAL YEAR TO MONTH data type 831
  - alignment 837
  - size on aligned row format systems 837
  - size on packed64 systems 837

**J**

- Join column
  - report/query analysis form term 147
- Join indexes 215
  - ad hoc join indexes 528
  - aggregate join indexes 541, 552
  - aggregate join indexes and tactical queries 565
  - aggregate join indexes defined using the EXTRACT function 556
  - and partitioned primary index 615
  - applications of join indexes 504
  - base join index definitions on foreign key-primary key equality predicates 519
  - block level compression of join indexes 502
  - built-in functions are resolved at the time a join index is created, not at run time 574
  - calculating the benefit percentage for a join index 535
  - collecting statistics on a join index 519
  - collecting statistics on a multitable join index 595
  - collecting statistics on a single-table join index 595
  - collecting statistics on join indexes 595, 596
  - column-partitioned join indexes 541, 547, 553
  - comparison of single-table join indexes with hash indexes 603
  - compressing join index column values 599
  - compressing join index rows 599
  - computing join index payback 522
  - computing the benefits of a join index with a cost/benefit analysis 534
  - computing the benefits of join indexes 535

computing the payback factor for join indexes 536  
 computing the query ratio for a join index 535  
 cost of creating a join index 522  
 cost of disk resources 522  
 cost of maintaining a join index 525, 526  
 cost of maintaining a join index as a function of hits/block 528  
 cost of maintaining a join index as a function of row size 529  
 cost of maintaining a join index as a function of the insertion method used to load a table 531  
 cost/benefit analysis 520, 534  
 creating an aggregate join index using a CREATE JOIN INDEX request 552  
 defined with inequality conditions 556  
 defining a join index using an outer join 558  
 defining a simple join index 549  
 defining fallback for join indexes 508  
 definition of a join index 191, 499  
 definition of a sparse join index 505  
 definition of an aggregate join index 553  
 definition of join index benefits 521  
 definition of join index cost 520  
 definition of the query ratio for a join index 535  
 design considerations for join indexes 517  
 distributing join index rows across the AMPs 602  
 equation for calculating the benefit percentage for a join index 536  
 equation for calculating the payback factor for a join index 536  
 equation for computing the join index payback factor 536  
 foreign key join indexes 528  
 functions of a single-table join index 546  
 functions of aggregate join indexes 552  
 global join indexes 500, 506, 903, 907  
 guidelines for collecting statistics on a single-table join index or its base table 596  
 guidelines for collecting statistics on multitable join indexes 595  
 guidelines for collecting statistics on single-table join index columns 596, 597  
 hash-ordered join indexes 540  
 how join indexes are stored 217  
 in-place join indexes 527  
 join constraints and join index definitions 519  
 join index row compression 598  
 join indexes and base tables 503  
 join indexes are not supported with FastLoad 613  
 join indexes are not supported with MultiLoad 613  
 join indexes are not supported with permanent journal recovery 613  
 join indexes are not supported with tables that have triggers 613  
 join indexes are not supported with the Archive and Restore utility 613  
 join indexes cannot be defined using a system-derived PARTITION or PARTITION#Ln column 573  
 join indexes compared with primary and secondary indexes 252  
 join indexes described 194  
 join indexes for supporting specific queries and query workloads 503  
 join indexes used to create a different row distribution geography than its underlying base table 540  
 join indexes used to vertically partition the columns of a base table 540  
 maintenance considerations for tactical query support 566  
 maintenance generalizations 533  
 mandatory use of join indexes 216  
 multitable aggregate join indexes 217, 541  
 multitable join indexes 194  
 multitable simple join indexes 217, 540  
 NUSIs and join indexes 516  
 organizing the storage of join indexes 598  
 performance and join indexes 509  
 PPI versus value-ordered NUSI for range query support 506  
 problem solving strategies related to aggregate join indexes 553  
 problem solving strategies related to simple join indexes 549  
 purpose of aggregate join indexes 553  
 purpose of join indexes 500  
 purpose of simple join indexes 541  
 purpose of single-table join indexes 546  
 reasons for using join indexes to denormalize the physical database schema 181  
 restricted data types for join index columns 573  
 restriction on number of join indexes per base table 572  
 restriction on outer joins in join index definitions 573  
 restriction on the number of columns per base table 573  
 restrictions for join indexes on tables with triggers 613  
 restrictions on join indexes 216  
 restrictions on ORDER BY clauses in join index definitions 593  
 restrictions on specifying a unique primary index for join indexes 574  
 restrictions on specifying unique secondary indexes for a join index 574  
 restrictions on the number of join indexes selected per table 575  
 restrictions on the use of aggregate functions in join index definitions 591  
 restrictions on WHERE clause predicates in sparse join index definitions 592  
 restrictions with permanent journal recovery 613  
 restrictions with the Archive and Restore utility 614

restrictions with the FastLoad utility 461  
 row structure of join indexes on aligned row format systems 777  
 row structure of join indexes on packed64 systems 775  
 row structure on aligned row format systems 777  
 row structure on packed64 systems 775  
 rules for using OUTER JOIN in join index definition 573  
 selecting a secondary index for a join index 594  
 selecting the primary index for a join index 594  
 single-table aggregate join indexes 217, 541  
 single-table join index maintenance costs 548  
 single-table join index support for parameterized queries 548  
 single-table join indexes 194, 546  
 single-table join indexes and tactical queries 564  
 single-table simple join indexes 216, 540  
 sorting join indexes in value order 519  
 space overhead formula for join indexes 599  
 sparse join indexes 541, 556  
 summary of join index benefits 534  
 summary of join index functions 502  
 tradeoff considerations for join indexes 615  
 types of join indexes 216  
 using join indexes to denormalize the physical database schema 181  
 using outer joins in a simple join index definition 518  
 value-ordered join indexes 540, 602  
**JOIN** relational operator 80  
**Joins**  
 definition of a join 80  
 few-AMP joins and tactical queries 901  
 join constraints and join index definitions 519  
 nested joins and tactical queries 902  
**JSON** data type 843

