Tera-Tom on Teradata Physical Implementation

Understanding is the key!

Second Edition

Published by

Coffing Publishing

Second Edition June, 2005

Written by W. Coffing
Web Page: www.Tera-Tom.com and www.CoffingDW.com
E-Mail address:
Tom.Coffing@CoffingDW.Com

Teradata®, NCR™, V2R4, V2R5 and BYNET® are registered trademarks of NCR Corporation, Dayton, Ohio, U.S.A., IBM® and DB2® are registered trademarks of IBM Corporation, ANSI® is a registered trademark of the American National Standards Institute. In addition to these products names, all brands and product names in this document are registered names or trademarks of their respective holders.

Coffing Data Warehousing shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of programs or program segments that are included. The manual is not a publication of NCR Corporation, nor was it produced in conjunction with NCR Corporation.

Coffing Publishing
7810 Kiester Rd.
Middletown, OH 45042

International Standard Book Number*: ISBN 0-9704980-5-5*

Printed in the United States of America

All terms mentioned in this book that are known to be trademarks or service have been stated. ***Coffing Publishing*** cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

About Coffing Data Warehousing's CEO Tom Coffing

Tom is President, CEO, and Founder of Coffing Data Warehousing. He is an internationally known consultant, facilitator, speaker, trainer, and executive coach with an extensive background in data warehousing. Tom has helped implement data warehousing in over 40 major data warehouse accounts, spoken in over 20 countries, and has provided consulting and Teradata training to over 8,000 individuals involved in data warehousing globally.

Tom has co-authored over 20 books on Teradata and Data Warehousing. To name a few:

- *Secrets of the Best Data Warehouses in the World*
- *Teradata SQL - Unleash the Power*
- *Tera-Tom on Teradata Basics*
- *Tera-Tom on Teradata E-business*
- *Teradata SQL Quick Reference Guide - Simplicity by Design*
- *Teradata Database Design - Giving Detailed Data Flight*
- *Teradata Users Guide -The Ultimate Companion*
- *Teradata Utilities - Breaking the Barriers*

Mr. Coffing has also published over 20 data warehousing articles and has been a contributing columnist to DM Review on the subject of data warehousing. He wrote a monthly column for DM Review entitled, "Teradata Territory". He is a nationally known speaker and gives frequent seminars on Data Warehousing. He is also known as "The Speech Doctor" because of his presentation skills and sales seminars.

Tom Coffing has taken his expert speaking and data warehouse knowledge and revolutionized the way technical training and consultant services are delivered. He founded CoffingDW with the same philosophy more than a decade ago. Centered around 10 Teradata Certified Masters this dynamic and growing company teaches every Teradata class, provides world class Teradata consultants, offers a suite of software products to enhance Teradata data warehouses, and has eight books published on Teradata.

Tom has a bachelor's degree in Speech Communications and over 25 years of business and technical computer experience. Tom is considered by many to be the best technical and business speaker in the United States. He has trained and consulted at so many Teradata sites that students affectionately call him Tera-Tom.

Teradata Certified Master

- Teradata Certified Professional
- Teradata Certified Administrator
- Teradata Certified Developer
- Teradata Certified Designer
- Teradata Certified SQL Specialist
- Teradata Certified Implementation Specialist

Table of Contents

Chapter 1 — The Rules of Data Warehousing 1
Teradata Certification 2
A Logical View of the Teradata Architecture 1
The Parsing Engine (PE) 2
The Access Module Processors (AMPs) 3
The BYNET 4
A Visual for Data Layout 5
Teradata Cabinets, Nodes, Vprocs, and Disks 6

Chapter 2 — Data Distribution Explained 7
Rows and Columns 8
The Primary Index 9
The Two Types of Primary Indexes 10
Unique Primary Index (UPI) 11
Turning the Primary Index Value into the Row Hash 13
The Row Hash Value determines the Row's Destination 14
The Row is Delivered to the Proper AMP 15
The AMP will add a Uniqueness Value 16
An Example of an UPI Table 17
An Example of a NUPI Table 18
How Teradata Retrieves Rows 19
Row Distribution 20
A Visual for Data Layout 21
Teradata accesses data in three ways 22
Data Layout Summary 23

Chapter 3 — V2R5 Partition Primary Indexes 25
V2R4 Example 26
V2R5 Partitioning 27
Partitioning doesn't have to be part of the Primary Index 28
Partition Elimination can avoid Full Table Scans 29
The Bad NEWS about Partitioning on a column that is not part of the Primary Index 30
Two ways to handle Partitioning on a column that is not part of the Primary Index 31
Partitioning with CASE_N 32
Partitioning with RANGE_N 33
NO CASE, NO RANGE, or UNKNOWN 34
Partitioning and Joins 35

Chapter 4 — Teradata Engine – Under the Hood 37
Full Cylinder Read 38
Table Header 39
Each Table is given a Table ID 40
How Data Blocks are Dynamically Built 41
Data Blocks 42
How Teradata Finds a Row of Data 43
The Master Index 44

The Cylinder Index 45
Cylinder Index Changes........ 46
How Teradata Writes to an AMP........ 47
Writing to Data Blocks of Equal Length 48
When a Data Block is not Big Enough for a Write........ 49
How Teradata Allocates Blocks........ 50
Block and Row Definitions........ 51
Large Row versus Oversized Row........ 52
Defragmentation 53
When a Cylinder becomes Full........ 54
A Node and its Memory Allocations 55

Chapter 5 — The Extended Logical Data Model 57
The Application Development Life Cycle........ 58
Asking the Right Questions 59
Logical Data Model 60
Primary Keys 61
Foreign Keys........ 62
Normalization 63
Extended Logical Data Model 64
The End Goal of the ELDM is to build Table Templates........ 65
Column ACCESS in the WHERE Clause 66
Data Demographics........ 67
Distinct Values........ 68
Typical Rows Per Value 70
Maximum Rows NULL 71
Change Rating........ 72
Extended Logical Data Model Template 73

Chapter 6 — The Physical Data Model........ 75
Step 1 – Look at Distribution........ 76
Step 2 – Eliminate based on Change Rating........ 79
Step 3 – NUSI Elimination via Value Access Frequency........ 81
Step 4 – Pick the Primary Index 83
Primary Index Factors........ 84
Why Join Access Frequency is Top Priority?........ 85
Why Value Access Frequency is Second Priority?........ 88
What have we learned about picking the Primary Index? 89
Step 5 — Pick Secondary Indexes........ 91
USI to eliminate Duplicate Row Checking........ 92
NUSI considerations........ 93
Multi-Column NUSI Columns used as a Covered Query........ 94
Value-Ordered NUSIs........ 95
A Strongly Vs Weakly Selective NUSI 95
A formula for calculating a strongly selective NUSI........ 96
Typical Row Size........ 97

Typical Block Size 98

Chapter 7 — Denormalization 101
Derived Data 102
Storing Aggregates 103
Pre–Joining Tables 104
Repeating Groups 105
Horizontal Partitioning 106
Vertical Partitioning 107
Covered Query 108
Single-Table Join Indexes 109
Multi-Table Join Indexes 110
Temporary Tables 111
Derived Tables 112
Volatile Temporary Tables 113
Global Temporary Tables 114

Chapter 8 — Secondary Indexes 115
Unique Secondary Index (USI) 117
USI Subtable Example 118
How Teradata retrieves an USI query 119
NUSI Subtable Example 120
How Teradata retrieves a NUSI query 121
Value-Ordered NUSI 122
How Teradata retrieves a Value-Ordered NUSI query 123
NUSI Bitmapping 124
Prototyping indexes with EXPLAIN 125
Secondary Index Summary 127

Chapter 9 — Join Strategies 129
A Join in Simple Terms 131
The key things to know about Teradata and Joins 132
Merge Join Strategies 133
Joins need the joined rows to be on the same AMP 134
Another Great Join Picture 135
Joining Tables with matching rows on different AMPs 135
Joining Tables with matching rows on different AMPs 136
Redistributing a Table for Join Purposes 137
Big Table Small Table Join Strategy 138
Big Table Small Table Duplication 139
Nested Join 140
Hash Join 141
Exclusion Join 142
Product Joins 143
Cartesian Product Join 144

Chapter 10 — Join Indexes 145
Three basic types of Join Indexes 146
Join Index Fundamentals 147
Join Indexes versus other objects 148
Multi-Table Join Index 149
Single-Table Join Indexes 150
Aggregate Join Index 151
Sparse Index 152
Sparse Index Picture 153
Global Join Index 154
Global Join Index Picture 155
Global Join Index – Multi-Table Join Back 156
Hash Indexes 157
Hash Indexes vs. Single-Table Join Indexes 158

Chapter 11 — Explains 159
The Teradata Optimizer "knows" how to Explain in Detail 160
Row Estimate Confidence Levels 162
Explain Terminology 163
Visual Explain 169

Chapter 12 — The Parsing Engine in Detail 171
The Parsing Engine (PE) goes through Six Steps 172
Each PE has a Plan Library called RTS Cache 173
The Parsing Engine has Data Dictionary Cache 174
Why the PE loves the Macro 175
The Parsing Engine in Detail 176
The Parsing Engine Knows All 177

Chapter 13 — Understanding Views and Macros 179
Views 179
Creating Simple VIEWs and VIEWs that Join Tables 180
How to Change a VIEW Using REPLACE 183
How to Drop a VIEW 185
View Aggregation and Nested Views 186
All About Macros 187
Macros that Use Parameters 188
Changing a MACRO Using REPLACE 189
How to Drop a MACRO 192

Chapter 14 - Locks 193
Teradata has Four locks at Three levels 194
Locks and their compatibility 195
How Teradata Locks Objects 196
Teradata Locks – First Come First Serve 197
Locking Queue Example 2 198

Locking Queue Example 3 199
Locking Modifier 200
The NOWAIT Option 201

Chapter 15 - Collect Statistics 203
Dynamic AMP Sampling 204
How Collect Statistics Works 205
Sample Statistics 206
Sample Statistics 206
What You Should COLLECT STATISTICS On 207
COLLECT STATISTICS DETAILED SYNTAX 208
COLLECT STATISTICS Examples 209

Chapter 16 - MISC 211
Identity Columns 211
Identity Columns Example 212
LIKE Clause 213
SUBSTRING and SUBSTR Functions 214
Referential Integrity 215
Soft Referential Integrity 216
Materialized Views 217
Compression 218
Implementing Compression 219
How Compression Works 220
Creating a Table With DATABLOCKSIZE 221
The ALTER TABLE Command 222
Index Wizard 222
Index Wizard 223

Chapter 1 — The Rules of Data Warehousing

"Let me once again explain the rules. Teradata rules!"

Tera-Tom Coffing

The Teradata RDBMS was designed to eliminate the technical pitfalls of data warehousing and it is parallel processing that allows Teradata to rule this industry. The problem with Data Warehousing is that it is so big and so complicated that there literally are no rules. Anything goes! Data Warehousing is not for the weak or faint of heart because the terrain can be difficult and that is why 75% of all data warehouses fail. Teradata data warehouses provide the users with the ability to build a data warehouse for the business without having to compromise because their database is unable to meet the challenges and requirements of constant change. That is why 90% of all Teradata data warehouses succeed.

Teradata allows businesses to quickly respond to changing conditions. Relational databases are more flexible than other database types and flexibility is Teradata's middle name. Here is how Teradata Rules:

- 8 of the Top 13 Global Airlines use Teradata
- 10 of the Top 13 Global Communications Companies use Teradata
- 9 of the Top 16 Global Retailers use Teradata
- 8 of the Top 20 Global Banks use Teradata
- 40% of Fortune's "US Most Admired" companies use Teradata
- Teradata customers account for more than 70% of the revenue generated by the top 20 global telecommunication companies
- Teradata customers account for more than 55% of the revenue generated by the top 30 Global retailers
- Teradata customers account for more than 55% of the revenue generated by the top 20 global airlines
- More than 25% of the top 15 global insurance carriers use Teradata

Teradata Certification

> *"If you want to have the potential to be rich then get a medical degree, a law degree or pass the six Teradata Certification Tests!"*
>
> **– Tera-Tom Coffing**

There are six Teradata Certification tests and they are listed below. **Pass all six tests makes you a Teradata Certified Master**. You will be on a Teradata Master Email list where you can post or answer questions from every Teradata Certified Master around the globe. To take a Teradata test you sign up online at **Prometric.com**. Once at the website Prometric.com then select the hyperlink **"Schedule a Test"**. You will be taken to a 3-step menu. Select **Information Technology** as your **AREA of STUDY**. Select **NCR Teradata** as the **TESTING PROGRAM**. Fill in your location of **STATE** and **COUNTRY** and then select NCR Teradata again and pick the test you want to take. You will be given **TEST SITE** locations to choose from and make an appointment to take your examination.

Teradata Certification Tests

1 – Teradata Basics
2 – Teradata Physical Implementation
3 – Teradata SQL
4 – Teradata Database Administration (DBA)
5 – Teradata Designer
6 – Teradata Application Development

Pass all six tests and you are a Teradata Certified Master for life!

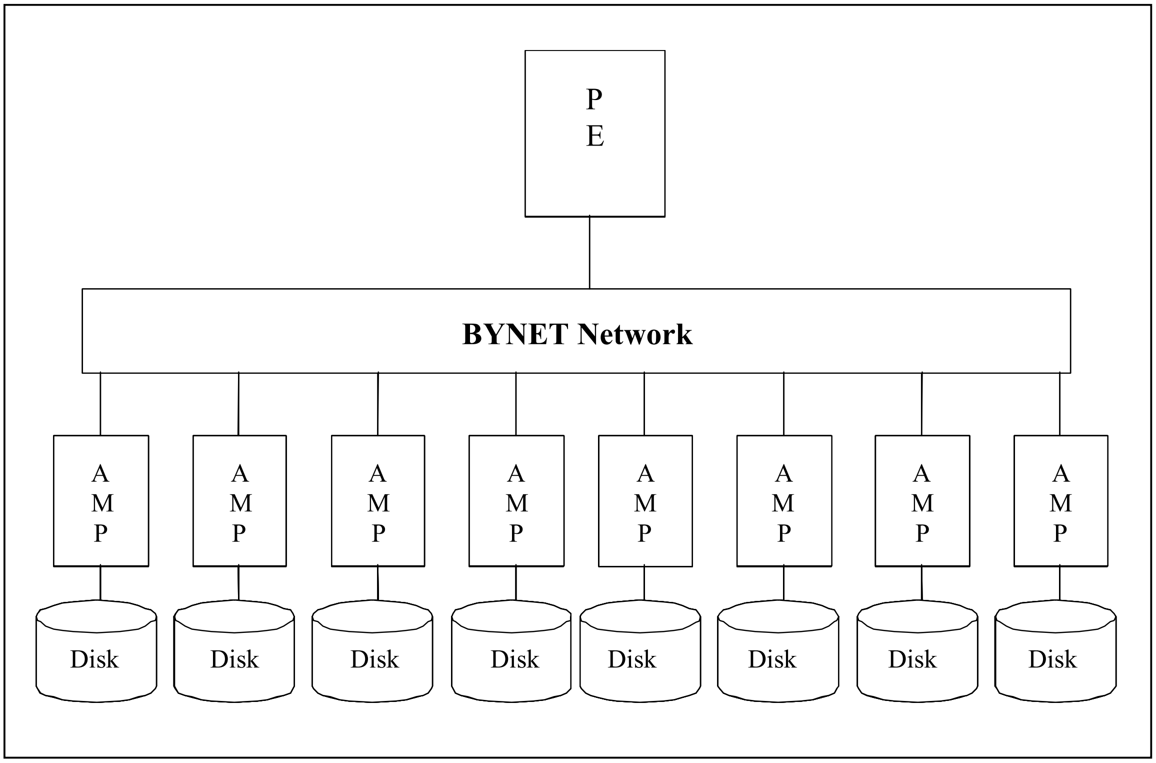
Pass the Teradata Basics and you are a Teradata Certified Professional
Pass Tests 1 and 2 and you are a Teradata Certified Implementation Specialist
Pass Tests 1 and 3 and you are a Teradata Certified SQL Specialist
Pass Tests 1, 2, and 4 and you are a Teradata Certified Administrator
Pass Tests 1, 2, and 5 and you are a Teradata Certified Designer
Pass Tests 1, 3, and 6 and you are a Teradata Certified Application Developer

A Logical View of the Teradata Architecture

"Kites rise highest against the wind – not with it."

– Sir Winston Churchill

Many of the largest data warehouses in the world are on Teradata. Teradata provides customers a centrally located architecture. This provides a **single version** of the **truth** and it **minimizes synchronization**. Having Teradata on your side is a sure win-ston. If Churchill had been a data warehouse expert, he would agree that most data warehouses eventually receive the blitz and stop working while Teradata has the strength from parallel processing to "never give up".



The user submits SQL to the Parsing Engine (PE). The PE checks the syntax and then the security and comes up with a plan for the AMPs. The PE communicates with the AMPs across the BYNET. The AMPs act on the data rows as needed and required. **AMPs temporarily store the answer sets in Spool Space**. Spool is unused space on the disk. Multiple users can never share spool files. The only way to **share spool files** is when **statements** are from the **same request parcel**.

The Parsing Engine (PE)

"The greatest weakness of most humans is their hesitancy to tell others how much they love them while they're alive."

– O.A. Battista

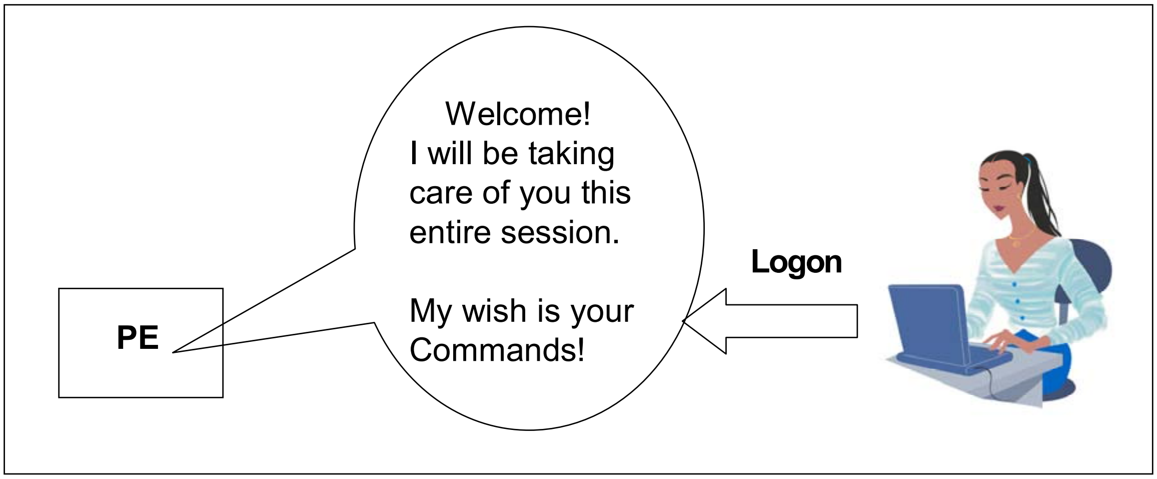
If you haven't told someone lately how much you love them you need to find a way. Leadership through love is your greatest gift. Teradata has someone who greets you with love with every logon. That person is the Parsing Engine (PE), which is often referred to as the optimizer. When you logon to Teradata the Parsing Engine is waiting with tears in its eyes and love in its heart ready to make sure your session is taken care of completely.

The Parsing Engine does three things every time you run an SQL statement.

- **Checks the syntax of your SQL**
- **Checks the security to make sure you have access to the table**
- **Comes up with a plan for the AMPs to follow**

The PE creates a PLAN that tells the AMPs exactly what to do in order to get the data. The PE knows how many AMPs are in the system, how many rows are in the table, and the best way to get to the data. The Parsing Engine is the best optimizer in the data warehouse world because it has been continually improved for over 25 years at the top data warehouse sites in the world.

The **Parsing Engine** verifies SQL requests for **proper syntax**, checks security, maintains up to **120 individual user sessions**, and breaks down the **SQL requests into steps.**


Welcome!
I will be taking care of you this entire session.

My wish is your Commands!
Logon
PE

The Access Module Processors (AMPs)

"A true friend is one who walks in when the rest of the world walks out."

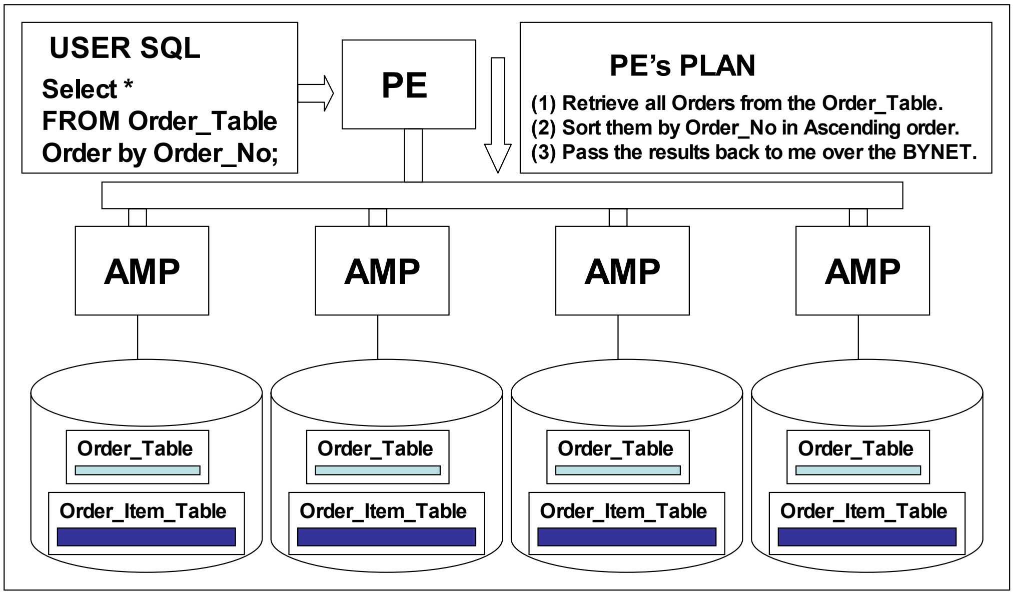
– Anonymous

The AMPs are truly mans best friend because they will work like a dog to read and write the data. (Their bark is worse then their byte). An AMP never walks out on a friend. The AMPs are the worker bees of the Teradata system because the AMPs read and write the data to their assigned disks. The Parsing Engine is the boss and the AMPs are the workers. The AMPs merely follow the PE's plan and read or write the data.

The **AMPs are** always **connected** to a singe **virtual disk** or **Vdisk**. The philosophy of parallel processing revolves around the AMPs. Teradata takes each table and spreads the rows evenly among all the AMPs. When data is requested from a particular table each AMP retrieves the rows for the table that they hold on their disk. If the data is spread evenly then each AMP should retrieve their rows simultaneously with the other AMPs. That is what we mean when we say Teradata was born to be parallel.

The AMPs will also perform output conversion while the PE performs input conversion. The AMPs do the physical work associated with retrieving an answer set.

The PE is the boss and the AMPs are the workers. **Could** you have a **Teradata system** without **AMPs**? No – who would **retrieve the data**? Could you have a **Teradata system** without **PEs**? Of course not – could you get along **without your boss**?!!!



The BYNET

"Not all who wander are lost."

– J. R. R. Tolkien

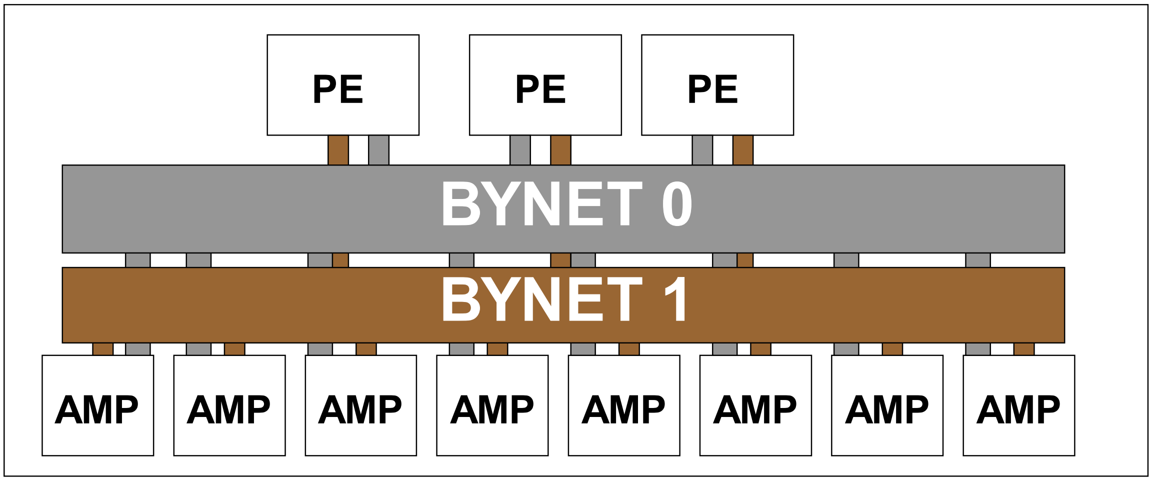
The BYNET is the communication network between AMPs and PE's. Data and communication never wanders and is never lost. How well does the BYNET know communication? It is the lord of the things! How often does the PE pass the plan to the AMPs over the BYNET? Every time – it makes it a hobbit!

The PE passes the PLAN to the AMPs over the BYNET. The AMPs then retrieve the data from their disks and pass it to the PE over the BYNET. The **BYNET** can communicate **point-to-point**, **multi-cast**, or **broadcast** with the **AMPs**.

The BYNET provides the communications between AMPs and PEs – so no matter how large the data warehouse physically gets, the BYNET makes each AMP and PE think that they are right next to one another. The BYNET gets its name from the Banyan tree. The Banyan tree has the ability to continually plant new roots to grow forever. Likewise, the BYNET scales as the Teradata system grows in size. The BYNET is scalable.

There are **always two BYNETs** for **redundancy** and extra **bandwidth.** AMPs and PEs can use both BYNETs to send and retrieve data simultaneously. What a network!

- The PE checks the user's SQL Syntax;
- The PE checks the user's security rights;
- The PE comes up with a plan for the AMPs to follow;
- The PE passes the plan along to the AMPs over the BYNET;
- The AMPs follow the plan and retrieve the data requested;
- The AMPs pass the data to the PE over the BYNET; and
- The PE then passes the final data to the user.

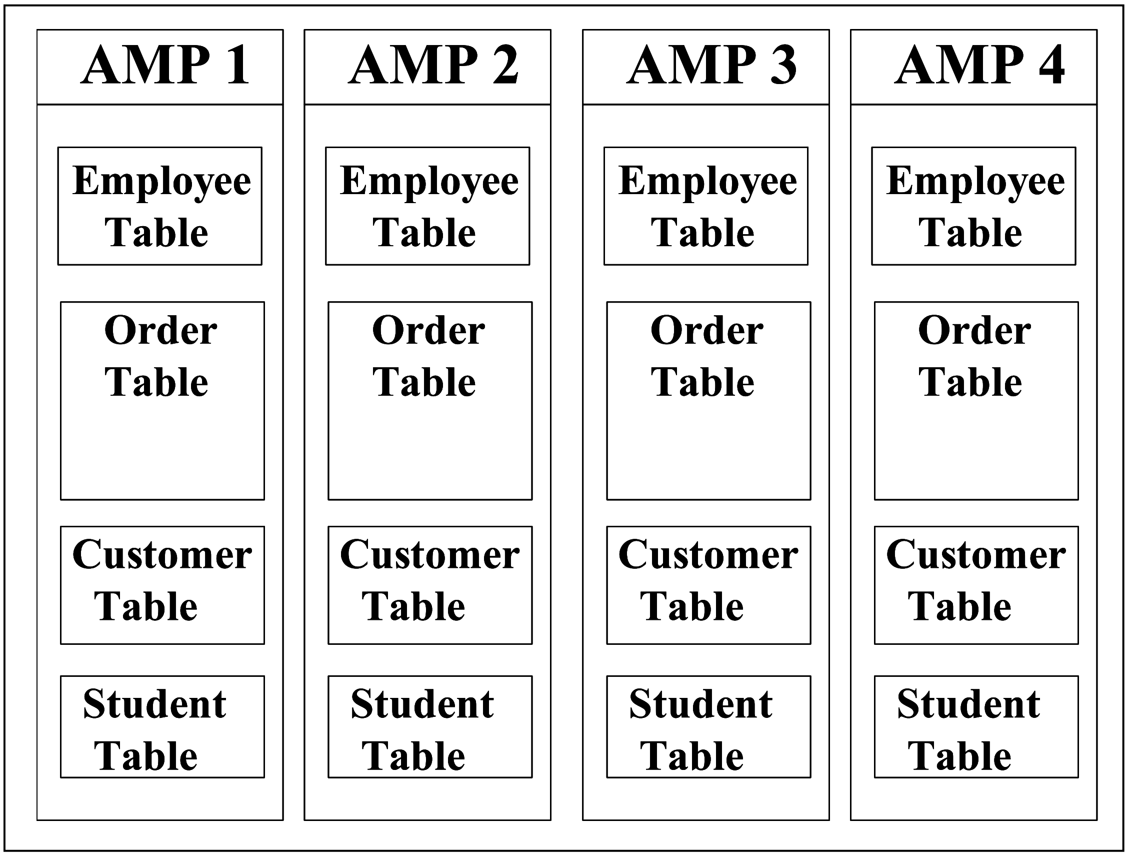


A Visual for Data Layout

"I saw the angel in the marble and carved until I set him free."

--Michelangelo

Teradata saw the users in the warehouse and parallel processed until it set them free. Free to ask any question at any time on any data. The Sistine Chapel wasn't painted in a day and a true data warehouse takes time to carve. Sculpt your warehouse with love and caring and you will build something that will allow your company to have limits that go well beyond the ceiling. Below is a logical view of data on AMPs. Each AMP holds a portion of a table. Each AMP keeps the tables in their own separate drawers.



- Each AMP holds a portion of every table.
- Each AMP keeps their tables in separate drawers.

An **AMP** actually holds three things on its **disk** and they are **Tables, Spool, and Secondary Index Subtables**.

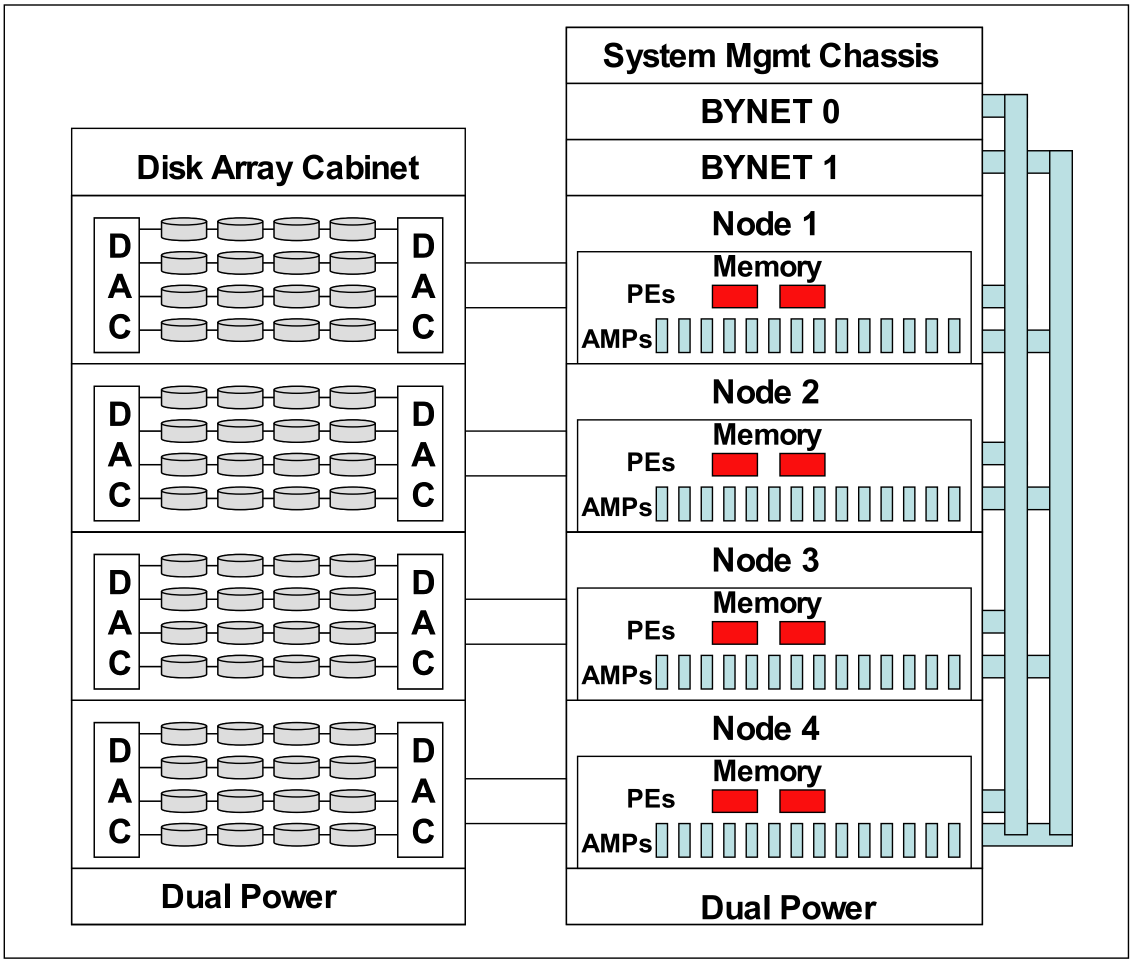
Teradata Cabinets, Nodes, Vprocs, and Disks

"The best way to predict the future is to create it."

- Sophia Bedford-Pierce

Teradata predicted data warehousing 20 years before its time by creating it. Who could have imagined 100 Terabyte systems back in the 1970's? Teradata did!

In the picture below we see a Teradata cabinet with four nodes. Each node has two Intel processors of lightning speed. Inside each nodes memory are the AMPs and PEs which are referred to as Virtual Processor or VProcs. Each node is attached to both BYNETs and each node is attached directly to a set of disks. Each AMP then has one virtual disk where it stores its tables and rows. If you want to expand your system then merely buy another Node Cabinet and another Disk Cabinet.



Chapter 2 — Data Distribution Explained

"There are three keys to selling real estate. They are location, location, and location."

Teradata knows a little about real estate because the largest and best data warehouses have been sold to the top companies in countries around the world. This is because Teradata was meant for data warehousing. When Teradata is explained to the business and they ask if they are interested in a purchase the word most often used is SOLD!

"There are three keys to how Teradata spreads the data among the AMPs. They are Primary Index, Primary Index, and Primary Index."

Every time we begin to teach a data warehousing class, an experienced Teradata manager or DBA will come up and say, "Please explain to the students the importance of the Primary Index.". The Primary Index of each table lays out the data on the AMPs! If you understand anything about Teradata make sure you understand the importance and role of the Primary Index.

Rows and Columns

"I never lost a game; time just ran out on me."

– Michael Jordan

Michael Jordan never lost a game; time just ran out on him; however, many data warehouses lose their game because managing the data can become so intense that life turns into sudden-death double overtime. Teradata allows the data to be placed by the system and not the DBA. Talk about a slam dunk!

Employee Table

EMP	DEPT	LNAME	FNAME	SAL
UPI				
1	40	BROWN	CHRIS	95000.00
2	20	JONES	JEFF	70000.00
3	20	NGUYEN	XING	55000.00
4	?	BROWN	SHERRY	34000.00

Teradata stores its information inside Tables. A table consists of rows and columns. A row is one instance of all columns. According to relational concepts column positions are arbitrary and a column always contains like data. Teradata does not care what order you define the columns and Teradata does not care about the order of rows in a table. Rows are arbitrary also, but once a row format is established then Teradata will use that format because a Teradata table can have only one row format.

There are many benefits of not requiring rows to be stored in order. Unordered data isn't maintained to preserve the order. Unordered data is independent of the query.

ROW	1	40	Brown	Chris	95000

Every AMP will hold a portion of every table. Rows are sent to their destination AMP based on the value of the column designated as the Primary Index.

The Primary Index

"Alone we can do so little; together we can do so much."

– Helen Keller

Teradata takes each table and spreads the table rows across the AMPs. When the table needs to be read, each AMP has to read only their portion of the table. If the AMPs start reading at the same time and there are an equal amount of rows on each AMP, then parallel processing works brilliantly. Alone an AMP can do so little, but the AMPs working together can accomplish the incredible. This brilliant feat begins with the Primary Index.

Each table in Teradata is required to have a Primary Index. The biggest key to a great Teradata Database Design begins with choosing the correct Primary Index. The Primary Index will determine on which AMP a row will reside. Because this concept is extremely important, let me state again that the Primary Index is the only thing that will determine on which AMP a row will reside.

Many people new to Teradata assume that the most important concept concerning the Primary Index is data distribution. INCORRECT! The Primary Index does determine data distribution, but even more importantly, the Primary Index provides the fastest physical path to retrieving data. The Primary Index also plays an incredibly important role in how joins are performed. Remember these three important concepts of the Primary Index and you are well on your way to a great Physical Database Design.

The Primary Index plays 3 roles:

- *Data Distribution*
- *Fastest Way to Retrieve Data*
- *Incredibly important for Joins*

Because the Primary Index is so important it **should NOT be** frequently **changed**. It should also be a column that is **frequently used in row selection**. This means that the SQL should utilize this column often in the WHERE clause. Although you might be tempted to pick a column with poor distribution you should not. This can cause errors stating **"Database FULL Conditions".**

The Two Types of Primary Indexes

"A man who chases two rabbits catches none."

Roman Proverb

Every table must have at least one column as the Primary Index. The Primary Index is defined when the table is created. There are only two types of Primary Indexes, which are a Unique Primary Index (UPI) or a Non-Unique Primary Index (NUPI). Because Teradata distributes the data based on the Primary Index columns value it is quite obvious that you must have a primary index and that there can be only one primary index per table.

"A man who chases two rabbits misses both by a HARE! A person who chases two Primary Indexes misses both by an ERR!"

Tera-Tom Proverb

The Primary index is the Physical Mechanism used to retrieve and distribute data. The primary index is limited to the number of columns in the primary index. In other words, the primary index is made up of from 1 to 16 columns. You can pick only one primary index and once it is picked you will need to access all columns in the WHERE clause for the primary index to be utilized on the retrieve.

Most databases use the Primary Key as the physical mechanism. Teradata uses the Primary Index. There are two reasons you might pick a **different Primary Index** then your **Primary Key**. They are (1) for **Performance** reasons and (2) **known access paths**.

Unique Primary Index (UPI)

"If you do what you've always done, you'll get what you've always got."

A Unique Primary Index means that the values for the selected column must be unique. The data will always be spread evenly if the Primary Index is unique. If you try and insert a row with a Primary Index value that is already in the table, the row will be rejected. Even though a Unique Primary Index will always spread the table rows evenly amongst the AMPs, please don't assume this is always the best thing to do.

"A Unique Primary Index will always do what it has always done and that is to spread the data evenly. Got it?"

Below is a table that has a Unique Primary Index. We have selected EMP to be our Primary Index. Because we have designated EMP to be a Unique Primary Index, there can be no duplicate employee numbers in the table.

Employee Table

EMP	DEPT	LNAME	FNAME	SAL
UPI				
1	40	BROWN	CHRIS	95000.00
2	20	JONES	JEFF	70000.00
3	20	NGUYEN	XING	55000.00
4	?	BROWN	SHERRY	34000.00

A Unique Primary Index (UPI) will always spread the rows of the table Evenly amongst the AMPs. UPI access is always a one-AMP operation. It also requires no duplicate row checking.

Non-Unique Primary Index

"Talk low, talk slow, and don't talk too much."

John Wayne with advice on acting

John Wayne was an original and will never be duplicated, but a NUPI does have duplicate data. Some people assume that a Non-Unique Primary Index means the data will have to duke it out for space on the AMPs. Although the data will not be spread evenly it is OK if the data does not produce a Hot AMP or have ridiculous distribution.

A Non-Unique Primary Index (NUPI) means that the values for the selected column can be non-unique. You can have many rows with the same value in the Primary Index. A Non-Unique Primary Index will almost never spread the table rows evenly. Please don't assume this is always a bad thing. Below is a table that has a Non-Unique Primary Index. We have selected LNAME to be our Primary Index. Because we have designated LNAME to be a Non-Unique Primary Index we are anticipating that there will be individuals in the table with the same last name.

Employee Table

EMP	DEPT	LNAME	FNAME	SAL
		NUPI		
1	40	BROWN	CHRIS	95000.00
2	20	JONES	JEFF	70000.00
3	20	NGUYEN	XING	55000.00
4	?	BROWN	SHERRY	34000.00

"A Non-Unique Primary Index (UPI) will almost NEVER spread the rows of the table evenly amongst the AMPs. Users might talk low because queries can run slow!"

Turning the Primary Index Value into the Row Hash

"Actions lie louder than words"

Carolyn Wells

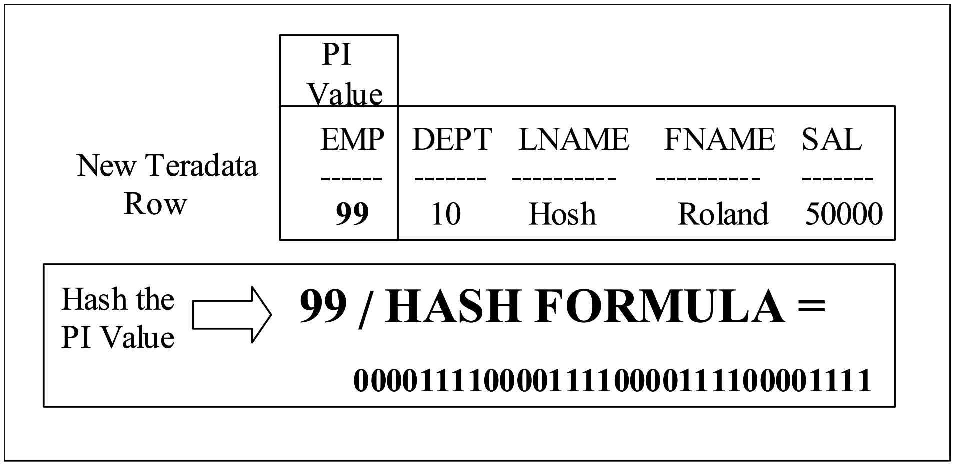
The Primary Index is the only thing that determines where a row will reside and the hashing algorithm never lies. It is important that you understand this process. When a new row arrives into Teradata, the following steps occur:

Teradata's PE examines the Primary Index value of the row. Teradata takes that Primary Index value and runs it through a Hashing Algorithm. The output of the Hashing Algorithm (i.e., a formula) is a 32-bit Row Hash.

The 32-bit Row Hash will perform two functions:

1. The 32-bit Row Hash will point to a certain spot on the Hash Map, which will indicate which AMP will hold the row.
2. The 32-bit Row Hash will always remain with the Row as part of a Row ID.

Hashing is a mathematical process where an **Index** (UPI, NUPI) is converted into a 32-bit row hash value. The key to this hashing algorithm is the **Primary Index**. When this value is determined, the output of this 32-bit value is called the **Row Hash**.



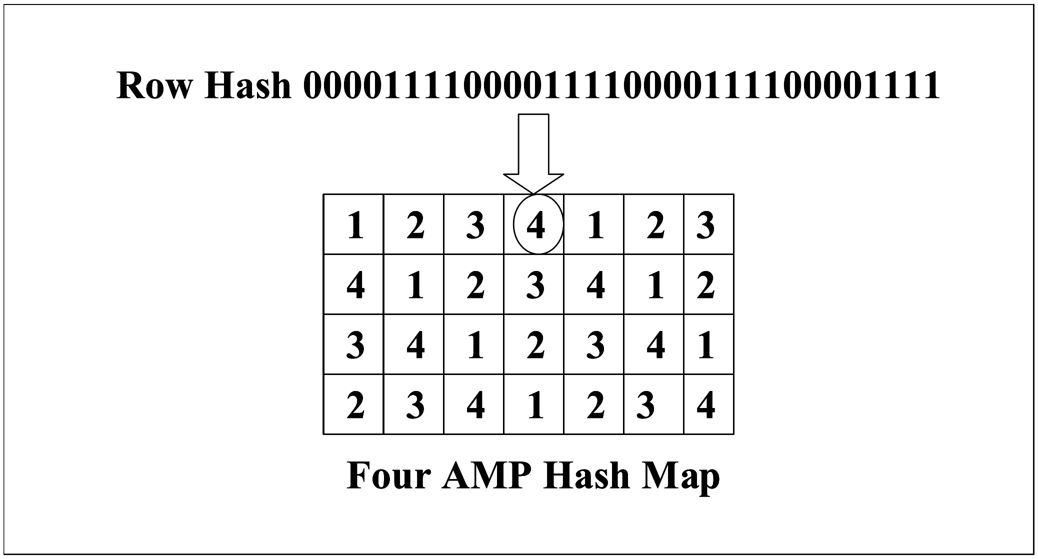
EMP	DEPT	LNAME	FNAME	SAL
99	10	Hosh	Roland	50000

A new row is going to be inserted into Teradata. The Primary Index is the column called EMP. The value in EMP for this row is 99. Teradata runs the value of 99 through the Hash Formula and the output is a 32-bit Row Hash. In his example our **32-bit Row Hash output**: 00001111000011110000111100001111.

The Hashing Algorithm will produce **random, but consistent** row hashes with an end goal to **produce a 32-bit row hash**.

The Row Hash Value determines the Row's Destination

The **first 16 bits** of the **Row Hash** (**a.k.a., Destination Selection Word**) are used to locate an entry in the Hash Map. This entry is called a Hash Map Bucket. The only thing that resides inside a Hash Map Bucket is the AMP number where the row will reside.



Row Hash 00001111000011110000111100001111

1	2	3	4	1	2	3
4	1	2	3	4	1	2
3	4	1	2	3	4	1
2	3	4	1	2	3	4

Four AMP Hash Map

The first 16 bits of the Row Hash of 00001111000011110000111100001111 are used to locate a bucket in the Hash Map. A bucket will contain an AMP number. We now know that employee 99 whose row hash is 00001111000011110000111100001111 will reside on AMP 4. **Note:** The AMP uses the entire 32 bits in storing and accessing the row.

If we took employee 99 and ran it through the hashing algorithm again and again, we would always get a row hash of 00001111000011110000111100001111.

If we take the row hash of 00001111000011110000111100001111 again and again, it would always point to the same bucket in the hash map.

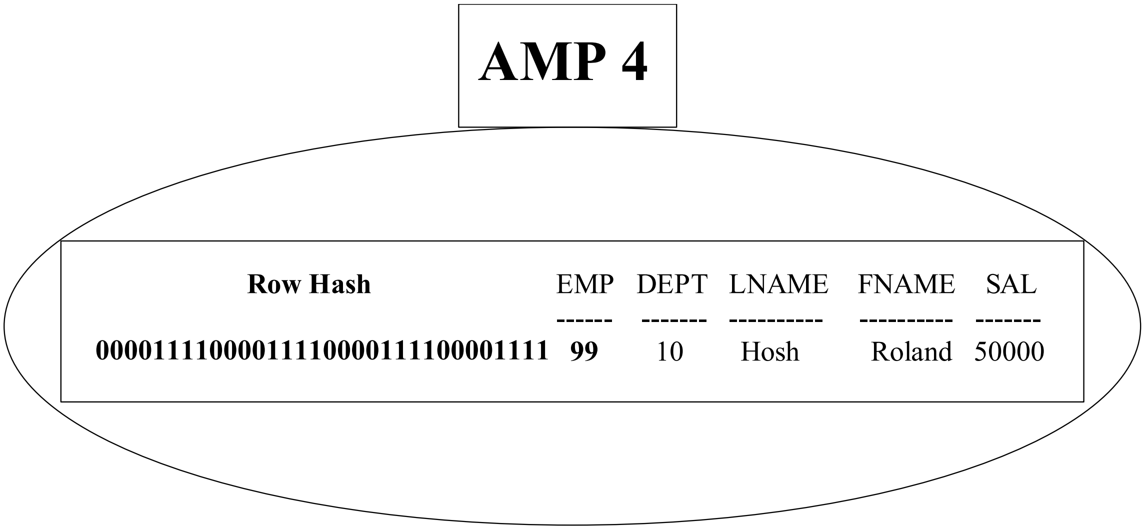
The above statement is true about the Teradata Hashing Algorithm. Every time employee 99 is run through the hashing algorithm, it returns the same Row Hash. This Row Hash will point to the same Hash Bucket every time. That is how Teradata knows which AMP will hold row 99. It does the math and it always gets what it always got!

Hash values are calculated using a hashing formula.

The Hash Map will change if you add additional AMPs.

The Row is Delivered to the Proper AMP

Now that we know that Employee 99 is to be delivered to AMP 4, Teradata packs up the row, places the Row Hash on the front of the row, and delivers it to AMP 4.



AMP 4

Row Hash	EMP	DEPT	LNAME	FNAME	SAL
00001111000011110000111100001111	**99**	10	Hosh	Roland	50000

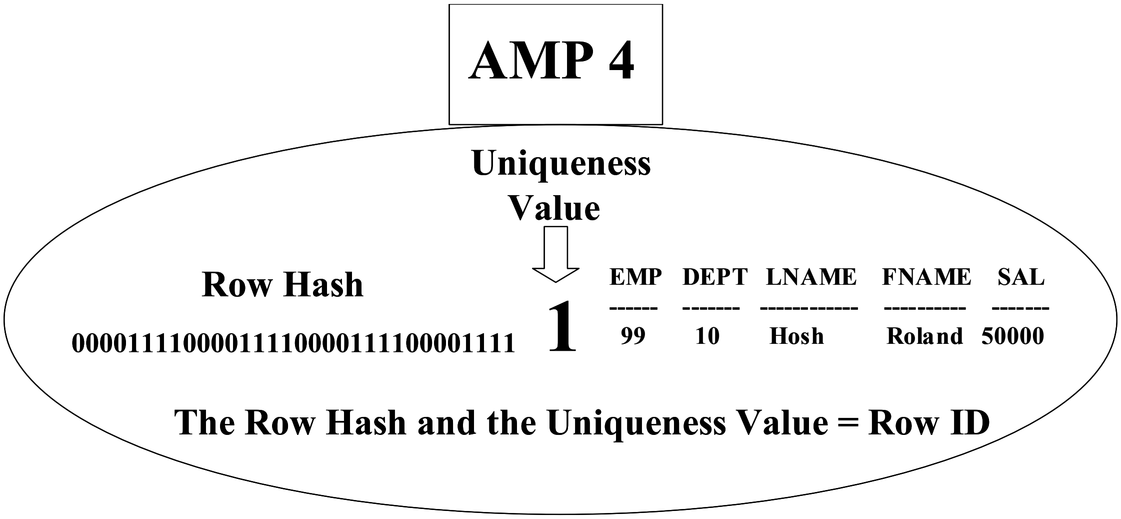
The entire row for employee 99 is delivered to the proper AMP accompanied by the Row Hash, which will always remain with the row as part of the Row ID.

Review:

- A row is to be inserted into a Teradata table
- The Primary Index Value for the Row is put into the Hash Algorithm
- The output is a 32-bit Row Hash
- The Row Hash points to a bucket in the Hash Map
- The bucket points to a specific AMP
- The row along with the Row Hash are delivered to that AMP

The AMP will add a Uniqueness Value

When the AMP receives a row it will place the row into the proper table, and the AMP checks if it has any other rows in the table with the same row hash. If this is the first row with this particular row hash, the AMP will assign a 32-bit uniqueness value of 1. If this is the second row hash with that particular row hash, the AMP will assign a uniqueness value of 2. The 32-bit row hash and the 32-bit uniqueness value make up the 64-bit Row ID. The Row ID is how tables are sorted on an AMP.



EMP	DEPT	LNAME	FNAME	SAL
99	10	Hosh	Roland	50000

The Row Hash always accompanies when an AMP receives a row.

The AMP will then assign a Uniqueness Value to the Row Hash. It assigns a 1 if the Row Hash is unique or a 2 if it is the second, or a 3 if the third, etc.

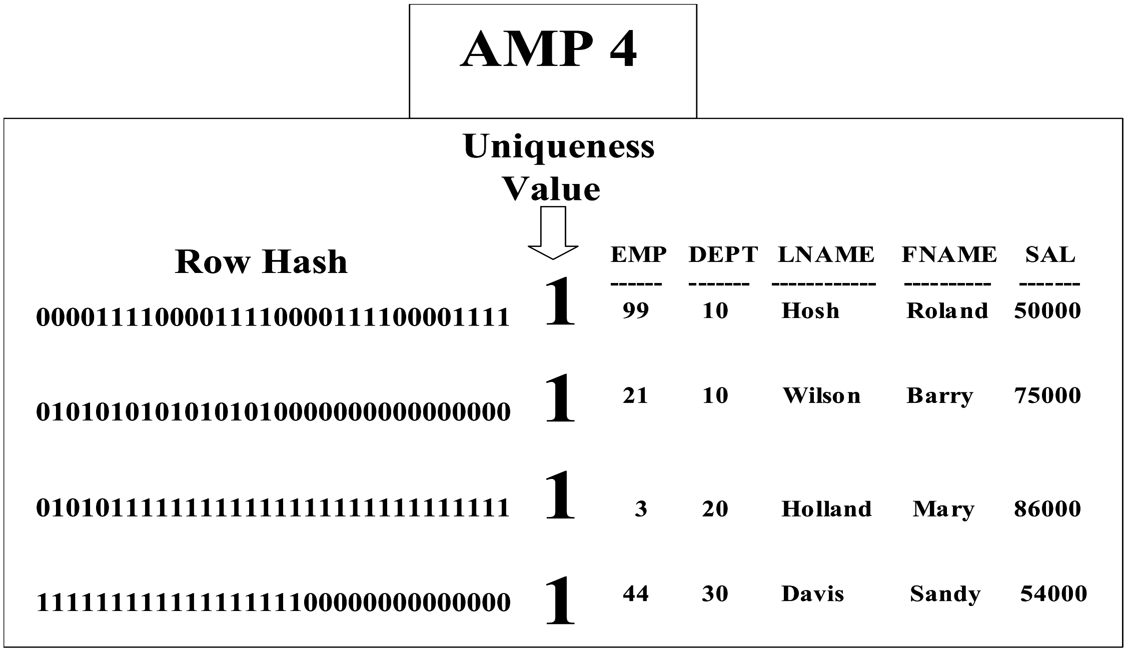
The **two** main components of the **ROW ID** are **Row Hash** and **Uniqueness Value**.

An Example of an UPI Table

"Never trust the advice of a man in difficulties"

Aesop (620 BC – 560 BC)

You can trust a Unique Primary Index because it has no difficulties in delivering data evenly. It is also trustworthy to its word that it won't allow any duplicate values. That is why you see the uniqueness values below all set to 1. Below is an example of a portion of a table on one AMP. The table has a Unique Primary Index of EMP.



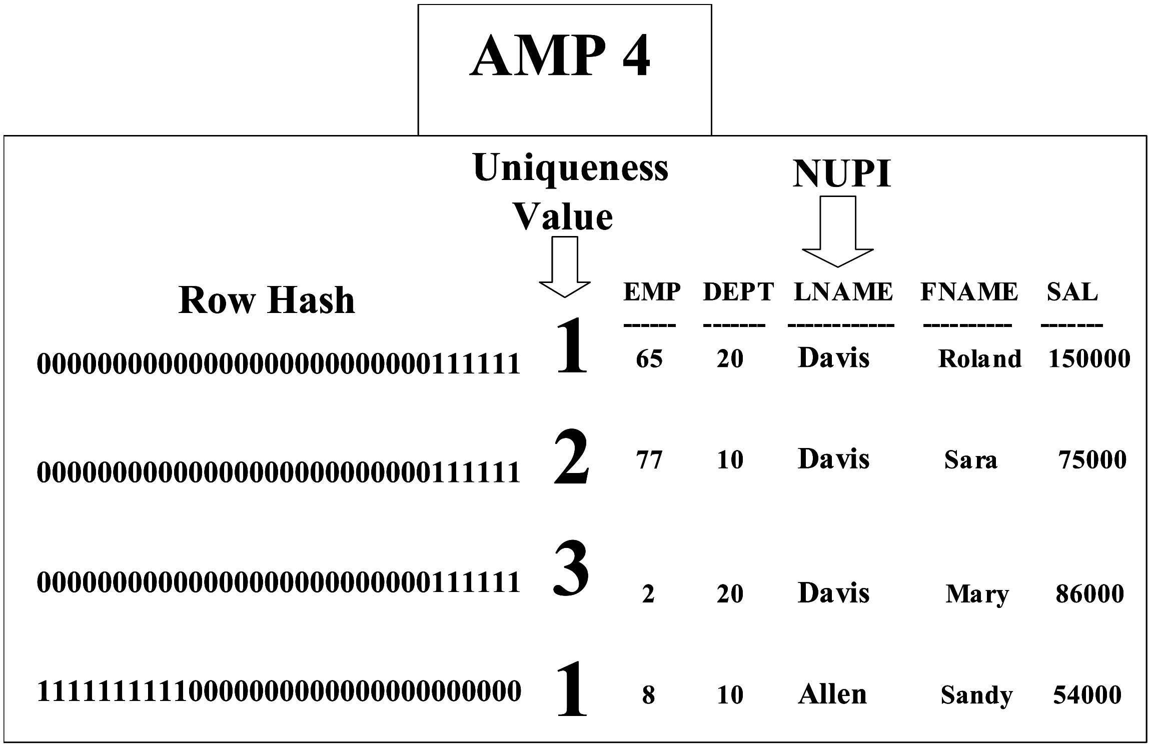
Row Hash		EMP	DEPT	LNAME	FNAME	SAL
		------	-------	------------	----------	-------
00001111000011110000111100001111	1	99	10	Hosh	Roland	50000
01010101010101010000000000000000	1	21	10	Wilson	Barry	75000
01010111111111111111111111111111	1	3	20	Holland	Mary	86000
11111111111111111100000000000000	1	44	30	Davis	Sandy	54000

The above Employee Table has a Unique Primary Index on the column EMP. Notice that Row ID sorts the portion of the table on AMP 4. Notice that the Uniqueness Value for each row is 1.

One of the important things to remember about hashing is that duplicate values are always hashed exactly if they are the same data types. **INTEGER** and **DATE** data types also generate the **same hash**. **CHAR** and **VARCHAR** also generate the same **hash results** for the same values.

An Example of a NUPI Table

Below is an example of a portion of a table on one AMP. The table has a Non-Unique Primary Index (NUPI) on the Last Name called LNAME.



Row Hash		EMP	DEPT	LNAME	FNAME	SAL
00000000000000000000000000111111	1	65	20	Davis	Roland	150000
00000000000000000000000000111111	2	77	10	Davis	Sara	75000
00000000000000000000000000111111	3	2	20	Davis	Mary	86000
11111111110000000000000000000000	1	8	10	Allen	Sandy	54000

The above Employee Table has a Non-Unique Primary Index on the column LNAME. Notice that each row with the LNAME of Davis has the exact same Row Hash. Notice that the Uniqueness Value for each Davis is incremented by 1.

Each time the LNAME is Davis, the Hashing Algorithm generates the Row Hash:

000000000000000000000000000011111

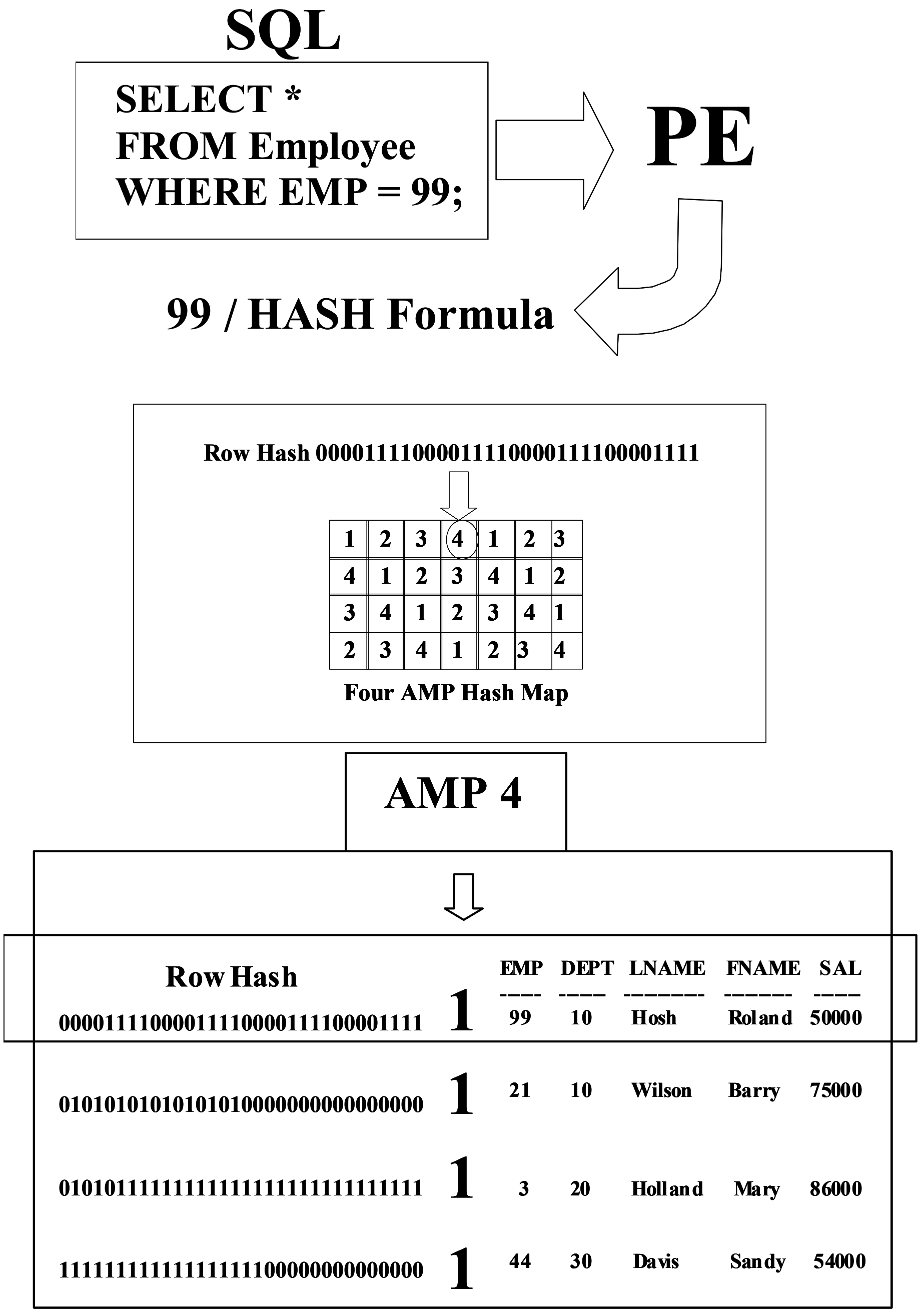
That Row Hash points to the exact same bucket in the Hash Map. This particular bucket in the Hash Map references (or points to) AMP 4.

The Row Hash accompanied each row to AMP 4. The AMP assigned Uniqueness Values of 1, 2 and 3 to the three rows with the LNAME of Davis.

Notice that Row ID sorts the portion of the table on AMP 4.

How Teradata Retrieves Rows

In the example below, a user runs a query looking for information on Employee 99. The PE sees that the Primary Index Value EMP is used in the SQL WHERE clause. Because this is a Primary Index access operation, the PE knows this is a one AMP operation. The PE hashes 99 and the Row Hash is 00001111000011110000111100001111. This points to a bucket in the Hash Map that represents AMP 4. AMP 4 is sent a message to get the Row Hash: 00001111000011110000111100001111 and make sure it's EMP 99.



```
SELECT *
FROM Employee
WHERE EMP = 99;
```

Row Hash 00001111000011110000111100001111

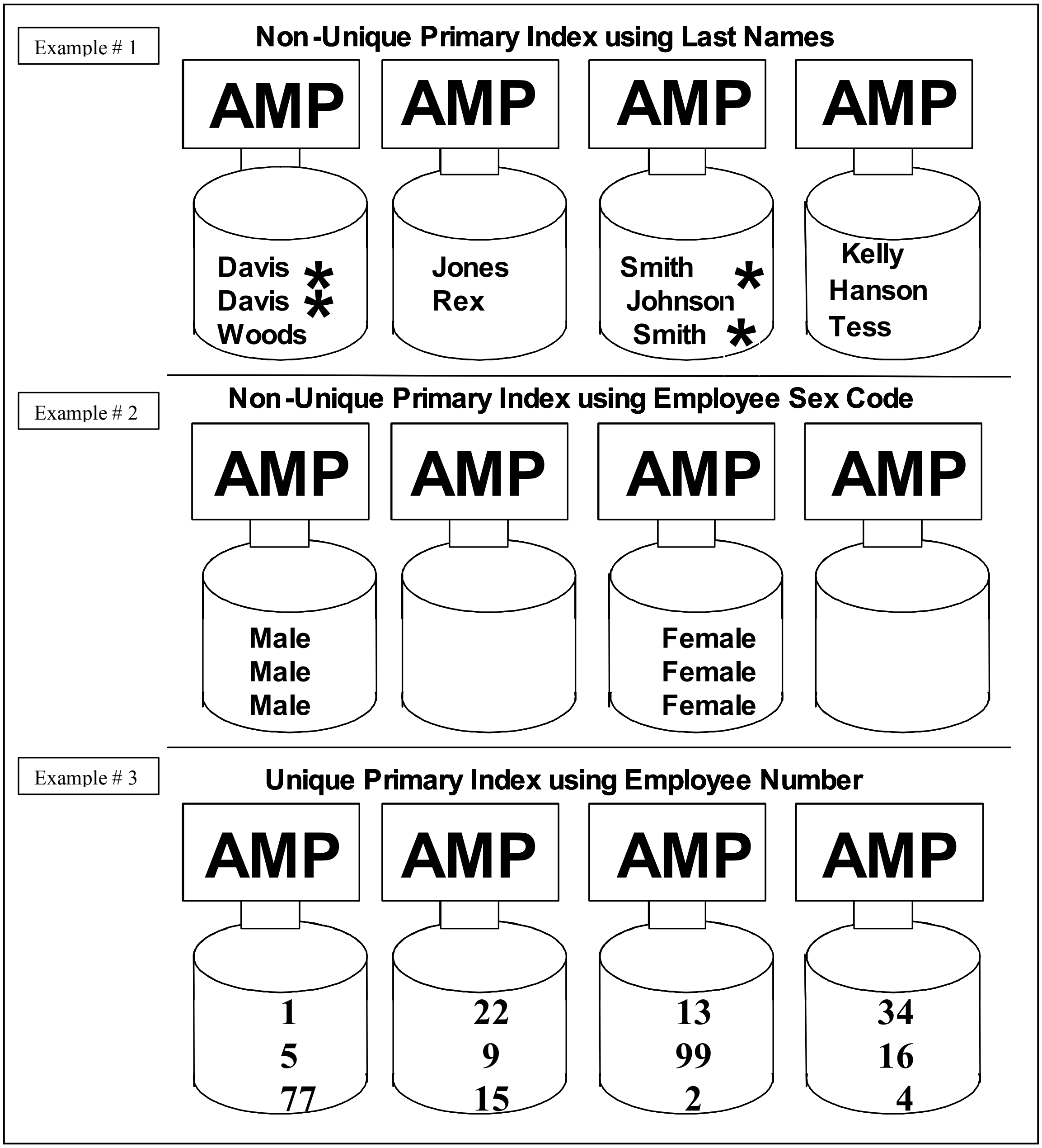
1	2	3	4	1	2	3
4	1	2	3	4	1	2
3	4	1	2	3	4	1
2	3	4	1	2	3	4

Four AMP Hash Map

Row Hash		EMP ----	DEPT ----	LNAME --------	FNAME ------	SAL ----
00001111000011110000111100001111	1	99	10	Hosh	Roland	50000
01010101010101010000000000000000	1	21	10	Wilson	Barry	75000
01010111111111111111111111111111	1	3	20	Holland	Mary	86000
11111111111111111100000000000000	1	44	30	Davis	Sandy	54000

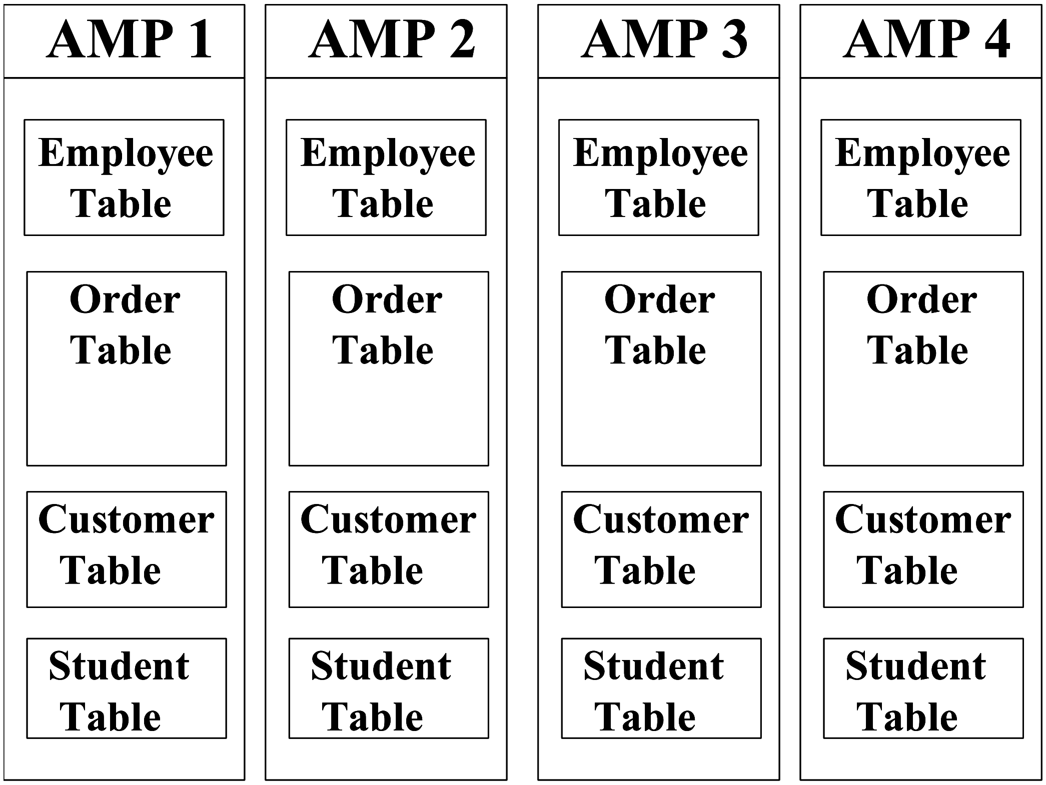
Row Distribution

In the examples below we see three different Teradata Systems. The first system has used Last_Name as a Non-Unique Primary Index (NUPI). The second example has used Sex_Code as a Non-Unique Primary Index (NUPI). The last example uses Employee_Number as a Unique Primary Index (UPI).



A Visual for Data Layout

Below is a logical view of data on AMPs. Each AMP holds a portion of a table. Each AMP keeps the tables in their own separate drawers. The Row ID is used to sort each table on an AMP.



AMP 1	AMP 2	AMP 3	AMP 4
Employee Table	Employee Table	Employee Table	Employee Table
Order Table	Order Table	Order Table	Order Table
Customer Table	Customer Table	Customer Table	Customer Table
Student Table	Student Table	Student Table	Student Table

- *Each AMP holds a portion of every table.*

- *Each AMP keeps their tables in separate drawers.*

- *Each table is sorted by Row ID.*

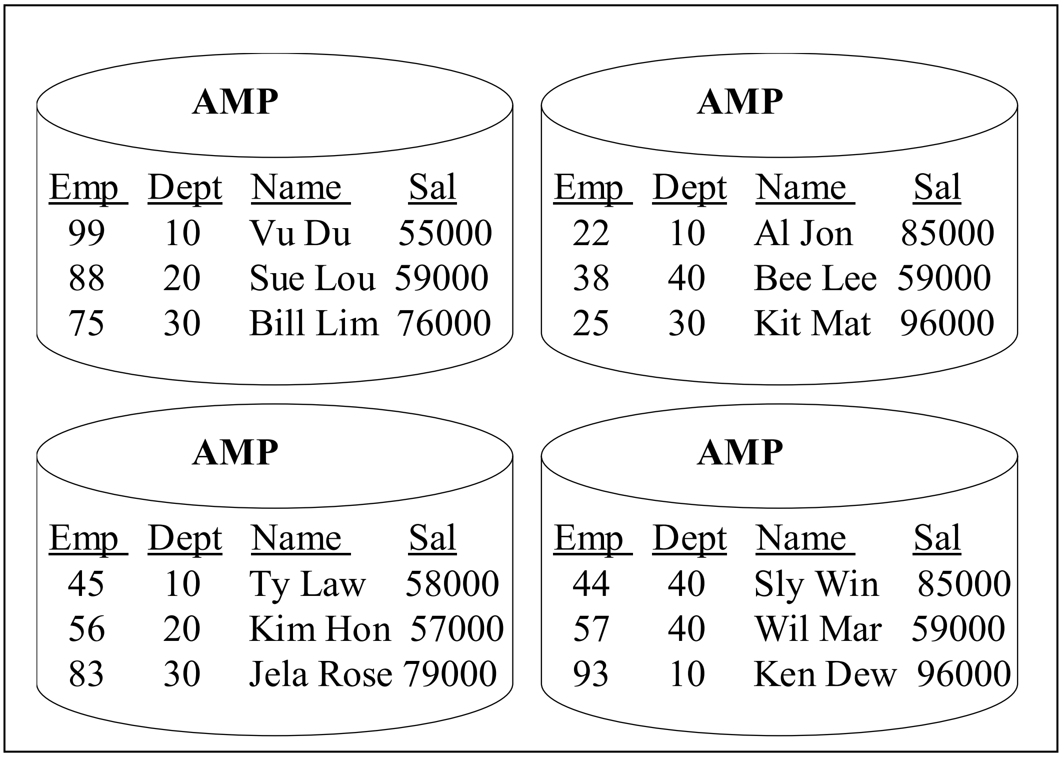
Teradata accesses data in three ways

- Primary Index (fastest)
- Secondary Index (second fastest way)
- Full Table Scan (slowest way)

Primary Index (fastest) - When ever a Primary Index is utilized in the SQL WHERE Clause the PE will be able to use the Primary Index to get the data with a one-AMP operation.

Secondary Index (next fastest) - If the Primary Index is not utilized sometimes Teradata can utilize a secondary index. It is not as fast as the Primary Index, but it is much faster than a full table scan.

Full Table Scan (FTS) (Slowest)
Teradata handles full table scans brilliantly because Teradata accesses each data row only once because of the parallel processing. Full Table Scans are a way to access Teradata without using an index. Each data block per table is read only once.



AMP

Emp	Dept	Name	Sal
99	10	Vu Du	55000
88	20	Sue Lou	59000
75	30	Bill Lim	76000

AMP

Emp	Dept	Name	Sal
22	10	Al Jon	85000
38	40	Bee Lee	59000
25	30	Kit Mat	96000

AMP

Emp	Dept	Name	Sal
45	10	Ty Law	58000
56	20	Kim Hon	57000
83	30	Jela Rose	79000

AMP

Emp	Dept	Name	Sal
44	40	Sly Win	85000
57	40	Wil Mar	59000
93	10	Ken Dew	96000

When Teradata does a Full Table Scan of the above how many rows are read? 12 How about per AMP? 3

Data Layout Summary

Teradata lays out data totally based on the Primary Index value. If the Primary Index is Unique, the data layout will be spread equally among the AMPs. If the Primary Index is Non-Unique, the data distribution across the AMPs may be skewed. A Non-Unique Primary Index is acceptable if the data values provide reasonably even distribution.

Every table must have a Primary Index and it is created at CREATE TABLE time. When users utilize a Unique Primary Index in the WHERE clause of their query, the query will be a one-AMP operation. Why?

A Unique Primary Index value only uses one AMP to return at most one row.

A Non-Unique Primary Index value also uses one AMP to return zero to many rows. The same values run through the Hashing Algorithm will return the exact same Row Hash. Therefore, like values will go to the same AMP. The only difference will be the Uniqueness Value.

Every row in a table will have a Row ID. The Row ID consists of the Primary Index Value Row Hash and the Uniqueness Value.

Primary Index	Number of AMPs	Rows Returned
UPI	1	0-1
NUPI	1	0-Many

Chapter 3 — V2R5 Partition Primary Indexes

"Life is a succession of lessons, which must be lived to be understood."

--Ralph Waldo Emerson

Teradata has lived and understood the data warehouse environment for decades over their competitors. One of the key fundamentals of the V2R5 release is in the ability to allow the AMPs to access data quicker with Partition Primary Indexes. **Partition Primary Indexes** are always and I mean always defined in the **Physical Model**.

In the past Teradata has hashed the Primary Index, which produced a Row Hash. From the Row Hash, Teradata was able to send the row to a specific AMP. The AMP would place a uniqueness value and the Row Hash plus the Uniqueness value made up the Row ID. The data on each AMP was grouped by table and sorted by ROW ID.

Through years of experience working with data warehouse user queries Teradata has decided to take the hashing to an additional level.

In the past you could choose a Unique Primary Index (UPI) or a Non-Unique Primary Index (NUPI). Now Teradata will let you choose either a Partition Primary Index (PPI) or a Non-Partition Primary Index (NPPI).

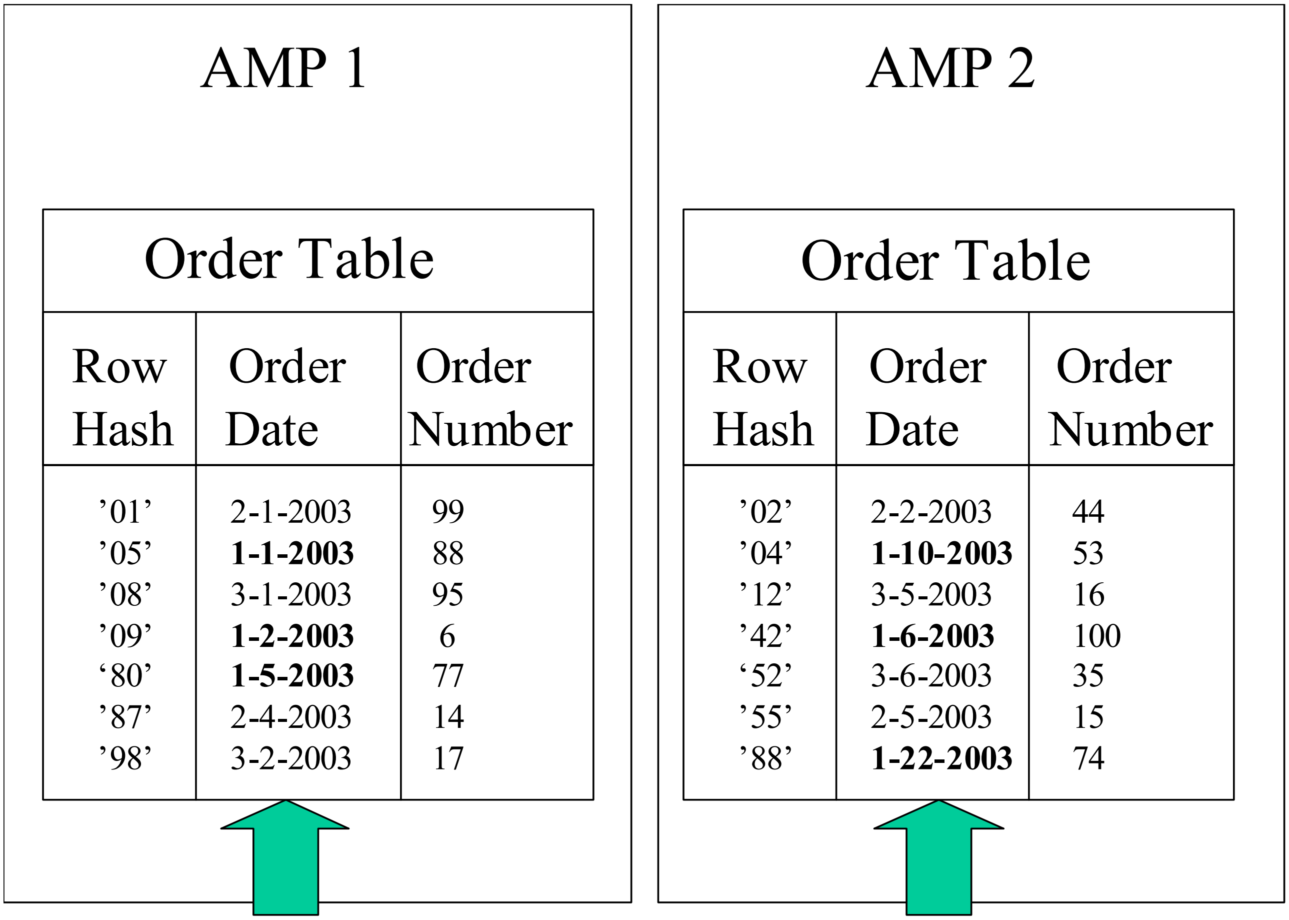
This allows for fantastic flexibility because user queries will often involve ranges or are specific to a particular department, location, region, or code of some sort. Now the AMPs can find the data quicker because the data is grouped in alphabetical order. You can avoid Full Table Scans more often.

An example is definitely called for here. I will show you a table that is hashed and another that has a Partition Primary Index.

V2R4 Example

If you are on a V2R4 machine then each table is distributed to the AMPs based on Primary Index Row Hash and then sorted on that AMP by Row ID. The example below is also a Non-Partitioned Primary Index in V2R5.

An Example of V2R4



AMP 1

Order Table

Row Hash	Order Date	Order Number
’01’	2-1-2003	99
’05’	**1-1-2003**	88
’08’	3-1-2003	95
’09’	**1-2-2003**	6
‘80’	**1-5-2003**	77
’87’	2-4-2003	14
’98’	3-2-2003	17

AMP 2

Order Table

Row Hash	Order Date	Order Number
’02’	2-2-2003	44
’04’	**1-10-2003**	53
’12’	3-5-2003	16
’42’	**1-6-2003**	100
‘52’	3-6-2003	35
’55’	2-5-2003	15
’88’	**1-22-2003**	74

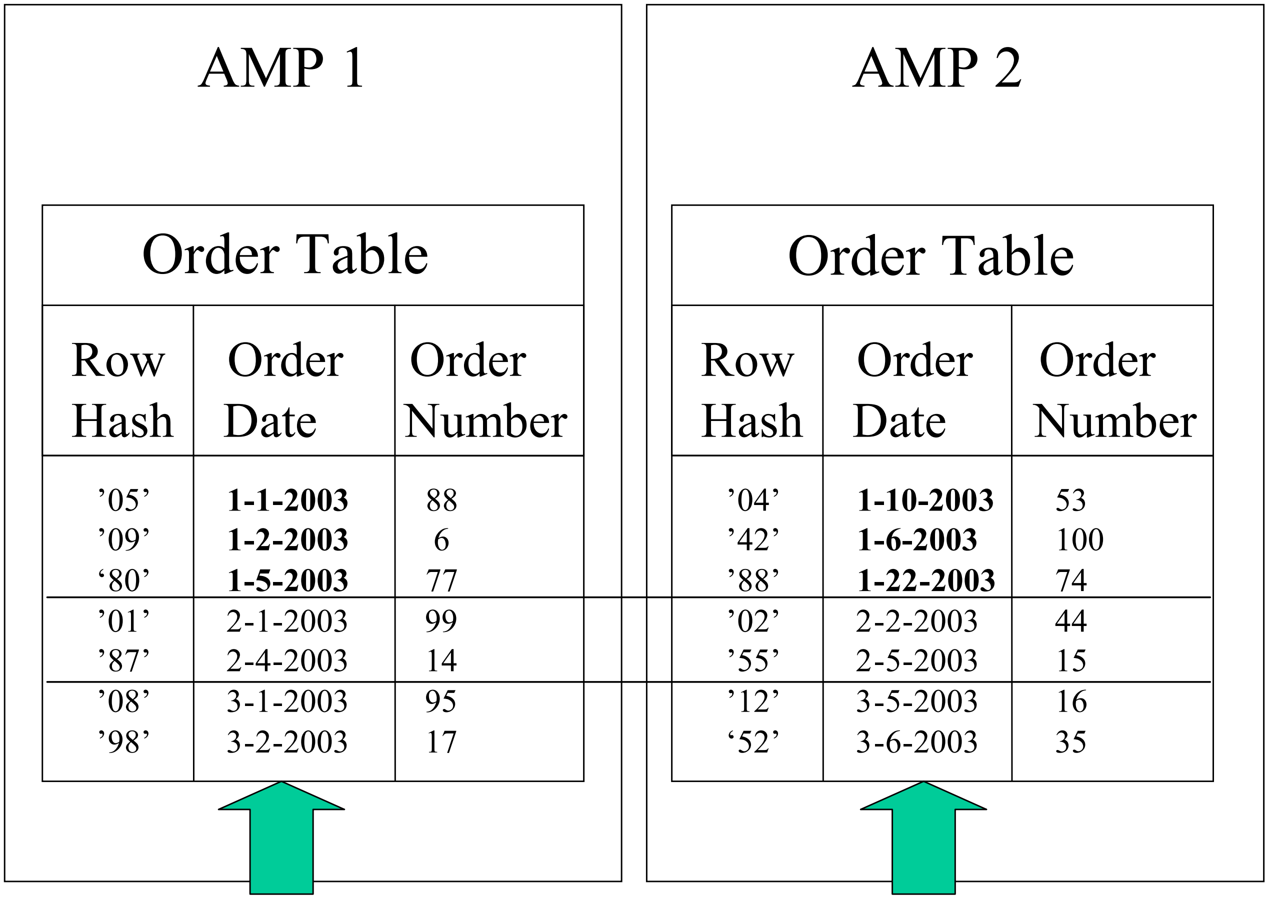
Primary Index is Order Date

Notice that the Primary Index is ORDER_DATE. The Order_Date was hashed and rows were distributed to the proper AMP based on Row Hash and then sorted by Row Id. Unfortunately the query below results in a full table scan to satisfy the query.

*SELECT * FROM Order_Table*
WHERE Order_Date between 1-1-2003 and 1-31-2003;

V2R5 Partitioning

Notice that the Primary Index is now a Partition Primary Index on ORDER_DATE. The Order_Date was hashed and rows were distributed to the same exact AMP as before. The only difference is that the data is sorted by Order_Date and not by Row Hash. The query below does **not** take a Full Table Scan because the January orders are all together in their partition. Partitioned Primary Indexes (PPI) are best for queries that specify range constraints.



AMP 1

Order Table

Row Hash	Order Date	Order Number
’05’	**1-1-2003**	88
’09’	**1-2-2003**	6
‘80’	**1-5-2003**	77
’01’	2-1-2003	99
’87’	2-4-2003	14
’08’	3-1-2003	95
’98’	3-2-2003	17

AMP 2

Order Table

Row Hash	Order Date	Order Number
’04’	**1-10-2003**	53
’42’	**1-6-2003**	100
’88’	**1-22-2003**	74
’02’	2-2-2003	44
’55’	2-5-2003	15
’12’	3-5-2003	16
‘52’	3-6-2003	35

Partition Primary Index is Order_Date

*SELECT * FROM Order_Table*
WHERE Order_Date between 1-1-2003 and 1-31-2003;

Partitioning doesn't have to be part of the Primary Index

"A Journey of a thousand miles begins with a single step."

-Lao Tzu

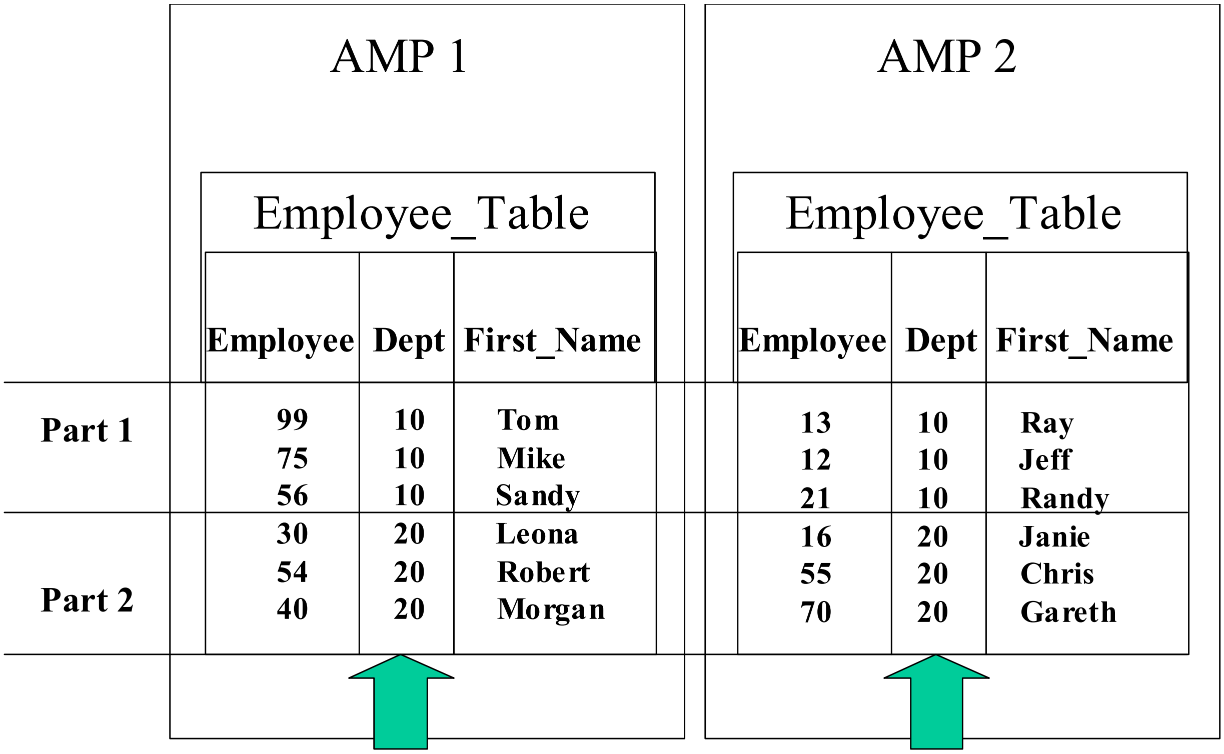
Understanding Teradata begins with a single step and that is reading and understanding this book. You will soon be a Teradata master and that is quite an accomplishment. understanding partitioning is easy once you understand the basic steps. You do not have to partition by a column that is the primary index. Here is an example:

```
CREATE SET TABLE EMPLOYEE_TABLE
(
EMPLOYEE          INTEGER   NOT NULL
,DEPT             INTEGER
,FIRST_NAME       VARCHAR(20)
,LAST_NAME        CHAR(20)
,SALARY           DECIMAL(10,2)
)
PRIMARY INDEX (EMPLOYEE)
PARTITION BY DEPT;
```

You can NOT have a UNIQUE PRIMARY INDEX on a table that is partitioned by something not included in the Primary Index.

Here is an interesting brain teaser? If a Primary Index is Non-Unique on a Partition Primary Index table can Teradata utilize the Primary Index column to access the row or rows? Absolutely! Teradata scans all **partitions** that have **not been eliminated** and the **Hashed Primary Index** value is scanned for the **Row Hash**.

Partition Elimination can avoid Full Table Scans



AMP 1

Employee_Table

	Employee	Dept	First_Name
Part 1	99	10	Tom
	75	10	Mike
	56	10	Sandy
Part 2	30	20	Leona
	54	20	Robert
	40	20	Morgan

AMP 2

Employee_Table

	Employee	Dept	First_Name
Part 1	13	10	Ray
	12	10	Jeff
	21	10	Randy
Part 2	16	20	Janie
	55	20	Chris
	70	20	Gareth

Partition Primary Index is Dept

How many partitions on each AMP will need to be read for the following query?

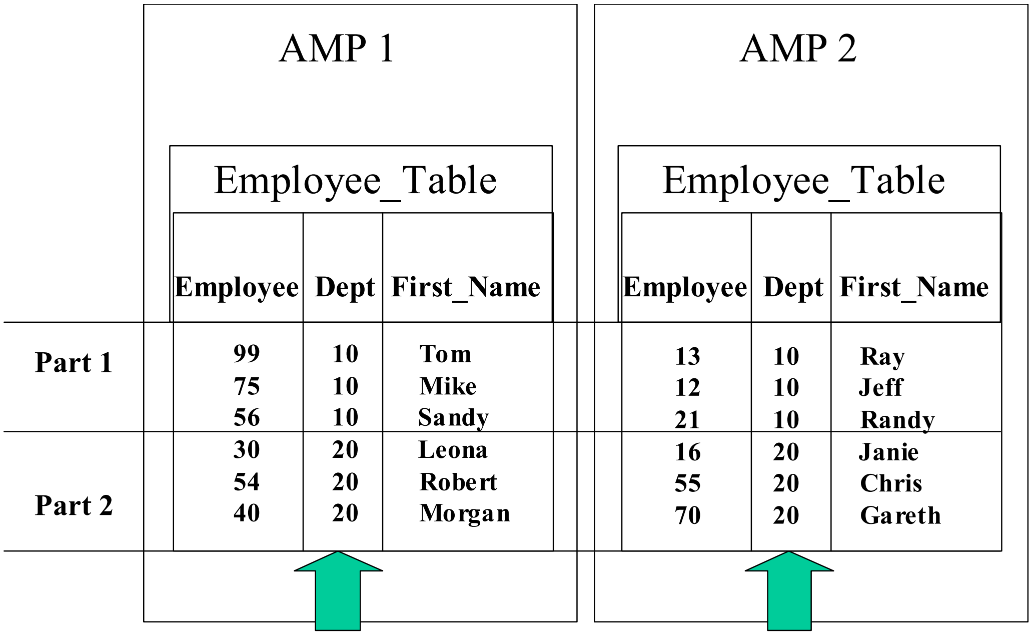
```
SELECT *
FROM Employee_Table
WHERE Dept = 20;
```

Answer: 1

Partition Primary Indexes reduce the number of rows that are processed by using partition elimination.

The Bad NEWS about Partitioning on a column that is not part of the Primary Index

Before you get too excited about partitioning by a column that is not part of the primary index you should remember “The Alamo”. This is because when you run queries that don’t mention the PARTITION COLUMN in your SQL you have to check every partition and this can be some serious battle. Partitions can range from 1-65,535 partitions. The example below will have to check every partition so be careful.



AMP 1

Employee_Table

	Employee	Dept	First_Name
Part 1	99	10	Tom
	75	10	Mike
	56	10	Sandy
Part 2	30	20	Leona
	54	20	Robert
	40	20	Morgan

AMP 2

Employee_Table

	Employee	Dept	First_Name
Part 1	13	10	Ray
	12	10	Jeff
	21	10	Randy
Part 2	16	20	Janie
	55	20	Chris
	70	20	Gareth

Partition Primary Index is Dept

```
SELECT *
FROM Employee_Table
WHERE employee = 99
```

GREAT Things about Partition Primary Indexes:

PPI avoids full table scans without the overhead of a secondary index and allows for instantaneous dropping of old data and rapid addition of newer data.

Remember these rules: A Primary KEY can’t be changed. A Primary Index always distributes the data. A Partition Primary Index (PPI) partitions data to avoid full table scans.

Two ways to handle Partitioning on a column that is not part of the Primary Index

You have two ways to handle queries when you partition by a column that is not part of the Primary Index.

a. You can assign a **Unique Secondary Index (when appropriate).**
b. You can include the **partition column in your SQL.**

Example 1:

```
CREATE UNIQUE INDEX (Employee)
On Employee_Table
```

```
SELECT *
FROM Employee_Table
WHERE employee = 99
```

Example 2:

```
SELECT *
FROM Employee_Table
WHERE employee = 99
AND Dept = 10;
```

↑ Partition is Dept

In all examples above only one partition would need to be read.

Partitioning with CASE_N

"You Cracked the Case Honey"

Vinny – My cousin Vinny

Teradata now allows you to crack the CASE statement as a partitioning option. Here are the fundamentals:

Use of CASE_N results in:

- Just like the CASE statement it evaluates a list of conditions picking only the first condition met.
- The data row will be placed into a partition associated with that condition.

```
CREATE TABLE Order_Table
(
Order_Number       Integer   NOT NULL
,Customer_Number Integer   NOT NULL
,Order_Date           Date
,Order_Total          Decimal (10,2)
)
PRIMARY INDEX(Customer_Number)
PARTITION BY CASE_N
   (Order_Total < 1000
    ,Order_Total < 5000
    ,Order_Total < 10000
    ,Order_Total < 50000, NO Case, Unknown
);
```

Note: We can't have a Unique Primary Index (UPI) here because we are partitioning by ORDER_TOTAL and ORDER_TOTAL is not part of the Primary Index.

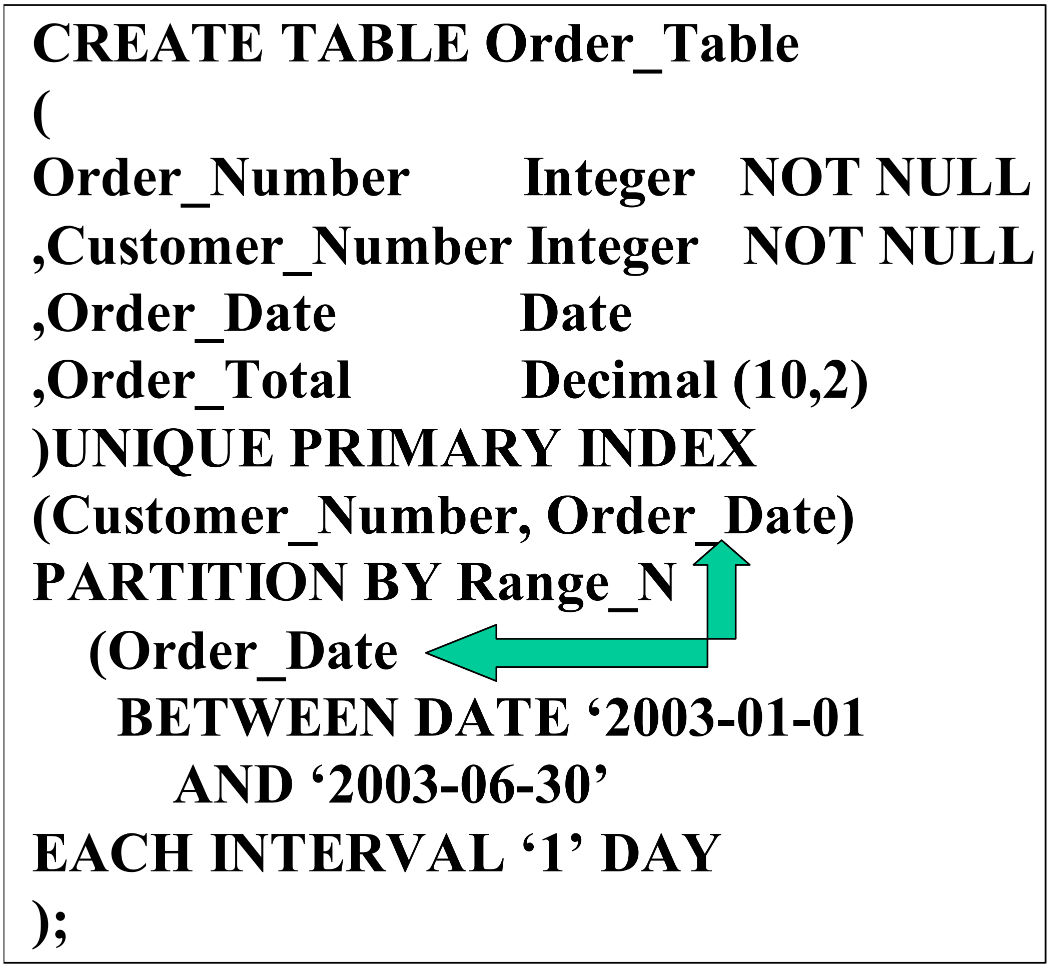
A CREATE Table statement will dictate whether or not a table is Set or Multi-Set and if it is a PPI or NPPI table. You have options also available to **override** system level specifications such as **Datablock Size** and **Freespace Percent**.

Partitioning with RANGE_N

Teradata also has a western theme because they allow your partitions to go “Home on the Range” by using the RANGE_N function. Here are the fundamentals. Use of RANGE_N results in:

- The expression is evaluated and associated to one of a list of ranges.
- Ranges are always listed in increasing order and can’t overlap
- The data row is placed into the partition that falls within the associated range.
- The test value in RANGE_N function must be an INTEGER or DATE. (This includes BYTEINT or SMALLINT)

In the example below please notice the arrows. They are designed to illustrate that you can use a UNIQUE PRIMARY INDEX on a Partitioned table when the Partition is part of the PRIMARY INDEX.



```
CREATE TABLE Order_Table
(
Order_Number       Integer   NOT NULL
,Customer_Number Integer   NOT NULL
,Order_Date          Date
,Order_Total          Decimal (10,2)
)UNIQUE PRIMARY INDEX
(Customer_Number, Order_Date)
PARTITION BY Range_N
   (Order_Date
     BETWEEN DATE ‘2003-01-01
        AND ‘2003-06-30’
EACH INTERVAL ‘1’ DAY
);
```

NO CASE, NO RANGE, or UNKNOWN

"We only have one person to blame, and that's each other"

Barry Beck, NY Ranger, on who started a fight during a hockey game

In no case does no range have anything to do with a New York Ranger, but if you specify NO Case, NO Range, or Unknown with partitioning their will be no fighting amongst the partitions. These keywords tell Teradata which penalty box or partition to place bad data.

A NO CASE or NO RANGE partition is for any value, which isn't true for any previous CASE_N or RANGE_N expression.

If UNKNOWN is included as part of the NO CASE or NO RANGE option with an OR condition, then any values that are not true for any previous CASE_N or RANGE_N expression and any unknown (e.g., NULL) will be put in the same partition. This example has a total of 5 partitions.

```
   Partition by CASE_N
           (  Salary < 30000,
              Salary < 50000,
              Salary < 100000
              Salary < 1000000,
              NO CASE OR UNKNOWN)
```

IF you don't see the OR operand associated with UNKNOWN then NULLs will be placed in the UNKNOWN Partition and all other rows that don't meet the CASE criteria will be placed in the NO CASE partition. This example has a total of 6 partitions.

```
Partition by CASE_N
           (  Salary < 30000,
              Salary < 50000,
              Salary < 100000
              Salary < 1000000,
              NO CASE, UNKNOWN)
```

Partitioning and Joins

"Outside of the killings, Washington has one of the lowest crime rates in the country"

Marion Barry, Mayor, Washington, D.C.