**K**

**Keys**  
 alternate keys 75, 206  
 candidate keys 76, 206  
 child keys 206  
 composite keys 76  
 definition of a key 77, 206  
 differences between keys and indexes 206  
 foreign keys 76, 206  
 grouping keys 206  
 hash keys 206  
 intelligent keys 77  
 natural keys 77, 207  
 order keys 207  
 parent keys 207  
 primary keys 78, 207  
 search keys 207  
 sort keys 207

summary of differences between keys and indexes 208  
 surrogate keys 78, 91, 207

**L**

**Life cycle**  
 database design life cycle 47  
**Limits**  
 database 927  
 session 938  
 system 921  
**LOB and XML subtables**  
 sizing a LOB or XML subtable 861  
**Lock Viewer Viewpoint portlet** 915  
**Logical data mart** 20  
**Logical database design** 123  
 activity transaction modeling as the first step in the transition from logical design to physical design 123  
 assigning domains during the logical design phase 132  
 design using the logical model instead of denormalization 180  
 formalizing relationships 53  
 normalization and logical database design 53  
**Logical database integrity** 629  
**LONG RAW**. See **VARBYTE**.  
**LONG VARBYTE** data type 829  
 alignment 829  
 size on aligned row format systems 829  
 size on packed64 systems 829  
**LONG VARCHAR** data type 131, 840  
**LONG**. See **LONG VARCHAR.\$nopage>VARCHAR2**. See **VARCHAR**.

**M**

**M:M relationships**  
 guidelines for drawing tables for users 70  
**Many-to-many relationships** 70  
**Master index** 246  
**Materialized views**. See **Join indexes**.  
**Maxima**  
 database maxima 927  
 session maxima 938  
 system maxima 921  
**MBB geospatial data type** 845  
**MBR geospatial data type** 845  
 Mixed inference rules for multivalued dependencies 83  
 Monitoring tactical queries 914  
**Multilevel partitioned primary indexes** 372  
 example from the insurance industry 378  
 example from the retail industry 380  
 example from the telecommunications industry 380  
 implicit table-level index CHECK constraint for an MLPI 374  
 MLPIs and the importance of partition order 382

- partition elimination with MLPIs 378
  - MultiLoad utility**
    - MultiLoad and block-level compression 695
    - MultiLoad and reserved query bands 695
    - MultiLoad and temperature level 695
    - restrictions with hash and join indexes 593, 608, 613, 614
    - restrictions with join indexes 461
  - Multitable aggregate join indexes 217
  - Multitable join indexes 194
  - Multitable simple join indexes 217
  - Multivalued dependencies
    - complement rule for multivalued dependencies 82
    - definition of multivalued dependencies 81
    - inference axioms for multivalued dependencies 81
- N**
- Named constraints 642
  - Naming columns**
    - conventions for naming columns 135
    - guidelines for naming columns 134
    - naming foreign key columns 136
  - n-ary relationship 67
  - Natural key**
    - definition of a natural key 77, 207
  - Nested joins**
    - tactical queries and nested joins 902
  - Net capacity usage**
    - definition of net capacity usage 727
    - detailed calculation of net capacity usage 730
    - how to calculate net capacity usage with no fallback 727
    - how to calculate row header space 728
    - how to calculate row vacated space 728
    - how to calculate table header space 728
    - how to calculate with fallback 728
  - NewHashBucketSize DBS Control parameter 229
  - Non-hash-partitioned row allocation to the AMPs for NoPI**
    - table rows 237, 238
  - Non-key attributes** 65
  - Non-prime tables** 66
  - Non-unique primary indexes** 192
  - Non-unique secondary indexes** 194, 214
    - importance of consecutive index numbers for value-ordered NUSIs 485
    - limitations on value-ordered NUSIs 485
    - NUSI bit mapping 479
    - query covering 482
    - typical uses of hash-ordered NUSIs 487
    - typical uses of value-ordered NUSIs 487
    - usage summary 497
    - value-ordered NUSIs and range conditions 484
  - NoPI tables 280
    - manipulating NoPI table rows 283
    - restrictions for hash indexes 606, 608, 611
  - rules and limitations for NoPI tables 282
  - uses for NoPI tables 281
  - USI access to NoPI tables 256
  - Normal forms**
    - 3NF 86
    - 5NF 993
    - BCNF 87
    - Boyce-Codd normal form 87
    - definition of third normal form 86
    - third normal form 86
  - Normalization** 76
    - advantages for physical database implementation 109
    - built-in Teradata architecture support for normalization 26, 73
    - conflicts with dependency preservation 105
    - cost of normalization 75
    - decomposing relations 83, 89
    - definition of normalization for the activity transaction modeling process 127
    - definition of normalization terms 75
    - derived columns and normalization 181
    - general procedure for achieving normalization 107
    - goals of normalization 89
    - horizontal decomposition 83
    - normalization and primary key values 137
    - normalization in the logical database design phase 53
    - objectives of normalization 83
    - preserving functional dependencies 103
    - procedure for normalizing a database schema 107
    - relation attribute 76
    - summary of advantages of normalization 109
    - vertical decomposition of relations 84
  - NPPI. See **Unpartitioned primary indexes**
  - Nulls** 684
    - alternatives to SQL nulls 679
    - bivalent and higher-valued logics and SQL nulls 676
    - collation sequence of SQL nulls 687
    - data types of null literals 687
    - excluding SQL nulls from a result set 688
    - how aggregate functions handle nulls 684
    - how ANSI DateTime data types work with nulls 684
    - how arithmetic functions handle SQL nulls 683
    - how arithmetic operators handle SQL nulls 683
    - how SQL aggregate functions handle nulls 684
    - how SQL arithmetic functions handle nulls 683
    - how SQL arithmetic operators handle nulls 683
    - how SQL CASE expressions handle nulls 685
    - how SQL comparison operators handle nulls 683
    - how SQL handles nulls for DateTime and Interval data 684
    - inconsistencies of SQL nulls 674
    - nulls as literals 686
    - presence bits for compressed nulls 780
    - searching for nulls and non-nulls in the same search condition 688