Teradata is so much better then other databases it is almost a crime. Teradata has the fastest join rates in the world. Teradata has always been the leader in data warehousing and one reason is because of their brilliant handling of joins. Teradata can join up to 64 tables in one query and do so with amazing speed and accuracy.

For **two rows to be joined** together, they must **physically reside on the same AMP**. Quite often they don't, so Teradata will either redistribute a table by hashing the joining column or it will duplicate the smaller table on all AMPs. In either case, the result is that the joining rows are on the same AMP even though they might be in spool temporarily. When two tables have the same primary index and the tables being joined are joined on that primary index, then no data needs to move because the joining rows are already on the same AMP.

Joining PPI tables and NPPI tables make this process a little more difficult. Teradata still has to make sure that two joining rows reside on the same AMP and here's how they accomplish it:

- It will redistribute one or both tables by hash code of the WHERE CLAUSE columns
- It will duplicate the smaller table across all the AMPs
- It will convert a PPI table to a NPPI table
- It will convert a NPPI table to a PPI table
- It will use a moving window to check out each partition

The way to speed this process up is through proper design. Design partition tables so that the most common **WHERE** conditions **eliminate** the **most partitions**. You can also choose **primary index columns** that will **minimize row redistributions**. The area that will take the most time and cost the system the most resources are when there is a join between an NPPI and PPI table with a large number of surviving partitions after constraints are applied. The result will be that **Disc I/Os** are **higher** and your **CPU utilization** will sky rocket **higher**.

Chapter 4 — Teradata Engine – Under the Hood

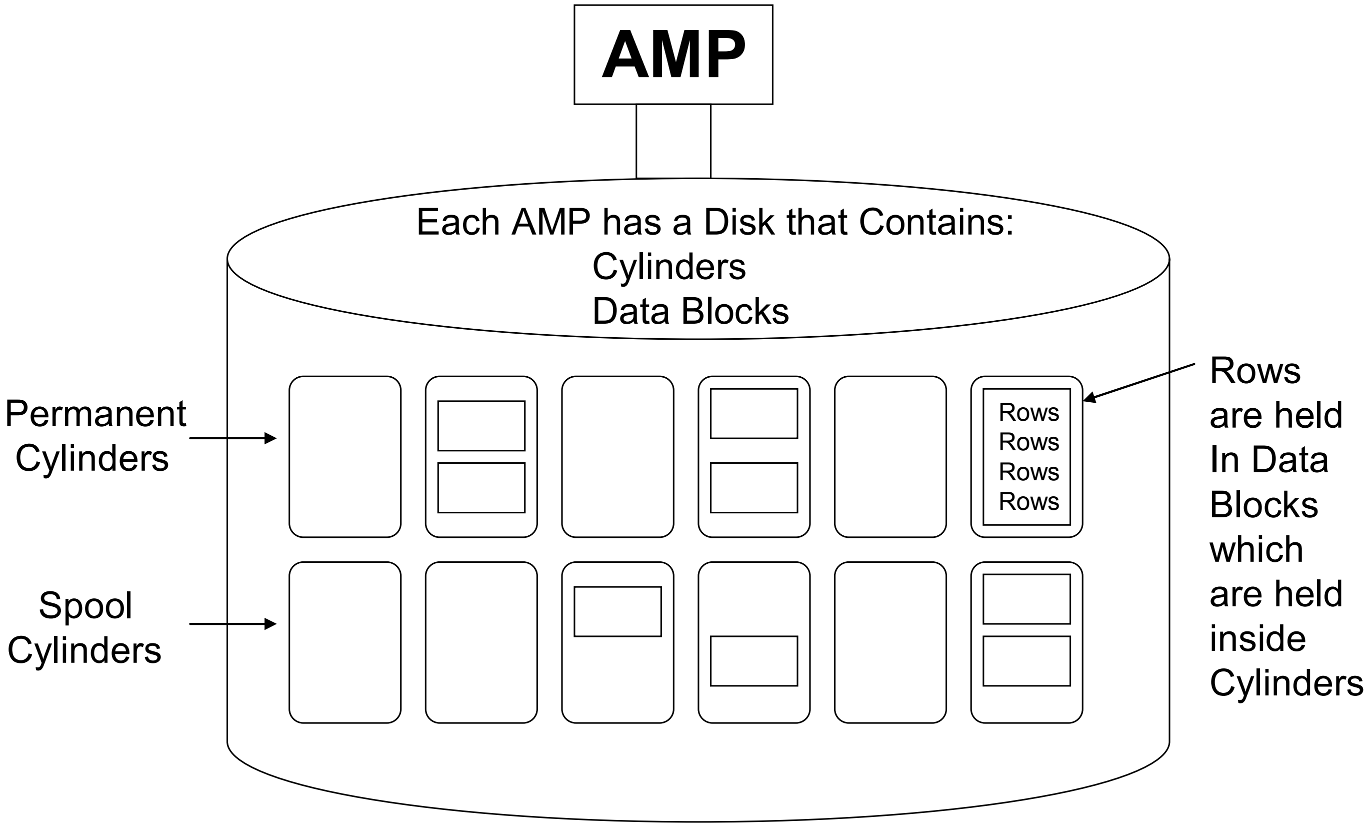
"You always pass failure on your way to success"

– Mickey Rooney 1920

Teradata is a big Mickey Rooney fan because failure with Teradata is never an option. Each AMP is attached to a virtual disk. This is obviously where each AMP stores their tables which consist of data rows. Each AMP disk is made up of thousands of cylinders. Inside the cylinders are where the data blocks are stored. Inside the data blocks are where the rows reside.

When an AMP wants to retrieve data from a particular table the Parsing Engine (PE) converts the table name to the Table ID. The AMP then uses the Master Index like a phone book to look up which cylinder or cylinders Chapter 3 — V2R5 Partition Primary Indexes the data blocks needed reside. Then the AMP uses the Cylinder Index to find the exact blocks that contain the needed rows.

Cylinders are broken down to either Permanent Table Cylinders or Spool Cylinders. Once a cylinder contains rows from a Permanent Table then that cylinder cannot be used for spool. Spool Cylinders, in other words are unused Permanent Cylinders.


AMP
Each AMP has a Disk that Contains:
Cylinders
Data Blocks
Permanent Cylinders
Spool Cylinders
Rows
Rows
Rows
Rows
Rows are held In Data Blocks which are held inside Cylinders

Full Cylinder Read

"Common sense is the collection of prejudices acquired by age eighteen."

– Albert Einstein

Teradata has the ability for what is called a Full Cylinder Read. A Full Cylinder Read allows for all the blocks in an entire cylinder to be read with a single I/O operation. Teradata used to require that each block in a cylinder be read with a separate I/O operation, but since V2R5 Teradata can read the entire cylinder on certain relevant queries. The best queries that can take advantage of a Full Cylinder Read are FULL TABLE SCAN operations utilizing: Aggregations, Large Table SELECTS, Merge INSERT/SELECT, Merge DELETE, INNER JOINS and OUTER JOINS.

The Full Cylinder Read is not always ON or automatic, but is set by the Teradata Database Customer Support group when configuring the machine by setting the CR FLAG to ON. Each AMP is normally allocated at least 40 MB of memory. If FSG Cache memory per AMP is below 32 MB then Full Cylinder Reads are automatically disabled. Full Cylinder Reads are allocated with memory being allocated to cylinder slots and these slots are used for the Full Cylinder Reads only. A USER can select between 2 and 40 cylslots per AMP and the more slots the faster the Full Cylinder Read.

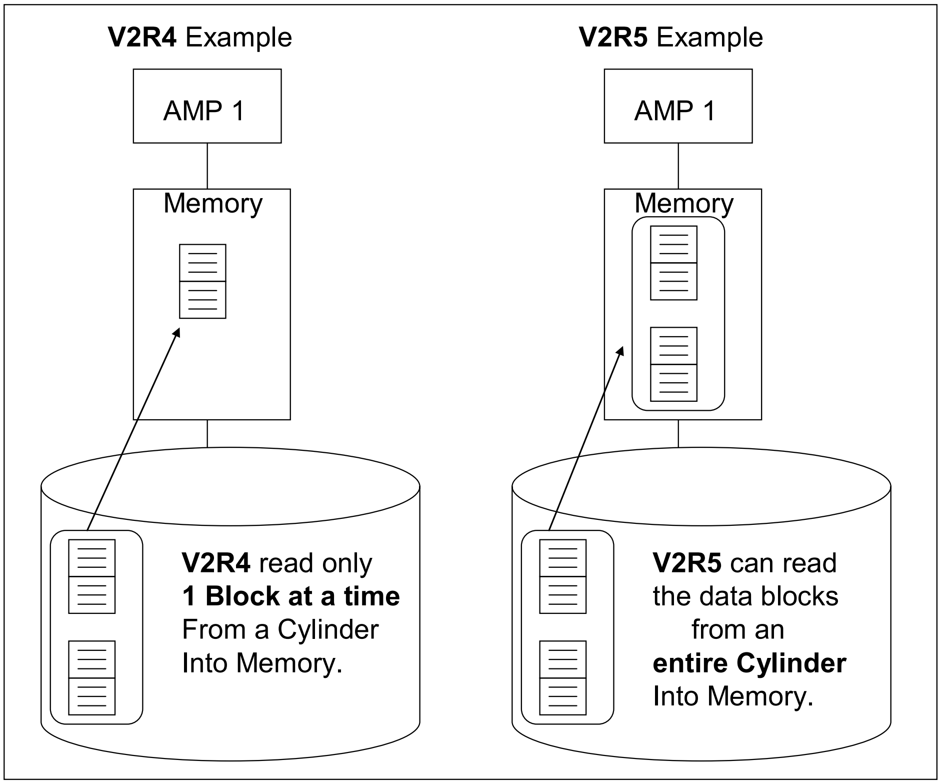


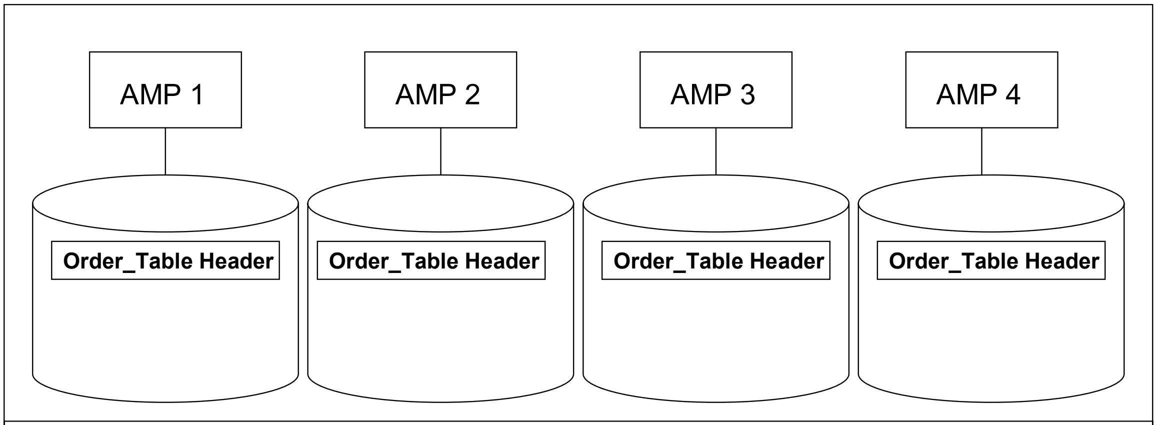
Table Header

"We hang the petty thieves and appoint the great ones to public office."

– Aesop 550 BC

When a table is created the first thing Teradata does is create a Table Header on each AMP. This header contains the column names, their data types, and compression information. Even though no rows have been added yet, the table still exists. When rows are added Teradata will create data blocks to store the rows. The Table Header is kept separate of the data blocks physically, but logically are connected.

If compression is used on a column in the table then each compressed value for a column is kept in the table header. In the actual data rows a bit is flipped from a 0 to a 1 at the beginning of the row which tells Teradata what value actually should reside in the data row. The combination of storing the compressed values in the Table Header and representing which value should be present by flipping a bit in the actual row allow for compression of values.



When a Table is Created a Table Header is created on each AMP. The Header knows the columns, their data types, and holds compression information.

Each Table is given a Table ID

> *"Nearly all men can stand adversity, but if you want to test a man's character, give him power."*
>
> **– Abraham Lincoln**

Teradata tracks every object by giving it a table ID. This comes from the Data Dictionary table called DBC.NEXT. The DBC.NEXT table is assigned to increment by 1 providing Teradata objects with the NEXT Table ID. When you write SQL accessing a particular table then Teradata converts the table name to its associated TABLE ID.

Remember, that each Teradata object is given a globally unique Teradata Table ID and these objects include: Databases, Users, Tables, Roles, Profiles, Views, Macros, Triggers, Join Indexes, Hash Indexes, and Stored Procedures. Each column and index within a table is assigned a unique numeric ID within its Table ID.

The Table ID is a 48-bit number and it consists of two major components. The first part of the 48-bit Table ID is the Unique Value given directly from DBC.Next. This portion of the number will also help Teradata track the table, which allows Teradata to know if the table is a Permanent Table, a Spool Table or a Permanent Journal Table.

The second part of the Table ID is a 16-bit Subtable ID that will tell Teradata if this portion of the table block is a Teradata Header, a Primary Data Row, a Fallback Data Row, or a secondary index block.

The actual Table ID and Row ID combined will allow Teradata to uniquely define every single row in an entire Teradata system.

Table ID 48 bits

32-Bit Number from DBC.NEXT	16-Bit Number describing the Block
Permanent Table	Table Header
Spool Table	Primary Data Rows
Permanent Journal	Fallback Data Rows
Global Temp Table	Secondary Index Rows

How Data Blocks are Dynamically Built

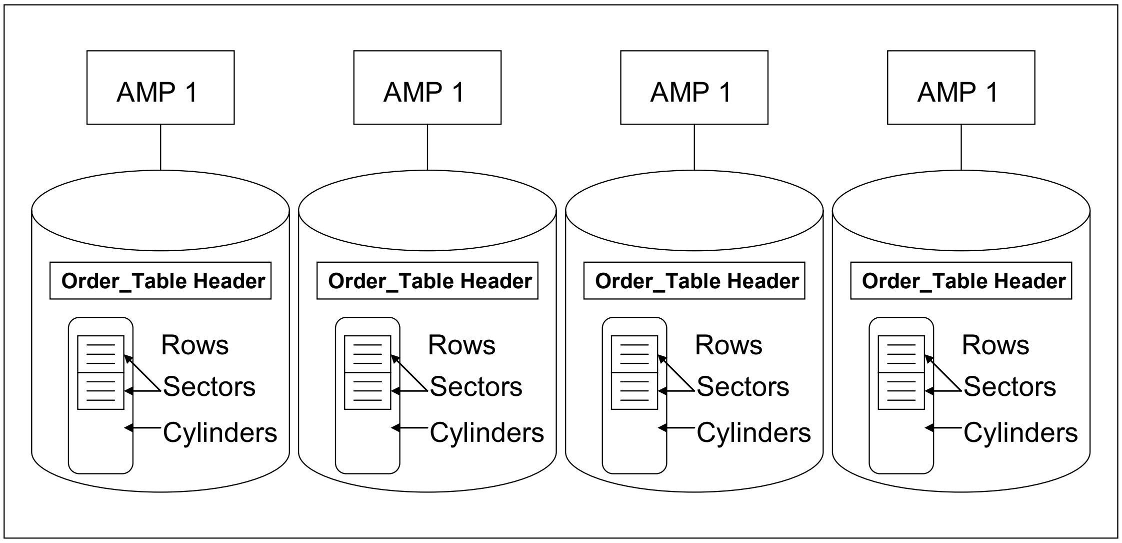
"Men are what mothers made them."

– Ralph Waldo Emerson

Once a table has been created a Table Header is created on each AMP. This Table Header basically contains the columns and their data types. When data rows begin being loaded or inserted each AMP allocates at least a 512 byte sector inside a disk cylinder. This 512 byte sector is considered a block of data. As more and more rows are added and the first 512 byte sector is filled the AMP will allocate another 512 byte sector. The two sectors now contain 1024 bytes and this is still considered one block of data. Teradata will continue to add sectors as needed until the block reaches the maximum block size. At that time Teradata will perform a block split.

This is a brilliant design because the Database Administrator does not have to allocate space for a table. Most databases require the DBA allocate and calculate the space for each table. This is a nightmare because many data warehouses can contain thousands of tables. Teradata was designed like the human body. As the body grows new cells are created. As the table grows new sectors are allocated.

The DBA can set the minimum sector allocation size so that when data is initially loaded or inserted multiple sectors can be allocated. When new sectors are needed then multiple sectors can be added instead of just one at a time. This can save time for very large tables.



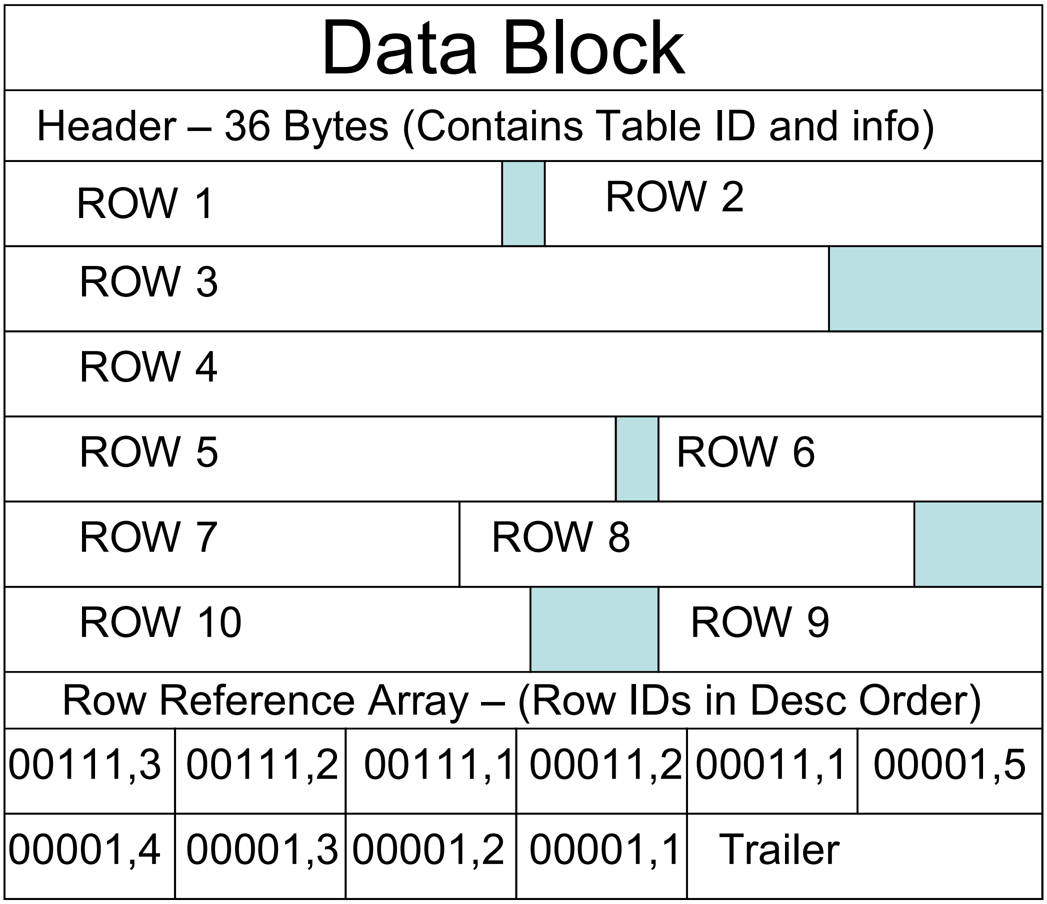
Cylinders hold disk sectors. One to many contiguous disk sectors make up a block of data. Rows are held inside blocks. Teradata allocates a 512 byte sector and when it fills another sector is added. When the maximum block size is reached Teradata splits the block.

Data Blocks

"A professional is someone who can do their best work when they don't feel like it."

– Johann Christoph Freidrich Von Schiller

A table might consist of several data blocks, but Teradata never shares a data block with multiple tables. When Teradata loads data the rows are placed in Row ID order. When inserts add a row to a table the row is added to the end of the data block. The Row Reference Array is always sorted by Row ID in Descending Order and it points to the exact location of the row. This allows Teradata to add rows with great speed, yet still allows for them to be sorted by Row ID. This allows Teradata to do a binary search when trying to locate a single row. A binary search is like trying to name a song in 7 notes. Teradata can go to the middle of the Row Reference Array and say, "Are you the row I am looking for?" The answer might be "Too Low". Then Teradata goes to the middle of the lower Row IDs and ask, "Are you the row I am looking for?" This game of High-Low continues until the row is found. This is magnitudes of order faster then scanning the entire block for the row.



How Teradata Finds a Row of Data

“Nobody forgets where they buried the hatchet”

– Frank McKinney “Kin” Hubbard

Teradata never has to bury the hatchet because it loves to chop through data. Teradata accesses data rows quickly by utilizing the Master Index to find the Cylinder. Then it uses the Cylinder Index to find the Data Block. It can then search the data block via a binary search to find a single row or easily scan an entire block with a cylinder or block read if necessary.

The PE comes up with a PLAN and sends a message to the AMP via the message passing layer containing

Table ID	Row Hash	Primary Index Value

The AMP accesses its **MASTER INDEX** in order to locate the **CYLINDER**.

The AMP accesses its **Cylinder INDEX** in order to locate the **Data Block**.

The AMP searches the **Data Block** with a binary search In order to locate the **Row.**

The **Master Index** is always memory resident. The **Cylinder Index** and **Data Blocks** are accessed via an AMP’s **FSG Cache** and may or may not be in memory.

The Master Index

"Rudeness is the weak person's imitation of strength"

– Eric Hoffer

Teradata never needs to be rude because it has strength in organizing data. Each AMP has a Master Index that is organized in alphabetical order. The Master Index knows the Table ID and tracks the Partition, Lowest Row ID and the Highest Partition and Row Hash. This allows an AMP to scan the Master Index just like you might a phone book. Just like a phone book would eventually provide you a number the Master Index provides an AMP with the cylinder or cylinders that contain the data block(s). This might now give you a better understanding as to why an AMP keeps its tables in separate data blocks and why it sorts those data blocks by Row ID. Teradata feels the need for speed when locating rows and the organization done up front will allow Teradata to fly.

The Master Index locates the Cylinder

LOWEST			HIGHEST			
Table ID	Part #	Row ID	Table ID	Part #	Row Hash	Cylinder #
1	0	00001, 1	1	0	00011, 1	10
2	0	00011, 1	2	0	01011, 1	15
22	0	00111, 1	22	0	01111, 2	31
100	0	00011, 1	100	0	00111, 1	40
100	0	00111, 2	100	0	01111, 1	41
100	0	01111, 2	100	0	11111, 1	42
200	1	00001, 1	200	12	00111, 1	66
200	13	01111, 1	200	24	10111, 1	70

Free Cylinder List

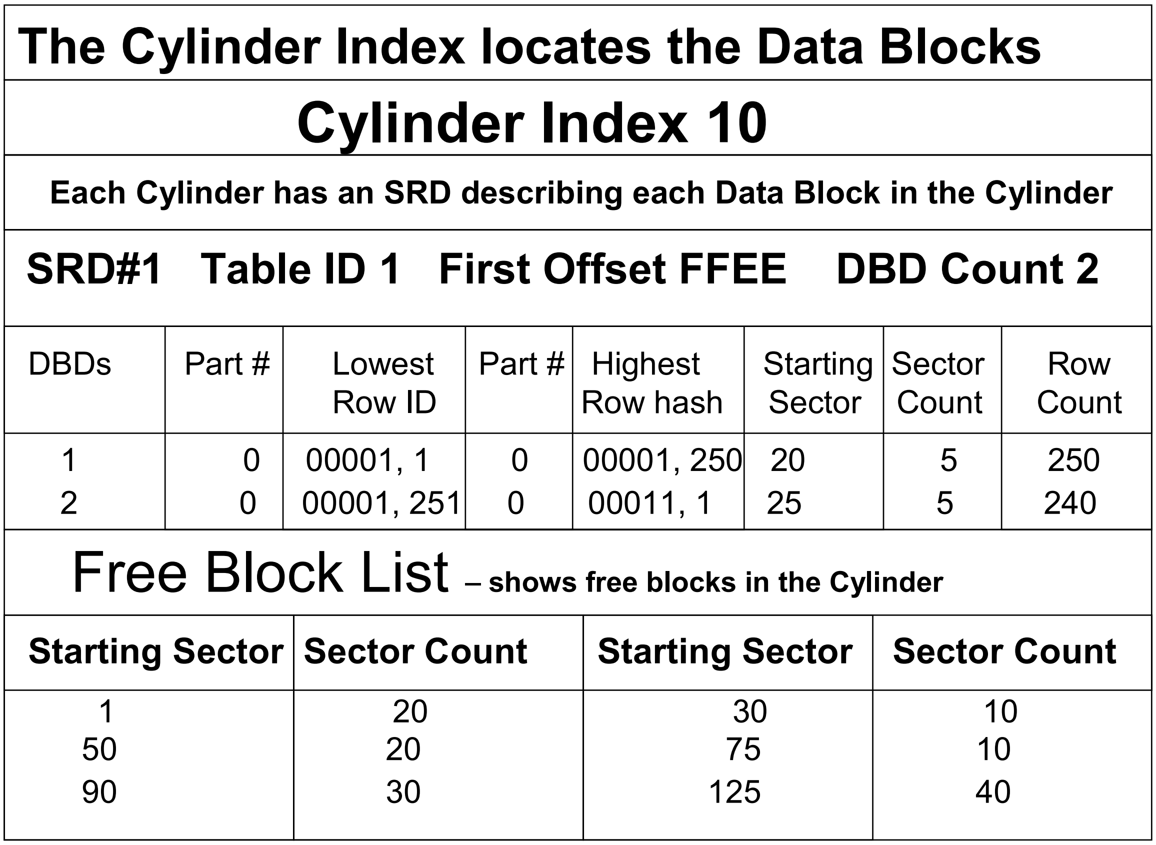
1, 2, 3, 4, 5, 6, 7, 8, 9,11, 12, 13, 14, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26 27, 28, 29, 30, 32,33, 34,35, 36,37, 38, 39, 43, 44, 45, 46, etc.

The Cylinder Index

"To escape criticism – do nothing, say nothing, be nothing."

– Elbert Hubbard

Teradata has had critics for over a decade, but now even the Gartner Group rates them as the number one data warehouse database. Each AMP has a Cylinder Index that is organized again in alphabetical order. The Cylinder Index knows the Table ID and tracks the Partition, Lowest Row ID and the Highest Partition and Row Hash. This allows an AMP to scan the Cylinder Index exactly like a phone book, but only better. Just like a phone book would eventually provide you a number the Cylinder Index provides an AMP with the data blocks and how many rows are inside each data block. This might now give you a better understanding as to why an AMP keeps its tables in separate data blocks and why it sorts those data blocks by Row ID. Each Cylinder Index is used to build the Master Index when the Teradata system boots up or has to re-boot. The combination of Master Index and Cylinder Index allow each AMP to find the proper Cylinder and the Data Blocks and corresponding rows inside that Cylinder.



The Cylinder Index locates the Data Blocks

Cylinder Index 10

Each Cylinder has an SRD describing each Data Block in the Cylinder

SRD#1 Table ID 1 First Offset FFEE DBD Count 2

DBDs	Part #	Lowest Row ID	Part #	Highest Row hash	Starting Sector	Sector Count	Row Count
1	0	00001, 1	0	00001, 250	20	5	250
2	0	00001, 251	0	00011, 1	25	5	240

Free Block List **– shows free blocks in the Cylinder**

Starting Sector	Sector Count	Starting Sector	Sector Count
1	20	30	10
50	20	75	10
90	30	125	40

Cylinder Index Changes

"When I was 14 I thought my parents were the stupidest people in the world. When I was 21 I was amazed at how much they learned in seven years."

– Mark Twain

With Teradata V2R4 and below there was only room for one Cylinder Index at the beginning of each cylinder. With V2R5, space for 2 Cylinder Indexes was allocated at the beginning of every cylinder. Teradata will choose to use both cylinder indexes based on the operating system in place with Teradata V2R5.1 and beyond.

When does a Cylinder Index change? When a current data block inserts a row that changes the Lowest Row ID or the Highest Row Hash. The Cylinder Index also changes when a new Data Block is written inside the Cylinder.

When does a Master Index change? Any time a Cylinder Index is updated.

UNIX MP-RAS – Only 1 Cylinder Index is used. When a Cylinder Index needs to be updated it is updated in place and also written to a buddy node.

Windows 2000 Systems – Teradata V2R5.1 and beyond use two Cylinder Indexes. When a Cylinder Index is updated the changes are written to the "Alternate Cylinder Index". This provides for better performance and Integrity.

How Teradata Writes to an AMP

> *"Oh, what a bitter thing it is to look into happiness through another man's eyes."*
>
> **– William Shakespeare**

Teradata will write data to a data block if the additional row can fit. If the row will not fit into the data block then Teradata will find a bigger block. This series of pages will give you an excellent idea of the options Teradata takes to make the write happen.

Read the data block into the AMP's Memory

Make the changes to the data block in memory and calculate the new block length.

If V2R5.1 or above and CHECKSUM is enabled for the table then allocate a new Data Block automatically.

If V2R5.1 or above and CHECKSUM = NONE and the New Block length = Old Block length:

> **UNIX MP-RAS systems – Update the Data Block from memory and the Cylinder Index if it has changed.**
>
> **For NON-UNIX MP-RAS systems – Allocate a new Data Block if the Cylinder Index will change.**

If CHECKSUM = NONE and the NEW Block length is different then the Old Block length then write the new Data Block and update the Cylinder Index. Place the Old Data Block on the FREE BLOCK LIST.

Writing to Data Blocks of Equal Length

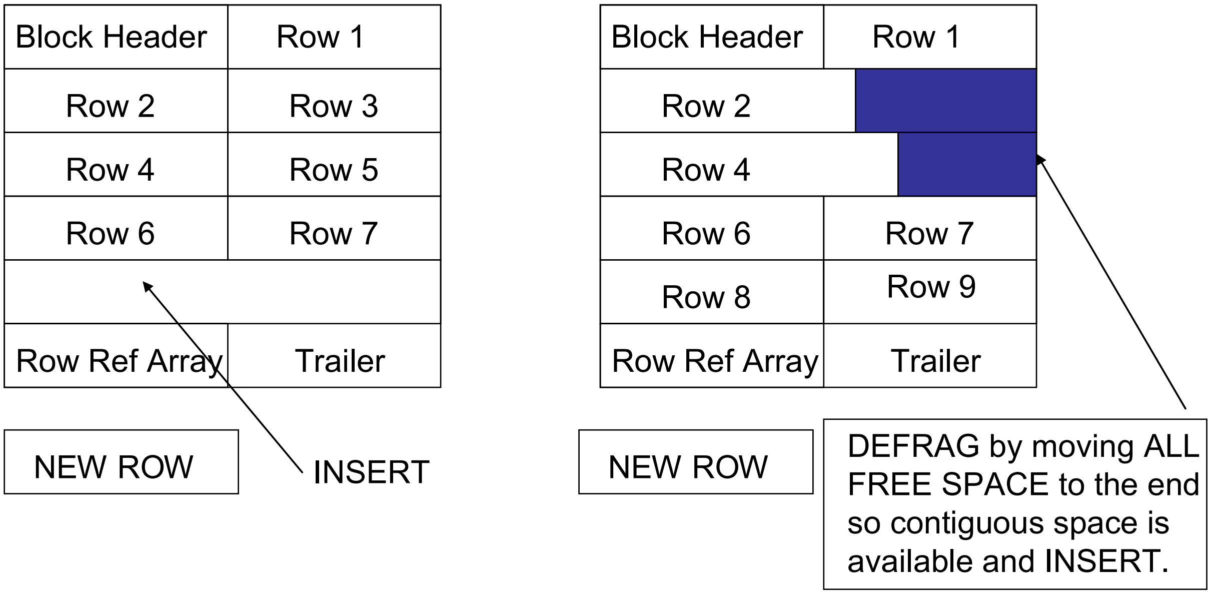
"To be or not to be – that is the question."

– William Shakespeare

The real question is to Defrag or not to Defrag. In our examples below we see two different rows being inserted into two different blocks. Both blocks have enough room, but the second example does not have contiguous space. In the second case Teradata will perform a Defrag on the block. A Defrag will move the rows up in the block with the goal of getting contiguous space at the end of the block. The row can then be inserted.

In the first example there is contiguous space so the row is easily inserted.

Both Data Blocks have enough space for their new rows, but the second example does not have enough contiguous space.



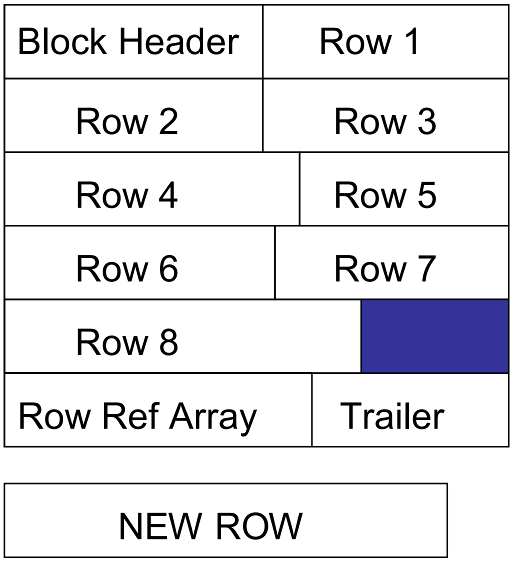
DEFRAG by moving ALL FREE SPACE to the end so contiguous space is available and INSERT.

When a Data Block is not Big Enough for a Write

"It is better to be deceived by one's friends than to deceive them."

– Johann Wolfgang von Goethe

When Teradata is writing to an existing data block and there is not enough room for the new row then Teradata will update the row in memory and calculate the new length. Then Teradata will look on the Free Block List to find a block of contiguous sectors large enough to house the block. The New Block will be written and the Old Block placed on the Free Block List.



Because the NEW ROW will not fit
In the current BLOCK, Teradata will
Perform the INSERT in memory. It
will determine the new length needed
And look on the FREE BLOCK LIST
for contiguous sectors large enough
to house the WRITE. It will then
WRITE the Data Block to its new home.

This is like adding a new member to your
family and now needing a bigger house.
Too bad there isn't a FREE HOUSE LIST.

How Teradata Allocates Blocks

"Quarrels would not last long if the fault were only on one side."

– Francois, Duc de La Rochefoucauld

When data is loaded or inserted the Teradata AMP allocates a 512-byte sector. Once the sector is filled then Teradata allocates another 512-byte sector and the block is considered 1024 in size. This adding of sectors on an as needed basis will continue until the block reaches what is called the Largest Block. At that point in time there is a block split and now the table on the particular AMP has two blocks holding the table rows. This theory continues and a single table can consist of hundreds if not thousands of blocks on each AMP.

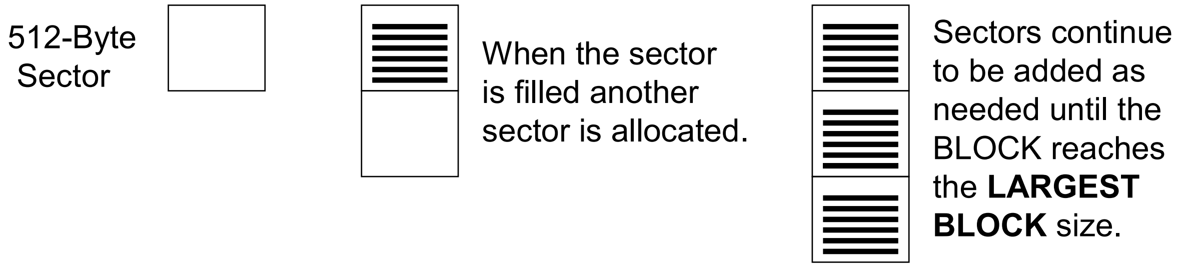
When Teradata loads the first row to an AMP the AMP allocates a single 512-byte sector and rows are loaded until the sector is filled.

Once the first sector is filled then Teradata allocates a second 512 byte Sector and rows continue to load until that sector is also filled.

Teradata will continue to allocate an additional sector until the block reaches the LARGEST BLOCK size.

Once the LARGEST BLOCK size is reached then Teradata will do a BLOCK SPLIT.

Now rows can be loaded to both blocks until one reaches the LARGEST BLOCK size and then another BLOCK SPLIT occurs.



512-Byte Sector

When the sector is filled another sector is allocated.

Sectors continue to be added as needed until the BLOCK reaches the **LARGEST BLOCK** size.

Block and Row Definitions

"I am not young enough to know everything."

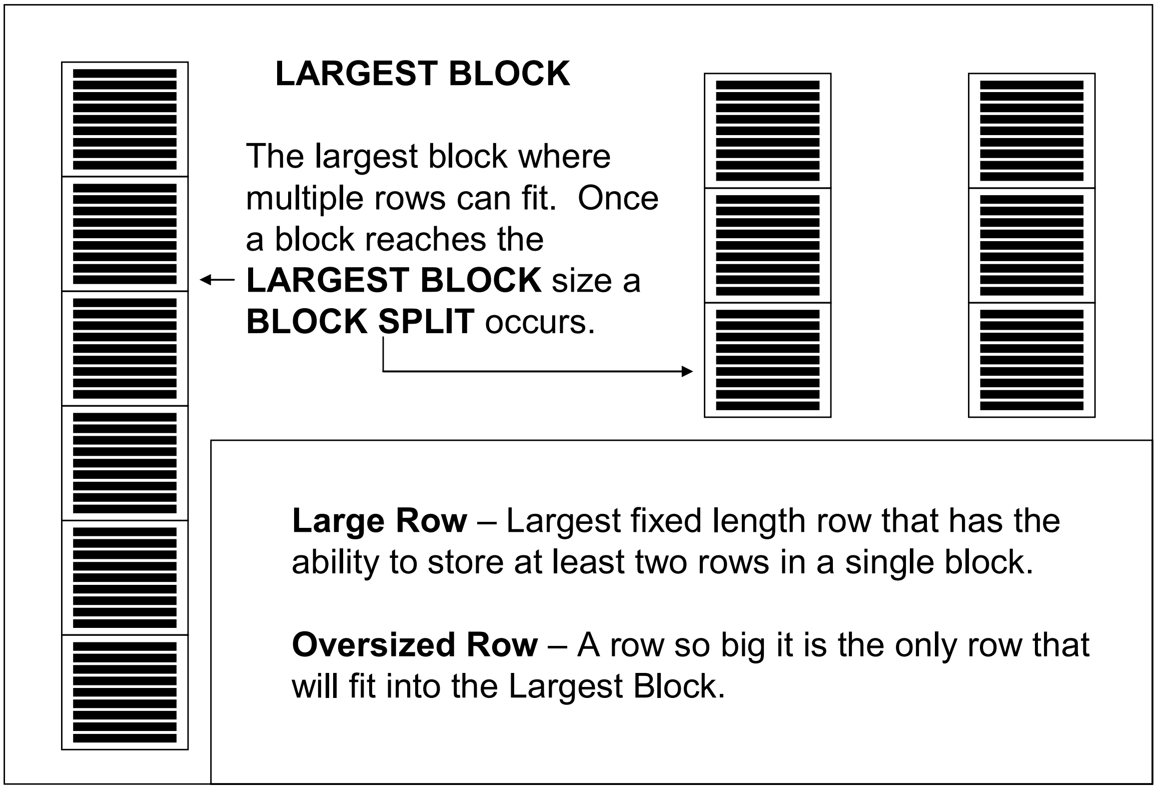
– Oscar Wilde

There are three terms that you should know and those are Largest Block, Large Row, and Oversized Row.

Largest Block – This is the Largest Multi-Row Block Allowed. This usually defaults to the size set in the PermDBSize. This system default can be changed with the DBSControl Utility. Users creating a table can also set this parameter at the table level by using the keyword DATABLOCKSIZE when creating the table.

Large Row – This is the largest fixed length row that has the ability to store at least two rows in a single block.

Oversized Row – This is a row so big that it is the only row that will fit in the block. Every time this row is read or updated it takes an entire I/O for just that row. In other words the entire block must be read into memory just to access this single row. This is an expensive read.



LARGEST BLOCK

The largest block where multiple rows can fit. Once a block reaches the **LARGEST BLOCK** size a **BLOCK SPLIT** occurs.

Large Row – Largest fixed length row that has the ability to store at least two rows in a single block.

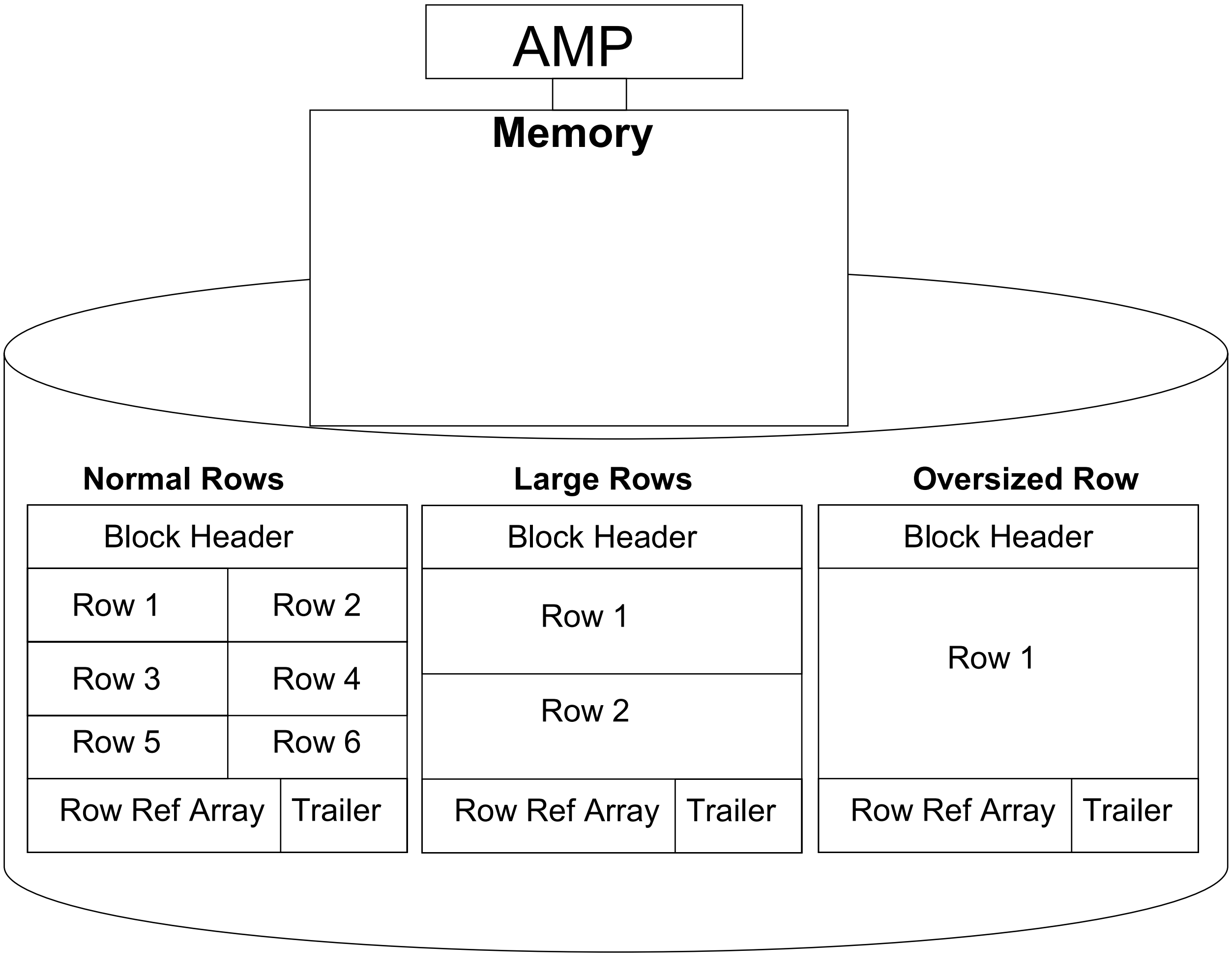
Oversized Row – A row so big it is the only row that will fit into the Largest Block.

Large Row versus Oversized Row

"The distance doesn't matter; it is only the first step that is difficult."

– Marie de Vichy-Chamrond

The terms Large Row and Oversized row can be confusing so let me explain. A large row is still small enough that two large rows can still fit inside a block that is less than the Largest Block Size. An Oversized Row is a row that is so large that only that row can fit inside the Largest Block Size. An Oversized Row is a problem because when ever it is read or updated an entire block is placed into memory (a single I/O) for the mere purpose of accessing a single row. The great thing about blocks are that most often many rows fit inside a block and when the rows need to be read or updated the block is placed into memory and then many rows are accessed nearly simultaneously.



Defragmentation

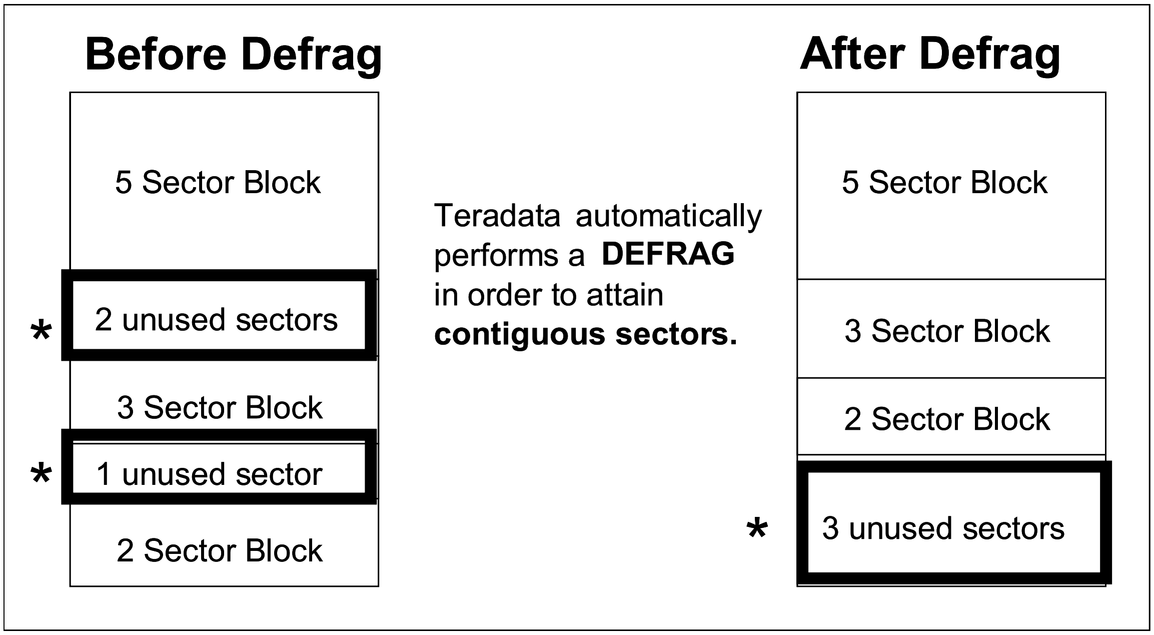
"All glory comes from daring to begin."

– Anonymous

Teradata does not ever need reorganizations because Teradata collects free blocks whenever the system is not busy. This collection is called a Defrag. The concept of a Defrag is shown below. The idea is to collect free sectors in a cylinder and place these free sectors together at the bottom of the cylinder. This is like taking several unused rooms from many different houses and combining them together to form one big house. This big house can now be the home of a large family.

Teradata actually accomplishes attaining contiguous sectors by performing an actual Defrag and collecting unused sectors an placing them at the bottom of the cylinder or by another technique involving adjacent sectors. When a sector is freed up and that freed sector is adjacent to other free sectors, then these sectors are combined into one entry on the free block list.

When a new row or an updated row causes a block to grow by one or more sectors the system does not look to see if the next sector is available. It merely looks on the free block list in the cylinder index for a group of contiguous free sectors that is large enough to house the block. Think of this as when your family grows and you need a bigger house merely moving instead of adding a room to your existing house.



When a Cylinder becomes Full

"Only dead fish swim with the stream."

– Anonymous

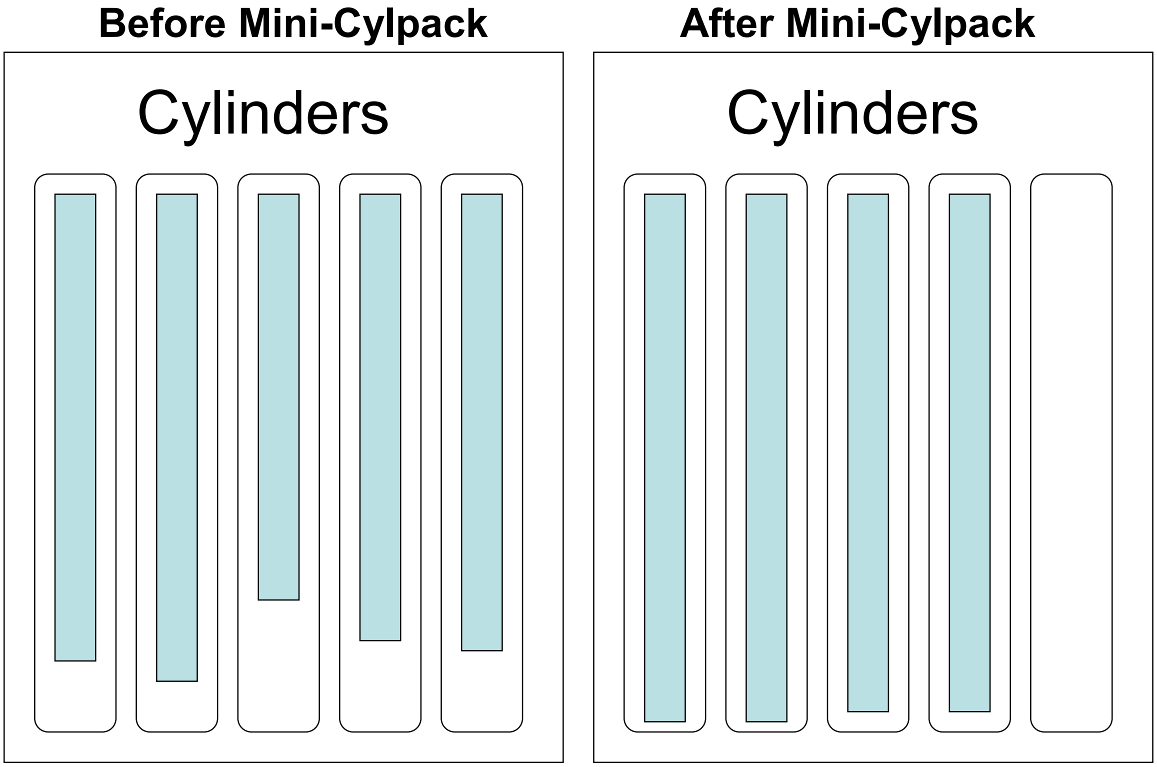
A Cylinder Full condition happens when there is no block on the Free Block List that has enough contiguous sectors big enough to house the new block on an INSERT or UPDATE. Teradata goes through up to four steps to fix the condition and they are listed in the order Teradata attempts to perform them as:

(1) Defragmentation – Collect all Free Blocks in the Cylinder and move them to the bottom of the cylinder by moving all data blocks towards the top of the cylinder.