- searching for SQL nulls using a SELECT request 687
- semantics of nulls in outer join operations 689
- semantics of SQL nulls 673
- types of missing values represented by SQL nulls 673
- unique primary indexes and nulls 675
- NUMERIC** data type 824, 828
  - alignment 827
  - size on aligned row format systems 827
  - size on packed64 systems 827
- Numeric data types**
  - DECIMAL** 828
  - DOUBLE PRECISION** 827
  - FLOAT** 827
  - NUMERIC** 828
  - REAL** 827
- NUPIs** 192, 265, 409
  - assigning a NUPI to a table 192
  - definition of a NUPI 192
  - duplicate NUPI row read I/O and number of rows inserted 445
  - duplicate row checks and NUPIs 448
  - eliminating duplicate row checks using NUPIs 449
  - guidelines for maximum number of NUPI duplicates 446, 447
  - minimizing the number of duplicate row checks using NUPIs 449
  - multicolumn NUPIs 449
  - performance effects of duplicate row checks and NUPIs 444
  - relative advantages of multicolumn NUPIs versus single column NUPIs 449
- NUSI**
  - join indexes and NUSIs 516
  - performance and 473
- NUSI.** See also **Secondary index**
- NUSIs** 194, 214, 467, 864
  - access to rows using a NUSI 471
  - ANDed predicate set selectivity and NUSIs 477
  - column candidacy for covered access using a NUSI 484
  - complex conditional expressions with NUSIs 480
  - computing NUSI bit maps 481
  - determining if NUSI bit mapping is being used by the optimizer 481
  - high NUSI selectivity 474
  - importance of consecutive index numbers for value-ordered NUSIs 485
  - limitations on value-ordered NUSIs 485
  - multiple and composite NUSI access performance compared 476
  - multiple NUSI selectivity and composite NUSI access selectivity compared 476
  - NUSI bit mapping 479
  - NUSI subtable row layout 468
  - NUSI subtable row layout field definitions 470
  - NUSI subtable row layout for PPI base tables on 64-bit systems 469
  - NUSI subtable row layout for PPI base tables on packed64 systems 469
  - NUSI subtable row layout for unpartitioned primary index base tables on 64-bit systems 469
  - NUSI subtable row layout for unpartitioned primary index base tables on packed64 systems 468
  - NUSI subtable space requirements 865
  - NUSI usage summary 497, 498
  - NUSI** with low selectivity 474
  - NUSIs**, tactical queries, and non-localization of work 899
  - ORed** predicate set selectivity and NUSIs 478
  - performance advantages of NUSI bit mapping 479
  - query covering 482
  - restrictions on NUSI bit mapping 479
  - selectivity considerations for NUSIs 474
  - special NUSI subtable sizing considerations 864
  - support for queries using BETWEEN, LESS THAN, GREATER THAN, or LIKE operators 478
  - typical uses of hash-ordered NUSIs 487
  - typical uses of value-ordered NUSIs 487
  - value-ordered non-unique secondary indexes defined on a single column with the ALL option 194
  - value-ordered non-unique secondary indexes defined on all columns with the ALL option 194
  - value-ordered NUSIs and range conditions 484

## O

- Object identifier**, see **OID**
- Octet** versus **byte**
  - conceptual difference 781, 987
  - presence bits 781, 987
- OID** 820, 862
- OLTP** 36
  - how OLTP differs from data warehousing support 48
  - OLTP and indexes 196
- ON clause**
  - how the Optimizer uses the ON clause 210
- One-to-one relationships**
  - guidelines for placing the foreign key 68
- Optimal data access using secondary indexes** 488
- ORDER BY clause**
  - restrictions on the use of ORDER BY clauses in join index definitions 593
  - value ordering 602
- Order independence in relations** 77
- Order key**
  - definition of an order key 207
- Outer joins**
  - semantics of SQL nulls in outer join operations 689
  - used to define join indexes 558

**P**

## Parent key

- definition of a parent key 207
- forming referential constraints 98

## PARTITION column

- collecting statistics on for a join index 519
- Partition elimination 400
  - delayed partition elimination 402
  - dynamic partition elimination 402
  - multilevel partitioned primary indexes and partition elimination 378
  - performance gains realized from partition elimination 401
  - static partition elimination 401

## Partition number

- component of RowID for PPI tables 773

## PARTITION system-derived column 801

- collecting statistics on PARTITION columns 805
- restrictions on use in hash index definitions 612

## PARTITION[#Ln] system-derived columns

- restrictions on use in hash index definitions 612

## Partitioned primary index

- hash indexes 615
- join indexes 615

## Partitioned primary indexes 192, 256, 263

- evaluating benefits of a single-level partitioned primary index versus a nonpartitioned primary index 370

evaluation scenario 1 427

evaluation scenario 2 431

evaluation scenario 3 432

evaluation scenario 4 434

hash-based row allocation to the AMPs for PPI rows 242

join index support for range queries 506

NUSI subtable row layout for PPI base tables on 64-bit systems 469

NUSI subtable row layout for PPI base tables on packed64 systems 469

NUSI subtable row layout for unpartitioned primary index base tables on 64-bit systems 469

NUSI subtable row layout for unpartitioned primary index base tables on packed64 systems 468

USI subtable row layout for PPI base tables on 64-bit systems 465

USI subtable row layout for PPI base tables on packed64 systems 464

USI subtable row layout for unpartitioned primary index base tables for packed64 systems 464

USI subtable row layout for unpartitioned primary index base tables on 64-bit systems 464

## Partitioning columns

- volatility as a principal selection criterion for partitioning columns 420

## Partition-level backup and restore 280

## Payback for a join index

- definition of join index payback 522

## Percent compression

- definition of percent compression 727

definition of percent compression for multi-value

compression 722

relationship of the percent compression to the compression ratio 727

## Performance considerations

- optimizing NUPI performance 449

performance considerations for non-unique primary indexes 442

performance considerations for unique primary indexes 442

Period data types 131, 140, 141, 212, 214, 216, 219, 250, 254, 262, 456, 612, 635, 636, 644, 650, 655, 657, 838

PERIOD(DATE) 838

PERIOD(TIME(precision) WITH TIMEZONE) 839

PERIOD(TIME(precision)) 838

PERIOD(TIMESTAMP(precision) WITH TIMEZONE) 839, 840

PERIOD(TIMESTAMP(precision)) 839

PERIOD(DATE) data type 838

PERIOD(TIME(precision) WITH TIMEZONE) data type 839

PERIOD(TIME(precision)) data type 838

PERIOD(TIMESTAMP(precision) ) data type 839

PERIOD(TIMESTAMP(precision) WITH TIMEZONE) data type 839, 840

PERIOD(TIMESTAMP(precision)) data type 839

PERM space. See Permanent space

## Permanent journal recovery

restrictions with hash indexes 613

restrictions with join indexes 613

## Permanent space

allocating PERM space 880

allocating permanent space for system users 878

calculating the total PERM space requirements for user

SysAdmin 889

estimating the total amount of PERM space required by user SysAdmin 889

QCD tables are stored in permanent space 870

Permutation rule for inclusion dependencies 633

Persistent spool space 868

## Physical database design 55

limiting column values using constraints in ALTER TABLE requests 136

limiting column values using constraints in CREATE TABLE requests 136

## PPI

see Partitioned primary indexes

PPI. See Partitioned primary indexes

Prejoins 179

- Presence bits
  - number of presence bits required to represent compressed values 784
  - presence bit mappings for COMPRESS and NULL 782
  - purpose of presence bits 780
- Presence bytes
  - how to calculate presence bytes 728
- Presence octets
  - how to calculate presence octets 728
- Primary index columns
  - volatility as a principal selection criterion for primary index columns 420
- Primary indexes 211
  - access demographics as a principal selection criterion for primary index columns 417
  - analyzing demographics 422
  - assigning a NUPI as a primary index 192
  - assigning major entities to UPIs 265
  - assigning minor entities to NUPIs 265
  - comparison of partitioned and unpartitioned primary index access for some typical SQL operations 403
  - consumption of disk space by primary indexes 450
  - creating the primary index using CREATE TABLE 414
  - default definitions 263
  - definition of a non-unique primary index 192
  - definition of a unique primary index 191
  - definition of the partitioned-unpartitioned dimension 194
  - definition of the unique - non-unique dimension for primary indexes 193
  - distribution demographics and the maximum number of rows null 425
  - distribution demographics and the number of distinct values 422
  - distribution demographics and the number of rows per distinct value 423
  - distribution demographics as a principal selection criterion for primary index columns 416
  - hashing of primary indexes 221
  - how primary indexes are stored 212
  - implicit table-level index CHECK constraint for multilevel partitioned primary indexes 374
  - importance of the partition order for multilevel partitioned primary indexes 382
  - locating rows using the primary index 437
  - mandatory use of primary indexes 212
  - maximum number of columns 262
  - maximum number per table 212
  - multilevel partitioned primary index example from the insurance industry 378
  - multilevel partitioned primary index example from the retail industry 380
  - multilevel partitioned primary index example from the telecommunications industry 380
  - multilevel partitioned primary indexes 372
  - multilevel partitioned primary indexes and partition elimination 378
  - optimal data access using a primary index 408
  - optimizing the performance of NUPIs 449
  - partitioned primary indexes 194, 255, 256, 263, 267
  - performance considerations for NUPIs 442
  - performance considerations for UPIs 442
  - primary indexes and uniform data distribution 408
  - primary indexes compared with secondary and join indexes 252
  - purpose of primary indexes 211, 262
  - relative advantages and disadvantages of NUPIs 449
  - restrictions for primary indexes 212
  - retrieving row using the primary index 437
  - selecting the primary index for a join index 594
  - selecting the primary index for queue tables 421
  - unique primary indexes 191
    - correcting uneven row distribution 245
  - unique primary indexes and nulls 675
  - unpartitioned primary indexes 194, 256, 263, 267
  - using HASHROW to evaluate a UPI choice 245
  - using the HASHAMP function to evaluate a UPI choice 245
  - using the HASHROW function to evaluate a UPI choice 244, 245
  - non-unique. See also NUPIs
- Primary key
  - assigning a USI to a primary key 489
  - defining bounds for primary keys 92
  - definition for the activity transaction modeling process 126
  - definition of a primary key 78, 207
  - domain rules and referential integrity 101
  - guidelines for selecting the primary key for a table 92
  - identifying the primary key for a table 91
  - primary key attribute 64
  - primary key values and normalization 137
  - primary keys and referential integrity 101
- PRIMARY KEY constraints 654
- Prime tables 65
- Principle of Interchangeability 663
- Principle of Orthogonal Design 664
- Principle of the Identity of Indiscernibles 664
- Principles of normalization 664
- Priority Scheduler utility 915
- PRODUCT relational operator 80
- PROJECT relational operator 80
- Projection rule for inclusion dependencies 633
- Pseudotransitivity rule
  - pseudotransitivity rule for functional dependencies 81
  - pseudotransitivity rule for multivalued dependencies 82

**Q**

## QCD

- capacity planning for a query capture database 870
- QCD sizing equations
- query capture database tables 870
- sizing a query capture database 870
- where QCD tables are stored 870

## QCF

- capturing query plan information using the Query Capture Facility 202
- QCF and Query Capture Databases (QCD) 202

## Queries

- ad hoc queries and data warehousing 44
- tactical queries 45
- using hash indexes to support specific queries and query workloads 605
- using join indexes to support specific queries and query workloads 503

Query capture database. See QCD

## Query complexity

- relationship of query complexity with the value of the result set it returns 42

## Query covering

- definition of partial NUSI query definition 483
- NUSI query covering 482

## Query ratio for a join index

- computing the query ratio for a join index 535
- definition of the query ratio for a join index 535

## Query\_Database rules 943

## Queue tables

- criteria for selecting a primary index for queue tables 421

**R**

## Ratios

- entity relationship ratios 64

## Read from fallback 672

## REAL data type 131, 824, 827

- alignment 827
- size on aligned row format systems 827
- size on packed64 systems 827

## Record Mode operation 888

## Reference Indexes 219, 653, 654, 867

- sizing equation for Reference Indexes 867

## REFERENCES constraints

- rules for column-level REFERENCES constraints 651

## Referential constraints

- checks on specifying referential constraints 653
- criteria for specifying referential constraints 100
- forming a referential constraint using a foreign key and a parent key 98