(2) Cylinder Migrate – Checks the logically adjacent cylinders to the left and right and if room is available then up to 10 data blocks are moved to the adjacent cylinder.

(3) Cylinder Split – Checks for a free cylinder and moves a maximum of 20 data blocks to the newly allocated cylinder. This is now considered a Permanent Cylinder.

(4) Mini-Cylpack – The goal of a mini-cylpack is to free up a single cylinder. This is accomplished by moving data blocks continually to the top of the first cylinder and then the second cylinder, and then the next cylinder until a single cylinder is empty.



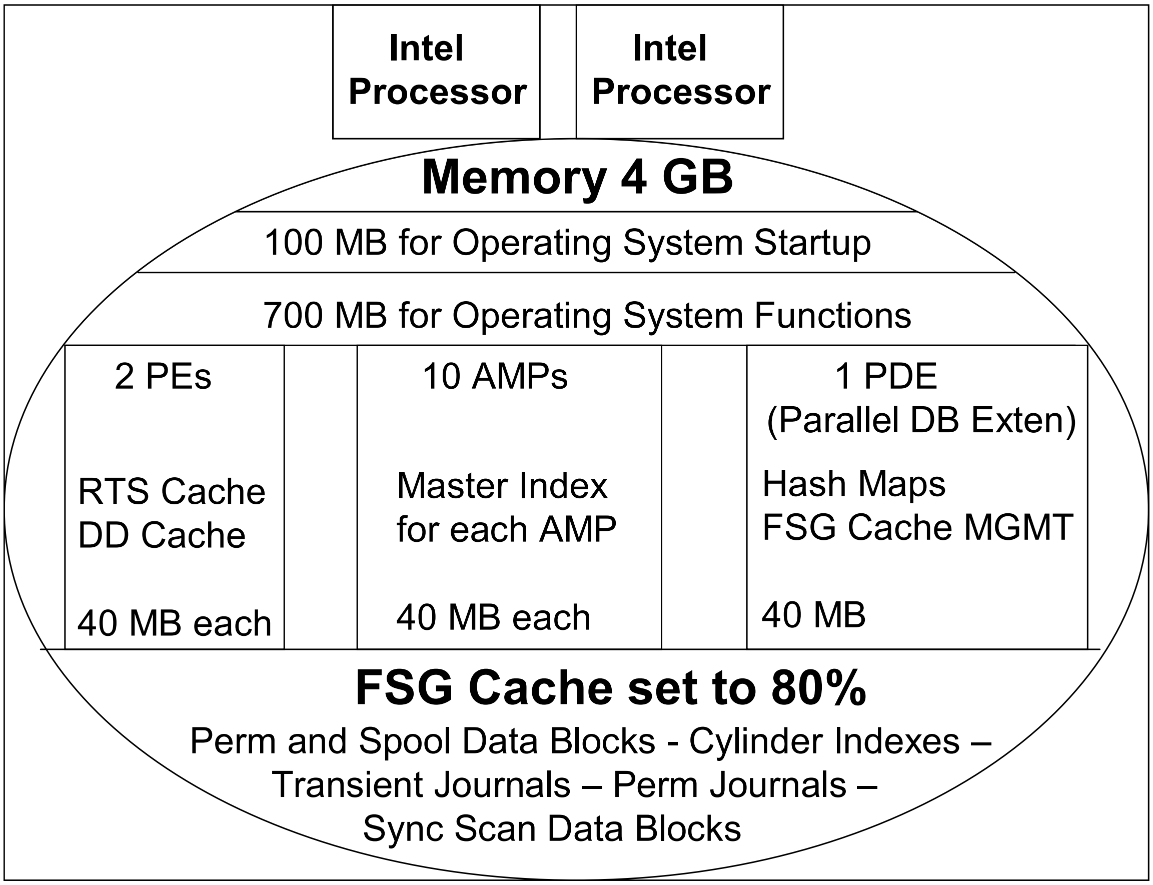
A Node and its Memory Allocations

"With 32-bit operating system limits the memory is like the human body and is often the second thing to go. What is the first? I Forget!"

- Tera-Tom Coffing

The AMPs and PEs are called Vprocs because they reside in the memory of a node. A 32-bit operating system can only have up to 4 Gigabytes of memory. This is the worst limitation on a Teradata data warehouse. The chart below lists how memory is used.

Node and Memory


Intel Processor
Intel Processor
Memory 4 GB
100 MB for Operating System Startup
700 MB for Operating System Functions
2 PEs
RTS Cache
DD Cache
40 MB each
10 AMPs
Master Index for each AMP
40 MB each
1 PDE (Parallel DB Exten)
Hash Maps
FSG Cache MGMT
40 MB
FSG Cache set to 80%
Perm and Spool Data Blocks - Cylinder Indexes – Transient Journals – Perm Journals – Sync Scan Data Blocks

Chapter 5 — The Extended Logical Data Model

"Too much of a good thing is just right."

– Mae West

Too much of a good thing is just right, but even a little of a bad thing will mess up your data warehouse. Building a poor Extended Logical Data Model makes about as much sense as Mae West running east looking for a Sunset! Things will be headed in the wrong direction. If however, you can produce quality in your ELDM, your warehouse is well on its way to being just right!

We have arrived at building the Extended Logical Data Model. The **Extended Logical Data Model** will serve as the **input to the Physical Data Model**. We are in a critical stage. We must produce excellence in this section because we don't want bad input going into our Physical Data Model. This comes from Computer Science 101 -Garbage In – Garbage Out (GIGO).

This chapter will begin with the Application Development Life Cycle, and then talk about the Logical Model and Normalization. From there, we get to the meat and discuss the metrics, which is a critical part of the Extended Logical Data Model. The ELDM will become our final input into the Physical Data Model.

The Application Development Life Cycle

"Failure accepts no alibis.
Success requires no explanation."

– Robert Rose

The design of the physical database is key to the success of implementing a data warehouse along with the applications that reside on the system. Whenever something great is built, whether a building, an automobile, or a business system, there is a process that facilitates its development. When designing the physical database, the process to conduct this is known as the **Application Development Life Cycle**. If you follow the process you will need no alibis because success will be yours.

The six major phases of this process are as follows:

- **Design** - Developing the Logical Data Model (LDM), Extended Logical Data Model (ELDM), and the Physical Data Model (PDM).

- **Development** - Generating the queries according to the requirements of the business users.

- **Test** - Measure the impact that the queries have on system resources.

- **Production** - Follow the plan and move the procedures into the production environment.

- **Deployment** - Training the users on the system application and query tools (i.e., SQL, MicroStrategy, Business Objects, etc.).

- **Maintenance** - Re-examining strategies such as loading data and index selection.

This chapter will focus on the design phase. Insight will be given on each of the three types of data models and any corresponding normalization or de-normalization processes. It is this foundation that will allow the remaining phases to stand or fall. A successful completion of the design phase is the beginning of a successful implementation.

Asking the Right Questions

"He who asks a question may be a fool for five minutes, but he who never asks a question remains a fool forever."

– Tom Connelly

The biggest key to this section is knowledge and not being afraid to ask the right questions. Knowledge about the user environment is vitally important. If you can ask the right questions, you will build a model that maps to the users needs. In addition, you will be able to deliver a world-class data warehouse that remains cool forever.

Here is how this works. The **logical modelers** will create a **logical data model**. Then it is up to you to ask the right questions and find out about the demographics of the data. Only then you can build the proper Physical Data Model.

Remember:

- *The **Logical Data Model** will be the input to the **Extended Logical Data Model.***
- *The **Extended Logical Data Model** will be **input to the Physical Data Model**.*

Logical Data Model

"When you are courting a nice girl an hour seems like a second. When you sit on a red-hot cinder a second seems like an hour. That's relativity."

– Albert Einstein

Albert Einstein must have been a modeler because he understood that each table has a relative and that some joins on hot cinder AMPs can take an hour if not logically and physically modeled correctly. Each table usually has a relation to one or more tables and that is exactly how joins take place.

The first step of the design phase is called the **Logical Data Model** (LDM). The LDM is a logical representation of tables that reside in the data warehouse database. Tables, rows and columns are the equivalent of files, records and fields in most programming languages. A properly normalized data model allows users to ask a wide variety of questions today, tomorrow, and forever.

The following illustration displays the Employee Table. The columns are **EMP**, **DEPT**, **LNAME**, **FNAME** and **SAL**:

Employee Table

EMP	DEPT	LNAME	FNAME	SAL
Primary Key	Foreign Key			
1	40	BROWN	CHRIS	95000.00
2	20	JONES	JEFF	70000.00
3	20	NGUYEN	XING	55000.00
4	10	JACOBS	SHERRY	34000.00

Notice that each of the four rows in the employee table is listed across all of columns and each row has a value for each column. A row is the smallest unit that can be inserted, or deleted from a table in a data warehouse database.

Primary Keys

A **Primary Key** is a **logical concept** that **uniquely identifies** a **row** in a table. All columns in this table that are not a part of the Primary Key should directly relate to the key. From the table below, we notice that employee 1 is Chris Brown who has a salary $95,000, and resides in department 40.

Employee Table

EMP	DEPT	LNAME	FNAME	SAL
Primary Key	Foreign Key			
1	40	BROWN	CHRIS	95000.00
2	20	JONES	JEFF	70000.00
3	20	NGUYEN	XING	55000.00
4	10	JACOBS	SHERRY	34000.00

Primary Keys have the following requirements:

1. No Duplicate values
2. No Null values
3. No Changes

The fact that duplicate values are not supported implies that **Primary Keys** are unique. In addition, null values will not be present for the primary key. Finally, a primary key value, once established in the table, cannot be changed. This ensures that data integrity is maintained.

A common question asked by those who are new to logical modeling is, “Can a Primary Key be made of more than one column?” Theoretically, there is no limit to the number of columns that can be used to make up a Primary Key. Keep in mind, there is only one Primary Key per table. However, the primary key can be made up of several columns. Teradata sets the Primary Key limitation to16 columns.

When doing the physical implementation and design of a Teradata Data Warehouse, it is extremely important to remember that a Primary Key is **logical**, and has no influence in **physically** distributing or accessing the data. A Primary Key is simply a logical concept that uniquely identifies the rows. It also provides for the relationships between tables.

It is extremely important to remember that Primary Keys are a Unique Identifier for a row. In addition, Primary Keys establish how tables are JOINED. Say it again with me, “The Primary Key will join to another table via a Foreign Key”.

Foreign Keys

Foreign Keys are similar to primary keys. The similarity is that **Foreign Keys** are really **Primary Keys** that reside **on another table**. The main purpose of this is to establish a relationship between the tables for Joins. Again, this is a vital thing to remember when we design our Primary Indexes. Foreign Keys are different because of the following:

1. A Foreign Key **can have duplicate values**
2. A Foreign Key **can be null**
3. A Foreign Key **can change**
4. A Foreign Key in one table must first exist as a Primary Key in its related table.

Employee Table

EMP	DEPT	LNAME	FNAME	SAL
PK	**FK**			
1	40	BROWN	CHRIS	95000.00
2	20	JONES	JEFF	70000.00
3	20	NGUYEN	XING	55000.00
4	?	JACOBS	SHERRY	34000.00

Department Table

DEPT	DEPT_NAME
PK	
10	Human Resources
20	Sales
30	Finance
40	Information Technology

Question, what is the department name for Chris Brown? Answer, when you match the foreign key value of 40 in the Employee Table with the Primary Key of 40 in the Department Table, you now have the answer. Remember that the non-key columns in the row relate to the Primary Key. Therefore, when we look up department 40 in the Department Table we can see that Chris Brown works in the Information Technology Department. The two tables are related via the Primary Key–Foreign Key relationship, which is based on the DEPT column.

You will find that most joins and subqueries utilize the Primary and Foreign Keys to join tables. It is important to design your tables with Primary Keys and Foreign Keys that are in the same domain. This means they have the same data types and range of values. It is also a great idea to give them the same names if you can. You can **avoid data conversion** by assigning data types at the **domain level**.

Normalization

"The noblest search is the search for excellence."

– Lyndon B. Johnson

Lyndon Johnson tried to give us the "Great Society" and normalizing your database fundamentally tries to make the tables work in database society greatly. If the modelers can't win the normalization war then they should not seek or should not accept another term as your modeler.

The term normalizing a database came from IBM's Codd. He was actually intrigued by President Nixon's term of trying to normalize relations with China so he began his search for excellence to normalize relations between tables. He called the process normalization. Most modelers believe the noblest search is the search for perfection so they often can take years to perfect the data model. This is a mistake. When creating a logical data model, don't strive for perfection; aim for excellence. And do it quickly!

Normalizing a database is a process of placing columns that are not key related (PK/FK) columns into tables in such a way that **flexibility is utilized**, **redundancy is minimized**, and **update anomalies** are vaporized! Industry experts consider Third Normal Form (3NF) mandatory, however it is not an absolute necessity when utilizing a Teradata Data Warehouse. Teradata has also been proven to be extremely successful with Second Normal Form implementations. The first four forms are described as follows:

1. **First Normal Form** (1NF) **eliminates repeating groups**, for each Primary Key there is only one occurrence of the column data.

2. **Second Normal Form** (2NF) is when the columns/attributes relate to the **entire Primary Key, not just a portion of it**. All tables with a single column Primary Key are considered second normal form.

3. **Third Normal Form** (3NF) states that all columns/attributes relate to the **entire primary key and no other key or column.**

4. **4Th Normal Form** (4NF) states that columns/attributes of a **multi-value dependency** will be **functionally dependent** upon each other.

Once the tables are defined, related, and normalized, the Logical Data Model is complete. The next phase involves the business users, which establish the foundation for creation of the Extended Logical Data Model.

Extended Logical Data Model

"Choice, not chance, determines destiny."

– Anonymous

The information you gather to create the Extended Logical Data Model will provide the backbone for the choices you will make in the Physical Data Model. Leave nothing to chance. You are in the process of determining the final outline of your data warehouse.

The completed Logical Data Model is used as a basis for the development of the Extended Logical Data Model (EDLM). The ELDM includes information regarding physical column access and data demographics for each table. This also serves as input to the implementation of the Physical Data Model. The ELDM contains demographic information about data distribution, sizing, and access.

The successful completion of the ELDM depends heavily on input from the business users. Users are very knowledgeable about the questions that need to be answered today. This knowledge in turn brings definition on how the warehouse will be utilized. In addition, this input provides clues about the queries that will be used to access the warehouse, the transactions that will occur, and the rate at which these transactions occur.

Inside the ELDM, each column will have information concerning expected usage and demographic characteristics. How do we find this information? The best way to resolve this is by running queries that use aggregate functions. The SAMPLE function can also be helpful when collecting data demographics. This information, along with the frequency at which a column will appear in the WHERE clause, is extremely important. The two major components of an ELDM are the following:

- ***COLUMN ACCESS*** *in the* ***WHERE*** *Clause or* ***JOIN*** *CLAUSE.*
- ***DATA DEMOGRAPHICS****.*

The End Goal of the ELDM is to build Table Templates

*"If you don't know where you're going,
Any road will take you there."*

– Lewis Carroll

We will be traveling down a road that you may not be familiar with, so before we get started with the Value Access and Data Demographic section of this book, let me show you our ending destination. The table template below is the road map for the Physical Design. The directions are as follows. For each table in our logical model, we will fill in the ACCESS and DATA DEMOGRAPHICS. When we have finished this process, we have completed the Extended Logical Data Model (ELDM). The next step is to utilize our ELDM table templates as input to the **Physical Database Design** portion where we choose our **Primary Indexes** and **Secondary Indexes**.

If we construct the proper Data Demographics and Value Access metrics, then your warehouse will be on easy street, and your users will be on the Fast-Query Freeway on the way to Promotion Parkway. Mess these metrics up and your warehouse will be on Hopeless Highway. In addition, your users will be constantly sitting in traffic along with your career heading straight for a dead end.

The Example Below illustrates an ELDM template of the Employee Table (assuming 20,000 Employees):

	EMP	DEPT	LNAME	FNAME	SAL
PK & FK	PK SA	FK			
ACCESS					
Value Acc Freq	6K	5K	100	0	0
Join Acc Freq	7K	6K	0	0	0
Value Acc Rows	70K	50K	0	0	0
DATA DEMOGRAPHICS					
Distinct Rows	20K	5K	12K	N/A	N/A
Max Rows Per Value	1	50	1K	N/A	N/A
Max Rows Null	0	12	0	N/A	N/A
Typical Rows Per Value	1	15	3	N/A	N/A
Change Rating	0	2	1	N/A	N/A

Column ACCESS in the WHERE Clause

"The only true wisdom is in knowing you know nothing."

– Socrates

Follow Socrates advice and assume you know nothing when starting this portion of the design process. However, If you make a column your Primary Index that is never accessed in the WHERE clause or used to JOIN to other tables, then Socrates had you in mind on the last three words of his quote.

It will be your job to interview users, look at applications, and find out what columns are being used in the WHERE clause for the SELECTs. The key here is to investigate what tables are being joined. It is also important to know what columns are joining the tables together. You will be able to find some join information from the users, but common sense plays a big part in join decisions. Understand how column(s) are accessed, and your warehouse is on its way to providing true wisdom!

COLUMN ACCESS in the WHERE CLAUSE:

Value Access Frequency	How frequently the table will be accessed via this column.
Value Access Rows * Frequency	The number of rows that will be accessed multiplied by the frequency at which the column will be used.
Join Access Frequency	How frequently the table will be joined to another table by this column being used in the WHERE clause.

Quite often, new designers to Teradata believe that selecting the Primary Index will be easy. They just pick the column that will provide the best data distribution. They assume that if they keep the Primary Index the same as the Primary Key column (which is unique by definition), then the data will distribute evenly (which it true). The Primary Index is about distribution, but even more important is **Join Access**. Remember this golden rule!

The Primary Index is the fastest way to access data and Secondary Indexes are next.

Data Demographics

"Don't count the days, make the days count."

– Mohammed Ali

Mohammed Ali has given you some advice that is reflective of your next objective. Count the **DATA** so you can make the **DATA** count. So how do you accomplish this task? The following will be your guide:

- Write SQL and utilize a wide variety of tools to get the data demographics.
- Combine these demographics with the column and join access information to complete the Extended Logical Data Model.
- Then use this information to create the Primary Indexes, Secondary Indexes, and other options in the Physical Database Design.

DATA DEMOGRAPHICS:

Distinct Values	The total number of unique values that will be stored in this column.
Maximum Rows per Value	The number of rows that will have the most popular value in this column.
Typical Rows per Value	The typical number of rows for each column value.
Maximum Rows NULL	The number of rows with NULL values for the column.
Change Rating	A relative rating for how often the column value will change. The value range is from 0-9, with 0 describing columns that do not change, and 9 describing columns that change with every write operation.

Data Demographics answer these questions:

- **How evenly will the data be spread across the AMPs?**
- **Will my data have spikes causing AMP space problems?**
- **Will the column change too often to be an index?**

Distinct Values

"Always remember that you are unique just like everyone else."

– Anonymous

Distinct Values relates to the **total number of unique values** that will be stored in this particular **column**. The more distinct values a column has, the better distribution it can potentially provide as an index. It is also an indicator of a Unique or Non-Unique candidate. If you have a million rows in a table and Column A has a million distinct values, then you know you have a **UNIQUE** Index candidate. The bottom line is that the more distinct values your column has, the better it will distribute. If you have a billion-row table and 500 million distinct values, you will probably have excellent distribution.

Here is a great rule to remember. **Try to never make a column your Primary Index if it has less Distinct Values then the number of AMPs**. The only exception to this rule is a very small table. Your system may have 2000 AMPs and you might have a state table with 50 rows.

The idea behind Distinct Values is to determine if a column is a Unique or Non-Unique candidate. It is also provides insight if the column will be able to distribute the data evenly. It could also be a hint for things to come when looking for Spikes or Hot AMPs.

A spike is an enormous amount of data values that could potentially cause one AMP to be overloaded. Remember that the Hashing Algorithm (combined with the Hash Map) determines data layout. A column chosen as the Primary Index will place all rows with the same Primary Index value on the same AMP. **When one AMP has more rows than any other AMPs, this is called a Hot AMP.**

A Hot AMP can cause your query to run out of Spool Space. It can also cause a Database or User to run out of Perm Space. Remember, both Spool and Perm space is equally divided over the number of AMPs in the system. For example, if you have 10 AMPs and 10 Gigabytes of Spool Space you really only have 1 Gigabyte of Spool per AMP. Spikes and Hot Amp situations indicate that Perm and Spool is not equally distributed among the AMPs. A good physical database design will avoid this problem.

Distinct Values provide answers to these questions:

- Will this column be a Unique Candidate?
- Does this column have potential to distribute well?
- Could this column cause potential data spikes?

Maximum Rows Per Value

"Those who dance are considered insane by those who cannot hear the music."

– George Carlin

I always felt that George Carlin was a stand-up guy who maximized his humor. The **Maximum Rows Per Value** in Data Demographics means the number of rows that will have the most popular value. This needs to be measured in conjunction with the **Typical Rows Per Value** demographic. Most people understand that if you have a million rows and a Maximum Row Per Value of 1 then you will have unique data in this column. It also means this column will be an excellent Primary Index candidate because it will spread the data evenly. What many do not understand is that a column might have 1,000 as the Maximum Rows Per Value, but 900 as the Typical Rows Per Value. This will most likely provide excellent distribution and you will get the last laugh.

Let me explain this in a different way. Imagine taking a deck of cards and distributing the cards one-by-one to four friends. When you were done dealing all the cards you could expect good distribution. Now imagine again we have 52 decks of cards. In this example, you take one deck at a time and distribute it to each friend. When you are done passing out all 52 decks of cards you again have great distribution. Just because you are passing larger quantities of like data to the AMPs does not mean the distribution will be bad. Remember that a high Maximum Rows Per Value means data could be clumsy and cumbersome.

Maximum Rows Per Value is a demographic that will show you if there are potential spikes in your data. If you have a Maximum Rows Per Value of 10,000 and a Typical Rows Per Value of 10 you are going to have a huge spike. If however, you have a Maximum Rows Per Value of 10,000 and a Typical Rows Per Value of 8,000, you probably will have excellent data distribution. Especially if you find that the number of rows in the table is in the billions.

If the **Maximum Rows Per Value** is one then the column will **be Unique**.

Always compare **Maximum Rows Per Value** with **Typical Rows Per Value** to see **potential data spikes** and whether the **distribution** will be **acceptable**.

Typical Rows Per Value

"I cannot imagine any condition which would cause this ship to founder. Modern shipbuilding has gone beyond that."

– E. I. Smith, Captain of the Titanic

The Typical Rows Per Value will set you on course for even distribution and make your data design unsinkable. Get this fundamental right and it is full speed ahead.

The Typical Rows Per Value will show the typical number of rows for each column value. When this is compared to the Maximum Rows Per Value you can see potential spikes in the data. In addition, this will provide a better feel if the column will potentially spread the data evenly.

If the **Typical Rows Per Value number is 1**, you can probably guess that the **column** will be **unique.** Don't make the mistake of thinking that if Typical Rows Per Value is high that it makes the data skewed. On larger tables, the Typical Rows Per Value could be enormous. However, when compared to the Maximum Rows Per Value, you might have extremely good distributed data.

The biggest key is to gather the demographics for columns and don't exclude any columns until you have seen the whole picture. I can't begin to tell you how often rookies make mistakes when picking Primary and Secondary indexes because they make assumptions.

If the **Typical Rows Per Value** is one then the column will probably **be Unique**.

Always compare **Typical Rows Per Value** with **Maximum Rows Per Value** to see **potential data spikes** and whether the **distribution** will be **acceptable**.

Maximum Rows NULL

"I'll Moider da bum..."

– Tony Galento, heavyweight boxer, when asked what he thought of William Shakespeare

You will get murdered if you have a heavy weight of nulls in a column you want to be your Primary Index. Having a small amount of nulls is OK, but not more then a 7 or 8 count. Thousands of NULL values will cause a TKO (Table KNulls Out).

The Maximum Rows NULL relates to the total number of NULL values that will be stored in this particular column. Having a NULL value is permissible even as a Primary Index in Teradata. You can even have one NULL in a Unique Primary Index. The biggest thing to consider is will the **NULL values** cause a **distribution spike** and create a **Hot AMP**.

A spike is an enormous amount of data values that could potentially cause one AMP to be overloaded. Remember that the Hashing Algorithm combined with the Hash Map determine data layout. A column chosen as the Primary Index will place all rows with the same Primary Index value on the same AMP. This also goes for NULL values. Each NULL value will produce the same Row Hash, point to the same bucket in the Hash Map, and be sent to the same AMP.

A **Hot AMP** can cause your query to run out of **Spool Space**. It can also cause Database or User to run out of Perm Space. Remember, both Spool and Perm space is equally divided over the number of AMPs in the system. For example, if you have 10 AMPs and 10 Gigabytes of Spool Space you actually have 1 Gigabyte of Spool per AMP.

Change Rating

"Things are more like they are now than they ever were before"

– Dwight D. Eisenhower, U.S. President

Personally, I like Ike, but change makes things more unlike now than they were before. A column with a high rate of change is not stable enough to be your Primary Index or storm the beaches of Normalization.

The **Change Rating** is a **relative rating** for **how often** the column value will **change**. The Change Rating can be a real heartbreaker. You see a column that has great potential as a Primary or Secondary Index and then the Change Rating bites you. Data that changes often cannot be a Primary Index. If the value changes quite often, it should not be a secondary index. When a Primary Index value changes, the base row needs to be rehashed and placed on a different AMP. Every secondary index on that table will also have to change to reflect the new location of the base row.

The value for the change rating is from 0-9, with 0 describing columns that don't ever change and 9 for columns that change with every write operation.

- PK and historical data columns are 0.
- Columns that usually don't change are 1.
- Columns that track updates are 9.

Here are some guidelines to remember about the change rating.

- Primary Indexes should be 0-2.
- Secondary Indexes should be 0-5.

If you find a column that is a potentially great Primary Index, but the change rating is greater than 2, exclude it from consideration. End of story! This is why the change rating can be a heartbreaker. If you find a column that is a potentially great Secondary Index, but the change rating is greater than 5, exclude it. End of story!

Your Primary and Secondary Indexes are your access paths to the data and are designed to speed up the queries and the system. When you place these indexes inside your warehouse and the columns change, then your system is bombarded with maintenance. The columns you were depending on for speed are now weighing down queries like an anchor.

Extended Logical Data Model Template

"There are not enough Indians in the world to defeat the Seventh Cavalry"

– George Armstrong Custer

Having a Logical Data Model Template will allow your decision making to be as sharp as an arrow. You will be able to back up your decisions with the facts so you can toot your own big horn.

Below is an example of an Extended Logical Data Model. The Value Access and Data Demographics have been collected. We can now use this to pick our Primary Indexes and Secondary Indexes. During the first pass at the table you should pick your potential Primary Indexes. Label them UPI or NUPI based on whether or not the column is unique. At this point in time, don't look at the change rating.

A great **Primary Index** will have:

- A **Value Access frequency** that is **high**
- A **Join Access frequency** that is **high**
- **Reasonable** distribution
- A change rating **below 2**

The Example Below illustrates an ELDM template of the Employee Table (assuming 20,000 Employees):

	EMP	DEPT	LNAME	FNAME	SAL
PK & FK	PK SA	FK			
ACCESS					
Value Acc Freq	6K	5K	100	0	0
Join Acc Freq	7K	6K	0	0	0
Value Acc Rows	70K	50K	0	0	0
DATA DEMOGRAPHICS					
Distinct Rows	20K	5K	12K	N/A	N/A
Max Rows Per Value	1	50	1K	N/A	N/A
Max Rows Null	0	12	0	N/A	N/A
Typical Rows Per Value	1	15	3	N/A	N/A
Change Rating	0	2	1	N/A	N/A

Once we have our table templates we are ready for the Physical Database Design. Read on, this is becoming interesting!

Chapter 6 — The Physical Data Model

"Nothing can stand against persistence; even a mountain will be worn down over time."

We have arrived at the moment of truth. We are arriving at the top of the mountain. It is now time to create the Physical Database Design model. This chapter will cover the keys to an excellent Teradata Physical Data Model. Let this chapter be your guide and the data warehouse will not be overwhelmed by user queries. The steps below will empower your user to levels and new heights of greatness. The biggest keys to a good physical model are choosing the correct:

- *Primary Indexes*
- *Secondary Indexes*
- *Denormalization Tables*
- *Derived Data Options*

Step 1 – Look at Distribution

"You can observe a lot by watching"

– Yogi Berra

You can observe a lot by watching the data distribution in your Teradata Data Warehouse. If you ever want to make the Database Hall of Fame then look at Data Distribution. If you are a hit at understanding data distribution you could be outstanding in your field.

No matter if the data is coming from an Oracle Database or the Mainframe, data demographics can be gathered from incoming data with a few simple SQL commands. Getting Value Access Frequency and Join Access Frequency can be much more difficult. This is why the first steps are not to take "Value Access" metrics into account. We will narrow down these fields before bringing in these metrics.

We will place the letters UPI or NUPI for every column that distributes well. In our later steps, we will eliminate candidates until we can pick a great Primary Index. This process will review every single column and if the column has good distribution, our next step is to automatically place an UPI or NUPI at the bottom of the chart in the column area (labeled PRI). If the column has poor distribution, we will leave the column blank. For columns with good distribution, if it is unique, put a UPI. If the column is non-unique, it will be a NUPI. When this process is completed, some columns will be listed as potential candidates as UPI or NUPI. Most likely, some columns won't have good distribution.

The second step of our Physical Database Design process is to place an USI or NUSI at the bottom of the chart (labeled SEC) for secondary indexes in every column where you have and UPI or NUPI in the above row. If you have an UPI, then place an USI in the column area for potential secondary indexes. If you have a NUPI, then place a NUSI in the column area for potential secondary indexes.

Lastly, look at Typical Rows Per Value to add any more potential NUSIs. Check to see if the Typical Rows Per Value is low compared to the Total Rows in the Table. For example, in a 2,000-row table, the Typical Rows Per Value would need to be 5-10. In a 2 Million-row table the Typical Rows Per Value could be higher. If the column has a relatively low Typical Rows Per Value, then automatically mark the column as a NUSI candidate. NUSIs are not concerned about data distribution.

For this process we will use two tables in our examples. They are the Employee table and the Customer Table.

Employee table has 20,000 rows

	EMP	DEPT	LNAME	FNAME	SAL
PK & FK	PK SA	FK			
ACCESS					
Value Acc Freq	6K	5K	100	0	0
Join Acc Freq	7K	6K	0	0	0
Value Acc Rows	70K	50K	0	0	0
DATA DEMOGRAPHICS					
Distinct Rows	20K	5K	12K	N/A	N/A
Max Rows Per Value	1	50	1K	N/A	N/A
Max Rows Null	0	12	0	N/A	N/A
Typical Rows Per Value	1	15	3	N/A	N/A
Change Rating	0	2	1	N/A	N/A
(PRI) Fill in UPI or NUPI					
(SEC) Fill in USI or NUSI					
Contact the Customer?					

Customer Table has 2,000,000 rows

	Cust_no	CName	Phone
PK & FK	PK SA		
ACCESS			
Value Acc Freq	1M	5K	0
Join Acc Freq	100K	0	0
Value Acc Rows	100M	5K	0
DATA DEMOGRAPHICS			
Distinct Rows	2M	1900K	1950K
Max Rows Per Value	1	2	1
Max Rows Null	0	12	500
Typical Rows Per Value	1	1	1
Change Rating	0	2	5
(PRI) Fill in UPI or NUPI			
(SEC) Fill in USI or NUSI			
Contact the Customer?			

Results from Step 1 - Distribution

Employee table has 20,000 rows

	EMP	DEPT	LNAME	FNAME	SAL
PK & FK	PK SA	FK			
ACCESS					
Value Acc Freq	6K	5K	100	0	0
Join Acc Freq	7K	6K	0	0	0
Value Acc Rows	70K	50K	0	0	0
DATA DEMOGRAPHICS					
Distinct Rows	20K	5K	12K	N/A	N/A
Max Rows Per Value	1	50	1K	N/A	N/A
Max Rows Null	0	12	0	N/A	N/A
Typical Rows Per Value	1	15	3	N/A	N/A
Change Rating	0	2	1	N/A	N/A
(PRI) Fill in UPI or NUPI	**UPI**	**NUPI**			
(SEC) Fill in USI or NUSI	**USI**	**NUSI**	**NUSI**		
Contact the Customer?					

Customer Table has 2,000,000 rows

	Cust_no	CName	Phone
PK & FK	PK SA		
ACCESS			
Value Acc Freq	1M	5K	0
Join Acc Freq	100K	0	0
Value Acc Rows	100M	5K	0
DATA DEMOGRAPHICS			
Distinct Rows	2M	1900K	1950K
Max Rows Per Value	1	2	1
Max Rows Null	0	12	500
Typical Rows Per Value	1	1	1
Change Rating	0	2	5
(PRI) Fill in UPI or NUPI	**UPI**	**NUPI**	**NUPI**
(SEC) Fill in USI or NUSI	**USI**	**NUSI**	**NUSI**
Contact the Customer?			

Both tables have had distribution analysis conducted for every potential Primary Index. We have automatically placed an equal Secondary Index. We also found a relatively low Typical Rows Per Value in the Employee table for LNAME and have put it as a NUSI.

Step 2 – Eliminate based on Change Rating

"Sure I'm helping the elderly. I'm going to be old myself some day"

– Lillian Carter, in her 80s

The great thing about age is that it never feels like we are getting older. We often don't see that we are changing either. We might not be getting older or changing, but your column values will change. The next step in our process is to eliminate Primary Index and Secondary Index candidates based on change rating.

The Change Rating is a relative rating for how often the column value will change. Data that changes often cannot be a Primary Index. If it changes quite often, it should not be a secondary index. When a Primary Index value changes, the base row needs to be rehashed and placed on a different AMP. Every secondary index on that table will also have to change to reflect the new location of the base row. This is the type of unnecessary steps that can cripple a data warehouse.

The value for the change rating is from 0-9, with 0 describing columns that don't ever change and 9 for columns that change with every write operation.

- PK and historical data columns are 0.
- Columns that usually don't change are 1.
- Columns that track updates are 9.

Here are some guidelines to remember about the change rating.

- Primary Indexes should be 0-2.
- Secondary Indexes should be 0-5.

Eliminate all Primary Index candidates if they have a change rating greater than 2.

Eliminate all Secondary Index candidates if they have a change rating greater than 5.

Results of Step 2 – Change Rating

Customer Table has 2,000,000 rows

	Cust_no	CName	Phone
PK & FK	PK SA		
ACCESS			
Value Acc Freq	1M	5K	0
Join Acc Freq	100K	0	0
Value Acc Rows	100M	5K	0
DATA DEMOGRAPHICS			
Distinct Rows	2M	1900K	1950K
Max Rows Per Value	1	2	1
Max Rows Null	0	12	500
Typical Rows Per Value	1	1	1
Change Rating	0	2	5
(PRI) Fill in UPI or NUPI	**UPI**	**NUPI**	**XXXX**
(SEC) Fill in USI or NUSI	**USI**	**NUSI**	**NUSI**
Contact the Customer?			

Employee table has 20,000 rows

	EMP	DEPT	LNAME	FNAME	SAL
PK & FK	PK SA	FK			
ACCESS					
Value Acc Freq	6K	5K	100	0	0
Join Acc Freq	7K	6K	0	0	0
Value Acc Rows	70K	50K	0	0	0
DATA DEMOGRAPHICS					
Distinct Rows	20K	5K	12K	N/A	N/A
Max Rows Per Value	1	50	1K	N/A	N/A
Max Rows Null	0	12	0	N/A	N/A
Typical Rows Per Value	1	15	3	N/A	N/A
Change Rating	0	2	1	N/A	N/A
(PRI) Fill in UPI or NUPI	**UPI**	**NUPI**			
(SEC) Fill in USI or NUSI	**USI**	**NUSI**	**NUSI**		
Contact the Customer?					

Our Customer table has the Phone eliminated in the NUPI. The change rating was above a 2. The Employee table has nothing eliminated.

Step 3 – NUSI Elimination via Value Access Frequency

"I am not young enough to know everything"

– Oscar Wilde (1854 – 1900)

Oscar Wilde must have raised a few teenagers to be that smart. Teenagers know a lot about life and they must have ACCESS to their friends. Most teenagers have certain friends that they ACCESS a lot. They are their best friends. Teradata users will have certain columns that are their friends in which they ACCESS a lot in queries. Eliminate columns from NUSI index consideration if users don't access the columns frequently. If columns are slacking then they should be lacking from NUSI consideration and should also get a haircut.

The next step in our process is to eliminate all NUSI candidates that do not have a value access frequency. The only reason to have a NUSI on a column is when the column has a high value access frequency. If there is no Value Access, then the column will not be a NUSI. Every time a NUSI is built a subtable must be created. If a column is not accessed in the WHERE clause of a query then the subtable will never be used. Why take up space when there is no need in the first place?

Results of Step 3 – NUSI Elimination

Customer Table has 2,000,000 rows

	Cust_no	CName	Phone
PK & FK	PK SA		
ACCESS			
Value Acc Freq	1M	5K	0
Join Acc Freq	100K	0	0
Value Acc Rows	100M	5K	0
DATA DEMOGRAPHICS			
Distinct Rows	2M	1900K	1950K
Max Rows Per Value	1	2	1
Max Rows Null	0	12	500
Typical Rows Per Value	1	1	1
Change Rating	0	2	5
(PRI) Fill in UPI or NUPI	**UPI**	**NUPI**	**XXXX**
(SEC) Fill in USI or NUSI	**USI**	**NUSI**	**XXXX**
Contact the Customer?			

Employee table has 20,000 rows

	EMP	DEPT	LNAME	FNAME	SAL
PK & FK	PK SA	FK			
ACCESS					
Value Acc Freq	6K	5K	100	0	0
Join Acc Freq	7K	6K	0	0	0
Value Acc Rows	70K	50K	0	0	0
DATA DEMOGRAPHICS					
Distinct Rows	20K	5K	12K	N/A	N/A
Max Rows Per Value	1	50	1K	N/A	N/A
Max Rows Null	0	12	0	N/A	N/A
Typical Rows Per Value	1	15	3	N/A	N/A
Change Rating	0	2	1	N/A	N/A
(PRI) Fill in UPI or NUPI	**UPI**	**NUPI**			
(SEC) Fill in USI or NUSI	**USI**	**NUSI**	**XXXX**		
Contact the Customer?					

Our Customer table has the Phone eliminated as a NUSI candidate because Phone has a Value Access Frequency of 0. The Employee table has eliminated LNAME as a NUSI.

Step 4 – Pick the Primary Index

"If I have seen farther than others, it is because I was standing on the shoulders of giants"

– **Isaac Newton**

Please understand the gravity of the situation and make the Primary Index the apple of your eye. You must analyze all the factors in order to make a Primary Index decision. The initial reaction here would be to make the Primary Index the column with the best Join Access Frequency. For example, you may encounter this situation where column A has a Join Access Frequency of 12, which means it is joined monthly. However, column B has a Value Access Frequency of 10,000. You probably decide on column B. If you have two columns (column D and column F) that both have a high Join Access Frequency, then you should probably analyze which one distributes the best. The following pages will provide the insight on the best direction to take. Take your time and look at your options so your warehouse doesn't take the fall.

By far the **best thing that** you can do to **affect performance** in your data warehouse is to **pick a great Primary Index**. The Primary Index is the gas that makes the car run, the steam that propels the steamboat, and the heart that pumps the blood. The Primary Index starts with the letter 'P' because it is what provides the performance in your data warehouse. If you miss the first buttonhole, you will have trouble buttoning your shirt. If you miss the Primary Indexes in your warehouse, you will probably lose your shirt!

If you pick a bad secondary index, you can drop the secondary index at any time. If you pick a bad Primary Index, you can't just drop the Primary Index. You will need to be a little more creative. To change a Primary Index, you will need to copy the table structure and create a new table with a better Primary Index. Then you have to either do an INSERT/SELECT into the new table or utilize an archive backup to load to the new table. Either process can takes extra time, work, and possibly space.

If you choose an INSERT/SELECT, you could run into space problems because you are temporarily duplicating the table during the INSERT/SELECT process until you drop the first table. Archives take an enormous amount of time and energy. As someone once said, "Sure I don't look busy – I did it right the first time." Make sure you pick a great Primary Index so you don't have to come back and do it again. We never have enough time to do things right, but we always have time to do them again.

Primary Index Factors

*The biggest gift a **Primary Index** can provide is **Join Access**.*

*When **joins** use the **Primary Index** in the **WHERE** or **ON** clause the table does **not** need to be **redistributed** or **duplicated** to perform the Join.*

*In **best-case scenarios**, when **two tables** being joined both **use their respective Primary Index columns** in the **WHERE** or **ON clause** the joins work like lightning.*

*A data warehouse Decision Support Environment will on average **join five tables** in an average query.*

Making your joins perform is the best determining factor in a high performance data warehouse.

Why Join Access Frequency is Top Priority?

"If I advance, follow me. If I stop, push me. If I fall, inspire me."

– **Robert Cushing**

If Robert Cushing would have written joins in Teradata he might have said, "If I write Advanced SQL, copy me. If it stops, raise my spool, and if it fails, blame it on a bad Primary Index". The absolute **most important factor** when picking a Primary Index is **Join Access Frequency**. When the Join Access Frequency is the basis for the Primary Index of two tables the join will be fast and will inspire all.

Join Access Frequency is vitally important and is definitely the biggest influencing factor when choosing a Primary Index. When the Primary Index is used in the WHERE or ON clause of a join, the table never needs to be redistributed or duplicated. Even better, when both tables in a join use their respective Primary Indexes, no data is ever redistributed or duplicated. These joins run at maximum speed with no effort.

In a data warehouse that provides decision support to hundreds or thousands of users, a wide variety of questions will be asked. It has been estimated that the average query in a large data warehouse will join 5 tables together. While joining tables is usually a database's worst nightmare, Teradata makes the process a dream come true. Teradata can join up to 64 tables. Teradata brilliantly uses the hash algorithm to redistribute or duplicate tables in preparation for the join, but nothing speeds up a join faster than having the Primary Index used in the WHERE or ON clause.

Making the joins run like lightning is the biggest gift you can give users. Your data warehouse begins to outdistance competitors data warehouse's by magnitudes of order when users can ask any question on any data at any time. The key to success is a normalized data model, and making sure that joins are efficiently performed. Knowing what columns are in the table will be your biggest key when deciding what your Primary Index will be. In addition, this is the best way to provide speed and performance in the data warehouse.

Most tables will be joined on either the Primary Key or Foreign Key, which is based on the Logical Model. Knowing this information will assist you if your Join Access Frequency is unable to be determined with help from your users. In this case, you can use common sense in determining the columns that will most likely be used for joins.

Pick the column that will provide the best join performance as your Primary Index.

A Real World Join Example

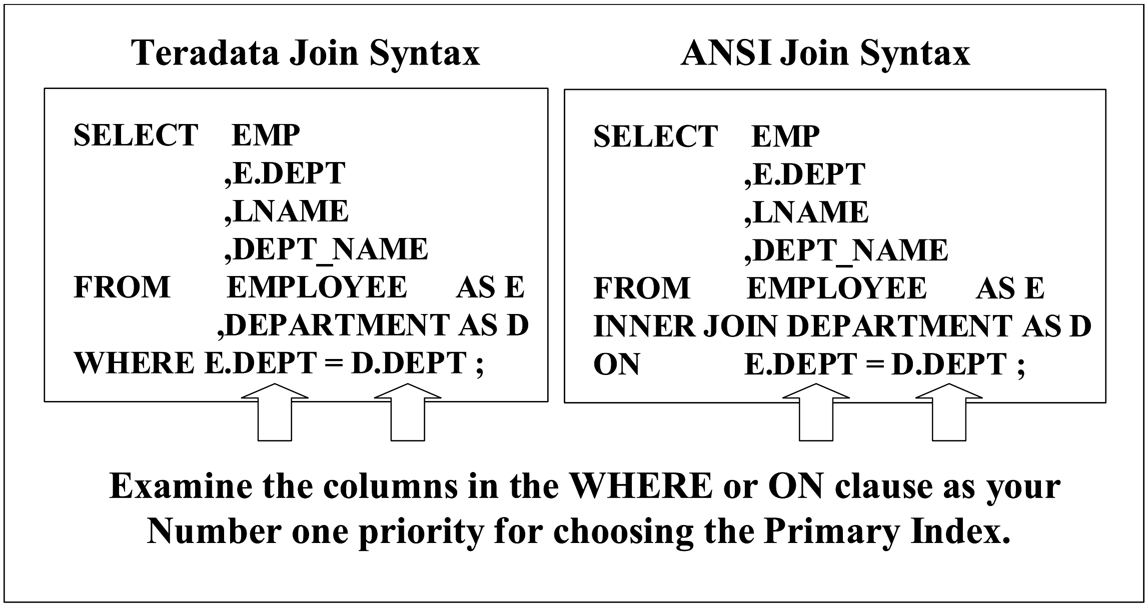
Below are two tables. They are the Employee and the Department tables. These two tables are joined together frequently.

Employee Table

EMP	DEPT	LNAME	FNAME	SAL
PK	**FK**			
1	40	BROWN	CHRIS	95000.00
2	20	JONES	JEFF	70000.00
3	20	NGUYEN	XING	55000.00
4	?	JACOBS	SHERRY	34000.00

Department Table

DEPT	DEPT_NAME
PK	
10	Human Resources
20	Sales
30	Finance
40	Information Technology



Teradata Join Syntax

```
SELECT   EMP
         ,E.DEPT
         ,LNAME
         ,DEPT_NAME
FROM     EMPLOYEE     AS E
         ,DEPARTMENT AS D
WHERE E.DEPT = D.DEPT ;
```

ANSI Join Syntax

```
SELECT   EMP
         ,E.DEPT
         ,LNAME
         ,DEPT_NAME
FROM     EMPLOYEE     AS E
INNER JOIN DEPARTMENT AS D
ON       E.DEPT = D.DEPT ;
```

Examine the columns in the WHERE or ON clause as your Number one priority for choosing the Primary Index.

In the previous example, the join would have been better if the DEPT column was the Primary Index for both tables. However, this may cause poor distribution and space problems. That is why you eliminate the poor distribution columns before looking at the Join Access. If DEPT in both tables had survived the distribution analysis, then DEPT column would be the Primary Index in both tables because the Join Access Frequency is high.

If DEPT did not survive the distribution analysis in one of the tables, it is still a good idea to use DEPT in the table where the column did survive the distribution analysis. This is because one of the tables in a join (WHERE or ON) clause is using the Primary Index. This would indicate that only one of the tables would need to be redistributed or duplicated.

Why Value Access Frequency is Second Priority?

"My best friend is the one who brings out the best in me!"

– **Henry Ford**

If you want to physically model your data warehouse to a T then realize that a user's second best friend are columns with a high Value Access Frequency. Also remember that if the Join Access Frequency is fairly low and the Value Access Frequency is extremely high, then you might have to make a "Ford Tough" decision as the warehouse is Built.

The Value Access Frequency would seem like an extremely important metric when picking the Primary Index. However, it is not as important as the Join Access Frequency. That is because the Join Access Frequency is important in maximizing the brilliant way Teradata handles joins. One of the even bigger reasons that Value Access Frequency is not as important in picking a Primary Index is because a Secondary Index can satisfy Value Access Frequency.

For the most part, the only thing that speeds up a join dramatically is when the Primary Index is in the WHERE or ON clause. Secondary Indexes do not really help joins. However, secondary indexes dramatically assist columns frequently accessed in the WHERE clause and can reduce the number of rows that have to participate in a join.

However, don't be too quick to eliminate Value Access Frequency from the Primary Index equation. If you have eliminated the columns in the Distribution analysis that would be used for joins, then Value Access is what you will utilize next to pick your Primary Index.

What have we learned about picking the Primary Index?

*If the **Distribution** for a Column is BAD, it has already been **eliminated** as a **Primary Index Candidate.***

*If you have a column with **high Join Access Frequency**, then this is your **Primary Index** of choice.*

If you have a column that has a high Value Access Frequency, you can always create a secondary index for it.

*If there are **no Join Columns** that survived the **Distribution Analysis**, then pick the column with the best **Value Access** as your **Primary Index**.*

If all of the above fail or two columns are equally important, then pick the column with the best distribution as your Primary Index.

Results of Step 4 – Picking a Primary Index

Customer Table has 2,000,000 rows

	Cust_no	CName	Phone
PK & FK	PK SA		
ACCESS			
Value Acc Freq	1M	5K	0
Join Acc Freq	100K	0	0
Value Acc Rows	100M	5K	0
DATA DEMOGRAPHICS			
Distinct Rows	2M	1900K	1950K
Max Rows Per Value	1	2	1
Max Rows Null	0	12	500
Typical Rows Per Value	1	1	1
Change Rating	0	2	5
(PRI) Fill in UPI or NUPI	**UPI**	**XXXX**	**XXXX**
(SEC) Fill in USI or NUSI	**XXXX**	**NUSI**	**XXXX**
Contact the Customer?			

Employee table has 20,000 rows

	EMP	DEPT	LNAME	FNAME	SAL
PK & FK	PK SA	FK			
ACCESS					
Value Acc Freq	6K	5K	100	0	0
Join Acc Freq	7K	6K	0	0	0
Value Acc Rows	70K	50K	0	0	0
DATA DEMOGRAPHICS					
Distinct Rows	20K	5K	12K	N/A	N/A
Max Rows Per Value	1	50	1K	N/A	N/A
Max Rows Null	0	12	0	N/A	N/A
Typical Rows Per Value	1	15	3	N/A	N/A
Change Rating	0	2	1	N/A	N/A
(PRI) Fill in UPI or NUPI	**UPI**	**XXXX**			
(SEC) Fill in USI or NUSI	**XXXX**	**NUSI**	**XXXX**		
Contact the Customer?					

We have picked our Primary Indexes and eliminated the Secondary Indexes for the Primary Index choices. For difficult decisions, place a check mark inside the box named “Contact the user for advice”.

Step 5 — Pick Secondary Indexes

A **Secondary Index** provides an **alternate path** to the **data**. They can greatly enhance performance, but there is a price to pay. All **secondary indexes** create a **secondary index subtable**. The price is additional space and overhead maintenance. You must be able to weigh the increased speed with the space and overhead maintenance. If the secondary index provides enhanced performance you will probably want it, but if it does not, then drop it. This is why the most important metric for secondary indexes is Value Access Frequency. We will discuss all options for the USI, and then the NUSI.

USI Considerations

There are three reasons to create a Unique Secondary Index on a column:

- You want to ensure that a column's values are unique.

- You have a column with a high value access frequency and you want users to get to the data quickly.

- You have a SET table that has a NUPI. Many of the NUPI values have a large amount of "Typical Rows Per Column" or there is a great deal of collisions or synonyms. Teradata will have to do Duplicate Row Checking to ensure no duplicate rows exist. This process can be time intensive. Placing an USI on another column eliminates this extra checking.

USI to Enforce Uniqueness

If a column has been designated as having no duplicates, such as the **Primary Key**, and it is important to **enforce the uniqueness**, the recommendation would be to **create a Unique Secondary Index (USI).** Teradata will ensure that no duplicate values are entered for all Unique Secondary Index columns. If a user tries to enter a value already in the table the transaction will be aborted and the user is sent a "Duplicate Value" message.

USI for High Value Access Frequency

If a column has a high Value Access Frequency and is unique, then it is the perfect USI. A Unique Secondary Index will speed up queries dramatically. A USI is always a two-AMP operation. When using an USI in the WHERE clause of a query, the answer set will never be more than one row. This is because the row you are after is unique, so there can only be one. An USI is an excellent choice if conditions permit.