## Referential integrity

- batch referential integrity 98, 99
- data integrity and referential integrity 98

definition of referential integrity 95

difficulties enforcing referential integrity with physical denormalization 115

enforcing referential integrity 96

importance of referential integrity 97

maintaining referential integrity using foreign keys 95

referencing (child) tables and referential integrity 97

referencing (parent) tables and referential integrity 97

referential integrity constraints 138

Referential Integrity Rule 95, 96, 98, 663

Reflexive rule for functional dependencies 81

Reflexive rule for inclusion dependencies 633

Reflexive rule for multivalued dependencies 82

Relational algebra

INTERSECTION relational operator 80

Relational algebra operators

DIFFERENCE 79

DIVIDE 80

JOIN 80

PRODUCT 80

PROJECT 80

RESTRICT 80

SELECT 80

UNION 80

Relational schema

definition of a relational schema 78

Relations

decomposing relations 83, 89

definition of a relation 78

DIFFERENCE relational operator 79

DIVIDE relational operator 80

INTERSECTION relational operator 80

JOIN relational operator 80

logical operations on relations 79

order independence property of relations 77

PRODUCT relational operator 80

PROJECT relational operator 80

relation values versus relations 627

relation variable versus relations 627

RESTRICT relational operator 80

SELECT relational operator 80

UNION relational operator 80

uniqueness property of relations 79

Relationships

cardinality of a relationship 67

definition of a binary relationship 67

definition of a relationship 62, 64

definition of a tertiary relationship 67

definition of a unary relationship 67

definition of an n-ary relationship 67

definition of connectivity 67

definition of degree 67

existence dependencies 67

identifying relationships using primary keys 137

- modeling 1:1 relationships 68
- modeling 1:M relationships 69
- modeling M:M relationships 70
- placing the foreign key in a 1:1 relationship 68
- recursive relationships 111
- relationship ratios 64
- terms associated with relationships 67
- Relvar
  - definition of a relvar 627
- Repeating group
  - definition of a repeating group 78
- Report/Query Analysis form 146
  - access column 147
  - activity transaction modeling form 146
  - completing the report/query analysis form 147
  - join column 147
  - purpose of the report/query analysis form 146
  - terminology used for the report/query analysis form 147
- Requirements analysis
  - compiling research information for a requirements analysis 53
  - developing an enterprise data model for requirements analysis 58
  - gathering input screens for a requirements analysis 53
  - interviewing notable employees for a requirements analysis 53
  - writing a requirements specification 53
- Requirements specification
  - writing a requirements specification 53
- RESTRICT relational operator 80
- ResUsage tables 914
- Row distribution
  - uneven row distribution 407, 416
- Row fields for base tables
  - column offsets 773
  - fixed length columns 773
  - flag byte 773
  - PPI RowID 772
  - presence bits 773
  - row length 772
  - row reference array 772
  - unpartitioned primary index RowID 772
  - VARCHAR columns 774
- Row hash
  - how row hash values are stored 197, 198
  - row hash value is a component of the RowID 772
  - RowID and rowhash values 197, 198
- Row header
  - number of presence bits required to represent compressed values 784
  - presence bits 780
  - presence bits and nullability 781
  - presence bits octet 781, 987
- Row header space
  - calculating row header space for net capacity usage 728
- Row Size Calculation form 170
- Row structure
  - aligned row format 740
  - byte alignment 770
  - packed64 row format 740
  - row components for aligned row format systems 771
  - row components for packed64 systems 771
  - row layout for a compressed join index with an
    - unpartitioned primary index on packed64 systems 776
  - row layout for compressed hash indexes with an
    - unpartitioned primary index on packed64 systems 776
  - row layout for uncompressed hash indexes with an
    - unpartitioned primary index on packed64 systems 775
  - row layout for uncompressed join indexes with a PPI on packed64 systems 775
  - row layout for uncompressed join indexes with an
    - unpartitioned primary index on packed64 systems 775
  - row structure for algorithmically compressing variable length columns 743
  - row structure for aligned row format systems 740
  - row structure for compressed hash indexes with an
    - unpartitioned primary index on aligned row format systems 779
  - row structure for compressed join indexes with an
    - unpartitioned primary index on aligned row format systems 779
  - row structure for compressing variable length columns 742
  - row structure for hash indexes on aligned row format systems 777
  - row structure for hash indexes on packed64 systems 775
  - row structure for join indexes on aligned row format systems 777
  - row structure for join indexes on packed64 systems 775
  - row structure for multi-value compressing and algorithmically compressing variable length columns 750
  - row structure for multi-value compressing variable length columns 747
  - row structure for packed64 systems 740
  - row structure for uncompressed join indexes with an
    - unpartitioned primary index on aligned row format systems 778
  - row structure of hash indexes on aligned row format systems 777
  - row structure of join indexes on aligned row format systems 777
  - row structure of join indexes on packed64 systems 775

Row vacated space  
   calculating the row vacated space for net capacity usage 728  
 RowID  
   value storage 197, 198  
 ROWID system-derived column 800  
 Row-level security  
   algorithmic compression and row-level security 696  
   autocompression and row-level security 710  
   block-level compression and row-level security 704  
   multi-value compression and row-level security 696, 700, 713  
   row compression and row-level security 709  
 Rows  
   compressing hash index rows 599  
   compressing join index rows 599  
   defining the row layout for NUSI subtables 470  
   defining the row layout for USI subtables 465  
   hash index row structure on aligned row format systems 777  
   join index row structure on aligned row format systems 777  
   length of unpartitioned primary index rows versus the length of partitioned primary index rows 771  
   locating using the primary index 437  
   maximum null 425  
   retrieving rows using the primary index 437  
   row size calculations 846  
   row size calculations for aligned row format systems 850  
   row size calculations for packed64 systems 846  
   row structure for aligned row format systems 755  
   row structure for compressed hash indexes with an unpartitioned primary index on packed64 architectures 776  
   row structure for hash indexes on packed64 systems 775  
   row structure for join indexes on packed64 systems 775  
   row structure for packed64 systems 753  
   row structure for partitioned primary index tables on aligned row format systems 756  
   row structure for partitioned primary index tables on packed64 architectures 754, 775  
   row structure for uncompressed hash indexes with an unpartitioned primary index on aligned row format systems 778  
   row structure for uncompressed join indexes with a PPI on aligned row format systems 778, 779  
   row structure for uncompressed join indexes with an unpartitioned primary index on aligned row format systems 778  
   row structure for unpartitioned primary index tables on aligned row format systems 756  
   row structure for unpartitioned primary index tables on packed64 architectures 775  
   row structure for unpartitioned primary index tables on packed64 systems 754

sizing of rows 846  
 sizing of rows on a packed64 system 846  
 sizing of rows on an aligned row format system 850  
 structure for a compressed join index with an unpartitioned primary index on packed64 architectures 776

## S

Search key  
   definition of a search key 207  
 Secondary indexes 213  
   accessing a row using a USI 458  
   accessing rows using a NUSI 471  
   aggregate NUSI operations 483  
   assigning USIs to primary keys 489  
   column candidacy for covered access using a NUSI 484  
   complex conditional expressions with NUSIs 480  
   computing NUSI bit maps 481  
   defining a non-unique secondary index 214  
   defining a unique secondary index 214  
   determining NUSI bit mapping using the EXPLAIN request modifier 481  
   differences in processing USIs and NUSIs 494  
   dual and composite NUSI access performance compared 476  
   hashing a unique secondary index 462  
   hashing a USI 462  
   hash-ordered non-unique secondary indexes defined on a single column with no ALL option 194  
   hash-ordered non-unique secondary indexes defined on a single column with the ALL option 194  
   hash-ordered NUSI defined on a single column with no ALL option 194  
   hash-ordered NUSI defined on a single column with the ALL option 194  
   high NUSI selectivity 474  
   high selectivity 488  
   how secondary indexes are stored 214  
   importance of consecutive index numbers for value-ordered NUSIs 485  
   index access and secondary indexes 204  
   limitations on value-ordered NUSIs 485  
   mandatory use of secondary indexes 213  
   maximum number per table 214  
   non-unique. See also NUSIs  
   non-unique secondary indexes 194, 467  
   NUSI bit mapping 479  
   NUSI ORed predicate set selectivity 478  
   NUSI query covering 482  
   NUSI subtable row layout 468  
   NUSI subtable row layout field definitions 470  
   NUSI subtable row layout for PPI base tables on 64-bit systems 469

NUSI subtable row layout for PPI base tables on packed64 systems 469  
 NUSI subtable row layout for unpartitioned primary index base tables on 64-bit systems 469  
 NUSI subtable row layout for unpartitioned primary index base tables on packed64 systems 468  
 NUSI subtable sizing equation 864  
 NUSI subtable space requirements 865  
 NUSI support for queries using BETWEEN, LESS THAN, GREATER THAN, or LIKE operators 478  
 NUSI usage summary 498  
 NUSI with low selectivity 474  
 NUSIs 194, 467  
 NUSIs ANDed predicate set selectivity 477  
 optimal data access using a secondary index 408  
 optimal data access using secondary indexes 488  
 performance advantages of NUSI bit mapping 479  
 purpose of NUSIs 192  
 purpose of secondary indexes 213  
 purpose of USIs 191  
 restrictions for NUSI bit mapping 479  
 restrictions on secondary indexes 214  
 secondary index access 490  
 secondary index usage summary 497  
 secondary indexes compared with primary and join indexes 252  
 selecting a secondary index for a base table 488  
 selecting a secondary index for a join index 594  
 selection criteria for secondary indexes 488  
 selectivity considerations for NUSIs 474  
 space considerations for secondary index subtables 457  
 special NUSI subtable sizing considerations 864  
 types of secondary indexes 214  
 typical uses of hash-ordered NUSIs 487  
 typical uses of value-ordered NUSIs 487  
 unique secondary indexes 194, 457  
 usage summary 497  
 USI 194  
 USI row layout field definitions 465  
 USI subtable row layout for PPI base tables on 64-bit systems 465  
 USI subtable row layout for PPI base tables on packed64 systems 464  
 USI subtable row layout for unpartitioned primary index base tables for packed64 systems 464  
 USI subtable row layout for unpartitioned primary index base tables on 64-bit systems 464  
 USI subtable sizing equation 863  
 USI subtable space requirements 863  
 USI usage summary 498  
 using a non-unique secondary index 192  
 using a NUSI 192  
 using unique secondary indexes to enforce row uniqueness 457  
 value-ordered non-unique secondary indexes and range conditions 484  
 value-ordered non-unique secondary indexes defined on a single column with the ALL option 194  
 value-ordered non-unique secondary indexes defined on all columns with the ALL option 194  
 value-ordered NUSI defined on a single column with no ALL option 194  
 value-ordered NUSI defined on a single column with the ALL option 194  
 SELECT relational operator 80  
 Selectivity  
     definition of selectivity 197  
     high NUSI selectivity 474  
     high selectivity index 197  
     low NUSI selectivity 474  
     low selectivity index 197  
     multiple NUSI selectivity and composite NUSI access selectivity compared 476  
     selectivity considerations for NUSIs 474  
     weak index selectivity 488  
 SET QUERY\_BAND  
     BlockCompression 695  
     TVSTemperature 695  
 Shared nothing architecture 27  
 Single-table aggregate join indexes 217  
 Single-table join indexes 194, 546  
     guidelines for collecting statistics on single-table join index columns 597  
     maintenance costs for single-table join indexes 548  
     single-table join index support for parameterized queries 548  
     tactical queries and single-table join indexes 564  
 Single-table simple join indexes 216, 540  
 Sizing equations  
     sizing equation for a NUSI subtable 864  
     sizing equation for a QCD 870  
     sizing equation for hash indexes 859  
     sizing equation for join indexes 859  
     sizing equation for LOB or XML subtables 861  
     sizing equation for LOB subtables 862  
     sizing equation for Reference Indexes 867  
     sizing equation for spool space 868  
     sizing equation for USI subtables 863  
     sizing equations for base tables 860  
 Skew 407, 416  
     uneven row distribution 407, 423, 425  
 SMALLINT data type 131, 824, 826  
     alignment 826  
     size on aligned row format systems 826  
     size on packed64 systems 826  
 Snowflake schema  
     dimensional modeling 189  
 Soft referential integrity. See Referential constraints

Sort key  
 definition of a sort key 207

Space requirements  
 estimating space requirements for spool space 886  
 estimating space requirements for the data dictionary 886  
 estimating spool space requirements 880  
 estimating temporary space requirements 881  
 estimating WAL log space requirements 881  
 hash indexes 859  
 space requirements for base tables 859  
 space requirements for fallback tables and index subtables 866  
 space requirements for join indexes 859  
 space requirements for non-fallback tables and index subtables 865  
 space requirements for NUSI subtables 865  
 space requirements for the CRASHDUMPS user 886  
 space requirements for user-defined routines 866  
 temporary space requirements 886  
 user spool space requirements 887