USI to eliminate Duplicate Row Checking

"Friends may come and go, but enemies accumulate"

– **Tom Jones**

It's not unusual for Teradata to have to check for duplicate rows in a set table. It's not unusual for this to slow Teradata down.

A very clever technique to utilize an USI is to prevent the time of checking for duplicate rows in a SET table. This is only done when you have a SET table that has a Primary Index that is a NUPI. If your table has a large amount of "Typical Rows Per Column", (i.e., duplicate values) or there is a great deal of collisions, then Teradata may spend a lot of time ensuring there are no duplicate rows. This is a byte-by-byte process. An USI on another column eliminates the extra checking.

Teradata allows for **two types of tables**: **SET** and **MULTISET**. A SET table does not allow two rows in the table to be exactly the same. A MULTISET table does allow two rows in the table to be exactly the same. Teradata originally only supported SET tables, so the database would reject any row that was exactly the same as any other row.

A SET table will allow duplicate Primary Index values when the table has a Non-Unique Primary Index (NUPI). Teradata will not however, allow for every byte in a row to match another row exactly in a set table. When a new row is inserted into a SET table, Teradata hashes the Primary Index. The row is then sent to the appropriate AMP and will check every row with the same hash value byte-by-byte to make sure a duplicate row does not enter the Teradata SET table. There could be hundreds to thousands to even millions of duplicate values in the Primary Index so this duplicate row checking can be exhausting.

To solve this problem you can place a Unique Secondary Index on one of the columns in the table. That way you ensure that no row could possibly be duplicated because the first check Teradata performs is to make sure the Unique Secondary Index column remains unique. There will never be a duplicate row check.

An even easier solution is to make the SET table a MULTISET Table.

NUSI considerations

You create a NUSI on a column that has a high Value Access Frequency. A NUSI will speed up access to non-unique columns used in the WHERE clause of a query. A NUSI can return multiple rows from the table so the process is not as fast as utilizing a USI, but it is enormously faster than doing a Full Table Scan.

Teradata accesses data by:

- **Primary Index**
- **Secondary Index**
- **Full Table Scan**

A NUSI can be effective for speeding up queries, but the Parsing Engine will make the decision if it will use the NUSI or do a Full Table Scan (FTS).

If the NUSI is strongly selective, the PE will most likely choose to use the NUSI. It is important to Collect Statistics on every NUSI in a table. If statistics are NOT collected, the PE is more likely do a Full Table Scan than use the NUSI.

If you are not sure about adding a NUSI to a column, you can prototype the NUSI. Take the SQL before adding the NUSI column and place the word EXPLAIN in front of the SQL. The PE will show you its plan. You can see estimated time and access paths. You will likely see that the plan will do "An All-AMPs retrieve scanning all rows". This is a full table scan. You can then add the NUSI to that column and then Collect statistics. The next step is to re-run the EXPLAIN. If the NUSI is being utilized, then it might be worth keeping. If the query is still doing the Full Table Scan (FTS), then drop the NUSI. Never keep a NUSI that is not being utilized because it takes up unnecessary space and maintenance.

Multi-Column NUSI Columns

You are usually better off staying away from multi-column NUSI indexes. It is better to create two different NUSI values, collect statistics on both individual columns, and let the PE analyze the best path. Quite often when queries are using an AND operator, the PE will utilize a bitmap plan which can save a lot of query time.

The above advice is sound, but read the next paragraph for a brilliant tip that does not follow the above advice.

Multi-Column NUSI Columns used as a Covered Query

The next tip is really clever, but it goes against the previous NUSI theory. We have always believed that it is better to have two separate NUSI columns than combine them together. Although this is a good rule of thumb to follow, we will now present a case where this is not true.

This case is called a **Covered Query**. When a user asks a question to Teradata, the PE will come up with a plan to get the data. Even though the users always query data from tables or views, the PE will sometimes get the answer set from the Secondary Index Subtable instead. This is called a "covered query".

Let's say that a company has a table with 256 columns (the maximum columns allowed). Let's also state that in most cases, the users typically query only about 12 columns from this table. An extremely clever trick is to take all twelve of the most popular columns and make a multi-column NUSI containing all 12 columns. When users query the base table and ask for any of the 12 columns, the PE will be able to take the information from the smaller secondary index subtable. Plus, anytime the base table changes, the changes will be reflected in the subtable. The syntax to create the subtable would be easy.

```
Command: CREATE INDEX (Colname, Colname2, Colname3, Colname4,
         Colname5, Colname6, Colname7, Colname8, Colname9,
         Colname10, Colname11, Colname12) ON TableName;
```

Then you could easily collect statistics:

```
Command: COLLECT STATISTICS ON TableName INDEX (Colname,
         Colname2, Colname3, Colname4, Colname5, Colname6,
         Colname7, Colname8, Colname9, Colname10, Colname11,
         Colname12);
```

You could then run some sample queries with the word EXPLAIN in front and you will probably notice that the query is being built from the subtable and not the base table.

Are you not now very glad you bought this book? I am determined to get you a raise! Tell your boss this idea came to you in a dream. The boss will think you're brilliant, until the boss reads this book too!

Value-Ordered NUSIs

"Look at life through the windshield, not the rearview mirror."

– **Byrd Baggett**

Italians are known for fast cars and fast driving so there is no need for a rear-view mirror because what's behind them doesn't matter. A value ordered NUSI never looks behind them in a secondary subtable because all values requested are in the proper order. Teradata can find the first value needed and then look forward until the destination is reached. This allows data to Arriva Derchi quickly.

A Value-Ordered NUSI is usually done on columns where the queries will be looking for a range. An excellent example might be a date column. The words Value-Ordered tell Teradata to store the secondary index subtable in an order based off of the column value and not the hash value. This gives the NUSI the ability to quickly find the rows that fall in the range of values that has been requested by the user's query. For more information about Value-Ordered NUSIs, see the secondary index chapter.

A Strongly Vs Weakly Selective NUSI

Teradata's Parsing Engine will do a Full Table Scan instead of using a NUSI if the NUSI is weakly selective. If you were in a large crowd of people mixed with an equal amount of men and women and wanted to find all the men, the query would probably be weakly selective because most likely about 50% of the people would be men. However, if you wanted to find all of the people from Siberia, you could have a strongly selective index. If Teradata can do a Full Table Scan as fast as if it were to use the index then the index is said to be weakly selective.

If a user runs a query that involves two NUSI values, but only one of the values is strongly selective, the PE will choose to use the index on the strongly selective value. This will return a reasonable amount of rows into spool where the remaining column can then be checked as a final filter. The Parsing Engine is very smart.

A formula for calculating a strongly selective NUSI

Below is a box with two formulas. Answer formula number one to get Answer 1. Then Answer formula number two to get Answer 2. If Answer 1 is less than Answer 2 you may have a strongly selective NUSI.

You can also create a NUSI, collect statistics and run an EXPLAIN to see if the PE thinks it is good enough to use.

Typical Rows Per Value / Total Rows in the Table

= ________________________ Answer 1

If Answer 1 is less than Answer 2 you have a Potentially good NUSI.

Typical Row Size / Typical Block Size

= ________________________ Answer 2

Formula number one has the numbers available to you from the data demographics in the Extended Logical Data Model. You might be wondering about the second formula. Where do you get the information for formula two? I will give you some estimates and rules of thumb on the next couple of pages and you can do some checking for yourself.

For example, it may be that (in most tables) the Typical Row Size is 480 bytes, so you would stick the number 480 in for Typical Row Size. The Typical Block Size is generally about 48 K, so you would stick 48000 in for Typical Block Size.

Typical Row Size

> *"He who controls the past commands the future. He who commands the future conquers the past."*
>
> **– George Orwell**

If you want your Teradata Warehouse to be above other warehouses then you must be its Big Brother and know everything about your data size.

You can estimate the Typical Row Size by running either the Help Column Command or the Show Table Command. Below the HELP column TableName.* command returns the Max_Length for each column. Add this up plus an additional 14 bytes (row overhead) and you will be close to your Typical Row Size.

Command: **HELP COLUMN Student_Table.* ;**

```
 6 rows returned

ColumnName    Type   Nullable   Format    Max_Length
------------------ -------- ----------- ---------- -----------------
Class_code     CF       Y        X(2)          2
First_name     CV       Y        X(12)        12
Grade_pt        D       Y      ---99           4
Last_name      CF       Y        X(20)        20
Student_ID      I       Y      -10(9)          4
```

Typical Block Size

Teradata brilliantly stores data in blocks and blocks inside cylinders. Most techies like me would say, “So What – Everyone else does also.” It is the way that Teradata proceeds to process it that makes the database special. Teradata places data inside a block and will continue to allow the block to grow as data is added until the block reaches a maximum block size. Then Teradata will do a block split. A block split takes half the data and stores it in a separate block. Now the data is comprised of two blocks of data. This can continue forever. Teradata can store incredible amounts of data. Some tables at Teradata data warehouse sites reach billions of rows.

Think of this concept as if someone was blowing up a balloon and they had an endless amount of breath. When the first balloon is filled, a second balloon would be made available and half of the air in the first balloon would be transferred. Then the person would continue to blow up both balloons simultaneously. When they both reached the maximum size they would split again.

Knowing what you now know, you can understand that most Teradata data warehouses have their maximum block size at 64K. That is 64,000 bytes. When blocks split from one block of 64 K they turn into two blocks of 32 K. When these two blocks reach maximum block size they both split into 4 blocks of 32 K.

That is why in most cases you can estimate the Typical Block size at 48 K because this is the average between 64K and 32K. So, on average you can run Formula 2 as:

480 divided by 48000 = Answer 2

Final Result for Primary and Secondary Indexes

Customer Table has 2,000,000 rows

	Cust_no	CName	Phone
PK & FK	PK SA		
ACCESS			
Value Acc Freq	1M	5K	0
Join Acc Freq	100K	0	0
Value Acc Rows	100M	5K	0
DATA DEMOGRAPHICS			
Distinct Rows	2M	1900K	1950K
Max Rows Per Value	1	2	1
Max Rows Null	0	12	500
Typical Rows Per Value	1	1	1
Change Rating	0	2	5
(PRI) Fill in UPI or NUPI	**UPI**	**XXXX**	**XXXX**
(SEC) Fill in USI or NUSI	**XXXX**	**NUSI**	**XXXX**
Contact the Customer?			

Employee table has 20,000 rows

	EMP	DEPT	LNAME	FNAME	SAL
PK & FK	PK SA	FK			
ACCESS					
Value Acc Freq	6K	5K	100	0	0
Join Acc Freq	7K	6K	0	0	0
Value Acc Rows	70K	50K	0	0	0
DATA DEMOGRAPHICS					
Distinct Rows	20K	5K	12K	N/A	N/A
Max Rows Per Value	1	50	1K	N/A	N/A
Max Rows Null	0	12	0	N/A	N/A
Typical Rows Per Value	1	15	3	N/A	N/A
Change Rating	0	2	1	N/A	N/A
(PRI) Fill in UPI or NUPI	**UPI**	**XXXX**			
(SEC) Fill in USI or NUSI	**XXXX**	**NUSI**	**XXXX**		
Contact the Customer?					

We ran our formula on both NUSI candidates and they both qualified. At this point our Primary Index and Secondary Indexes are done. The columns were not even questionable, so I don't need to place a check in the Contact the user box.

Chapter 7 — Denormalization

"Most databases denormalize because they have to, but Teradata denormalizes because it wants to."

– Steve Wilmes - CTO Coffing DW

Denormalization is the process of implementing methods to improve SQL query performance. The biggest keys to consider when **deciding to denormalize** a table are **PERFORMANCE** and **VOLATILTITY**. Will performance improve significantly and does the volatility factor make denormalization worthwhile?

Improved performance is an admirable goal, however, one must be aware of the hazards of denormalization. Denormalization will always reduce the amount of flexibility that you have with your data and can also complicate the development effort. In addition, it will also increase the risk for data anomalies. Lastly, it could also take on extra I/O and space overhead in some cases.

Others believe that **denormalization** has a **positive effect on application coding** because some feel it will **reduce the potential for data problems**.

Either way you should consider denormalization if users run certain queries over and over again and speed is a necessity. The key word here is performance. Performance for known queries is the most complete answer. If you don't know what queries to expect, then there is no reason to denormalize. However, if users execute certain queries on a regular basis and they need more performance, denormalization is a sound practice. To be able to denormalize, you must know your users applications or requirements.

Teradata has some clever ways to allow designers to keep their normalized model in place, and take advantage of denormalization. This is best done with a join index described later in this book. Below are some other methods:

- **Derived Data**
- **Temporary Tables (Derived, Volatile or Global)**
- **Pre-Joining Tables**

It is a great idea whenever you denormalize from your logical model to include the denormalization in **"The Denormalization Exception Report"**. This report keeps track of all deviations from 3rd normal form in your data warehouse.

Derived Data

"Never mix alcohol with calculated data because it is bad to drink and derive."

– Tera-Tom Coffing

If you drink and derive you should stand alone and let someone else derive. **Derived data** is **data** that is **calculated from other data**. For instance, taking all of the employee's salaries and averaging them would calculate the Average Employee Salary. It is important to be able to determine whether it is better to calculate derived data on demand or to place this information into a summary table.

The key factors for deciding whether to **calculate** or **store stand alone derived data** are:

- **Response Time Requirements**
- **Access Frequency of the request**
- **Volatility of the column**

Response Time Requirements – Derived data can take a period of time to calculate while a query is running. If user requirements need speed and their requests are taking too long, then you might consider denormalizing to speed up the request. If there is no need for speed, then be formal and stay with normal.

Access frequency of the request – If one user needs the data occasionally then calculate on demand, but if there are many users requesting the information daily, then consider denormalizing so many users can be satisfied.

Volatility of the column – If the data changes often, then there is no reason to store the data in another table or temporary table. If the data never changes and you can run the query one time and store the answer for a long period of time then you may want to consider denormalizing. If the game stays the same there is no need to be formal – make it denormal.

When you look at the above considerations you begin to see a clear picture. If there are several requests for derived data, and the data is relatively stable, then denormalize and make sure that when any additional requests are made the answer is ready to go.

Storing Aggregates

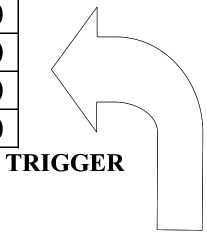
"The mathematical sciences particularly exhibit order, symmetry, and limitation; and these are the greatest forms of the beautiful."

– Aristotle (384-322 BC)

Below is the Order_Table and the Order_Line_Item_Table. This technique exhibits Orders, has symmetry, no limitations, and is a beautiful technique.

Order_Table

Order_No	Customer_No	Order_Date	Order_Total
100	101200	980101	50000.50
200	200200	980102	52200.50
300	300200	980103	8000.50
400	101200	980103	250000.50
500	300200	980104	50.50



Order_Line_Item_Table

Order_No	Line_Item	Product_No	Price	Quantity	Line_Cost
100	1	77	10000.10	3	30000.30
100	2	99	1000.01	20	20000.20
200	1	44	52200.50	1	52200.50
300	1	43	2000.25	2	4000.50
300	2	334	500.00	8	4000.00
400	1	134	1000.00	200	200000.00
400	2	456	500.00	100	50000.50
400	3	123	0.50	1	0.50
500	1	675	25.25	2	50.50

In the Order_Table, we have decided to track the Order_Total. This information is taken from the Order_Line_Item_Table. To keep the Order_Total updated in the Order_Table, we attach a TRIGGER that adds or subtracts the Order_Total when a line item is added or subtracted from the Order_Line_Item_Table.

Pre–Joining Tables

Pre-joining tables is the method of joining two or more tables together so join queries can run much quicker. A Pre-join table example between the Employee and Department tables is displayed below:

TABLE 1 – Employee Table

EMP	DEPT	LNAME	FNAME	SAL
PK	FK			
1	40	BROWN	CHRIS	65000.00
2	20	JONES	JEFF	70000.00
3	20	NGUYEN	XING	55000.00
4	40	JACOBS	SHERRY	30000.00
5	10	SIMPSON	MORGAN	40000.00
6	30	HAMILTON	LUCY	20000.00

TABLE 2 – Department Table

DEPT	DNAME
Primary Key	
10	HR
20	Sales
30	Finance
40	IT

TABLE 3 – Employee_Department Table (Pre-Joined Table)

EMP	DEPT	DNAME	LNAME	FNAME	SAL
PK	FK				
1	40	IT	BROWN	CHRIS	65000.00
2	20	Sales	JONES	JEFF	70000.00
3	20	Sales	NGUYEN	XING	55000.00
4	40	IT	JACOBS	SHERRY	30000.00
5	10	HR	SIMPSON	MORGAN	40000.00
6	30	Finance	HAMILTON	LUCY	20000.00
…	…	…	…	…	…

What we lose here is space and I/O, and problems if a department changes names. However, we can gain greater speed if these two tables were joined often. This is a great technique. We would however place this table information in our “Denormalization Exception Report”.

Repeating Groups

"We are what we repeatedly do. Excellence therefore is not an act but a habit."

– Aristotle

There is a company who has always excelled for customers repeatedly and every employee has made it a habit. Below we have a table that a major retailer from Arkansas uses. Tera-Tom trained there in the early 1990's and it was a very exciting trip. He learned as much from them as they learned from him. This great retailer likes to track the best and the worst selling products per week. They get together on Saturday for a managers' meeting to discuss how to help the worst and emulate the best. The table below actually has around 150,000 rows because this retailer is tracking 150,000 items. Both the item numbers and quantities sold have been made up to protect the innocent.

Product_ID	Mon	Tue	Wed	Thur	Fri	Sat	Sun
100100	5050	6000	4500	4345	7000	12000	9000
100200	100	125	235	400	443	780	650
100300	15400	12435	13900	17400	23444	32333	6050
200100	8	5	9	7	13	17	13
300200	100000	99665	38882	67454	98745	150003	200124
300300	6050	7200	4440	4321	7540	3400	6000
400100	210	135	265	420	543	754	620
400200	5050	6000	4500	4345	7000	12000	9000
400300	138	522	932	72	135	174	135

The above table actually saves space because if the table did not use repeating groups there would be 7 times the rows. The Product_ID would be repeated 7 times and because there would be 7 times more rows you can add around 14 extra bytes of row overhead per row. Instead of 150,000 rows this table would have over 1 million rows.

The above table is not about saving space. The customer knows that the users need information about products on a weekly basis. They also need to compare how Monday did versus Tuesday, etc. This table will allow users to get the information about a product during a specific week by reading only one row and not seven. Although this table violates even first normal form it can be a great technique for known queries.

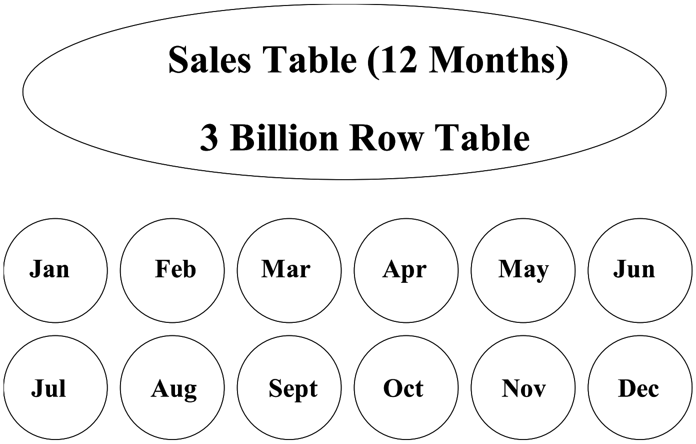
One of the problems with repeating groups is that they can sometime expand farther than expected. For example, we might track phone numbers in a table. We make room for 3 different phone numbers and then we discover somebody has 4 different telephone numbers. The above table is sound because we can be pretty sure that there will never be more then 7 days in a week, unless you are Paul McCartney or John Lennon.

Horizontal Partitioning

"It's not the data load that brings us down.
It's the way you carry it."

– Gareth Walter - COO Coffing DW

Teradata is the database of choice at many of the largest data warehouses in the world. Each and every quarter, a Teradata database is breaking a new threshold somewhere. Teradata holds tables at some sites that are billions of rows in size. Tables this large can become difficult to load and time consuming to query. **Horizontal partitioning** breaks up the **rows of one huge table** and places them in **separate partitions or tables**. This allows the data-loading specialists to lighten the load and users to bring down their query times.



Horizontal Partitioning

We have taken a 3 billion row yearly Sales table and broken it into 12 separate monthly Sales tables. This allows the load specialists to load monthly data into an empty table with FastLoad. This is magnitudes of order faster than having to load data into a huge yearly table with MultiLoad.

The users also benefit because their monthly queries run about 12 times faster. The only negative is that when users want to compare one month to another they have to join the tables together. The good news though is because the tables each have the same Primary Index the joins work very efficiently.

This is where you must know your user environment. If the majority of the sales queries are for a specific month then horizontal partitioning is a brilliant idea.

Vertical Partitioning

There are companies today that have tables that are extremely wide. This means they have many columns. Some tables even reach the maximum columns in a single table. In many cases, users will actually only query 8 of these columns on a regular basis. Vertical partitioning is a clever way to get the most frequently accessed columns quickly and easily, yet still logically link the tables together as one. Below is a table called Table A. This table has 256 columns. We will break it into two separate tables with the only common column being the Primary Key.

Table A 256 Columns (Before)

Col 1	Col 2	Col 3	Col 4	Col 5	Col 6	Col 7	Col 8	●	●	Col 252	Col 253	Col 254	Col 255	Col 256
Primary Key														

Table A with Vertical Partitioning

Col 1	Col 2	Col 3	Col 4	Col 5	Col 6	Col 7	Col 8
Primary Key							

Same Primary Key

Table A2 with Vertical Partitioning

Col 1	Col 9	Col 10	Col 11	Col 12	Col 13	Col 14	Col 15	●	●	Col 252	Col 253	Col 254	Col 255	Col 256
Primary Key														

Now that we have vertically partitioned our 256-column table into two separate tables, users can query their 8 columns much faster. If users need additional information contained in columns 9 through 256 they can perform a join of the two tables. Once again, knowing the user environment can rapidly speed up performance on known queries. We will use another technique similar next in something called a “covered query”.

Covered Query

This is an example that can speed up users queries just like vertical partitioning, but with some additional benefits. The users generally only need 8 of these columns in the majority of their queries. The 256 columns make the queries slower.

Vertical Partitioning would break the two tables up and place the top 8 columns in one table and the remaining columns in another table. Instead we will use a "covered query". If the Teradata Parsing engine (PE) can use a Secondary Index subtable to get all the data needed to satisfy a query, it is called a "Covered Query". Users never query the secondary index subtable, but the PE is so smart that if the data needed to satisfy a query can come from the smaller subtable, the PE will make the decision on its own to go there instead. All we have to do is create a Non-Unique Secondary Index, collect statistics on the index and the job is done.

The brilliance behind this technique is that users don't need to change their current applications or queries. Also, any updates to Table A will be reflected and maintained in the secondary index subtable. The only penalty is the space a secondary subtable takes up. Remember that Teradata only allows a maximum of 16 columns in an index. If the users have more than 16 columns that are queried the most often, you will need to use the vertical partitioning technique on the previous page.

Table A 256 Columns

Col 1	Col 2	Col 3	Col 4	Col 5	Col 6	Col 7	Col 8	●	●	Col 252	Col 253	Col 254	Col 255	Col 256
Primary Key														

Secondary Index Subtable

Col 1	Col 2	Col 3	Col 4	Col 5	Col 6	Col 7	Col 8

**Users Still Query TableA,
But anytime a query asks
For any combination of the
Top 8 columns the PE will
Retrieve the information
From the smaller subtable!**

```
CREATE INDEX (Col1, Col2, Col3, Col4, Col5, Col6, Col7, Col8)
        ON TableA;

COLLECT STATISTICS ON TableA
   INDEX ((Col1, Col2, Col3, Col4, Col5, Col6, Col7, Col8);
```

Single-Table Join Indexes

Below are three tables. They are Customer_Table, Order_Table and the Order_Line_Item_Table. Because the Order_Table is joined quite often to both the Customer_Table and the Order_Line_Item_Table, we had a difficult time picking the Primary Index in the Order_Table. Both Order_No and Customer_No had a High Join Access Frequency. Each table is scaled down for illustration purposes.

Customer_Table

Customer_No	Customer_Name	Phone
101200	101200	980101
200200	200200	980102
300200	300200	980103

Order_Table

Order_No	Customer_No	Order_Date	Order_Total
100	101200	980101	50000.50
200	200200	980102	52200.50
300	300200	980103	8000.50
400	101200	980103	250000.50
500	300200	980104	50.50

Order_Line_Item_Table

Order_No	Line_Item	Product_No	Price	Quantity	Line_Cost
100	1	77	10000.10	3	30000.30
100	2	99	1000.01	20	20000.20
200	1	44	52200.50	1	52200.50
300	1	43	2000.25	2	4000.50
300	2	334	500.00	8	4000.00
400	1	134	1000.00	200	200000.00
400	2	456	500.00	100	50000
400	3	123	0.50	1	0.50
500	1	675	25.25	2	50.50

Let's assume that we made the Primary Index Order_No in our Order_Table. The Order_Table joins extremely fast to the Order_Line_Item_Table because both tables use the Order_No as their respective Primary Indexes. The Customer_Table does not join well with the Order_Table. We can create a Single Join-Index on Order_Table and make the Join-Index on Customer_No. What we have essentially done is made a duplicate copy of the Order_Table with a Primary Index on Customer_No. Wow! Both joins now work like lightning, except now lightning does strike twice! For more information on Join indexes see the Join Index chapter.

Multi-Table Join Indexes

Below are two tables. They are Customer_Table and the Order_Table. Because the Order_Table is joined quite often to the Customer_Table and because both have different Primary Indexes, the joins are taking too much time. The users have requested that something be done. A Join Index will pre-join the tables together, be transparent to the users, and maintain integrity when data is added or deleted from either table. Each table is scaled down for illustration purposes.

Customer_Table

Customer_No	Customer_Name	Phone
101200	101200	980101
200200	200200	980102
300200	300200	980103

Order_Table

Order_No	Customer_No	Order_Date	Order_Total
100	101200	980101	50000.50
200	200200	980102	52200.50
300	300200	980103	8000.50
400	101200	980103	250000.50
500	300200	980104	50.50

```
CREATE JOIN INDEX Ord_Cust_Index AS
            SELECT  C.Customer_No,
                    ,Customer_Name
                    ,Phone
                    ,Order_No
                    ,Order_Date
                    ,Order_Total
           FROM Order_Table as O
           INNER JOIN Customer_Table as C
           ON C.Customer_No = O.Customer_No;
```

Anytime a user joins the Customer_Table to the Order_Table, the Parsing Engine (PE) will tell the AMPs to get the data from their Join Index table. In other words there will be no join because the rows are already physically joined together. For more information on Join indexes see the Join Index chapter.

Temporary Tables

Setting up Derived, Volatile or Global Temporary tables allow users to use a temporary table during their session. This is a technique where everyone wins. A great example might be this: Let's say you have a table that has 120,000,000 rows. Yes, the number is 120 million rows. It is a table that tracks detail data for an entire year. You have been asked to run calculations on a per month basis. You can create a temporary table, insert only the month you need to calculate, and run queries until you logoff the session. Your queries in theory will run twelve times faster. After you logoff, the data in the temporary table goes away.

TABLE 1 - Employee Table

EMP	DEPT	LNAME	FNAME	SAL
PK	FK			
1	40	BROWN	CHRIS	65000.00
2	20	JONES	JEFF	70000.00
3	20	NGUYEN	XING	55000.00
4	40	JACOBS	SHERRY	30000.00
5	10	SIMPSON	MORGAN	40000.00
6	30	HAMILTON	LUCY	20000.00

TABLE 2 - Department Table

DEPT	DEPT_NAME
PK	
10	Human Resources
20	Sales
30	Finance
40	Information Technology

TABLE 3 – Dept_Salary Table (Temporary Table)

DEPT	Sum_SAL	Count_Sal	Avg(Sal)
10	40000.00	1	40000
20	125000.00	2	75000
30	20000.00	1	20000
40	95000.00	2	47500

Derived Tables

Derived tables are **temporary tables** that are utilized within a **user's query**. When a derived table is created within a user's query, it is local to that query. This means that no other user can access this derived table. Derived tables can be very useful because they give the user an alternative to creating views or using permanent temporary tables.

Derived tables are created dynamically and **materialized** in the **user's spool space**. This means that there are **no special privileges required** to create this type of table. All the user needs is enough spool space to create the table. The user does not have to worry about dropping these types of tables because they are only materialized for a specific SQL query and then automatically dropped when the query is completed. Yes, it's true! **Rows** are actually **deleted** when the **query ends** and the table is gone!

An example of a derived table would be as follows. Make sure you pay close attention to the syntax after the FROM statement.

```
SELECT Avgsal FROM
(SELECT AVG(salary) FROM Employee_table) as CDW(Avgsal);
```

The above example uses a derived table named CDW with a column alias called Avgsal and its data value is the AVERAGE (AVG) aggregate.

```
SELECT Last_name, Dept, Salary, Avgsal
FROM (SELECT Dept_No, AVG(salary)
          FROM Employee_Table
          GROUP BY Dept_no)
          AS CDW(Dept, Avgsal),
         Employee_table
WHERE Dept = Dept_no
AND      Salary > Avgsal;
```

This example uses the same derived table named CDW with column aliases called Avgsal and Dept, which will be joined to the Employee table. This query will find all the employees who make more than the average salary in their particular department.

The key thing to remember is that the derived table starts after the **FROM** keyword. After the FROM keyword, parentheses will be around the SELECT statement, which creates the derived table. The SELECT of a derived table does not have to be a simple. A derived table can support inner joins, outer joins, subqueries, correlated subqueries, set operators, aggregates, and OLAP functions. When the query is over, the table is dropped and the data is gone.

Volatile Temporary Tables

Volatile tables have two characteristics in common with derived tables. They are **materialized in spool** and are **unknown to the Data Dictionary**. Because they are **unknown to the Data Dictionary**, **Rows are lost** following all system restarts. However, unlike a derived table, a volatile table may be utilized multiple times, and in more than one SQL statement throughout the life of a session. This feature allows for additional queries to utilize the same rows in the temporary table without requiring the rows to be rebuilt. The ability to use the rows multiple times is the biggest advantage over derived tables. **Rows** are always automatically **deleted** when a user **logs off**.

An example of how to create a volatile table would be as follows:

```
CREATE VOLATILE TABLE Sales_Report_vt, LOG
                (
                        Sale_Date       DATE
                        ,Sum_Sale       DECIMAL(9,2)
                        ,Avg_Sale       DECIMAL(7,2)
                        ,Max_Sale       DECIMAL(7,2)
                        ,Min_Sale       DECIMAL(7,2)
                )
ON COMMIT PRESERVE ROWS ;
```

Now that the **Volatile Table has been created**, the table must be **populated** with an **INSERT/SELECT** statement like the following:

```
INSERT INTO Sales_Report_vt

                SELECT  Sale_Date
                           ,SUM(Daily_Sales)
                           ,AVG(Daily_Sales)
                           ,MAX(Daily_Sales)
                           ,MIN(Daily_Sales)

                 FROM Sales_Table
                GROUP BY Sale_Date;
```

The LOG option indicates there will be transaction logging of before images. The ON COMMIT PRESERVE ROWS means that at the end of a transaction, the rows in the volatile table will not be deleted. The information in the table remains for the entire session. Users can ask questions to the volatile table until they **log off** or at **system end. Then the table and data go away.**

Global Temporary Tables

Global Temporary Tables are similar to volatile tables in that they are local to a user's session. However, when the table is created, the **definition is stored** in the **Data Dictionary**. In addition, these tables are **materialized** in a permanent area known as **Temporary Space**. Because of these reasons, global tables can survive a system restart and the table definition will not be discarded at the end of the session. However, when a session normally terminates, the rows inside the Global Temporary Table will be removed. Lastly, Global tables require no spool space. They use **Temp Space**.

Users from other sessions cannot access another user's materialized global table. However, unlike volatile tables, once the table is de-materialized, the definition still resides in the Data Dictionary. This allows for future materialization of the same table. Rows are automatically **deleted** at **session end**, but the tables definition stays in the data dictionary. That way Global Temporary Tables are **always empty** when **first materialized**. Although rows are automatically deleted at session end the table can **optionally be emptied** at the end of **each** and every **transaction**. If the global table definition needs to be dropped, then an explicit DROP command must be executed.

An example of how to create a global temporary table would be as follows:

```
CREATE GLOBAL TEMPORARY TABLE Sales_Report_gt, LOG
                        (
                         Sale_Date      DATE
                         ,Sum_Sale      DECIMAL(9,2)
                         ,Avg_Sale      DECIMAL(7,2)
                         ,Max_Sale      DECIMAL(7,2)
                         ,Min_Sale      DECIMAL(7,2)
                          )
              PRIMARY INDEX(Sale_Date)
             ON COMMIT PRESERVE ROWS ;
```

Now that the Global Temporary Table has been created, the table must be populated with an INSERT/SELECT statement like the following:

```
INSERT INTO Sales_Report_gt
                  SELECT  Sale_Date
                              ,SUM(Daily_Sales)
                              ,AVG(Daily_Sales)
                             ,MAX(Daily_Sales)
                             ,MIN(Daily_Sales)
                        FROM Sales_Table
                        GROUP BY Sale_Date ;
```

Chapter 8 — Secondary Indexes

"I don't skate to where the puck is, I skate to where I want the puck to be."

– Wayne Gretzky

What Wayne Gretzky is saying is that he finds the best path to the goal and expects the puck to be there when he arrives for the shot. Secondary indexes are similar because they define a path that will deliver the data quickly to meet the users' expected goals. A secondary index is an alternate path to the data. They can be defined as a Unique Secondary Index (USI) or a Non-Unique Secondary Index (NUSI). Without any secondary indexes, your data warehouse could be skating on thin ice!

When it comes to working with large amounts of data that is centrally located, demands for performance to access this data is key. So what can a user do to influence the way data is accessed? The first rule of thumb, which is essential when it comes to working with centralized databases today, is to know your data. Second, understand how Teradata manages data distribution and what a user can do to enhance performance. A query that utilizes a Primary Index in the WHERE column is the fastest path to the data. A query that utilizes a Secondary Index will provide an alternate path to the data and be the second fastest access method. This chapter is dedicated to secondary indexes.

Secondary Indexes

Secondary Indexes provide another path to access data. Let's say that you were planning a road trip to your hometown. To determine the best way to get there, you need to utilize a map. This map will give you many alternatives to plan your trip. In this case, you need to get there, ASAP. So you choose the best route to get there in the shortest period of time. Secondary indexes work very similar to this above example because they provide another path to the data. Teradata allows up to 32 secondary indexes per table. Keep in mind that the base table data rows aren't redistributed when secondary indexes are defined. The value of secondary indexes is that they reside in a subtable and are stored on all AMPs, which is very different from how the primary indexes (part of base table) are stored. Keep in mind that Secondary Indexes (when defined) do take up additional space.

Secondary Indexes are frequently used in a WHERE clause. The Secondary Index can be changed or dropped at any time. However, because of the overhead for index maintenance, it is recommended that index values should not be frequently changed.

There are two different types of Secondary Indexes, Unique Secondary Index (USI), and Non-Unique Secondary Index (NUSI). Unique Secondary Indexes are extremely efficient. A USI is considered a two-AMP operation. One AMP is utilized to access the

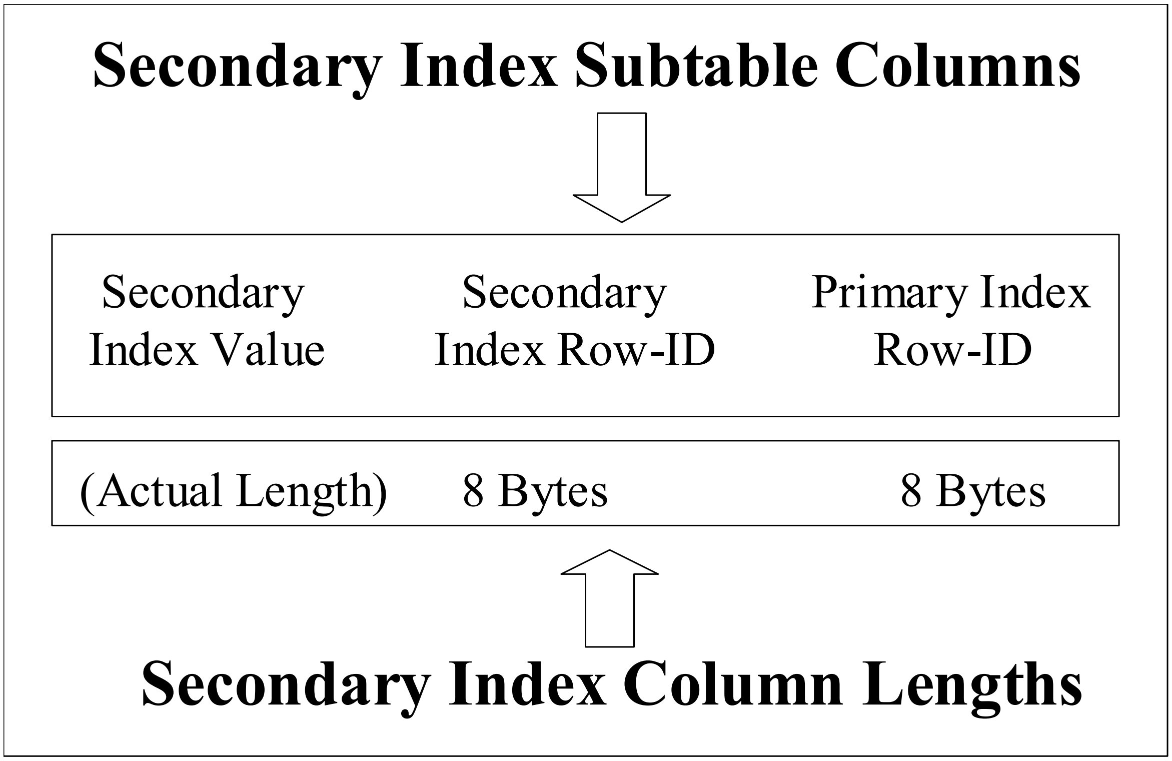
USI subtable row (in the Secondary Index subtable) that references the actual data row, which resides on the second AMP.

A Non-Unique Secondary Index is an All-AMP operation and will usually require a spool file. Although a NUSI is an All-AMP operation, it is faster than a full table scan.

Secondary indexes can be useful for:

- Satisfying complex conditions
- Processing aggregates
- Value comparisons
- Matching character combinations
- Joining tables

Below is a general illustration of a secondary index subtable row:



Secondary Index Subtable Columns

Secondary Index Value	Secondary Index Row-ID	Primary Index Row-ID
(Actual Length)	8 Bytes	8 Bytes

Secondary Index Column Lengths

Unique Secondary Index (USI)

"Life is not the candle or the wick, it's the burning."

– David Joseph Schwartz

A **Unique Secondary Index (USI)** is at most a **two-AMP operation** that will always return a maximum of one row. A USI will make your queries burn bright with speed. If you want users to return don't put a candle in the window, put an USI in the warehouse.

When a user creates a Unique Secondary Index, Teradata automatically creates a Secondary Index Subtable. The **USI subtable** will contain the:

Secondary Index Value
Secondary Index Row ID
Primary Index Row ID

When a user writes an SQL query that has a USI in the WHERE clause, the Parsing Engine will hash the Secondary Index Value. The output is the Row Hash of the USI. The PE creates a request containing the Row Hash and gives the request to the Message Passing Layer (which includes the BYNET software and network). The Message Passing Layer uses a portion of the Row Hash to point to a bucket in the Hash Map. That bucket contains an AMP number to which the PE's request will be sent. The AMP gets the request and accesses the Secondary Index Subtable pertaining to the requested USI information. The AMP will check to see if the Row Hash exists in the subtable and double check the subtable row with the actual secondary index value. Then, the AMP will create a request containing the Primary Index Row ID and send it back to the Message Passing Layer. This request is directed to the AMP with the base table row, and the AMP easily retrieves the data row.

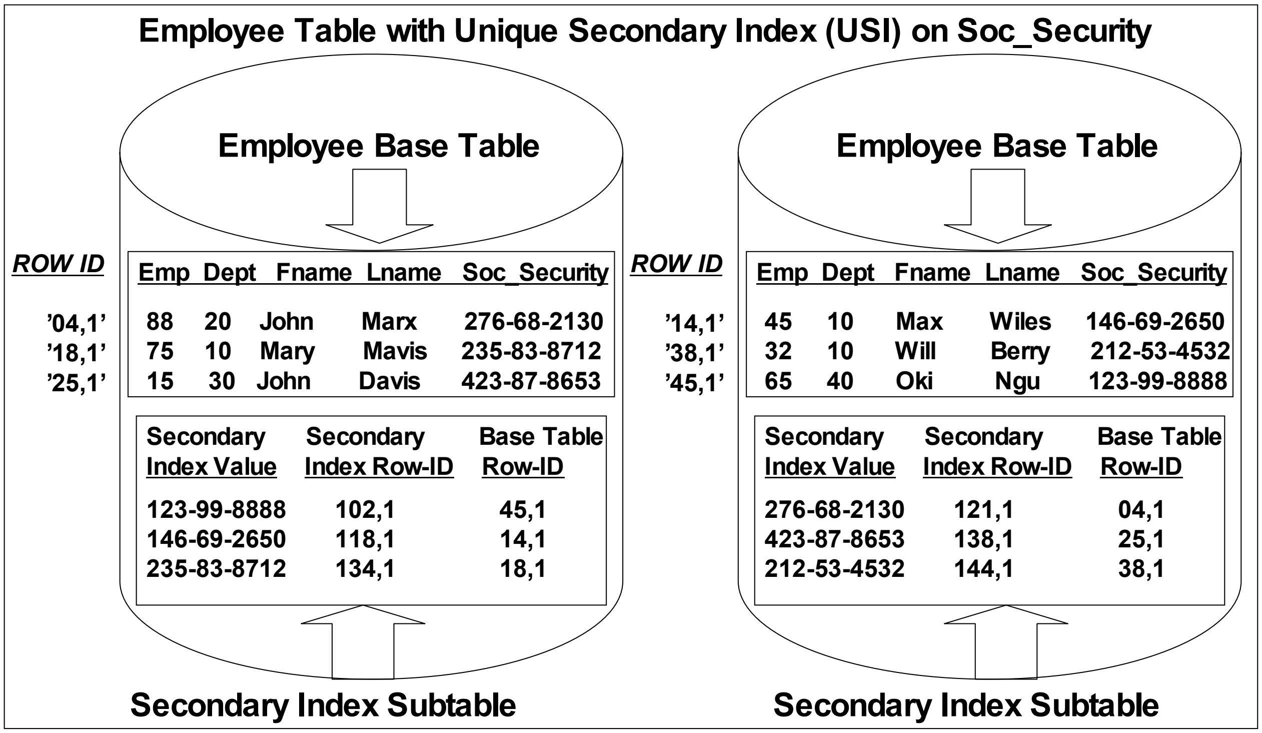
Unique Secondary Indexes **do many great things** such as:

- To provide an **alternative path** to retrieving data by providing an option, that allows **avoidance** of the treacherous **Full Table Scan (FTS)**.

- It provides a **Uniqueness constraint** to a column that will ensure a column value is unique in a table. Users **can't insert rows** with a duplicate USI value preserving data integrity.

USI Subtable Example

When a USI is designated on a table, each AMP will build a subtable. If you create 32 USI indexes on a table, then each AMP will build 32 separate subtables. Therefore, choose your Secondary Indexes wisely, because space is used when these indexes are created. When a user inputs SQL that utilizes a USI in the WHERE clause, then Teradata will know that either one row or no rows can be returned. Reason, the column in the WHERE clause is unique. The following example illustrates the how a USI Subtable is created and how it works to speed up queries.



Employee Table with Unique Secondary Index (USI) on Soc_Security

ROW ID	Emp	Dept	Fname	Lname	Soc_Security
'04,1'	88	20	John	Marx	276-68-2130
'18,1'	75	10	Mary	Mavis	235-83-8712
'25,1'	15	30	John	Davis	423-87-8653

Secondary Index Value	Secondary Index Row-ID	Base Table Row-ID
123-99-8888	102,1	45,1
146-69-2650	118,1	14,1
235-83-8712	134,1	18,1

ROW ID	Emp	Dept	Fname	Lname	Soc_Security
'14,1'	45	10	Max	Wiles	146-69-2650
'38,1'	32	10	Will	Berry	212-53-4532
'45,1'	65	40	Oki	Ngu	123-99-8888

Secondary Index Value	Secondary Index Row-ID	Base Table Row-ID
276-68-2130	121,1	04,1
423-87-8653	138,1	25,1
212-53-4532	144,1	38,1

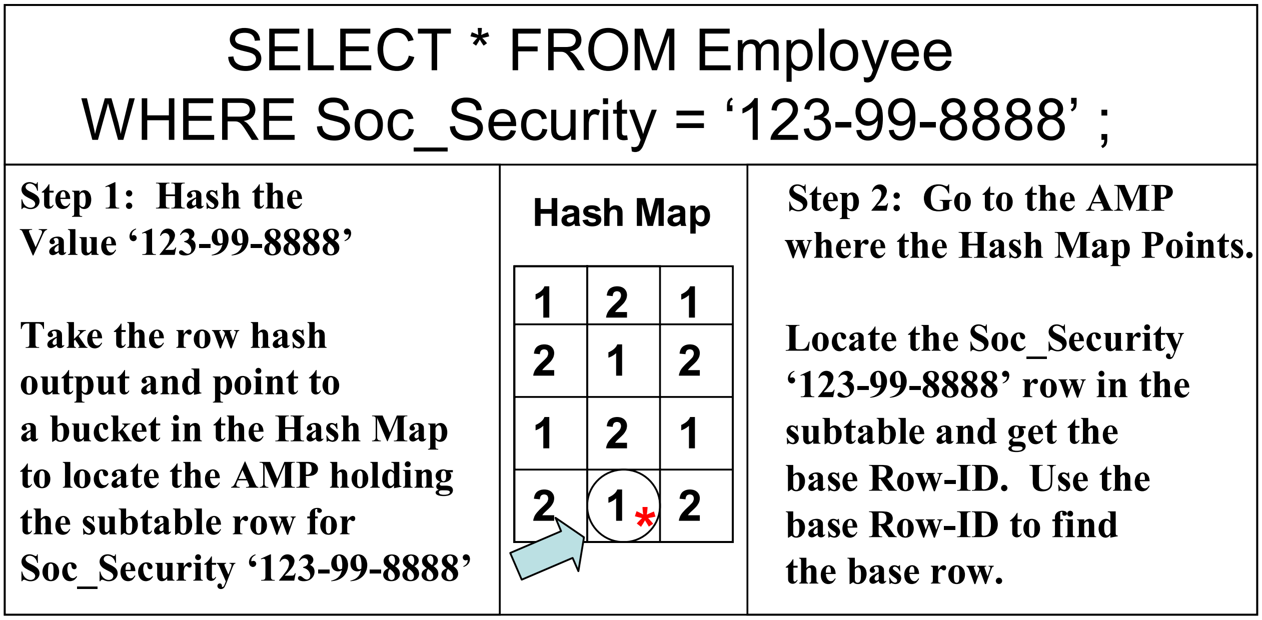
- When A **USI is created** Teradata will immediately build a secondary index **subtable on each AMP**.

- Each AMP will then **hash the secondary index value** for each of their rows in the base table. In our example, each AMP hashes the Soc_Security column for all employee rows they hold.

- The output of the Soc_Security hash will utilize the hash map to point to a specific AMP and that AMP will hold the secondary index subtable row for the secondary index value.

How Teradata retrieves an USI query

When an USI is used in the WHERE clause of an SQL statement, the PE Optimizer recognizes the Unique Secondary Index. It will perform a **two-AMP operation** to find the base row. Teradata knows it is looking for only one row and it can find it easily. It will **hash** the secondary index value and the hash map will point to the AMP where the row resides in the subtable. The **subtable row** will hold the **base table Row-ID** and Teradata will then find the base row immediately.



SELECT * FROM Employee
WHERE Soc_Security = ‘123-99-8888’ ;

Step 1: Hash the Value ‘123-99-8888’

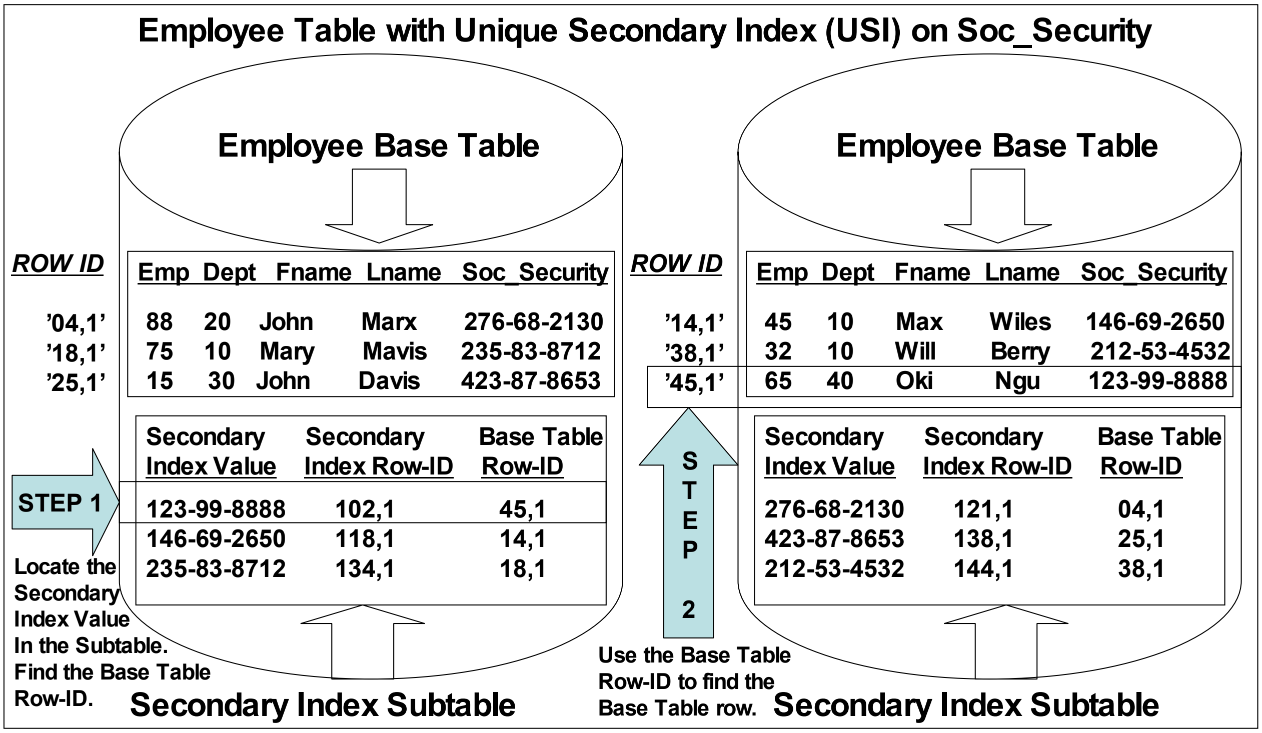
Take the row hash output and point to a bucket in the Hash Map to locate the AMP holding the subtable row for Soc_Security ‘123-99-8888’

Hash Map

1	2	1
2	1	2
1	2	1
2	1*	2

Step 2: Go to the AMP where the Hash Map Points.

Locate the Soc_Security ‘123-99-8888’ row in the subtable and get the base Row-ID. Use the base Row-ID to find the base row.



Employee Table with Unique Secondary Index (USI) on Soc_Security

Employee Base Table

ROW ID	Emp	Dept	Fname	Lname	Soc_Security
’04,1’	88	20	John	Marx	276-68-2130
’18,1’	75	10	Mary	Mavis	235-83-8712
’25,1’	15	30	John	Davis	423-87-8653

Secondary Index Value	Secondary Index Row-ID	Base Table Row-ID
123-99-8888	102,1	45,1
146-69-2650	118,1	14,1
235-83-8712	134,1	18,1

Secondary Index Subtable

STEP 1

Locate the Secondary Index Value In the Subtable. Find the Base Table Row-ID.

Employee Base Table

ROW ID	Emp	Dept	Fname	Lname	Soc_Security
’14,1’	45	10	Max	Wiles	146-69-2650
’38,1’	32	10	Will	Berry	212-53-4532
’45,1’	65	40	Oki	Ngu	123-99-8888

Secondary Index Value	Secondary Index Row-ID	Base Table Row-ID
276-68-2130	121,1	04,1
423-87-8653	138,1	25,1
212-53-4532	144,1	38,1

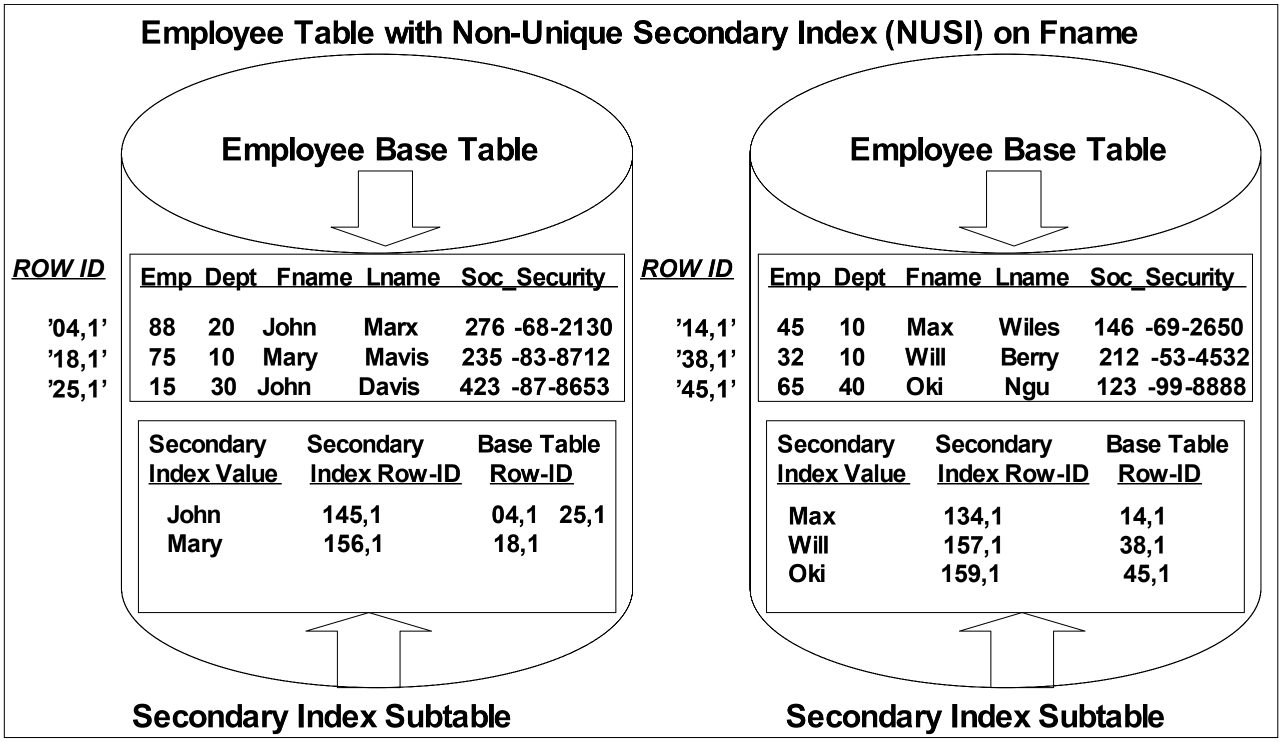
Secondary Index Subtable

STEP 2

Use the Base Table Row-ID to find the Base Table row.

NUSI Subtable Example

When a Non-Unique Secondary Index (NUSI) is designated on a table, each AMP will build a subtable. The NUSI subtable is said to be **AMP local** because each AMP will create it's secondary index subtable to **point to its own base rows**. In other words, the base **table row** and the **index subtable row** are on the **same AMP** and this why it is called **AMP Local**. Think of a NUSI as a **physical group of data** that is **related**. When a user inputs SQL that utilizes a NUSI in the WHERE clause, then Teradata will have each AMP check its subtable to see if it has any qualifying rows. Only the AMPs that contain the values needed will be involved in the actual retrieve.



Employee Table with Non-Unique Secondary Index (NUSI) on Fname

ROW ID	Emp	Dept	Fname	Lname	Soc_Security
'04,1'	88	20	John	Marx	276 -68-2130
'18,1'	75	10	Mary	Mavis	235 -83-8712
'25,1'	15	30	John	Davis	423 -87-8653

Secondary Index Value	Secondary Index Row-ID	Base Table Row-ID
John	145,1	04,1 25,1
Mary	156,1	18,1

ROW ID	Emp	Dept	Fname	Lname	Soc_Security
'14,1'	45	10	Max	Wiles	146 -69-2650
'38,1'	32	10	Will	Berry	212 -53-4532
'45,1'	65	40	Oki	Ngu	123 -99-8888

Secondary Index Value	Secondary Index Row-ID	Base Table Row-ID
Max	134,1	14,1
Will	157,1	38,1
Oki	159,1	45,1

- When A NUSI is created Teradata will immediately build a secondary index subtable on each AMP.

- Each AMP will hold the secondary index values for their rows in the base table only. In our example, each AMP holds the Fname column for all employee rows in the base table on their AMP (AMP local).

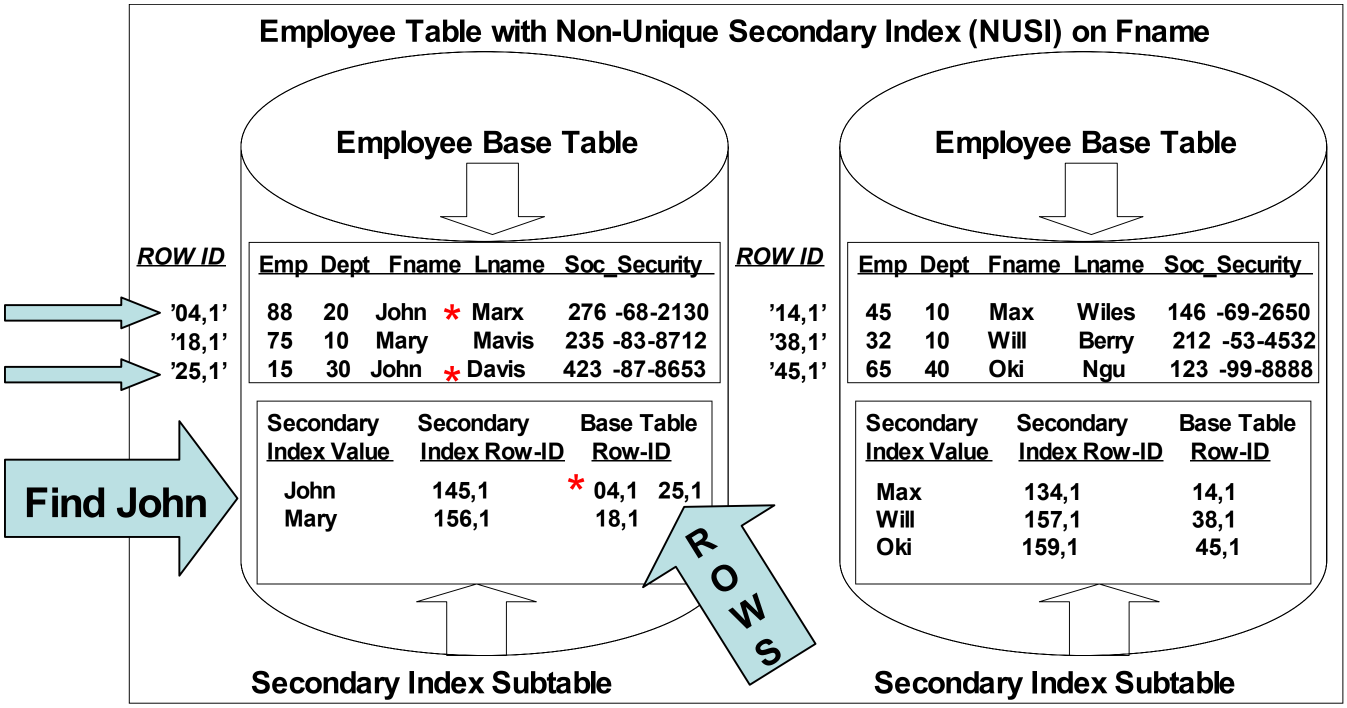
- Each AMP Local Fname will have the Base Table Row-ID (pointer) so the AMP can retrieve it quickly if needed. If an AMP contains duplicate first names, only one subtable row for that name is built with multiple Base Row-IDs.

How Teradata retrieves a NUSI query

When an NUSI is used in the WHERE clause of an SQL statement, the PE Optimizer recognizes the Non-Unique Secondary Index. It will perform an all AMP operation to look into the subtable for the requested value. If it contains the value it will continue participation. If it does not contain the requested value it will no longer participate. A NUSI query is an **All AMP operation**, but **not a Full Table Scan (FTS).**

```
SELECT * FROM Employee
WHERE Fname = 'John' ;
```

Step 1		Step 2
Step 1: Hash the Value 'John' for speed. **Take the row hash for John and have each AMP check its subtable to see if it has a John.**		**Step 2: All AMPs who contain a 'John' will retrieve their rows.** **Any AMP that does not contain the name John will no longer participate in the query.**



Employee Table with Non-Unique Secondary Index (NUSI) on Fname

ROW ID	Emp	Dept	Fname	Lname	Soc_Security
'04,1'	88	20	John *	Marx	276 -68-2130
'18,1'	75	10	Mary	Mavis	235 -83-8712
'25,1'	15	30	John *	Davis	423 -87-8653

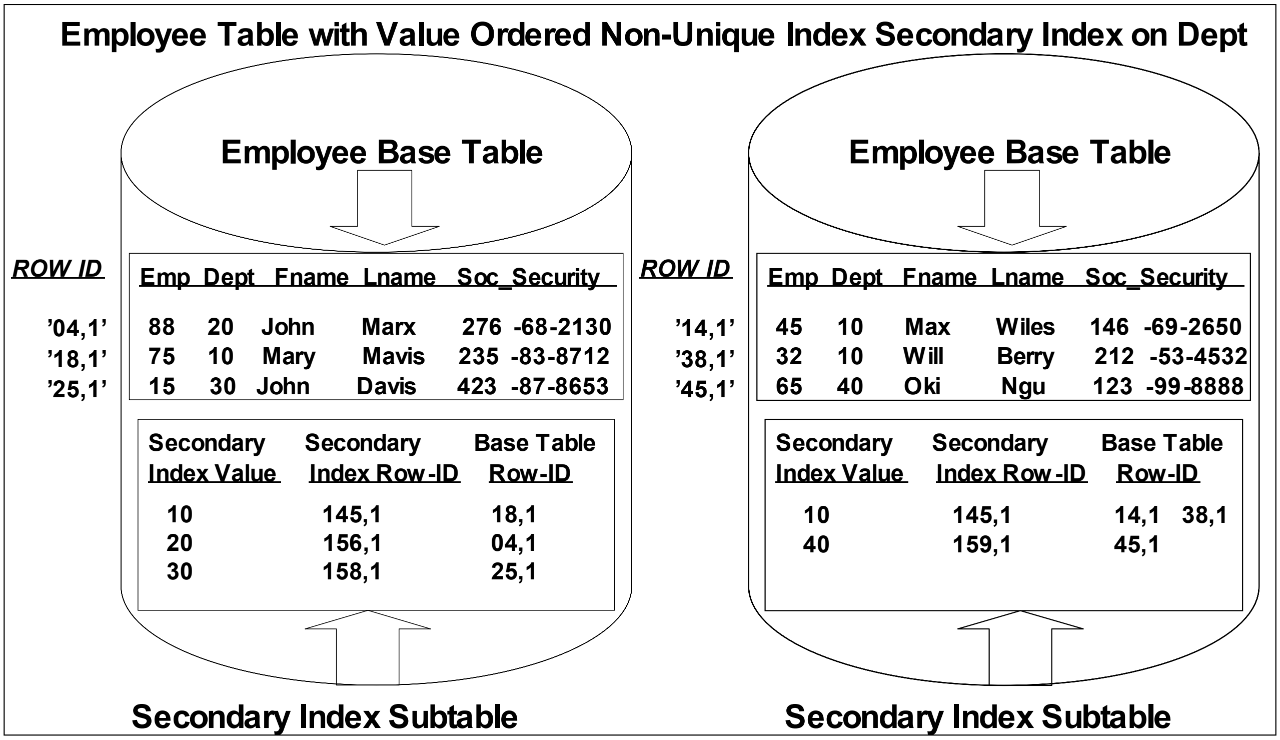
Secondary Index Value	Secondary Index Row-ID	Base Table Row-ID
John	145,1	* 04,1 25,1
Mary	156,1	18,1

ROW ID	Emp	Dept	Fname	Lname	Soc_Security
'14,1'	45	10	Max	Wiles	146 -69-2650
'38,1'	32	10	Will	Berry	212 -53-4532
'45,1'	65	40	Oki	Ngu	123 -99-8888

Secondary Index Value	Secondary Index Row-ID	Base Table Row-ID
Max	134,1	14,1
Will	157,1	38,1
Oki	159,1	45,1

Value-Ordered NUSI

When a Value Ordered Non-Unique Secondary Index (Value Ordered NUSI) is designated on a table, each AMP will build a subtable. The NUSI subtable is said to be AMP local because each AMP will create its secondary index subtable to point to its own base rows. In other words, every row in an AMPs NUSI subtable will reflect and point to the base rows it owns. It is called a Value Ordered NUSI because instead of the subtable being sorted by Secondary Index Value HASH it is sorted numerically.



Employee Table with Value Ordered Non-Unique Index Secondary Index on Dept

ROW ID	Emp	Dept	Fname	Lname	Soc_Security
’04,1’	88	20	John	Marx	276 -68-2130
’18,1’	75	10	Mary	Mavis	235 -83-8712
’25,1’	15	30	John	Davis	423 -87-8653

Secondary Index Value	Secondary Index Row-ID	Base Table Row-ID
10	145,1	18,1
20	156,1	04,1
30	158,1	25,1

ROW ID	Emp	Dept	Fname	Lname	Soc_Security
’14,1’	45	10	Max	Wiles	146 -69-2650
’38,1’	32	10	Will	Berry	212 -53-4532
’45,1’	65	40	Oki	Ngu	123 -99-8888

Secondary Index Value	Secondary Index Row-ID	Base Table Row-ID
10	145,1	14,1 38,1
40	159,1	45,1

- When A Value Ordered NUSI is created Teradata will immediately build a secondary index subtable on each AMP and **sort it in order.**

- Each AMP will hold the secondary index values for their rows in the base table only. In our example, each AMP holds the Dept column for all employee rows in the base table on their AMP (AMP local).

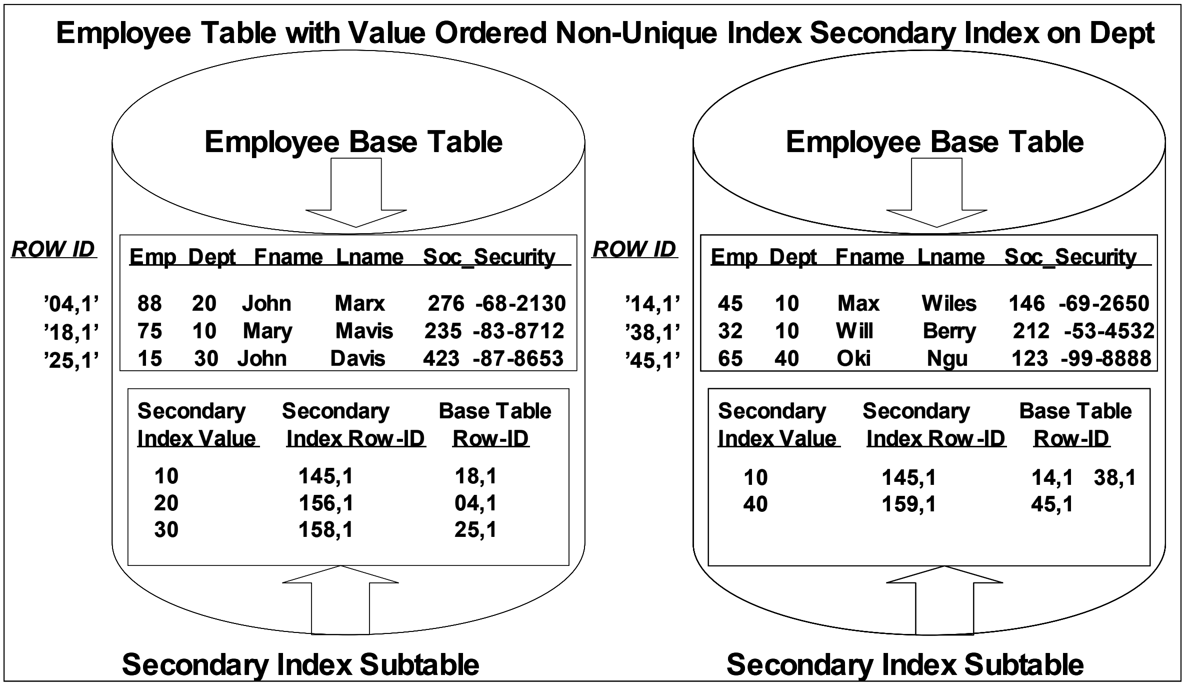
- Each AMP Local Dept will have the Base Table Row-ID (pointer) so the AMP can retrieve it quickly if needed. This is excellent for **Range queries** because the **subtable is sorted numerically by Dept**.

How Teradata retrieves a Value-Ordered NUSI query

When a Value Ordered NUSI is used in the WHERE clause of an SQL statement, the PE Optimizer recognizes the Value Ordered Non-Unique Secondary Index. It will perform an all AMP operation to look into the AMP-Local subtable for the requested value. It is excellent at checking ranges because all subtable rows are in order. If an AMP contains the value or values requested it will continue participation. If it does not contain the requested value or values it will no longer participate. A **Value-Ordered NUSI** query is an **All AMP operation**, but very **seldom a Full Table Scan (FTS)**. A Value Ordered NUSI must be non-unique and it must be a numeric data type. A DATE column type is considered numeric and there for may be a Value-Ordered NUSI.

```
SELECT * FROM Employee
WHERE Dept  BETWEEN 10 AND 20
```

Step 1: Check the subtable for Dept values ranging from 10 to 20 **If no rows are found then The AMP should no longer Participate in the query.**		**Step 2: If the AMP has qualifying rows then retrieve the rows in the range.**



Employee Table with Value Ordered Non-Unique Index Secondary Index on Dept

ROW ID	Emp	Dept	Fname	Lname	Soc_Security
’04,1’	88	20	John	Marx	276 -68-2130
’18,1’	75	10	Mary	Mavis	235 -83-8712
’25,1’	15	30	John	Davis	423 -87-8653

Secondary Index Value	Secondary Index Row-ID	Base Table Row-ID
10	145,1	18,1
20	156,1	04,1
30	158,1	25,1

ROW ID	Emp	Dept	Fname	Lname	Soc_Security
’14,1’	45	10	Max	Wiles	146 -69-2650
’38,1’	32	10	Will	Berry	212 -53-4532
’45,1’	65	40	Oki	Ngu	123 -99-8888

Secondary Index Value	Secondary Index Row-ID	Base Table Row-ID
10	145,1	14,1 38,1
40	159,1	45,1

NUSI Bitmapping

"It is only possible to live happily ever after on a day-to-day basis."

– Margaret Bonnano

You can allow your users to live happily ever after by collecting statistics on all Non-Unique Secondary Indexes. This provides the Parsing Engine (Optimizer) the ability to do NUSI bitmapping. NUSI Bitmapping does exist although some believe it is nothing more then a fairy tale.

One of the great things about NUSIs is that when multiple NUSI values are compared with the AND operator, the PE may choose a plan to create a bit-map. NUSI bitmapping will look for common Row IDs between multiple NUSI values. This is done internally using the "Intersect" operator. Why this is so fast is because the AMPs can simultaneously run the multiple queries in parallel and intersect the results.

Here is an SQL example that might use a bit map:

```
Command:     SELECT * FROM Employee
             WHERE Dept = 10
             AND Salary > 55000
             AND Hire_Date > '2001/01/01';
```

If all three columns of Dept, Salary, and Hire_Date are NUSIs, and statistics were collected on all three columns, then the PE may choose to do a bitmap.

The only way to determine if bitmapping is taking place is to use the EXPLAIN statement. The benefits of bitmapping are speed. Bitmapping truly reflects the brilliance behind the Teradata parallel architecture.

A weakly selective Non-Unique Index means that the index is almost useless. It would be like asking a live audience for all participants who breathe. **NUSI bitmapping** can take two or more **weakly selective NUSI columns** and **AND them** together to form a **strongly selective** combination.

Prototyping indexes with EXPLAIN

If you place the word EXPLAIN in front of your SQL, you can see the PE's plan in English. This is a way to prototype an index. For example, your application needs a boost of speed, so you decide to create some secondary indexes. You should run the EXPLAIN command before creating the indexes. Then create the indexes and run collect statistics. Then it is time to run the EXPLAIN again. If the plan utilizes the indexes and speeds up the query time, then you might want to consider keeping the index. If the PE doesn't utilize the secondary index, then drop the index.

The EXPAIN is an excellent facility for the following activities:

- Determining Access Paths
- Validating the use of indexes.
- To determine locking profiles.
- Showing Triggers and Join Index access.
- Estimate the query runtime.

```
EXPLAIN SELECT LNAME FROM Employee
WHERE SocSecNo = '276-68-7654';

Explanation
--------------------------------------------------
1) First, we do a Two-AMP RETRIEVE step from
Training.Employee by way of unique index # 20
"Training.Employee.SocSecNo = 276-68-7654" with no
residual conditions. The estimated time for this step is
0.09 seconds.

-> The row is sent directly back to the user as the
result of statement 1. The total estimated time is
0.09 seconds.
```

The above example shows that the EXPLAIN will use the secondary index for this query. Place the word EXPLAIN in front of your query and the query will not actually run, but the PE's plan will be delivered.

Chart for Primary and Secondary Access

The chart below shows that Primary Index access is a one-AMP operation. For Unique Secondary Index (USI) access, it is a two-AMP operation. For Non-Unique Secondary Index (NUSI) access, it is an all-AMP operation, but not a Full Table Scan (FTS). Keep this chart near and dear to your heart.

Primary Index	**Number of AMPs**	**Rows Returned**
UPI	1	0-1
NUPI	1	0-Many
USI	2	0-1
NUSI	All	0-Many

Secondary Index Summary

- You can have up to 32 secondary indexes for a table.
- Secondary Indexes provide an alternate path to the data.
- The two types of secondary indexes are USI and NUSI.
- Every secondary index defined causes each AMP to create a subtable.
- USI subtables are hash distributed.
- NUSI subtables are AMP local.
- USI queries are Two-AMP operations.
- NUSI queries are All-AMP operations, but not Full Table Scans.
- Value-Ordered NUSIs can be any non-unique index of numeric type that is 4 bytes or less.
- Always Collect Statistics on all NUSI indexes.
- The PE will decide if a NUSI is strongly selective and worth using over a Full Table Scan.
- Use the Explain function to see if a NUSI is being utilized or if bitmapping is taking place.

Chapter 9 — Join Strategies

"An invasion of Armies can be resisted, but not an idea whose time has come."

– Victor Hugo

A world-class data warehouse built for decision support must be able to efficiently join tables or it is just a giant data mart. Most large data warehouses that are not implemented using Teradata cannot perform joining of tables without crippling their data warehouse. Joins are an idea whose time has come!

So far you have learned the fundamentals of Teradata. You have also learned about how the Primary Index lays out the data with each table being sorted by Row ID. You have also learned all about Secondary Indexes. If you can take what you have learned so far and take the next step and understand joins you will be ready to model the database.

In this chapter, I want you to focus on the Merge Joins and the 4 strategies to make the Merge Joins happen. If you can see the data movement in these joins, you will understand how to impact the performance greatly.

Join Types vs. Join Strategies

Teradata's Optimizer has the ability to interpret a user's join types and then make decisions on what is the best path or join strategy to take in order complete the query. Basically, joins are combining rows from two or more tables. The key here is that these two tables have a common trait, albeit a same column that both tables have. As we go through this chapter, examples will point to these join types. However, our main focus in this chapter is to analyze how Teradata determines the join strategy based on the user's input. This chapter will also make recommendations on how the user can influence the join strategy. Keep in mind that the optimizer will have the last say on all joins completed in Teradata. Teradata allows up to 64 tables to be joined in a single query. As discussed, some of the common join types are:

- **Inner (may be a self join)**
- **Outer (Left, Right, Full)**
- **Exclusion**
- **Cross (may be a Cartesian)**

When the user inputs a join type, Teradata will then utilize clever join plans, or strategies, to perform the joins. These join strategies are:

- **Merge (Exclusion)**
- **Nested**
- **Row Hash**
- **Product (including Cartesian Product joins)**

A Merge Join is the most **popular join plan** and is usually based on equality (not always). A **Merge Join** requires that the data from the **tables be sorted** before attempting the **join operation**. A Merge Join **always requires rows** from the two tables to **be on the same AMP**. It will take the joining columns in the WHERE or ON clause and either **redistribute them by row hash** into a spool file or duplicate them across all AMPs. The bottom line is that when **neither join column** is a **Primary Index** and the join is based on equality or non-equality in a **merge join** or **exclusion merge join**, the qualified rows from both tables must be **rehashed** and then **redistributed** or **duplicated** and then always **sorted**.

A **Nested Join** is only used when the user specifies a **constant value** in the WHERE clause. This is because a Nested Join always uses at least one **Unique Index**.

A **Row Hash** join always copies the **smaller table into memory**.

A **product or Cartesian Product Join** will **duplicate** the **smaller table** on **every AMP**.

During this chapter, you will come to appreciate how brilliantly Teradata performs joins through **redistribution, duplication, sorting, and hashing**. Two other features that can potentially eliminate join steps and speed things up are **Soft Referential Integrity** or in the creation of a **multi-table join index**. These options will be discussed in later chapters.

As stated, our main goal in this chapter will be to focus on the four join strategies. However, before we go forward and discover these join strategies, let’s quickly review how a join works in Teradata.

A Join in Simple Terms

A join will take a row from one table and join it to another row in another table. Here is an example of a join. We will do this in our heads. Below are two tables. They are named Employee and Department. What is the Department Name that Vu Chang is in?

Employee Table

EMP	DEPT	LNAME	FNAME	SAL
Primary Key	Foreign Key *			
1	10	JONES	DAVE	45000.00
2	20	SMITH	MARY	50000.00
3	30	CHANG	VU	65000.00
4	10	WILSON	SUE	44000.00

Department Table

DEPT	DEPT_NAME
PK	
10	SALES
20	MARKETING
30	FINANCE
40	HUMAN RESOURCES

Vu Chang is in the Finance department. You did a join by taking a row from the Employee table and joining it with a row in the Department table. You were able to do this because the two tables had a relation in the column DEPT. You were able to find Vu Chang in the Employee table and see that he is in DEPT 30. Then you were able to scan the Department table until you found DEPT 30 to see that Mr. Chang was in FINANCE. Congratulations you have done a join.

The above example is called a Merge Join. **Merge joins** are always sorted by **Row Hash** or **Row Key** (for Partition Primary Index tables) and they can be based on a join constraint of either **equal** or NOT **equal**.

The key things to know about Teradata and Joins

Each AMP holds a portion of a table.

Teradata uses the Primary Index to distribute the rows among the AMPs.

Each AMP keeps their tables separated from other tables like someone might keep clothes in a dresser drawer.

Each AMP sorts their tables by Row ID.

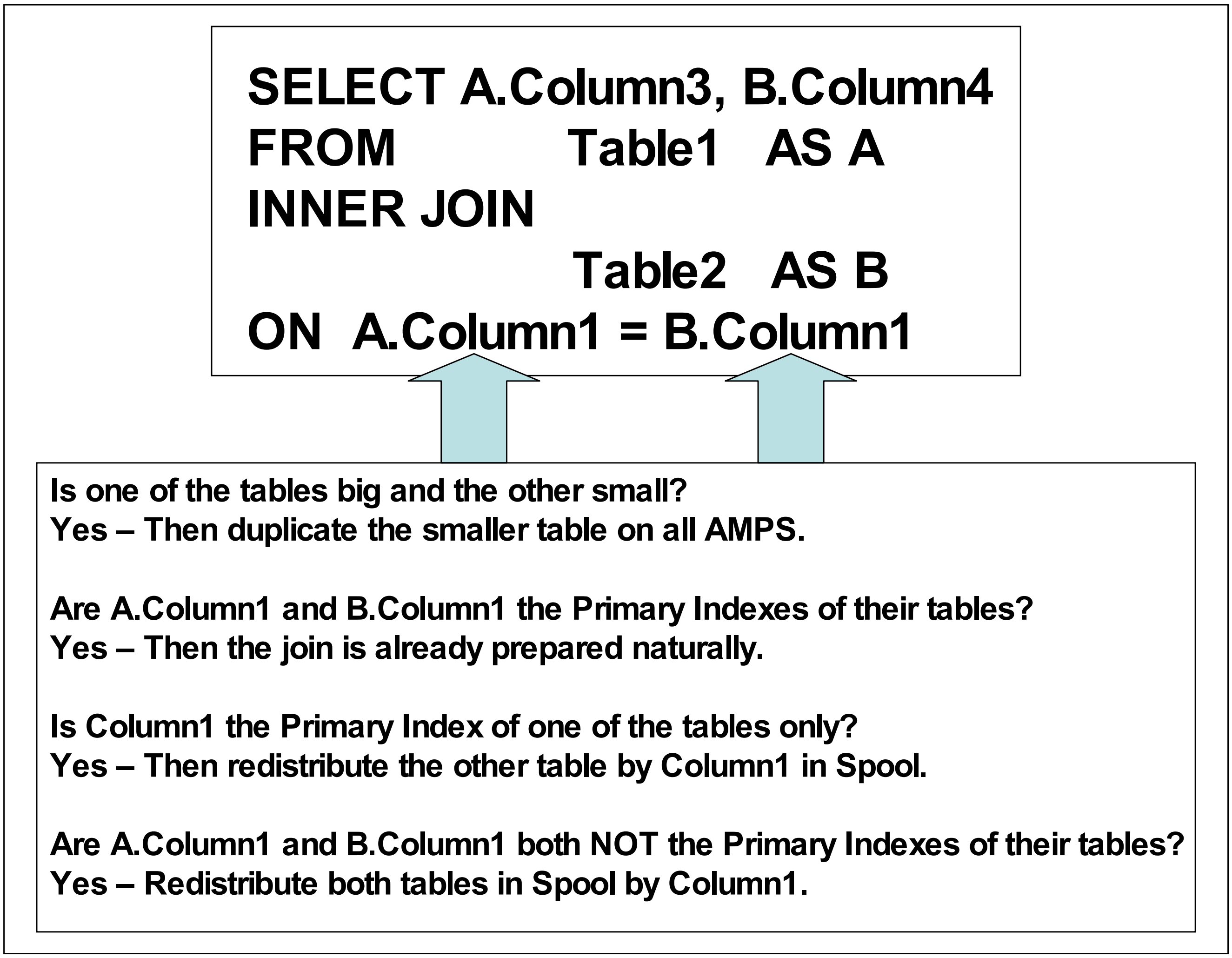
For a JOIN to take place the two rows being joined must find a way to get to the same AMP.

If the rows to be joined are not on the same AMP, Teradata will either redistribute the data or duplicate the data in spool to make that happen.

Merge Join Strategies

A **Join** that has uses the **Primary Index** of both tables in the **WHERE clause** never uses a **SPOOL** File. If two tables being joined have the **same Primary Index** value and they are **joined on that value** then Teradata performs a **Merge Join** with a **row hash match scan.** **A Merge Join** could also make use of a **Primary Index** on each of two tables **with matching Partitioned Primary Indexes.**

Let's say you have a table with a **multi-column composite primary key**. You can minimize join costs by **minimizing row redistribution**. To **minimize row redistribution**, the table should be defined with a **NUPI** on the columns most **frequently joined**.



```
SELECT A.Column3, B.Column4
FROM          Table1   AS A
INNER JOIN
              Table2   AS B
ON  A.Column1 = B.Column1
```

Is one of the tables big and the other small?
Yes – Then duplicate the smaller table on all AMPS.

Are A.Column1 and B.Column1 the Primary Indexes of their tables?
Yes – Then the join is already prepared naturally.

Is Column1 the Primary Index of one of the tables only?
Yes – Then redistribute the other table by Column1 in Spool.

Are A.Column1 and B.Column1 both NOT the Primary Indexes of their tables?
Yes – Redistribute both tables in Spool by Column1.

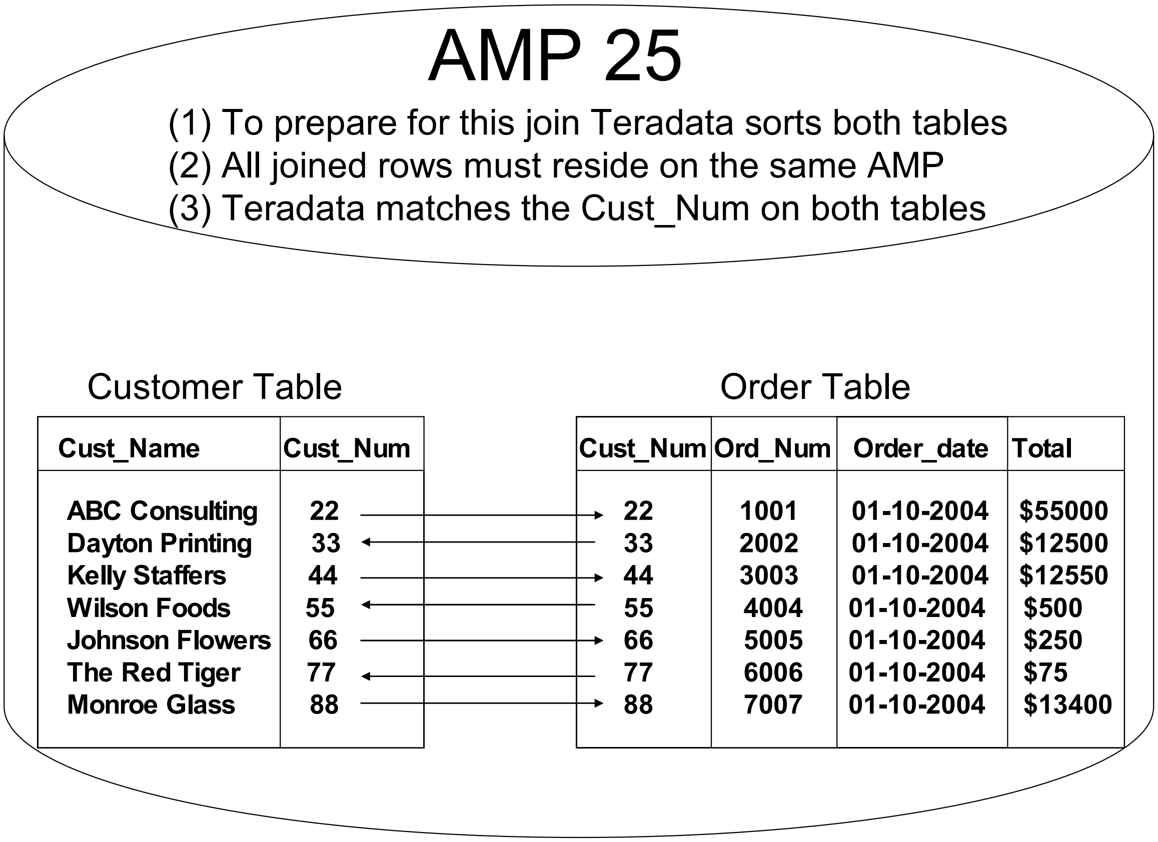
Joins need the joined rows to be on the same AMP

This is an extremely important chapter because Teradata is brilliant at joins. If you can understand the upcoming pages you will have gained great knowledge. Knowledge that really separates the good ones from the great ones.

When Teradata joins two tables it must place the joining rows on the same AMP. If the rows are not naturally on the same AMP then Teradata will perform two strategies to get them placed together. Teradata will redistribute one or both of the tables in spool or it will copy the smaller table to all of the AMPs.

In our picture below we are joining the Customer table with the Order table. AMP 25 is ready for the join because the rows it holds are sorted and the matching rows are on the same AMP. This was accomplished by only three possibilities:

- The two tables had their Primary Index as Cust_Num so the rows naturally resided together.
- Teradata redistributed one or both tables by rehashing them by Cust_Num in spool.
- Teradata placed the smaller table on all AMPs



AMP 25

(1) To prepare for this join Teradata sorts both tables
(2) All joined rows must reside on the same AMP
(3) Teradata matches the Cust_Num on both tables

Customer Table

Cust_Name	Cust_Num
ABC Consulting	22
Dayton Printing	33
Kelly Staffers	44
Wilson Foods	55
Johnson Flowers	66
The Red Tiger	77
Monroe Glass	88

Order Table

Cust_Num	Ord_Num	Order_date	Total
22	1001	01-10-2004	$55000
33	2002	01-10-2004	$12500
44	3003	01-10-2004	$12550
55	4004	01-10-2004	$500
66	5005	01-10-2004	$250
77	6006	01-10-2004	$75
88	7007	01-10-2004	$13400

Another Great Join Picture

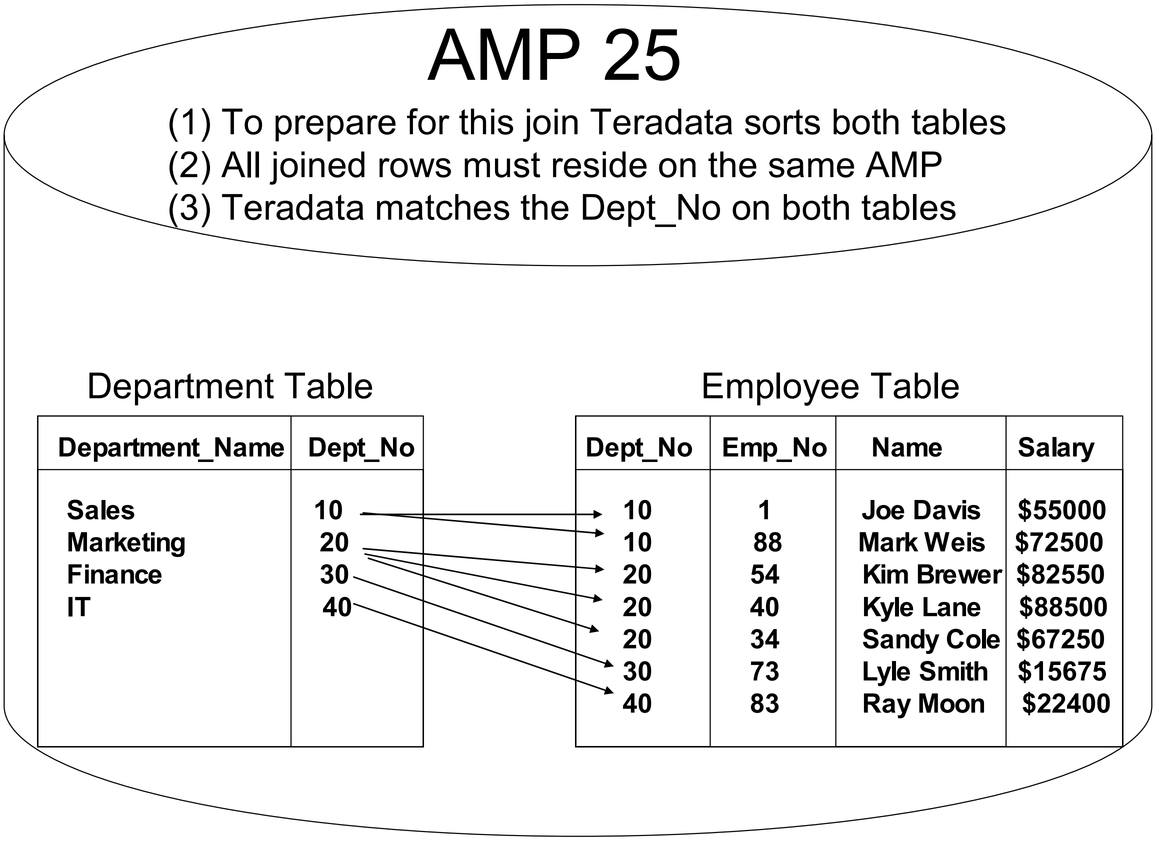
Our picture below shows AMP 25 again and we are joining the Department table with the Employee table. This is a one to many row join because one department holds many employees. Teradata was able to place the matching rows on the same AMP and did so by three different possibilities:

- The two tables had their Primary Index as Dept_no so the rows naturally resided together.

- Teradata redistributed one or both tables by rehashing them by Dept_no in spool.

- Teradata placed the smaller Department table on all AMPs.

```
SELECT E.Name, E.Salary, D.Department_Name
FROM    Employee as E
INNER JOIN
             Department as D
ON       E.Dept_No = D.Dept_No
```



(1) To prepare for this join Teradata sorts both tables
(2) All joined rows must reside on the same AMP
(3) Teradata matches the Dept_No on both tables

Department Table

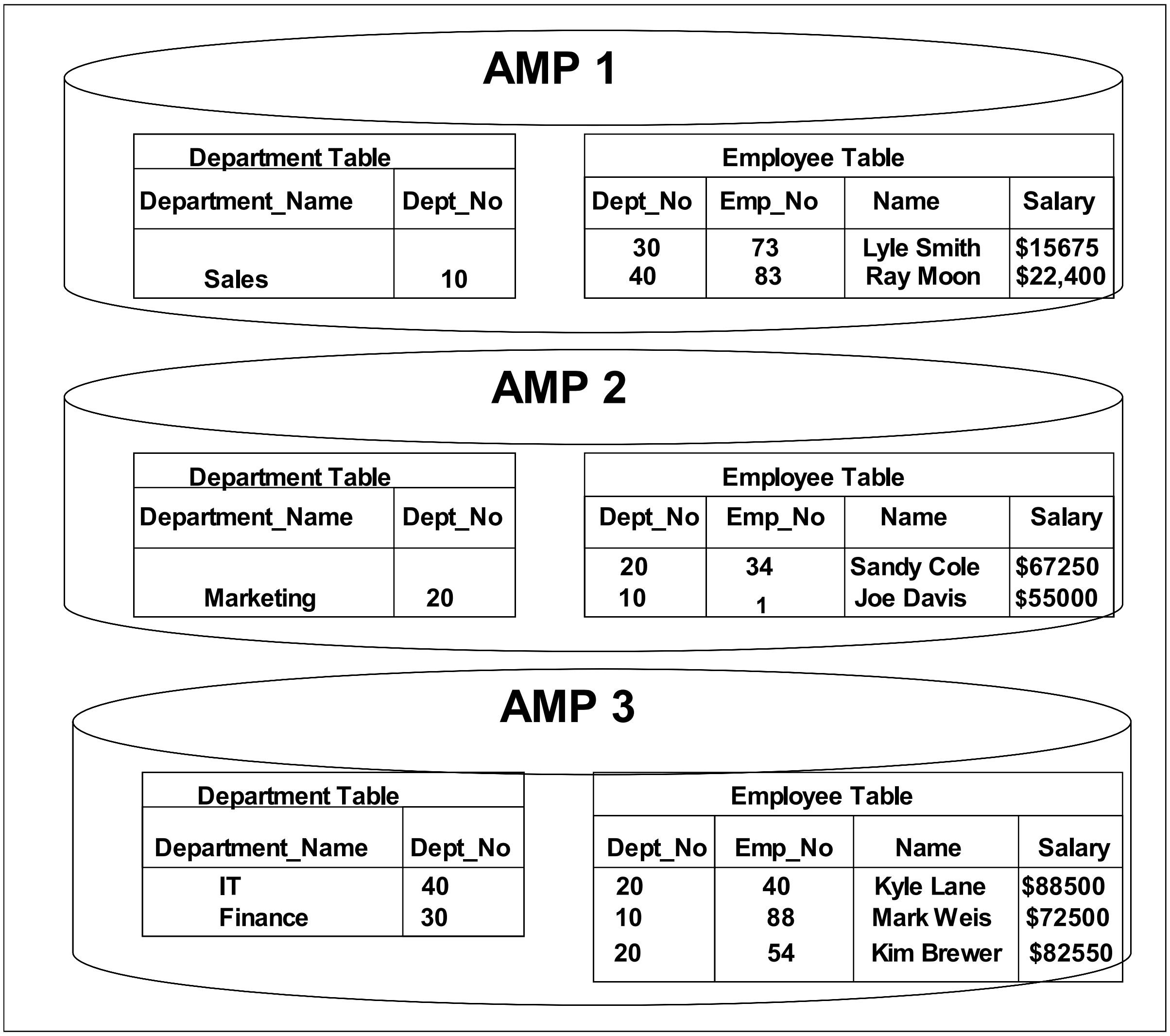
Department_Name	Dept_No
Sales	10
Marketing	20
Finance	30
IT	40

Employee Table

Dept_No	Emp_No	Name	Salary
10	1	Joe Davis	$55000
10	88	Mark Weis	$72500
20	54	Kim Brewer	$82550
20	40	Kyle Lane	$88500
20	34	Sandy Cole	$67250
30	73	Lyle Smith	$15675
40	83	Ray Moon	$22400

Joining Tables with matching rows on different AMPs

Our picture below shows a 3 AMP system. Each AMP holds rows from both the Department Table and the Employee Table. We will join the two tables based on the equality WHERE E.Dept_No = D.Dept_No. Since the join syntax will join on the Dept_No column from both tables, Teradata will realize that the Dept_No column is the Primary Index of the Department Table, but not the Employee Table. Teradata will need to redistribute the Employee Table in spool and rehash it by Dept_No in order to get the rows on the same AMPs (We will show that picture on the next page).



AMP 1

Department Table

Department_Name	Dept_No
Sales	10

Employee Table

Dept_No	Emp_No	Name	Salary
30	73	Lyle Smith	$15675
40	83	Ray Moon	$22,400

AMP 2

Department Table

Department_Name	Dept_No
Marketing	20

Employee Table

Dept_No	Emp_No	Name	Salary
20	34	Sandy Cole	$67250
10	1	Joe Davis	$55000

AMP 3

Department Table

Department_Name	Dept_No
IT	40
Finance	30

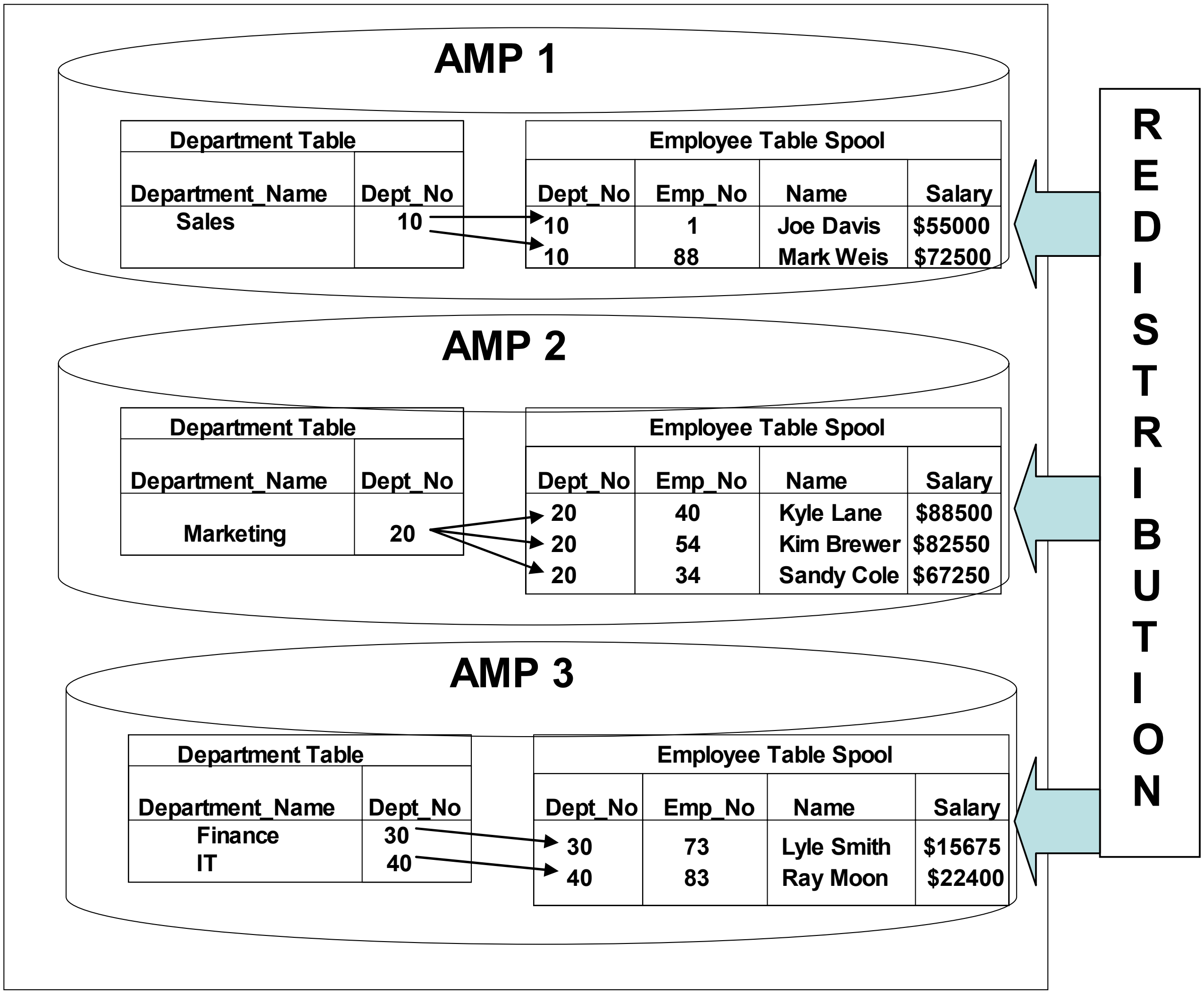
Employee Table

Dept_No	Emp_No	Name	Salary
20	40	Kyle Lane	$88500
10	88	Mark Weis	$72500
20	54	Kim Brewer	$82550

Redistributing a Table for Join Purposes

Our picture below shows that Teradata has redistributed the Employee table into Spool and rehashed by Dept_No. Now the joining rows are on the AMP with their joining row partners.

```
SELECT E.Name, E.Salary, D.Department_Name
FROM    Employee as E
INNER JOIN
            Department as D
ON      E.Dept_No = D.Dept_No
```



AMP 1

Department Table

Department_Name	Dept_No
Sales	10

Employee Table Spool

Dept_No	Emp_No	Name	Salary
10	1	Joe Davis	$55000
10	88	Mark Weis	$72500

AMP 2

Department Table

Department_Name	Dept_No
Marketing	20

Employee Table Spool

Dept_No	Emp_No	Name	Salary
20	40	Kyle Lane	$88500
20	54	Kim Brewer	$82550
20	34	Sandy Cole	$67250

AMP 3

Department Table

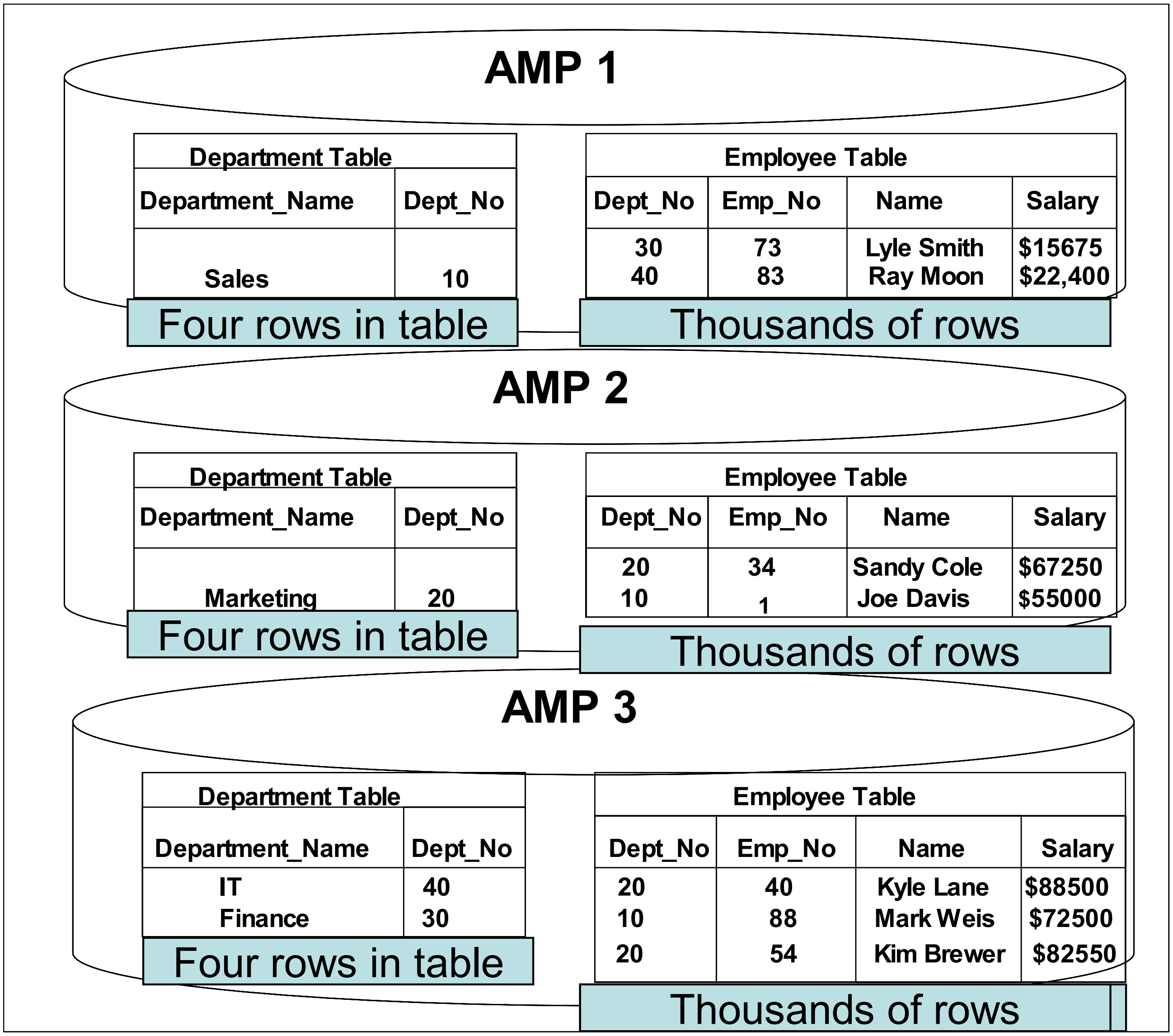
Department_Name	Dept_No
Finance	30
IT	40

Employee Table Spool

Dept_No	Emp_No	Name	Salary
30	73	Lyle Smith	$15675
40	83	Ray Moon	$22400

Big Table Small Table Join Strategy

Our picture below shows a 3 AMP system. Each AMP holds rows from both the Department Table and the Employee Table. We will join the two tables based on equality WHERE E.Dept_No = D.Dept_No. Let's pretend that the Department Table is small and that the Employee Table is extremely large. In the case of a big table small table join, Teradata will duplicate the smaller table on all AMPs. (We will show that Teradata places the Department table on each AMP in our next picture).