Sparse join indexes 505, 556

Specifications  
 database limits 927  
 database maxima 927  
 session limits 938  
 session maxima 938  
 system limits 921  
 system maxima 921

Spool file values  
 compressing spool file values 792  
 compression of spool file values 792

Spool files  
 compression of spool file values 792  
 reuse of spool files 34  
 sizing spool files 868

Spool space  
 allocating spool space 869, 880  
 estimating spool space requirements 880  
 guidelines for allocating spool space 869  
 intermediate spool space 868  
 limits of spool space 869  
 persistent spool space 868  
 rules for spool space equations 887  
 sizing spool space 868  
 sources of spool space 869  
 spool space output 868  
 spool space requirements 887  
 tables area 887  
 volatile spool space 869

SQL and indexes 209

SQL and performance  
 identity column 819  
 index access 204  
 join indexes 509

join indexes and NUSI, compared 516  
 NUSIs and performance 473  
 partition-level backup and restore 280  
 USI maintenance performance 467  
 USI rollback performance 467

SQL nulls  
 alternatives to SQL nulls 679  
 collation sequence of SQL nulls 687  
 data types of null literals 687  
 excluding SQL nulls from a result set 688  
 how arithmetic functions handle SQL nulls 683  
 how arithmetic operators handle SQL nulls 683  
 how SQL aggregate functions handle SQL nulls 684  
 how SQL CASE expressions handle nulls 685  
 how SQL comparison operators handle SQL nulls 683  
 how SQL handles nulls for DateTime and Interval data 684  
 inconsistencies of SQL nulls 674  
 searching for SQL nulls using a SELECT request 687  
 semantics of nulls in outer join operations 689  
 semantics of SQL nulls 673  
 SQL nulls and bivalent and higher-valued logics 676  
 SQL nulls as literals 686  
 types of missing values represented by SQL nulls 673

SQL statements  
 CREATE JOIN INDEX 695  
 INSERT 695  
 INSERT ... SELECT 695  
 SET QUERY\_BAND and BlockCompression 695  
 SET QUERY\_BAND and TVSTemperature 695  
 ST\_GeOMETRY geospatial data type 845

Standalone derived data  
 denormalizing standalone derived data 182  
 handling guidelines for standalone derived data 182

Standards Planning And Requirements Committee. See  
 ANSI/SPARC architecture

Star schema  
 dimensional modeling 188

Static partition elimination 401

Structured UDT data type  
 alignment 843  
 size on aligned row format systems 843  
 size on packed64 systems 843

Structured UDTs  
 storage structure for nested structured UDTs 793  
 storage structure for structured UDTs 792  
 storage structured for nested structured UDTs 793

Summary data  
 summary data has a low information content 41  
 why you should not store only summary data in the database 40

Super key  
 definition 78

Surrogate key  
 definition of a surrogate key 78, 207

- how surrogate keys are used 91
  - Synchronization of parallel operations
    - spool file reuse 34
    - synchronized BYNET operations 34
    - synchronized table scans 34
  - Synchronized BYNET operations 34
  - Synchronized table scans 34
  - SYSADMIN system user 878, 879
    - allocating permanent storage space 879
  - System disks
    - contents of system disks 875
    - disk 0 boot disk 875
    - disk 1 for dumps and memory swapping 875
    - types of system disks 875
  - System form 144
    - activity transaction modeling system form 144
    - information recorded on the system form 144
    - purpose of the system form 144
  - System users
    - DBC 878
    - SYSADMIN 879
    - SYSTEMFE 879
  - System-derived columns 800
    - collecting statistics on PARTITION columns 805
    - identity columns 818
    - PARTITION column 801
    - ROWID column 800
  - SYSTEMFE system user 878
    - allocating permanent storage space to SYSTEMFE 879
- T**
- Table capacity
    - calculating the uncompressed table capacity 725
  - Table form 152
    - access information 158
    - basic information 155
    - column-level information 156
    - completing access information for the table form 158
    - completing column-level information for the table form 156
    - completing data demographics for multicolumn objects 166
    - completing data demographics information for single-column objects 158
    - completing miscellaneous column-level information for the table form 157, 171
    - completing the table form 152
    - data demographics for multicolumn objects 165
    - data demographics for single-column objects 158
    - data demographics information for the table form 154
    - information on the table form 153
    - miscellaneous column-level information 157
    - miscellaneous information on the table form 153
  - Tables
    - base tables 500, 503
    - column-partitioned tables 280, 281, 282
    - compressibility of table columns 781
    - determining table sizes empirically 872
    - determining the available user table data space 883
    - determining user overhead 883
    - evaluating sizing of legacy tables 873
    - fallback space considerations for tables and index subtables 866
    - full-table scan table access 258
    - join index access 257
    - maximum number of join indexes per base table 572
    - naming associative tables 66
    - non-fallback space considerations 865
    - NoPI tables 280, 281, 282, 283
    - nullability of table columns 781
    - NUPI access to tables 256
    - NUSI access to tables 257
    - QCD 870
    - sizing base tables 859
    - sizing equations for base tables 860
    - summary of principal points for table sizing 873
    - summary of table access methods 255
    - UPI access to tables 256
    - USI access to tables 256
  - Tables area
    - CRASHDUMPS user 886
    - user spool space 887
    - user temporary space requirements 886
  - Tactical queries 45
    - aggregate join index support for tactical queries 565
    - canary queries and tactical queries 915
    - definition of tactical queries 893
    - effect of data volume growth on response times of tactical queries 895
    - effect of growth in number of concurrent users on response times of tactical queries 895
    - effects of configuration expansion on response times of tactical queries 895
    - few-AMP joins and tactical queries 901