AMP 1

Department Table

Department_Name	Dept_No
Sales	10

Four rows in table

Employee Table

Dept_No	Emp_No	Name	Salary
30	73	Lyle Smith	$15675
40	83	Ray Moon	$22,400

Thousands of rows

AMP 2

Department Table

Department_Name	Dept_No
Marketing	20

Four rows in table

Employee Table

Dept_No	Emp_No	Name	Salary
20	34	Sandy Cole	$67250
10	1	Joe Davis	$55000

Thousands of rows

AMP 3

Department Table

Department_Name	Dept_No
IT	40
Finance	30

Four rows in table

Employee Table

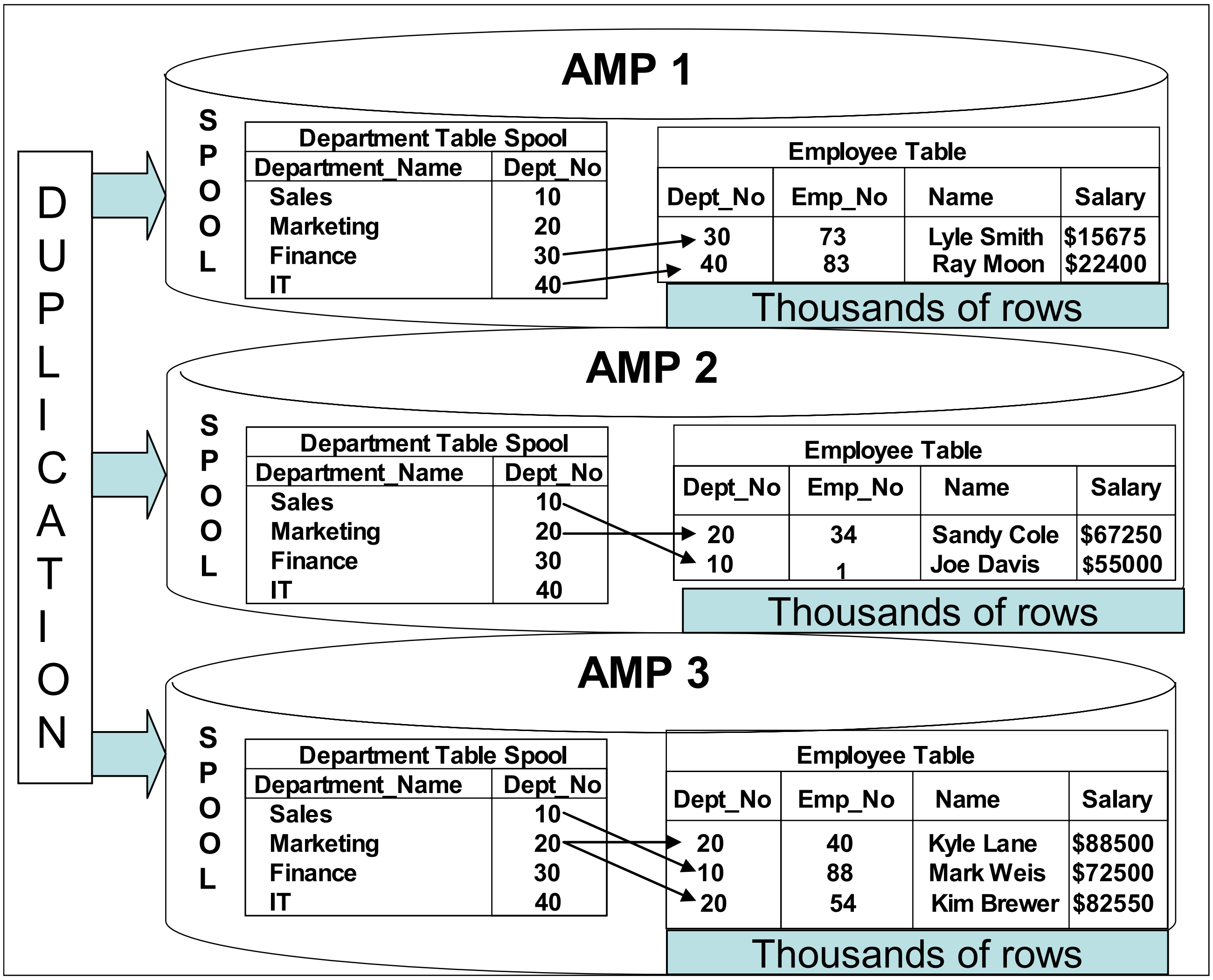
Dept_No	Emp_No	Name	Salary
20	40	Kyle Lane	$88500
10	88	Mark Weis	$72500
20	54	Kim Brewer	$82550

Thousands of rows

Big Table Small Table Duplication

Our picture below shows that Teradata will duplicate the Department table on all AMPs .

There are only **three join types** that will take the smaller table and **DUPLICATE ACROSS ALL AMPS** and they are a **merge** join, **hash** join, and it is always done on any **product join**. That is why **Merge, Hash,** and **Product joins** always require using **ALL AMPs**.



AMP 1

Department Table Spool	
Department_Name	Dept_No
Sales	10
Marketing	20
Finance	30
IT	40

Employee Table			
Dept_No	Emp_No	Name	Salary
30	73	Lyle Smith	$15675
40	83	Ray Moon	$22400

Thousands of rows

AMP 2

Department Table Spool	
Department_Name	Dept_No
Sales	10
Marketing	20
Finance	30
IT	40

Employee Table			
Dept_No	Emp_No	Name	Salary
20	34	Sandy Cole	$67250
10	1	Joe Davis	$55000

Thousands of rows

AMP 3

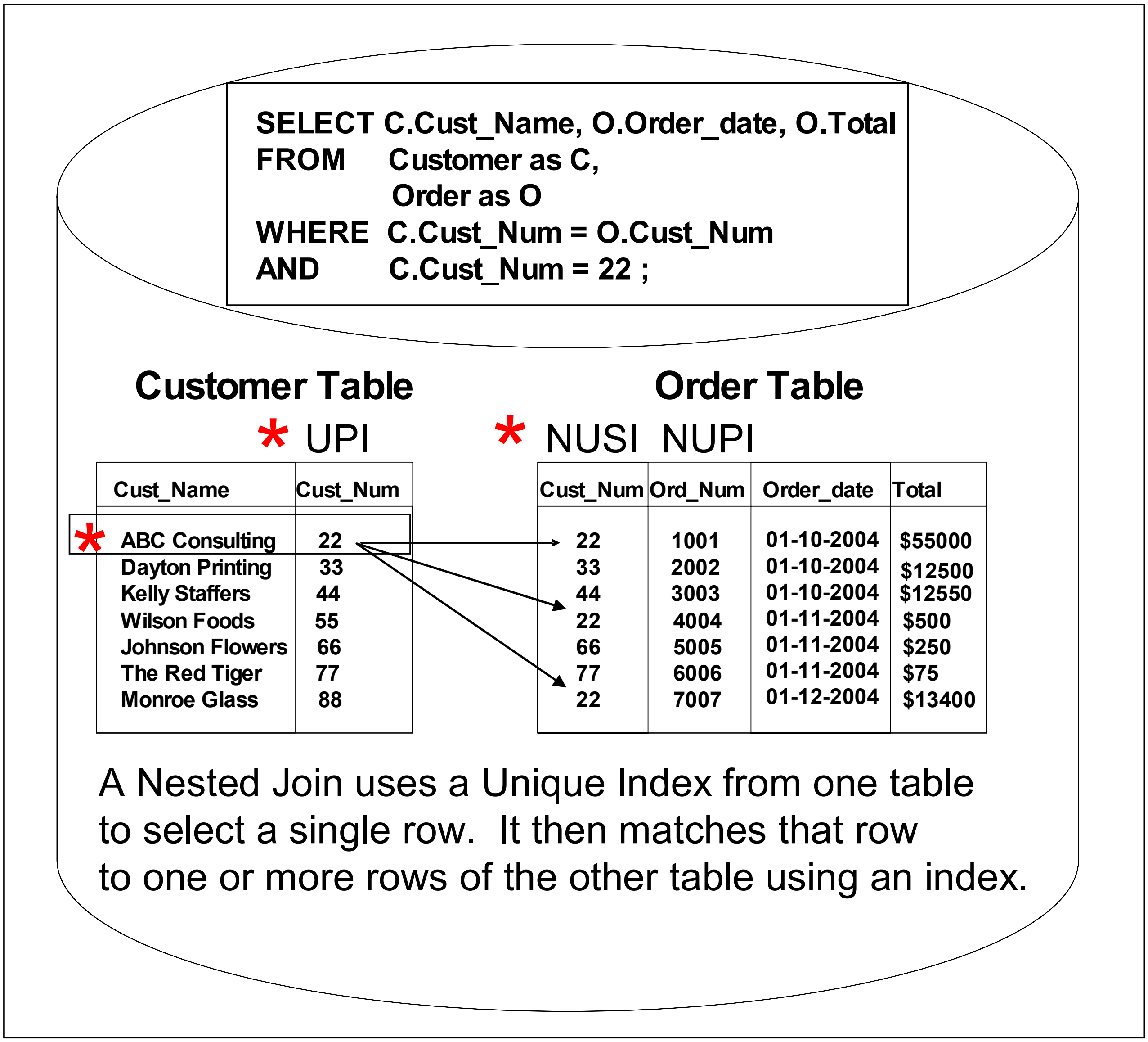
Department Table Spool	
Department_Name	Dept_No
Sales	10
Marketing	20
Finance	30
IT	40

Employee Table			
Dept_No	Emp_No	Name	Salary
20	40	Kyle Lane	$88500
10	88	Mark Weis	$72500
20	54	Kim Brewer	$82550

Thousands of rows

Nested Join

A nested join strategy is probably the most precise join available. The **Nested Join** is designed to utilize a **unique index** type (Either Unique Primary Index or Unique Secondary Index) from **one** of the tables in the join statement in order to retrieve a **single row**. It then matches that **row** to **one or more rows** on the other table being used in the join **using an index**.



```
SELECT C.Cust_Name, O.Order_date, O.Total
FROM    Customer as C,
        Order as O
WHERE  C.Cust_Num = O.Cust_Num
AND     C.Cust_Num = 22 ;
```

Customer Table

Cust_Name	Cust_Num
ABC Consulting	22
Dayton Printing	33
Kelly Staffers	44
Wilson Foods	55
Johnson Flowers	66
The Red Tiger	77
Monroe Glass	88

Order Table

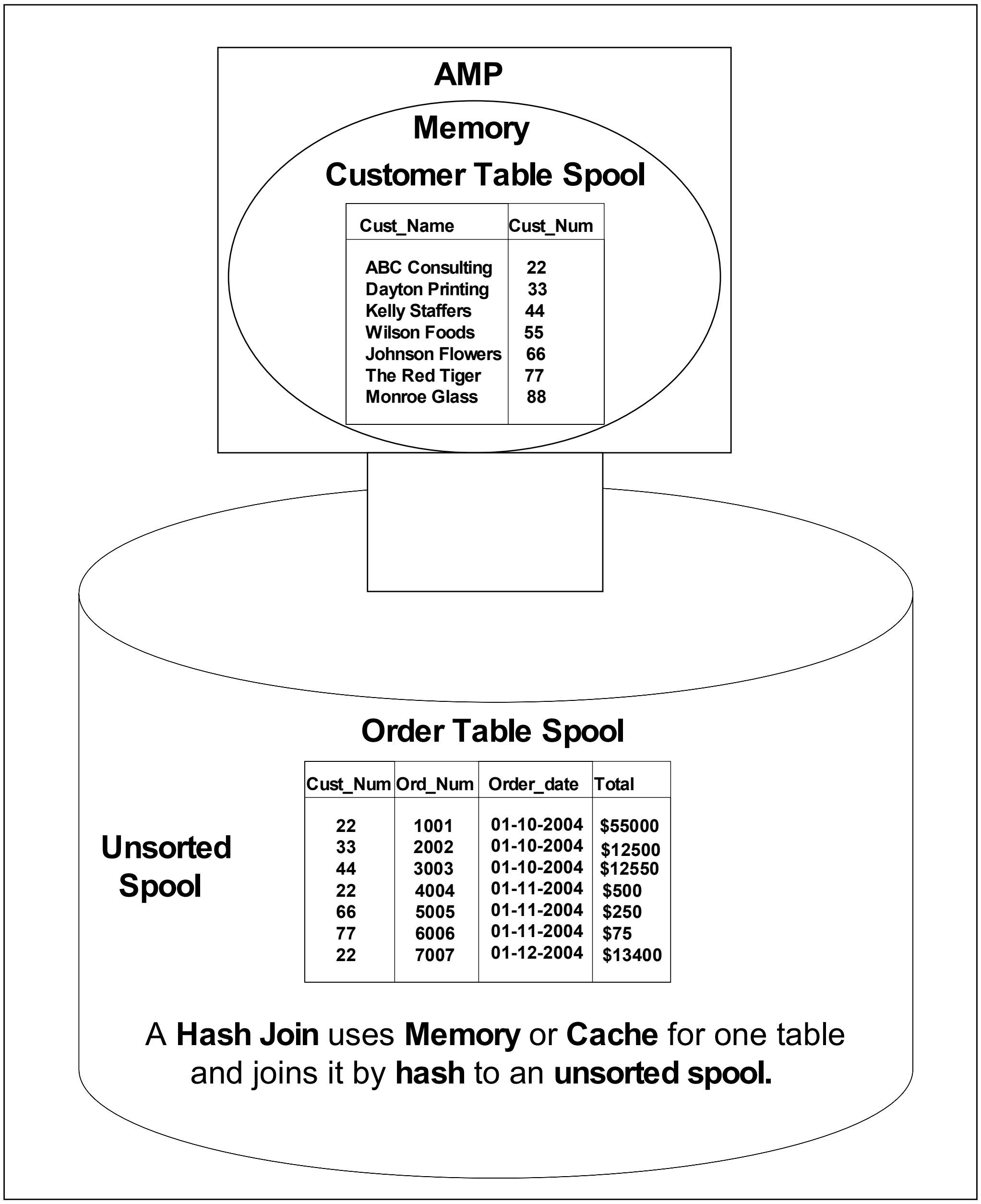
Cust_Num	Ord_Num	Order_date	Total
22	1001	01-10-2004	$55000
33	2002	01-10-2004	$12500
44	3003	01-10-2004	$12550
22	4004	01-11-2004	$500
66	5005	01-11-2004	$250
77	6006	01-11-2004	$75
22	7007	01-12-2004	$13400

A Nested Join uses a Unique Index from one table to select a single row. It then matches that row to one or more rows of the other table using an index.

Hash Join

The Hash Join is part of the Merge Join Family. A Hash Join can only take place if one or both of the tables on each AMP can fit completely inside the AMP's memory.

A **Hash Join** uses **Memory** or **Cache** for one table and joins it by **hash** to an **unsorted spool** and does so quite quickly. Hash Joins are generally **faster** than Merge Joins because a **Hash Join** does not need to **sort** the **larger table**.



Cust_Name	Cust_Num
ABC Consulting	22
Dayton Printing	33
Kelly Staffers	44
Wilson Foods	55
Johnson Flowers	66
The Red Tiger	77
Monroe Glass	88

Cust_Num	Ord_Num	Order_date	Total
22	1001	01-10-2004	$55000
33	2002	01-10-2004	$12500
44	3003	01-10-2004	$12550
22	4004	01-11-2004	$500
66	5005	01-11-2004	$250
77	6006	01-11-2004	$75
22	7007	01-12-2004	$13400

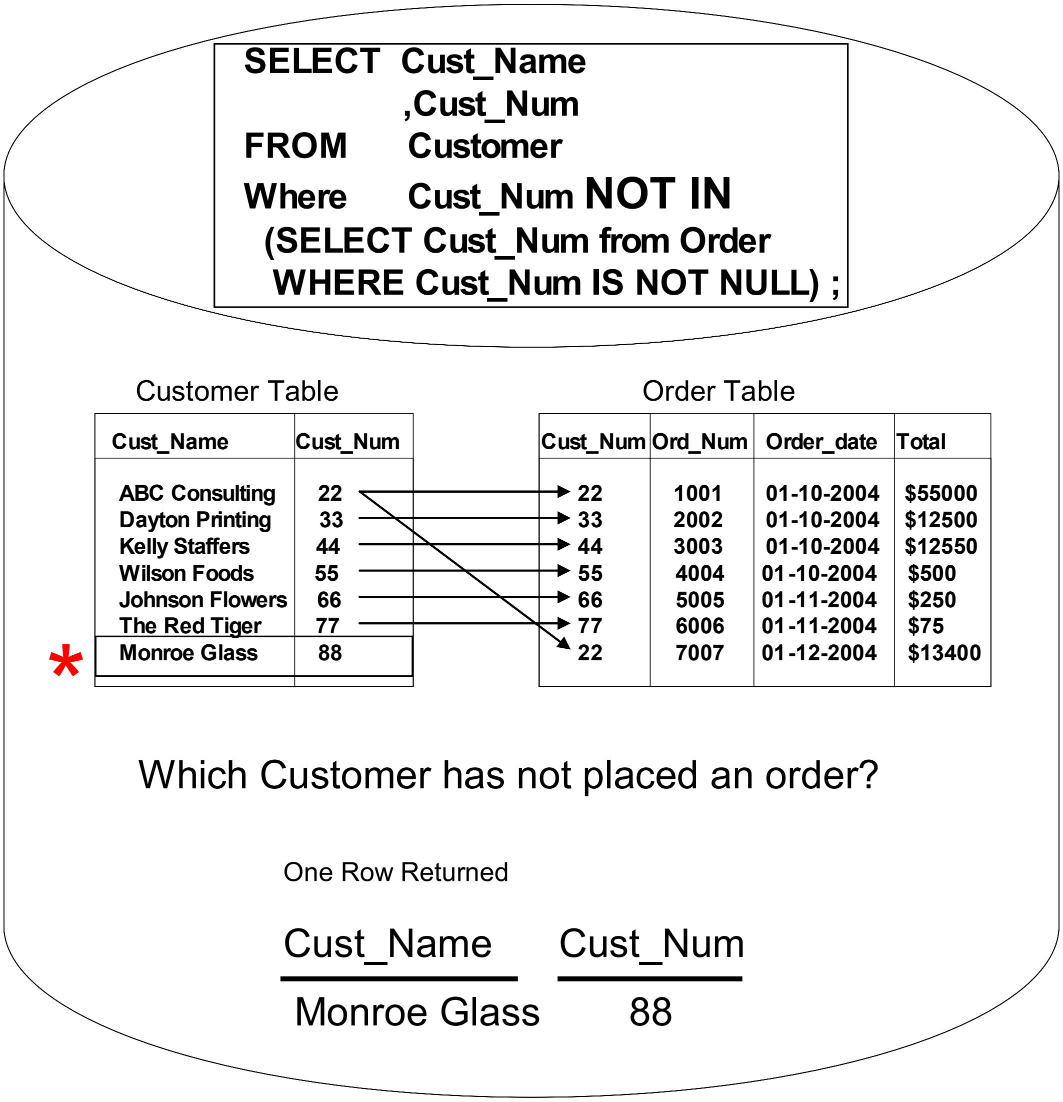
A **Hash Join** uses **Memory** or **Cache** for one table and joins it by **hash** to an **unsorted spool.**

Exclusion Join

Exclusion Joins are only used on queries with the NOT IN, EXCEPT, or MINUS operators and are based on set subtractions. Anytime a NULL is used in a NOT IN list then the results will be nothing. An additional WHERE with an IS NOT NULL is common practice to avoid no results returning.

Here is the way that exclusion joins work. Teradata is looking for rows from one table that are NOT IN another table. There are 3 rules that apply to qualifying a row.

1. Any matches disqualifies a row
2. Any unknown disqualifies a row
3. Rows that don't have matches qualify



```
SELECT  Cust_Name
        ,Cust_Num
FROM    Customer
Where   Cust_Num NOT IN
  (SELECT Cust_Num from Order
   WHERE Cust_Num IS NOT NULL) ;
```

Customer Table

Cust_Name	Cust_Num
ABC Consulting	22
Dayton Printing	33
Kelly Staffers	44
Wilson Foods	55
Johnson Flowers	66
The Red Tiger	77
* Monroe Glass	88

Order Table

Cust_Num	Ord_Num	Order_date	Total
22	1001	01-10-2004	$55000
33	2002	01-10-2004	$12500
44	3003	01-10-2004	$12550
55	4004	01-10-2004	$500
66	5005	01-11-2004	$250
77	6006	01-11-2004	$75
22	7007	01-12-2004	$13400

Which Customer has not placed an order?

One Row Returned

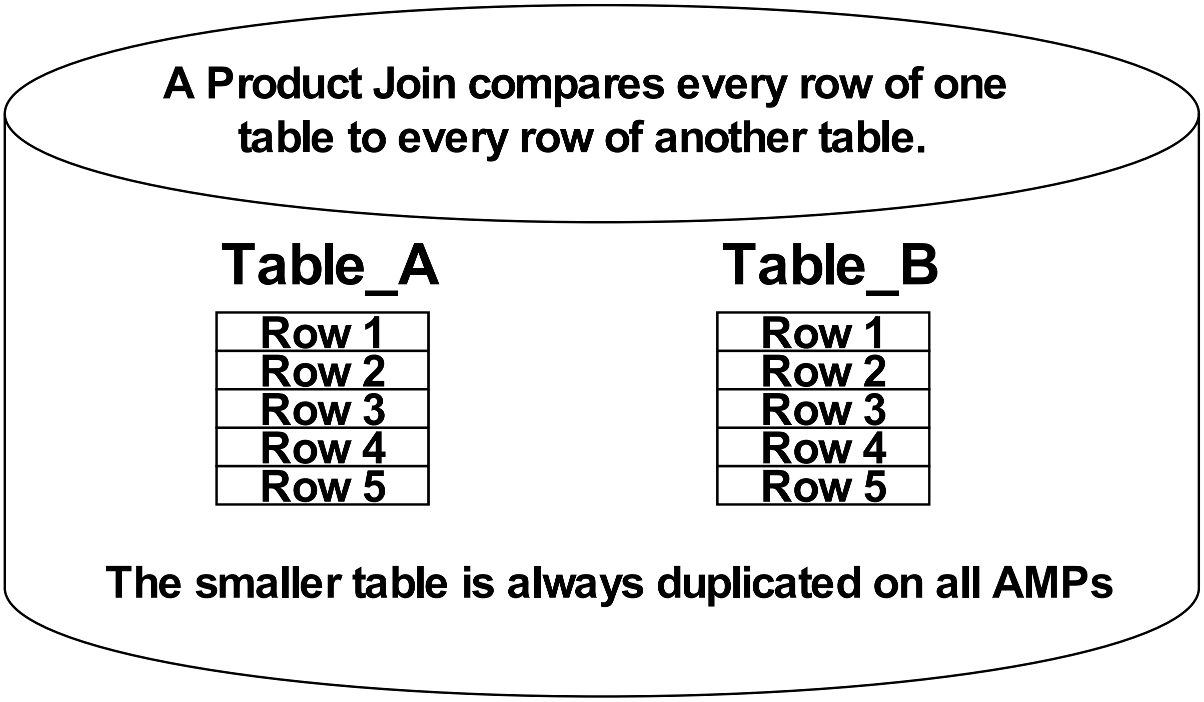
Cust_Name	Cust_Num
Monroe Glass	88

Product Joins

Product Joins compare every row of **one table** to **every row** of **another table**. They are called product joins because they are a product of the number of rows in table one multiplied by the number of rows in table two. For example, if one table has five rows and the other table has five rows, then the Product Join will compare 5 x 5 or 25 rows with a potential of 25 rows coming back.

To **avoid** a **product join**, check your syntax to ensure that the join is based on an **EQUALITY** condition. A **Product Join** always results when the join condition is based on **Inequality**. The reason the optimizer chooses **Product Joins** for join conditions other than equality is because **Hash Values** cannot be compared for **greater than** or **less then** comparisons.

A **Product Join**, **Merge Join**, and **Exclusion Merge** Join always requires **SPOOL** Files. **A Product Join** will usually be utilized when the **JOIN constraint** is **"OR"ed**.



A Product Join compares every row of one table to every row of another table.

Table_A	Table_B
Row 1	Row 1
Row 2	Row 2
Row 3	Row 3
Row 4	Row 4
Row 5	Row 5

The smaller table is always duplicated on all AMPs

Cartesian Product Join

"The superior man is modest in his speech, but exceeds in his actions."

– Confucius (551 BC – 479 BC)

A Cartesian Product join will always exceed in its actions because it **joins every row** from **one table** with **every row** in the **joining table**. It is usually a mistake and inferior coding.

Just as discussed with Product Joins above, a Cartesian Product Join is usually something you want to avoid. If we decided to run this query in Michigan this would be called the big "Mistake on the Lake". A Cartesian Product Join will join every row in one table to every row in another table. The only thing that decides the number of rows will be the total number of rows from both tables. If one table has 5 rows and another has 10 rows, then you will always get 50 rows returned. Imagine this situation – we have a table with 10 million rows and another with 25 million rows and a Cartesian product join is written (by accident) against these two tables. What will be the result? Well, based on the example above, you will get back about 250 Trillion Rows (250,000,000,000,000)! This is definitely NOT the correct answer this user would want. This is why spool space limitations are utilized.

```
SELECT        Emp
              ,E.Dept
              ,Name
              ,DeptName
FROM           Employee_Table    AS  E
              Department_Table AS D;
```

About 99% of the time, a Cartesian Product Join is a major problem. The recommendation is avoid these types of queries whenever possible. The reason is because all rows in both tables will be joined. So how do you avoid writing a Cartesian Product Join?

To avoid a Cartesian Product Join, check your syntax to ensure that the join is based on an EQUALITY condition. In the join syntax example above, the WHERE clause is missing. Because this clause is missing, a common domain condition between the two tables (i.e., e.dept = d.dept) does not exist, the result is a product join. Another cause of a product join is when aliases are not used after being established.

Chapter 10 — Join Indexes

"I have found the best way to give advice to your children is to find out what they want and then advise them to do it."

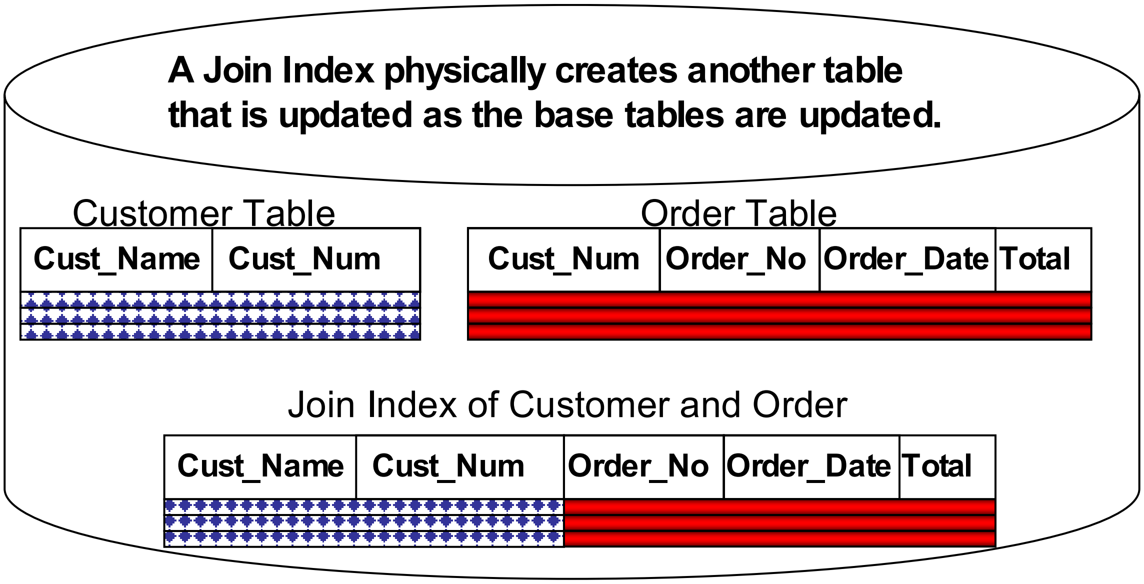
– Harry S. Truman (1884 - 1972)

Dewey have a great technique for you? Join Indexes are a relatively new feature that increases the efficiency and performance of queries containing joins. Teradata accesses the join index instead of resolving all the joins in the participating base tables.

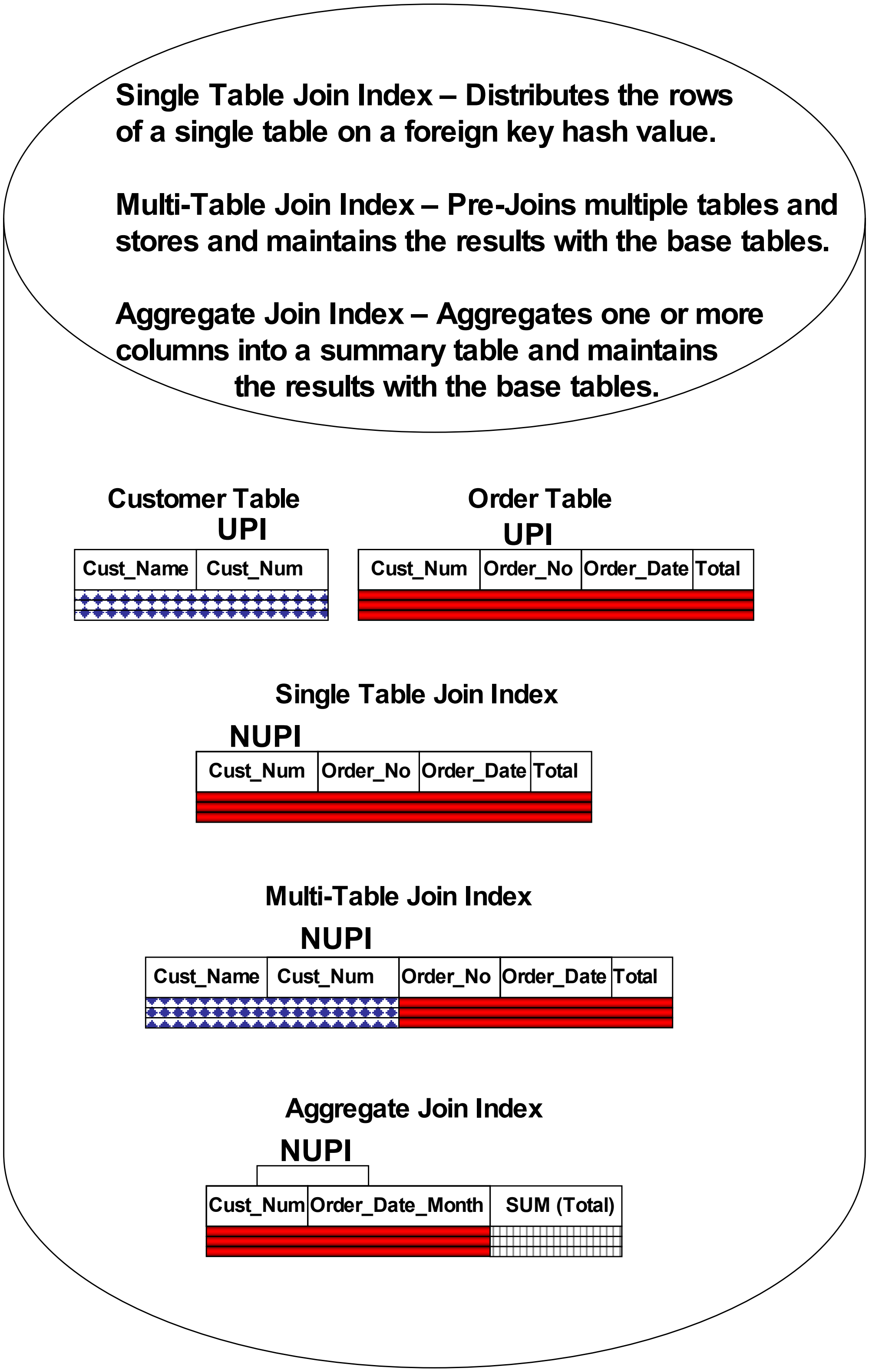
A Join Index can have **repeating values** and Join Indexes are **automatically updated** when the **base tables change**. They actually create a new physical table. Users don't access the Join Index table, but the Teradata does when appropriate.

Think of join indexes as aggregates or summary tables that users don't have to maintain because Teradata automatically manages the entire process. In fact, a user cannot view the row contents of join indexes even if they wanted to. Their operation is entirely transparent to the user. After deleting a base table row, you will not have to update the aggregate or joined table – say goodbye to those pesky "temporary" tables that need manual refreshing on a daily basis or whenever the contributing table rows were changed.

A **Join Index** can have a **different Primary Index** then the base table.


A Join Index physically creates another table
that is updated as the base tables are updated.
Customer Table
Cust_Name
Cust_Num
Order Table
Cust_Num
Order_No
Order_Date
Total
Join Index of Customer and Order
Cust_Name
Cust_Num
Order_No
Order_Date
Total

Three basic types of Join Indexes



Single Table Join Index – Distributes the rows of a single table on a foreign key hash value.

Multi-Table Join Index – Pre-Joins multiple tables and stores and maintains the results with the base tables.

Aggregate Join Index – Aggregates one or more columns into a summary table and maintains the results with the base tables.

Customer Table
UPI

Cust_Name	Cust_Num

Order Table
UPI

Cust_Num	Order_No	Order_Date	Total

Single Table Join Index
NUPI

Cust_Num	Order_No	Order_Date	Total

Multi-Table Join Index
NUPI

Cust_Name	Cust_Num	Order_No	Order_Date	Total

Aggregate Join Index
NUPI

Cust_Num	Order_Date_Month	SUM (Total)

Join Index Fundamentals

Join Index drawbacks

Join index implementation must be thought out thoroughly. Simply throwing a solution at a problem without carefully weighing its costs and rewards is an invitation for trouble. While join indexes are truly useful, they do not enhance performance in every situation. It would be inefficient to create join indexes for 90% of all feasible joins – like denormalization, such a proposal would require exponential amounts of storage.

Space consumption poses one of the most important concerns when using Join Indexes. Although not directly available through a query, Teradata must still store every row of a Join Index on disk. This is done much like any table row is stored – hashed to an AMP. When a Join Index is defined you are looking at twice the amount of space needed per column. If the system is running low on physical disk storage, Join Indexes may do more harm than good.

Join Indexes also require a certain amount of overhead and upkeep. Teradata transparently maintains all join indexes so that the index rows are updated when the base rows change. This is beneficial from a human maintenance perspective because it limits the occurrence of update anomalies. However, the system overhead involved in this automatic maintenance is an important consideration.

When to use a Join Index

The benefits and drawbacks of a Join index are discussed in detail above. You should consider using a join index in the following circumstances:

- **Specific, large tables** are **frequently** and **consistently joined** in which the result set includes **a large number of joins** from joined tables.

- A table is consistently joined to other tables on a column other than its Primary Index.

- Queries all request a small, consistent subset of columns from joined tables containing many columns.

- The **retrieval benefits are greater** than the **cost of setting up, maintaining** and **storing the join index**.

Join Indexes versus other objects

Join Indexes versus Views

Join Index	View
Rows are physically stored on the disks.	Rows are compiled each time the view is referenced.
Ability to have a Primary Index.	Cannot have a Primary Index.
Uses up a Perm space.	Uses only Spool space, while the query is being executed.
Main function is to increase access speed to data.	Main function is to manipulate how data is seen in reports, and for security.
Rows are not accessible to users	Rows are accessible to users with the proper rights.

Join Indexes versus Summary Tables

Join Index	Summary Tables
Rows are physically stored on the disks.	Rows are physically stored on the disks.
Ability to have a Primary Index.	Ability to have Primary Index.
Uses up Perm space.	Uses up Perm Space
Main function is to increase access speed to data – maintained by RDBMS.	Main function is to increase access speed to data – users have to maintain.
Rows are not accessible to users	Rows are accessible to users with the proper rights.

Join Indexes versus Temporary Tables

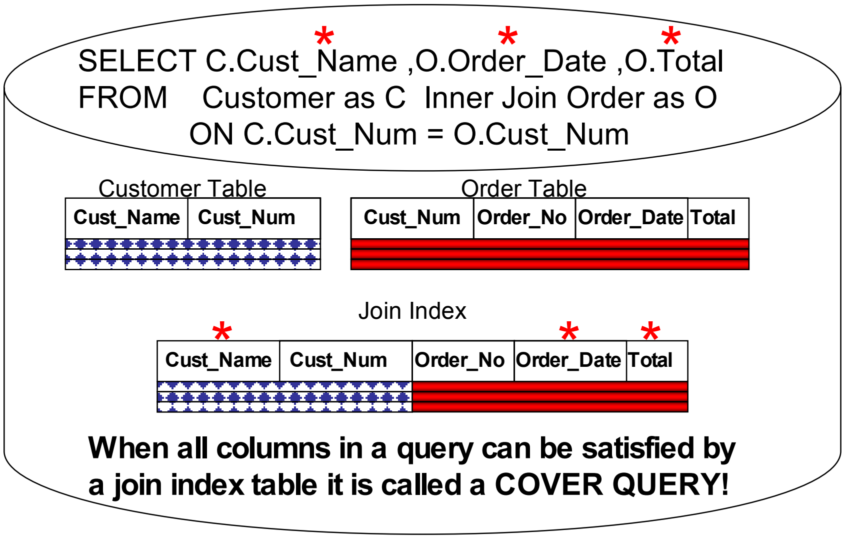
Join Index	Temporary Tables
Rows are physically stored on the disks.	Rows are in Spool or Temp space.
Ability to have a Primary Index.	Ability to have Primary Index.
Uses up Perm space.	Does not use Perm Space.
Main function is to increase access speed to data.	Main function is to allow for a quick and disposable look at the data.
Rows are not accessible to users	Rows are accessible to users with the proper rights.

Multi-Table Join Index

"Beware of the young doctor and the old barber."

– Benjamin Franklin (1706 - 1790)

If you want to discover electricity in your warehouse place the right keys and string them together in a join index. A multi-table join index is defined as creating a join index that involves more than one table, generally for joins of known queries. The essence behind a multi-table join index is that Teradata stores an answer set of an often-used query on disk. For example, join indexes for an outer join have the ability to preserve the unmatched rows resulting from the join. This is beneficial because this allows the join index to optimize more queries that have few join conditions. When all columns required by a query are available via a **Join Index** this is an example of a **Cover Query**.



```
SELECT C.Cust_Name ,O.Order_Date ,O.Total
FROM    Customer as C  Inner Join Order as O
             ON C.Cust_Num = O.Cust_Num
```

Customer Table

Cust_Name	Cust_Num

Order Table

Cust_Num	Order_No	Order_Date	Total

Join Index

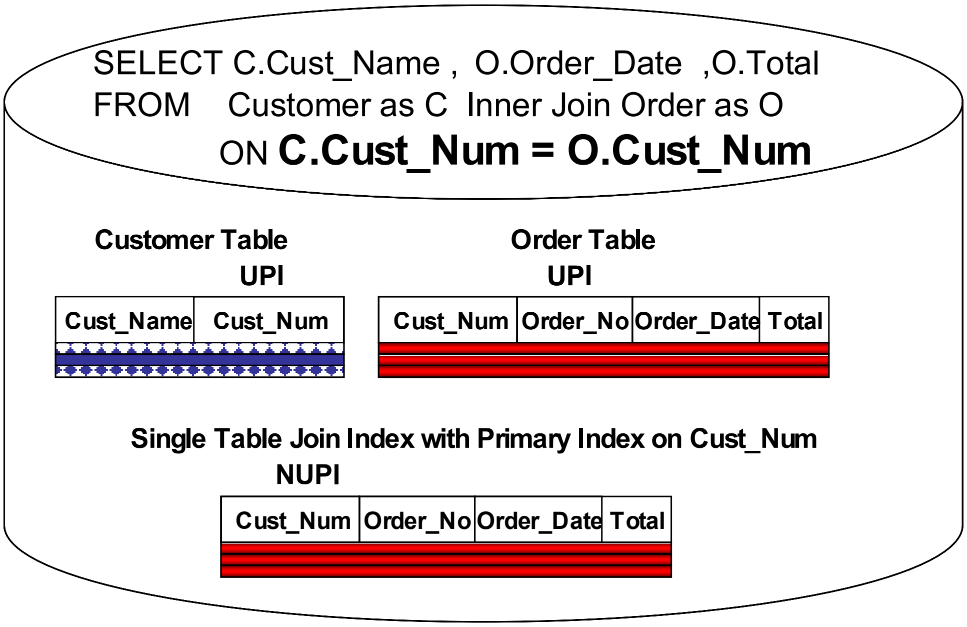
Cust_Name	Cust_Num	Order_No	Order_Date	Total

When all columns in a query can be satisfied by a join index table it is called a COVER QUERY!

```
CREATE JOIN INDEX Multi_Join_Index AS
SELECT                    C.Cust_Name
                         ,C.Cust_Num
                         ,O.Order_no
                         ,O.Order_Date
                         ,O.Total
FROM                     Customer as C
INNER JOIN               Order      as O
ON                       C.Cust_Num = O.Cust_Num
Primary Index(C.Cust_Num)       ;
```

Single-Table Join Indexes

A Single-Table Join Index is defined on **all** or **some** of the columns of one table with the **primary index** generally being a **foreign key** in that table. A great reason for a Single-Table Join index is a base table that is joined often in queries, but with the Primary Index not being in the join criteria. The joins may also pull from only half the columns in that table as well. A single-table join index can be created including only the desired rows and the Primary Index that matches the join criteria.



```
SELECT C.Cust_Name ,  O.Order_Date  ,O.Total
FROM    Customer as C  Inner Join Order as O
          ON C.Cust_Num = O.Cust_Num
```

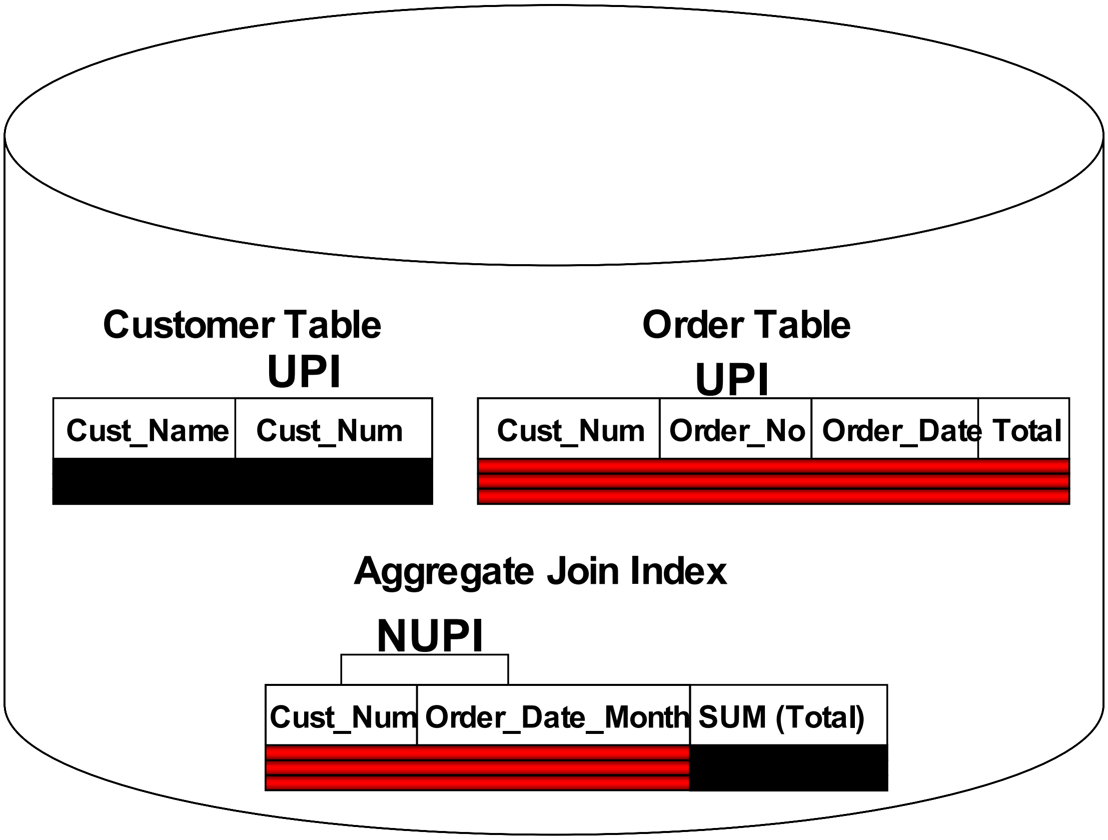
When the Customer and Order Tables are joined the rows will be on the same AMP

```
CREATE JOIN INDEX Single_Join_Index AS
SELECT                        Cust_Num
                             ,Order_no
                             ,Order_Date
                             ,Total
FROM                         Order
Primary Index (Cust_Num)       ;
```

Aggregate Join Index

Aggregate join indexes literally **pre-join and** summarize **aggregated tables** without requiring any physical summary tables. Summary tables often possess inaccurate data due to being merely a snapshot of the base tables. If the base table rows change, the summary table won't reflect this change without human intervention from a DBA or user. And even if such tables are frequently refreshed, human error can cause for anomalies to be overlooked. The following restrictions apply to aggregate join indexes:

- COUNT and SUM are only permitted.
- The DISTINCT command is not permitted.
- Resulting COUNT and SUM columns should be stored as type FLOAT to avoid overflow.