- global join indexes and tactical queries
- Group AMP operations and tactical queries 900, 901
- monitoring tactical queries 914
- NUSI access and non-localization of work for tactical queries 899
- priority scheduler utility and tactical queries 915
- ResUsage tables and tactical queries 914
- scalability considerations for tactical queries 894
- single-table join index support for tactical queries 564
- USI access and work localization 898
- Target Level Emulation 203
  - capturing environmental cost data using Target Level Emulation 203
- TEMP space. See Temporary space
- Temporary space
  - allocating TEMP space 880
  - estimating TEMP space requirements 881
  - TEMP space requirements 886
- Teradata Database
  - database size considerations 875
  - session specifications 938
  - specifications 927
  - system specifications 921
- Teradata Index Wizard utility 201
  - column change ratings 160
  - using the Teradata Index Wizard utility to analyze and suggest secondary indexes for tables 488
- Teradata parallel architecture
  - BYNET support of the Teradata parallel architecture 29
  - internodal communication in an MPP system 29
  - support for full normalization 73
  - synchronization of parallel operations 34
  - Teradata parallel architecture support for data placement support 26
  - Teradata parallel architecture support for full normalization of the physical database 25
  - Teradata parallel architecture support for normalization 26
  - Teradata parallel architecture support for request parallelism 32
- Teradata Parallel Data Pump utility
  - Teradata Parallel Data Pump is not restricted for use with tables on which a hash or join index is defined 614
- Teradata Parallel Transporter utility
  - restrictions for hash indexes 613
  - restrictions for join indexes 613
- Teradata philosophy of database design 25, 73
- Tertiary relationship 67
- Third normal form
  - definition of third normal form 86
- TIME data type 131, 830
  - alignment 831
  - size on aligned row format systems 831
  - size on packed64 systems 831
- TIME WITH TIME ZONE data type 830
  - alignment 831
  - size on aligned row format systems 831
  - size on packed64 systems 831
- TIMESTAMP data type 131, 830
  - alignment 831
  - size on aligned row format systems 831
  - size on packed64 systems 831
- TIMESTAMP WITH TIME ZONE data type 830
  - alignment 831
  - size on aligned row format systems 831
  - size on packed64 systems 831
- Transitive dependencies
  - definition of transitive dependencies 81
- Transitivity rule for functional dependencies 81
- Transitivity rule for inclusion dependencies 633
- Transitivity rule for multivalued dependencies 82
- Triggers
  - restrictions for triggers on tables with hash indexes 613
  - restrictions for triggers on tables with join indexes 613
  - restrictions on tables with hash indexes 613
  - restrictions on tables with join indexes 613
  - triggers and business rules 142
- Tuple
  - definition of a tuple 78
- TVSTemperature reserved query band 694

## U

- UDTs
  - storage structure for nested structured UDTs 793
  - storage structure for structured UDTs 792
  - UDTs as domains 124, 129, 130, 132, 134, 138, 139
- Unary relationship 67
- UNION relational operator 80
- Union rule
  - Union rule for functional dependencies 81
  - Union rule for multivalued dependencies 82
- UNIQUE constraints 656
  - UNIQUE constraints and NOT NULL 656
- Unique Name Assumption 630
  - definition of the Unique Name Assumption 630
- Unique primary indexes 191, 192
  - unique primary indexes and nulls 675
- Unique Secondary Index. See USIs
- Unique secondary indexes 191, 194, 214, 457
  - usage summary 497
  - USI subtable row layout for packed64 systems 463
  - using unique secondary indexes to enforce row uniqueness 457
- Uniqueness
  - description of uniqueness 79
  - determining primary index disk usage 450
  - uniqueness property of relations 79

- Uniqueness value
  - assigning values to 198
  - forming RowID 197, 772
  - hashing algorithm 227, 772
- Unpartitioned primary indexes 194
- UPDATE STATISTICS. See COLLECT STATISTICS (Optimizer Form).
- UPIs 100, 191, 192, 265, 409
- Upsert\_Database
  - rules 944
- User DBC
  - allocating permanent storage space for user DBC 878
  - allocating permanent storage space for user SysAdmin 889
  - permanent space usage for user DBC 879
  - spool space usage for user DBC 879
  - temporary space usage for user DBC 879
- User spool space
  - user spool space requirements 887
- User temporary space
  - requirements for user TEMP space 886
- User-defined data types
  - UDTs and domains 129, 130, 131, 132, 134, 136
- User-defined routines
  - sizing user-defined routines 866
- Users. See System users
- USI maintenance performance 467
- USI rollback performance 467
- USI subtable sizing equations
  - USIs 863
- USIs 100, 191, 194, 214, 457, 863
  - accessing a row using a USI 458
  - defining a USI 214
  - hashing a USI 462
  - row layout field definitions for USI subtables 465
  - tactical queries and work localization 898
  - USI subtable row layout for packed64 systems 463
  - USI usage summary 497, 498
  - using USIs to enforce row uniqueness 457
- V**
- Value of a result set
  - relationship of the value of a result set to query complexity 42
- VARBYTE data type 130, 131, 829
  - alignment 829
  - size on aligned row format systems 829
  - size on packed64 systems 829
- VARCHAR data type 131, 840
- VARCHAR(n) CHARACTER SET GRAPHIC data type
  - alignment 842
  - size on aligned row format systems 842
  - size on packed64 systems 842
- VARCHAR(n) CHARACTER SET LATIN data type
  - alignment 841
  - size on aligned row format systems 841
  - size on packed64 systems 841
- VARCHAR(n) CHARACTER SET UNICODE data type
  - alignment 842
  - size on aligned row format systems 842
  - size on packed64 systems 842
- Views
  - always use views to access data 121
  - denormalized views 112, 115
  - denormalized views provide the illusion of a denormalized physical database schema 185
  - dimensional views 186
  - how to use views to avoid denormalizing the physical database schema using prejoins 180
  - using global temporary tables instead 183
  - using global temporary tables instead of views 183
  - using prejoins to denormalize the physical database schema 179
  - using views for flexible data access 121
  - using views to denormalize the physical database schema 115
  - using views to isolate users 941
  - using views to provide the illusion of denormalizing the physical database schema 185
- Volatile spool space 869
- W**
- WAL log
  - calculating the data block size for the WAL log 882
  - estimating WAL log space requirements 881
- WHERE clause
  - how the Optimizer uses the WHERE clause 210
  - invoking full table scans in WHERE clause predicates 210
  - invoking indexes in WHERE clause predicates 210
  - purpose of the WHERE clause in SQL DML 210
  - restrictions on the use of WHERE clause predicates in sparse join index definitions 592
- Write Ahead Logging 770
  - portion of data space required for WAL overhead 883