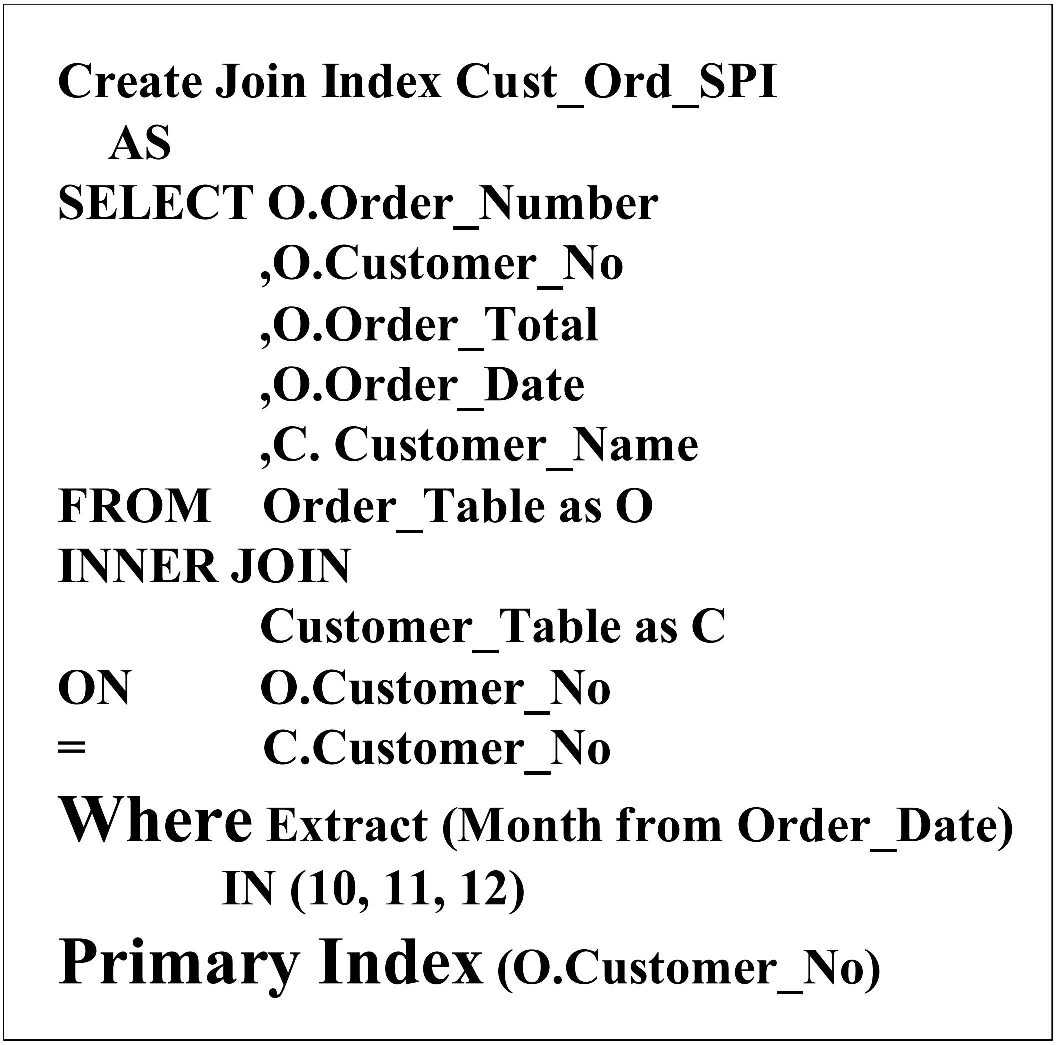


```
 CREATE JOIN INDEX Agg_Join_Index AS
 SELECT   Cust_Num
          ,Extract (Month from Order_Date)
                        AS MON
          ,Sum (Total)
 FROM     Order
GROUP BY 1, 2
```

Sparse Index

A **Sparse Index** is a **join index** with a **WHERE Clause**. If V2R4 when you created a join index of two tables they were joined in their entirety. If you had a yearly table joined to another yearly table you had two twelve month tables physically joined together. The good news was that when queries joined the two tables the queries ran fast. The bad news was that even if you only ran queries asking for the most recent quarter of information you still had to maintain all twelve months in the join index table. Now, with Teradata V2R5 you can join index only the rows you want and do so with a WHERE Clause. A Sparse Index:

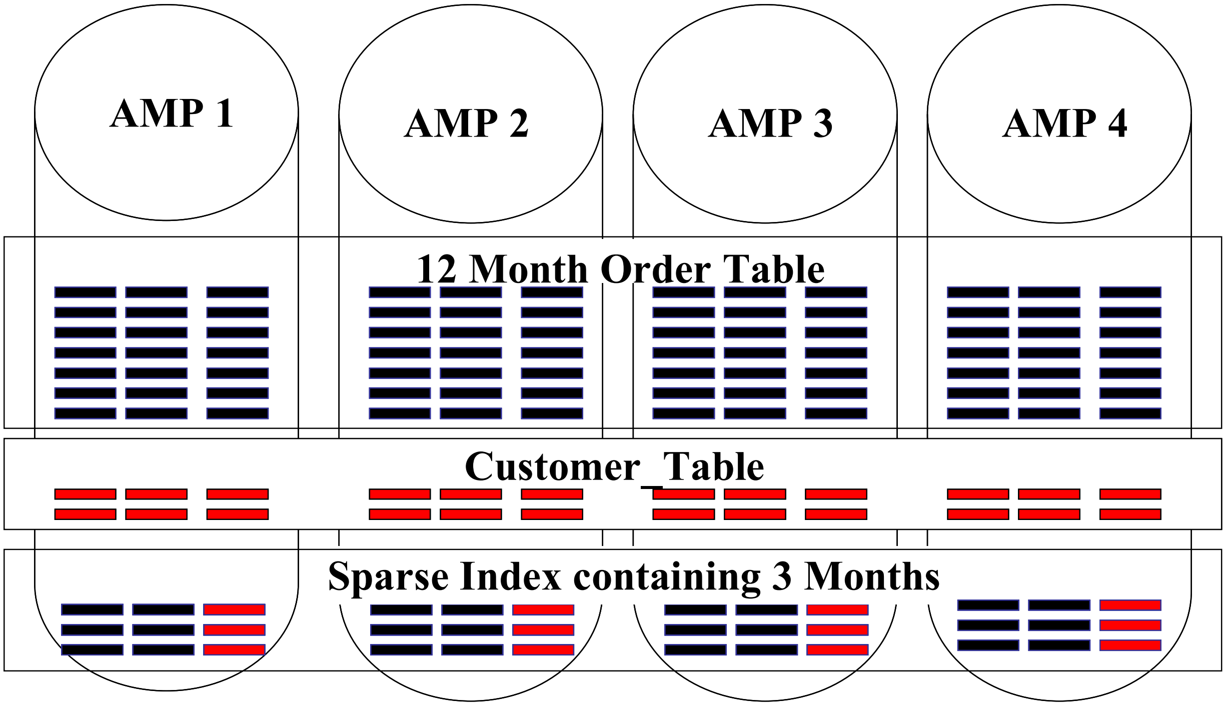
- Will have a WHERE clause in the Join Index
- Can have a Non-Unique Primary Index named in the join index



```
Create Join Index Cust_Ord_SPI
   AS
SELECT O.Order_Number
             ,O.Customer_No
             ,O.Order_Total
             ,O.Order_Date
             ,C. Customer_Name
FROM    Order_Table as O
INNER JOIN
             Customer_Table as C
ON           O.Customer_No
=            C.Customer_No
Where Extract (Month from Order_Date)
           IN (10, 11, 12)
Primary Index (O.Customer_No)
```

Sparse Index Picture

- Less maintenance and overhead because they are smaller
- Queries run faster because the Join Index table is smaller
- Create for queries with high volumes
- Sparse Indexes take up less space than Join Indexes
- Always collect statistics on the index even if only a single column is indexed
- You can redefine the WHERE clause
- You will get better UPDATE performance on the base tables if the index is strongly selective.



Global Join Index

Teradata is by nature born to be parallel. Generally, when you are accessing multiple rows you usually utilize All-AMPs. The Global Join Index allows you to minimize the number of AMPs you use because Global Join indexes are like Hashed NUSI's. Here is how it works!

A Global Join Index:

- Uses a single-table Join Index as an access path to a base table

- Creates a global index on a single table or a set of joined tables

- Hashes rows by an indexed column or set of columns

- Each Index row points back to the real base row in the real base table

- Will have rows hashed to a different AMP then the rows on the base table

- Is very similar to a hashed NUSI but most likely won't access all the AMPs

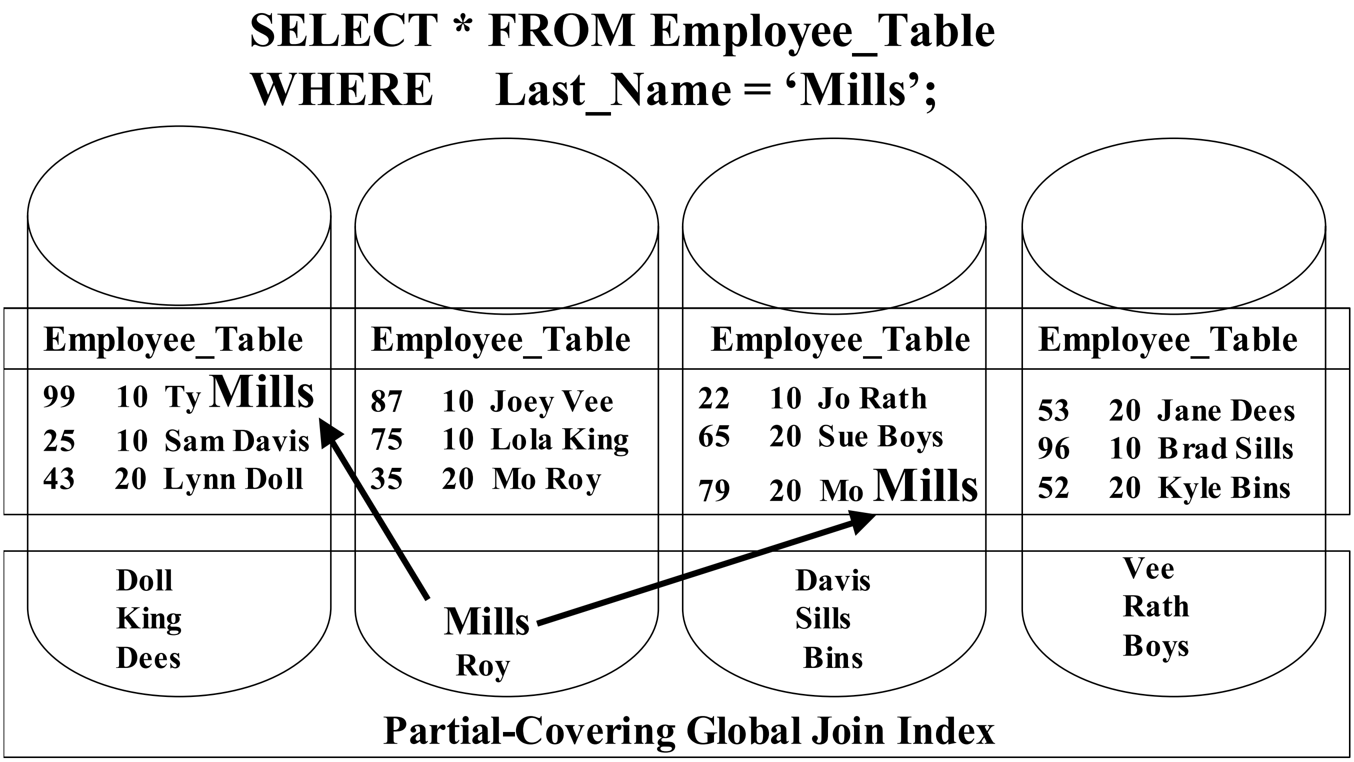
- Is also similar to a hashed USI, but most likely will be more than a 2-AMP operation

- Is outstanding for ODS queries with an equality condition on a strongly selective column because it changes two things:

    1. Table-level locks change to row hash locks
    2. All-AMP queries are now Group-AMP queries

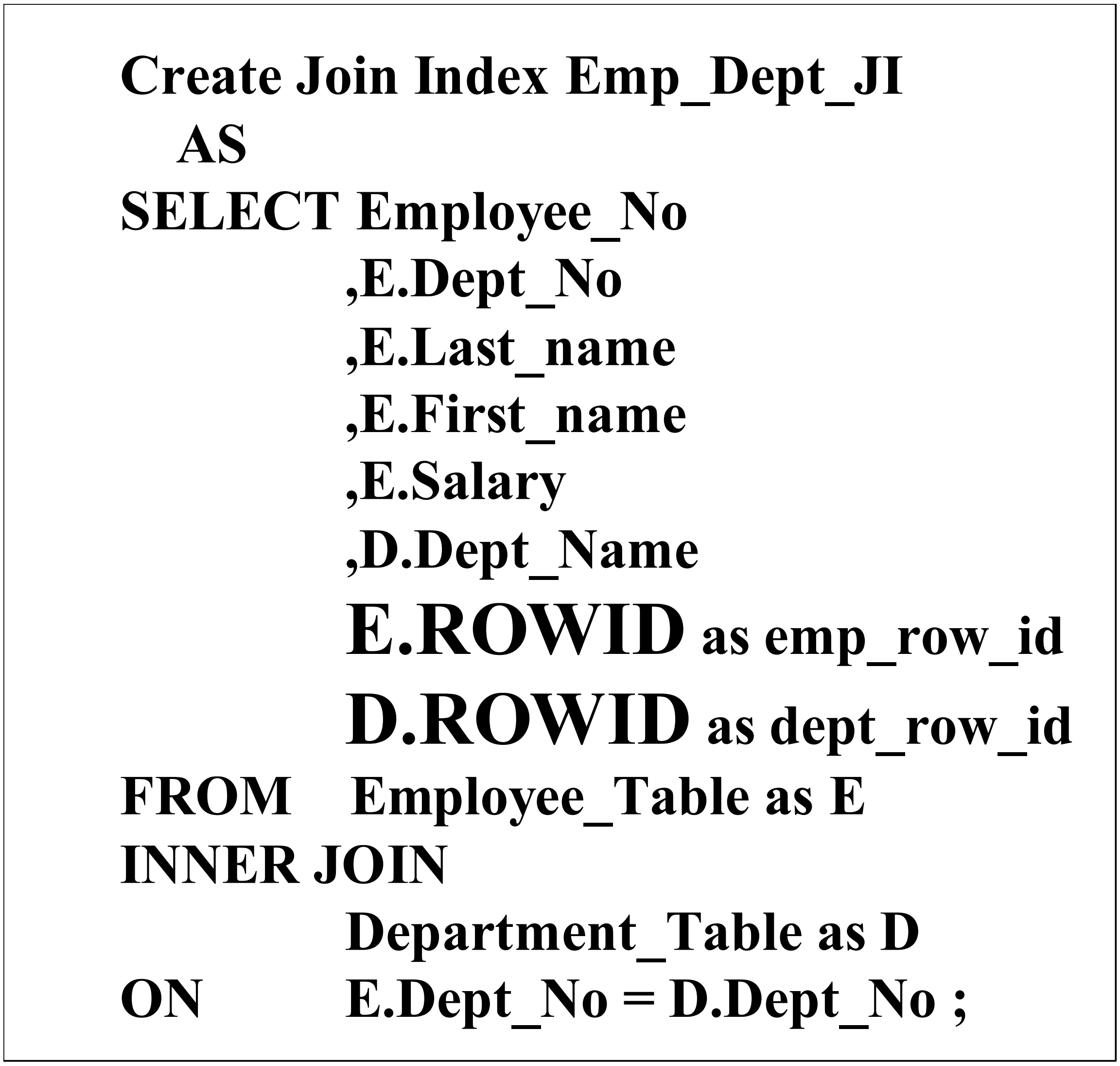
Global Join Index Picture

A Global Join index was created on the Last_Name of the Employee_Table. The Join Index table will hash the last name and store it in a separate sub-table. The sub-table will contain the Last_Name, Last_Name Row-ID, and Base Row-IDs for every base row with that name. In the example below the name Mills hashes to the sub-table on AMP2. The Sub-table row contains the Base Table Row-IDs for all rows that have the Last_Name of Mills.



Global Join Index – Multi-Table Join Back

In the previous slides we saw a single table Global Join Index. A single table approach could not be used to fully cover joins. Multi-table join indexes can be used for join back. Join Back means the join index includes the base Row-ID where additional columns can be found quickly. Because they involve multiple columns from both tables and retain the base tables Row-IDs they can cover queries and join back to the Base Table for any remaining columns needed. Here is an example of the syntax. Notice that Row-IDs are also asked for in the syntax.



```
Create Join Index Emp_Dept_JI
   AS
SELECT Employee_No
             ,E.Dept_No
             ,E.Last_name
             ,E.First_name
             ,E.Salary
             ,D.Dept_Name
             E.ROWID as emp_row_id
             D.ROWID as dept_row_id
FROM    Employee_Table as E
INNER JOIN
             Department_Table as D
ON           E.Dept_No = D.Dept_No ;
```

Hash Indexes

Hash indexes are similar to single-table simple join indexes in that they are used for denormalizing a single table by storing rows in a separate physical object from the base table. Hash indexes are limited to a SINGLE table and are beneficial in improving efficiency.

Like join indexes, Hash index rows are stored in a different object than the base table rows. This allows for faster access without needing access to the underlying rows. Also like their relative, they function as an alternative to the base rows instead of providing a different access path.

Like join indexes, hash indexes are known as "covered queries" because they are utilized when they contain all the columns requested by the query. If all of the columns are covered, the optimizer will usually choose to access the rows via the hash index instead of the base table rows. In a situation where the hash index partially covers the query, the optimizer may use the Row ID included with the Hash Index to access the other columns in the data row.

Join indexes and hash indexes are both transparently maintained by Teradata. When the base table changes, the Hash Index table is automatically updated. This automatic task is referred to as an update burden. Being that Hash Indexes are strikingly similar in functionality to secondary indexes, they should be carefully considered because they carry this update burden.

Hash indexes however, can offer marginal performance gains over a secondary index when certain columns are frequently needed from one table to join with many others. In these situations, the Hash index partially covers the query, but fully covers the rows from the base table, effectively eliminating any need to access the base table.

Hash Indexes:

- Can be ordered by hash or by values to facilitate range queries.
- Automatically has the Row ID of the base table row included in the Hash Index which the RDBMS software can use to access columns not included in a "covered query"

Hash Indexes vs. Single-Table Join Indexes

Hash indexes are nearly identical to single-table join indexes in the way they function and behave. By defining either index on a set of table columns, one can reduce the amount of columns and rows that participate in a join. See the previous section for a more detailed description on how such indexing can be used to redistribute table columns among AMPs to improve join performance.

The big difference between Hash and Single-Table Join Indexes is how they are defined:

Command: **CREATE HASH INDEX ([column1],[column2]…)**
ON [tablename];

This is much simpler syntax than defining a single-table simple join index and thus one of the reasons one would chose a hash-index over a join index.

The following list summarizes the similarities of hash and single-table join indexes.

- Both improve query performance
- Both are ways to achieve denormalized results without physically doing so.
- Both are maintained automatically by Teradata and transparent to the user.
- Both are physical objects that comprise their own table in PERM.
- Neither can be explicitly queried or directly updated.
- Both can be FALLBACK protected.
- The following commands can be used on either:
    - SHOW HASH/JOIN INDEX
    - COLLECT STATISTICS
    - DROP STATISTICS
    - HELP INDEX
- Both are not permitted in MultiLoad and FastLoad utilities.
- Both can be compressed

However, Hash Indexes provide two additional capabilities which are:

- Can be ordered by hash or by values to facilitate range queries.

- Automatically has the Row ID of the base table row included in the Hash Index which the RDBMS software can use to access columns not included in a "covered query"

Chapter 11 — Explains

"Lucy – You got some Splaining to do."

– Ricky Ricardo – I love Lucy

Teradata loves the "I love Lucy" show and has dedicated itself to do some splaining of each and every query. You can learn a lot from the explain command and also have a ball in doing so.

What does an Explain do? Usually I think of this as when I am discussing how to configure a computer component to my dad. Now my father is not the greatest with computers, but he is works really hard at it all the time. He always told me that you could never teach "an old dog new tricks". Well now my dad does Email, surfs the web, buys stocks, and has even learned how to send attachments to me. I would say that he has learned some new tricks. However, it took me the longest time to EXPLAIN to him in detail the steps to do all these things with his computer. Now he does these all of these things automatically with no hesitation. So the moral of the story is dads can learn new things if someone takes the time to explain things step by step.

Well EXPLAINS in Teradata are not that much different. They are a step-by-step analysis of the queries being executed in the database. In addition, once one gets familiar with the explain syntax and understand the key words to look for, then deciphering the EXPLAINS are automatic and easy. This is the purpose of this chapter.

Basically, the EXPLAIN statement is the Parsing Engines (PE's) plan to the AMPs. EXPLAIN outlines in detail which indexes will be used to process the request. It could also identify intermediate spool files that would be generated in the process. In addition, EXPLAINS could show whether the statements in a transaction would be completed in parallel.

How do you activate the EXPLAIN? Simple, by inserting the word "EXPLAIN" in front of the query, you will activate the PE's plan. When this is done, the query will not run. It will only show the PE's Plan in English.

The EXPLAIN is an excellent facility for the following activities:

- Determining Access Paths
- Validating the use of indexes
- To determine locking profiles
- Showing Triggers and Join Index access
- Estimate the query runtime

In addition, the EXPLAIN will always identify a DDL request (CREATE, DROP, or ALTER) by showing a data dictionary table write lock.

The Teradata Optimizer "knows" how to Explain in Detail

"Toto – We're not in Kansas anymore"

– Dorothy in the Wizard of Oz

The Teradata Optimizer also went to Oz and received the brain behind the Teradata database. It is all knowing and powerful. The optimizer determines the plan of attack to resolve any question asked of it. The Teradata Optimizer is basically known as the PE. One thing for sure is that the optimizer is never lion when it gives users a toto number of rows estimate.

The Teradata Optimizer is made up of thousands of lines of code that has been developed over 23 years. The Teradata Optimizer is a cost-based optimizer. A cost-based optimizer searches for the lowest cost plan to resolve a question in a timely manner. Remember time is money, therefore the optimizer will assign the costs to the steps and then look ahead to see if there are better choices available. Once the plan choices have been assessed, the Teradata Optimizer will choose the steps and start issuing the commands to get the job done ASAP. Keep in mind the Optimizer will not store the plan. Each plan is always dynamically regenerated each time a question comes to the optimizer that requires an answer. The reason is that plans can change based on what is happening on the system. In addition, confidence factors and ranges are assigned to the question (query) being posed to the optimizer.

So how does the optimizer assign confidence factors and ranges in addition to figuring out the plan of attack?

When a query is processed, execution times and row count are only estimates. These estimates are based on the following criteria:

- Are statistics collected? – Statistics are critical to the Optimizer for understanding the path and plan to execute a query.

- Is Teradata busy processing other requests? – This definitely influences how Teradata will execute the query.

- Is the channel or network busy? – As stated above – Teradata will choose an alternate route if the channel or network is overloaded.

In addition, the Parsing Engine (PE) knows the configuration of the Data Warehouse and the System Hardware Information such as the following:

- **Number of nodes in the system**
- **Number and type of CPU's per node**
- **Number of configured AMP Vprocs**
- Disk array configuration
- Interconnect configuration
- Amount of memory and memory configuration

With all this knowledge and power, how could the optimizer fail? The optimizer has the ability to answer some of the most complex questions in the world in a timely manner. This is probably why Teradata is being used at two of the world's largest warehouses being utilized today. This depth of knowledge of its environment and surrounding allows the optimizer to excel over the competition.

Row Estimate Confidence Levels

"You are educated when you have the ability to listen to almost anything without losing your temper or self-confidence."

– Robert Frost

Teradata is like a poet who doesn't know it because it will lose its confidence if statistics have not been collected.

There once was a query from one
Where an explain was done just for fun
The confidence was low
And the query ran slow
Cause the statistics collected were none

– Robert Frost after a Teradata class with Tera-Tom

As stated, Teradata is all knowing to its environment in addition to being all-powerful when it comes to making decisions. Teradata makes decisions based on the best choice that is available to resolve a query. The optimizer can be influenced to make better decision if the users can pass additional knowledge. The major influencing factor that can be communicated to the optimizer is the understanding of the number of rows that a table has before it is queried. The key thing a user can do to assist the optimizer in this decision making process is to collect statistics. When statistics are collected, this knowledge is communicated to the optimizer. The more the optimizer knows, the better the guarantee that the estimate will be right on the mark. If we issue an explain command in front of a query statement, there are a several estimated confidence messages that the optimizer will relay to the user which are:

- **High Confidence** - Statistics available on an Index or Column.
- **Low Confidence** - Random sampling of INDEX, or Statistics available but AND/OR condition exists.
- **No Confidence** - Random sampling based on AMP row count. **No statistics are collected.**

Explain Terminology

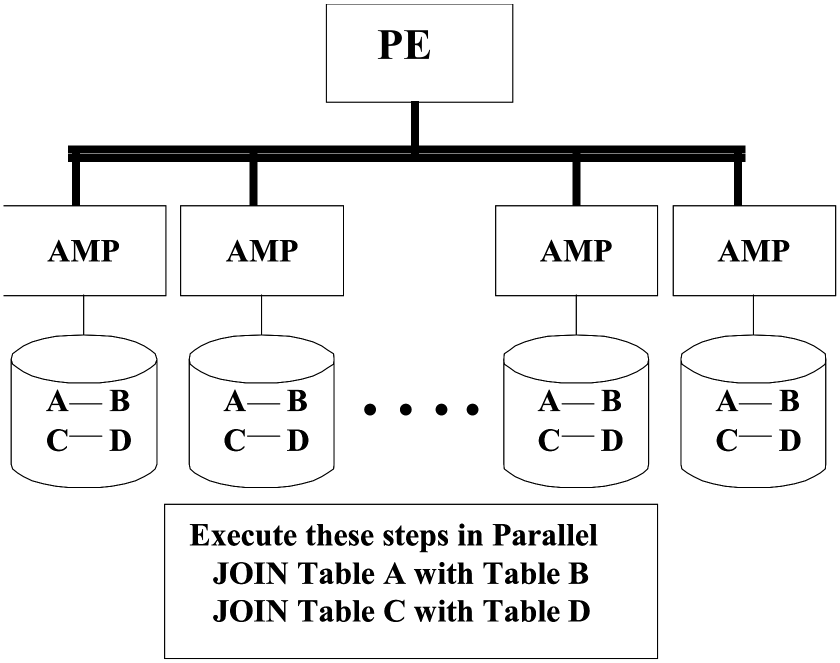
EXPLAINS, as discussed before, are a step-by-step analysis of the queries being executed in the database. Understanding the EXPLAIN Terminology is key for a user to can gain valuable insight on how the system is going to process the query. This will also provide the user the ability to apply the techniques learned in the previous chapters about how to influence the Optimizer. Once one gets familiar with the EXPLAIN syntax and understands the key words to look for, then deciphering the EXPLAINS are automatic and easy. The terminology and explanations behind these valuable key words are shown below.

Explain Statement	Explain Definition
All-AMPs JOIN STEP BY WAY OF AN ALL-ROWS SCAN	On each AMP on which they reside, spooled rows and/or primary table rows are searched row by row; rows that satisfy the join condition are joined.
All-AMPs JOIN STEP BY WAY OF A ROWHASH MATCH SCAN	The first row is retrieved from the first table; that row's hash code is used to locate a row from the second table.
All-AMPs RETRIEVE STEP BY WAY OF AN ALL ROWS SCAN	All rows of a table are selected row by row on all AMPs on which the table is stored.
ESTIMATED SIZE	This value, based on any statistics collected for a table, is used to estimate the size of the spool file needed to accommodate Spooled data. If statistics have not been collected for a table, this estimate may be grossly incorrect.

DUPLICATED ON ALL AMPS… A spool file containing intermediate data that is used to produce a result is copied to all AMPs containing data with which the intermediate data is compared. The above statement means **“Data Movement”.**

ESTIMATED TIME	This approximate time is based on average times for the sub-operations that comprise the overall operation, and the likely number of rows involved in the operation. The accuracy of the time estimate is also affected by the accuracy of the estimated size.
WITH NO RESIDUAL CONDITIONS	All applicable conditions have been applied to the rows.
A **SINGLE PARTITION** OF or **PARTITION NUMBER** OF	The table involved is a **PPI table** and it indicates the AMPs will only access a **single partition** or **number of partitions** in a table. The query has a **range constraint** or an **equality constraint** on the **partitioning column**.
HIGH CONFIDENCE	High Confidence means statistics have been collected. This can mean **restricting conditions** exist on an index or **column** that has **collected statistics**. It could also mean that one input object has **high confidence** and other input objects have **high** or **join index confidence**.

LAST USE	A spool file is no longer needed. It will be freed up after this step.
WITH A RESIDUAL CONDITION	WHERE phrases could not be resolved using an index. These conditions become "residual conditions" and are applied to the row after they are retrieved using the listed index.
END TRANSACTION	All Transactions are released and changes are committed.

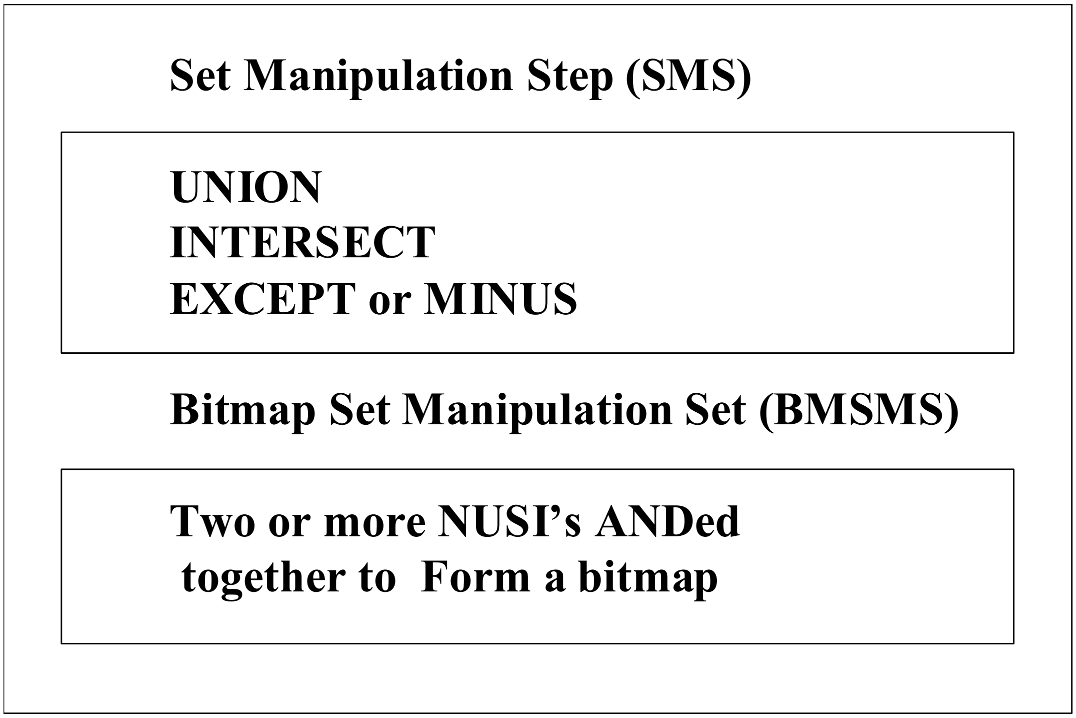


Execute these steps in Parallel
JOIN Table A with Table B
JOIN Table C with Table D

EXECUTE THE FOLLOWING STEPS IN PARALLEL… means **Execute** the following steps in **parallel**... This identifies a set of steps that are processed concurrently. The explanatory text immediately following the list describes the execution of each step.

Parallel steps are **sequentially numbered** steps that can **execute simultaneously**, taking advantage of Teradata Parallel Architecture. Parallel steps are steps whose **output may even be used later** in the plan.

ELIMINATING DUPLICATE ROWS	Eliminating duplicate rows in the spool files. This is done by using the DISTINCT command.
BY WAY OF THE SORT KEY	A sort is done on the field mentioned.
WE DO AN ABORT TEST	A test that is caused by a Rollback or Abort statement.
WHICH IS REDISTRIBUTED BY HASH CODE TO ALL AMPS	Redistributing each of the data rows for a table to a new AMP based on the hash value for the column or columns involved in a join.



Set Manipulation Step (SMS)

UNION
INTERSECT
EXCEPT or MINUS

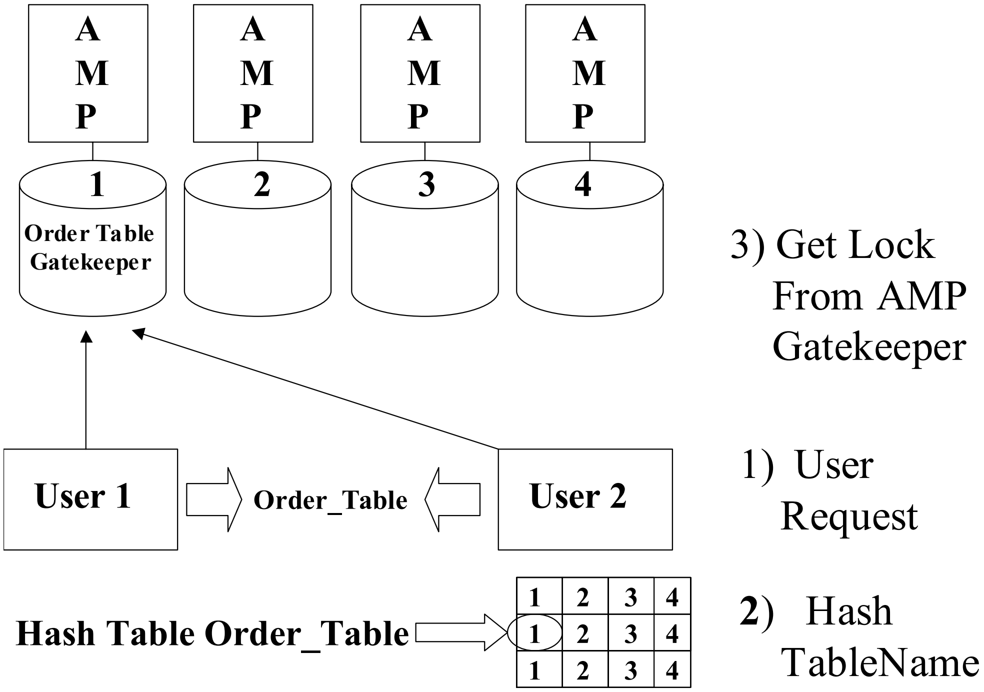
Bitmap Set Manipulation Set (BMSMS)

Two or more NUSI's ANDed together to Form a bitmap

WE DO AN SMS… A Set Manipulation Step caused by a UNION, MINUS, or INTERSECT operation. A **BMSMS** means two or more **NUSI's** are "AND"ed to form a **bitmap**.

WE DO A BMSMS… A way of handling two or more weakly selective secondary indexes that are linked by an AND in the WHERE clause.	
WHICH IS BUILT LOCALLY ON THE AMPs	Each AMP builds a portion of a spool file from the data found on its local disk.
BY WAY OF A **TRAVERSAL** OF **INDEX #N** EXTRACTING **ROW IDs** ONLY	**A spool file** is built by way of **row IDs** found from a **secondary index**. These Row IDs will be used to extract data later.
AGGREGATE INTERMEDIATE RESULTS ARE COMPUTED LOCALLY	Aggregation requires no redistribution.
AGGREGATE INTERMEDIATE RESULTS ARE COMPUTED **GLOBALLY**	Aggregation requires **local** and **global aggregation** phases for **computation**.
BY WAY OF A ROW HASH MATCH SCAN	A merge join is done on hash values.
WHERE UNKNOWN COMPARISON WILL BE IGNORED	Comparisons involving NULL values will be ignored.

If you are looking at an Explain and you see that a **Join Step** uses the term **"Residual Conditions"** it means that conditions must be applied **after** the join condition are all applied. An Explain will never use the word **subquery** so it is difficult to sometimes tell if the system is doing a **join or a subquery**. You can only tell by the **type of join step** that **is identified**.



1	2	3	4
1	2	3	4
1	2	3	4

3) Get Lock From AMP Gatekeeper

1) User Request

2) Hash TableName

WE LOCK A DISTINCT <DBNAME> PSUEDO TABLE… A way of preventing global deadlocks specific to an MPP DBS.

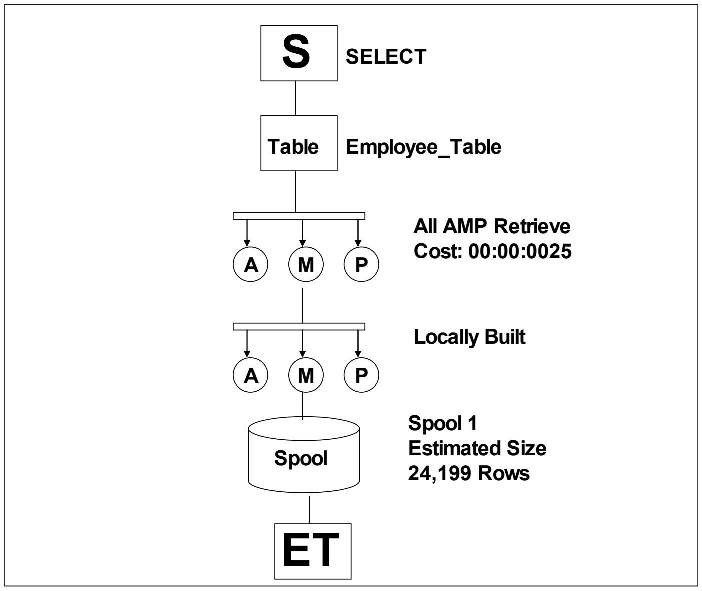
Visual Explain

Teradata has always been brilliant at explaining exactly what is going on with a query. Other databases have EXPLAIN capabilities, but they're too hard to understand. Teradata has taken it one step further with VISUAL EXPLAIN. **Visual Explain depicts** the PE optimizers execution plan to access data in a **visual format** with **easy to read icons**. This feature graphically represents the steps and can even perform comparisons of multiple plans. Visual Explain can also capture query plans in an emulated database environment.

How does Teradata do it? Visual Explain reads from the Query Capture Database (QCD) and turns it into a series of easy to read icons. To view a plan using Visual Explain, the execution plan information must be captured in the QCD using the Query Capture Feature (QCF). Some of the menu options with Visual Explain are:

- Show **Connectors**
- **Open Plan** from database
- Textual Output
- Setup QCD
- **Security**
- Explain **textual difference**
- **Compare Options**

Although Visual Explain is extremely versatile, it can't do everything. It does not identify **why the optimizer chooses a plan**, but only delivers the plan it chooses. You can use Visual Explain on any SQL statement except the keyword "**EXPLAIN**". Visual Explain merely translates the optimizer **output parser tree** into **icons**. You can see each step and its **estimated costs** in this version of Visual Explain.



Chapter 12 — The Parsing Engine in Detail

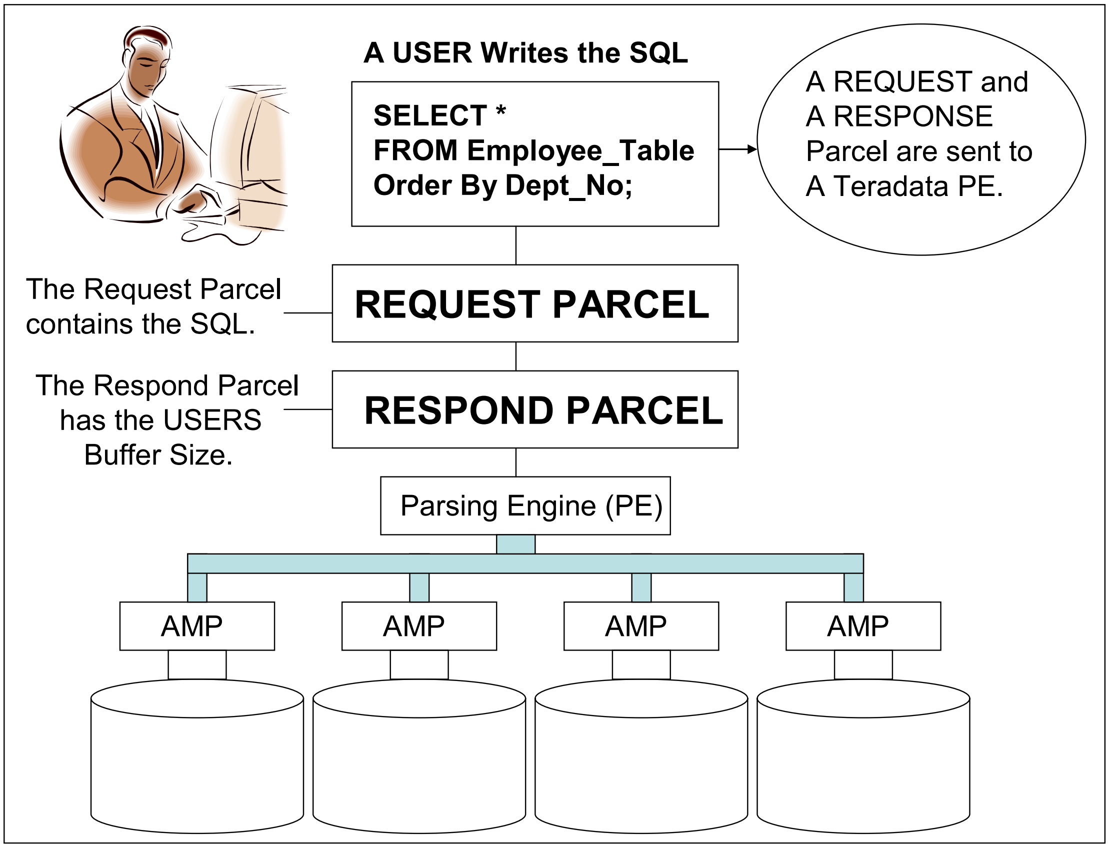
"It's always been and always will be the same in the world: the horse does the work and the coachman is tipped."

– Anonymous

When a user enters SQL and runs the query a Request and Respond Parcel are sent to a Teradata Parsing Engine. The Request Parcel will contain the SQL and the Respond Parcel will describe the USER buffer size. Now that Teradata knows the USER buffer size a Success/Failure Parcel will be sent back to the USER along with the SQL answer.

If the SQL is a Multi-Request Transaction then both SQL statements will most likely be sent in one Request Parcel.

If the USER created a macro that passed parameters then the USER SQL will generate a Data Parcel also. The Data Parcel contains the parameters. This allows Teradata to utilize the same plan, but with different data parameters.



```
SELECT *
FROM Employee_Table
Order By Dept_No;
```

A REQUEST and A RESPONSE Parcel are sent to A Teradata PE.

The Request Parcel contains the SQL.

The Respond Parcel has the USERS Buffer Size.

The Parsing Engine (PE) goes through Six Steps

"When a person speaks of strength they whisper their weakness."

– John M. Shanahan

Fundamentally we teach you that the Parsing Engine does three major things: Checks the SQL syntax, checks the security, and comes up with a plan for the AMPs to follow. This is actually accomplished with six steps:

Syntaxer – Checks the Syntax.

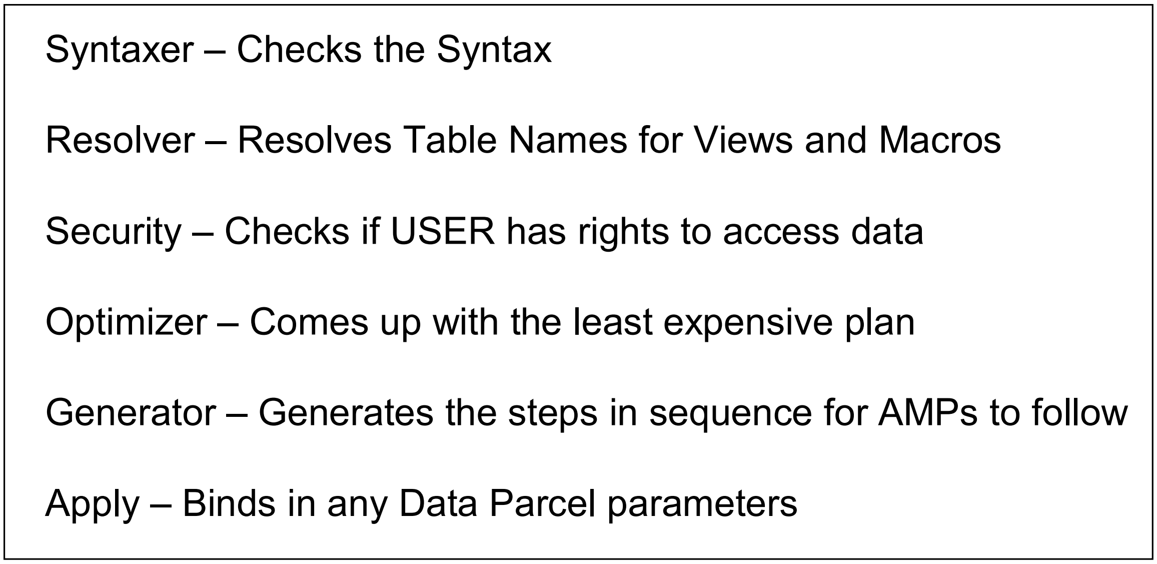
Resolver – Takes Views and Macros and from those objects finds the real tables being accessed.

Security – Checks to see if the user has the proper permissions to access the tables.

Optimizer – Comes up with the best plan for the AMPs to execute.

Generator – Creates the actual steps in sequence for the AMPs to follow.

Apply – Binds the data values only if parameters are passed.



Syntaxer – Checks the Syntax

Resolver – Resolves Table Names for Views and Macros

Security – Checks if USER has rights to access data

Optimizer – Comes up with the least expensive plan

Generator – Generates the steps in sequence for AMPs to follow

Apply – Binds in any Data Parcel parameters

Each PE has a Plan Library called RTS Cache

"Whoever gossips to you will gossip of you."

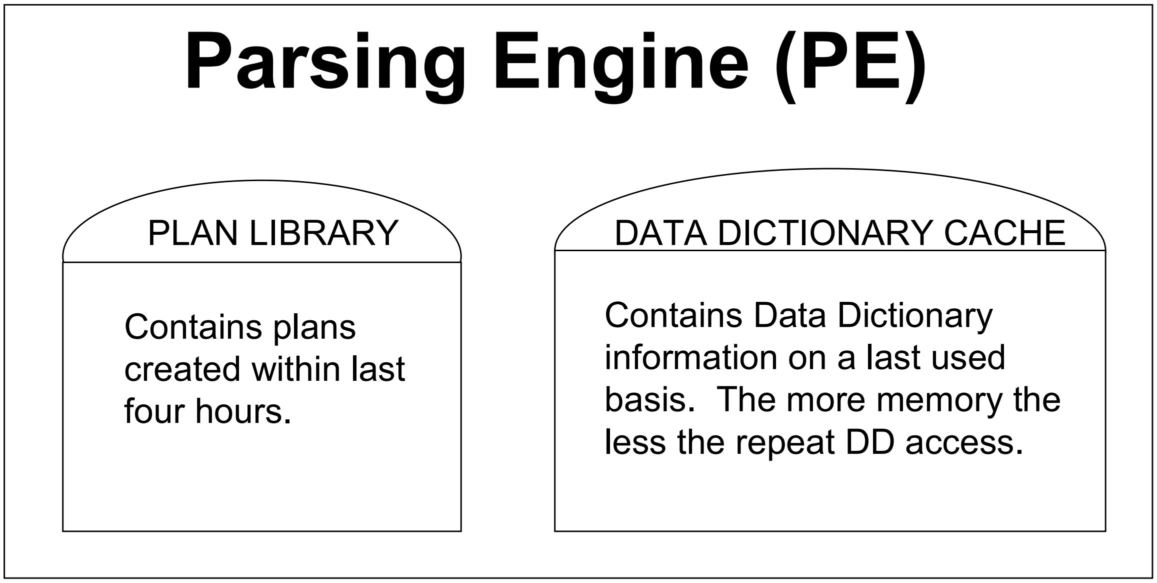
– Spanish Proverb

Each individual Parsing Engine will receive a portion of memory in a Node and this memory is referred to as CACHE. There are actually two types of cache. They are Data Dictionary cache which holds Data Dictionary information the PE has requested and Request-To-Step cache which is the Plan Library.

When SQL is sent to a Parsing Engine the PE must come up with a plan for the AMPs to follow. That plan is stored inside the RTS cache or Plan Library. When a PE sees that it has processed the same SQL before it can merely check security and then use the same plan as before, thus saving valuable time.

A Teradata PE recognizes previous SQL because it actually hashes the SQL and comes up with a 32-bit row hash. It stores the row hash and the SQL inside the plan library along with the plan for the AMPs to execute.

The Plan Library is flushed every four hours or whenever a DDL statement spoils the plan. For example, if someone recollects statistics then the old plan may no longer apply and is dismissed. The Plan Library is flushed every four hours to ensure freshness. A plan that was valid four hours ago may no longer be the least expensive approach.


Parsing Engine (PE)
PLAN LIBRARY
Contains plans created within last four hours.
DATA DICTIONARY CACHE
Contains Data Dictionary information on a last used basis. The more memory the less the repeat DD access.

The Parsing Engine has Data Dictionary Cache

"A minute's success pays the failure of years."

– Robert Browning

The Parsing Engine uses Data Dictionary Cache called DD Cache. The Parsing Engine accesses the Data Dictionary to check Database, Table and Column names. It also checks access rights and if statistics were collected. The DD cache that comes with each Parsing Engine (PE) holds the most recently accessed information in order to speed things up.

The DD Cache holds Table Names and their respective Table IDs. The DD Cache also holds data demographic information on tables without Collected Statistics. The DD Cache is also purged every four hours and it also spoils certain entries if DDL requests have been made.

The more Data Dictionary Cache you allocate to the PE's the better the performance. The amount of DD Cached is determined by the DBSControl utility in the Performance area parameter titled Dictionary/CacheSize. If you are not sure what to put in this field then go with 1024 KB.

Data Dictionary Cache allows the PE to hold valuable Data Dictionary information in memory. The DD Cache holds Table Names along with their Table IDs. It also holds information about tables where statistics were NOT collected. Tables utilized in DD Cache are:

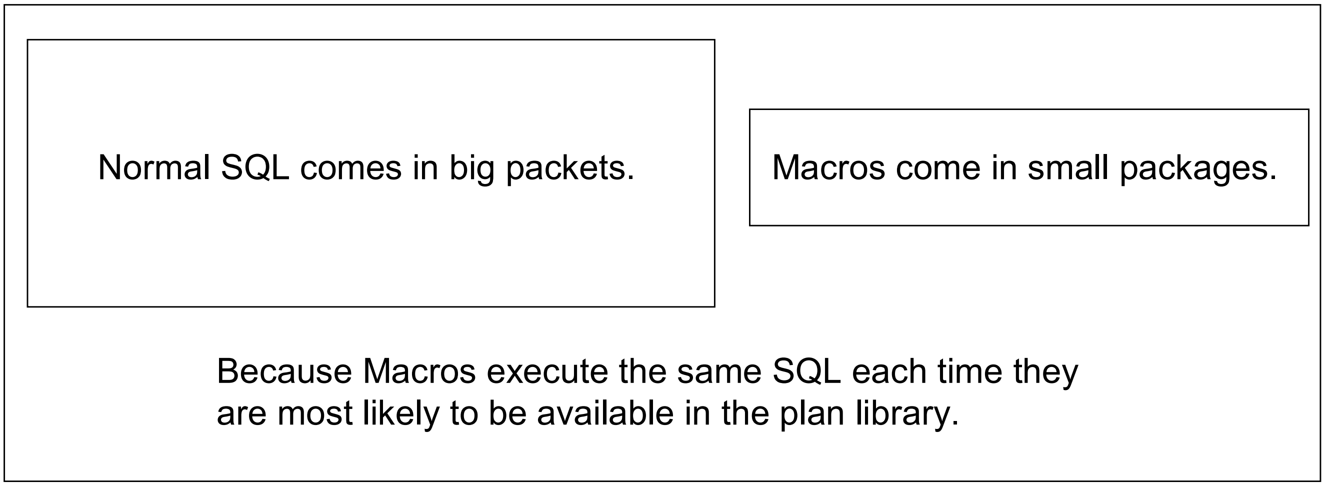
Dbase
TVM
AccessRights
RoleGrants
Indexes
TVFields

Why the PE loves the Macro

"Never engage in a battle of wits with an unarmed person."

– Anonymous

The PE's love macros because of a wide variety of reasons which include the fact macros are packaged in small packets. This reduces network traffic dramatically. Macros also use the exact same SQL each time they are executed so they often reside in the Plan Library called Request-To-Step Cache.



Normal SQL comes in big packets.

Macros come in small packages.

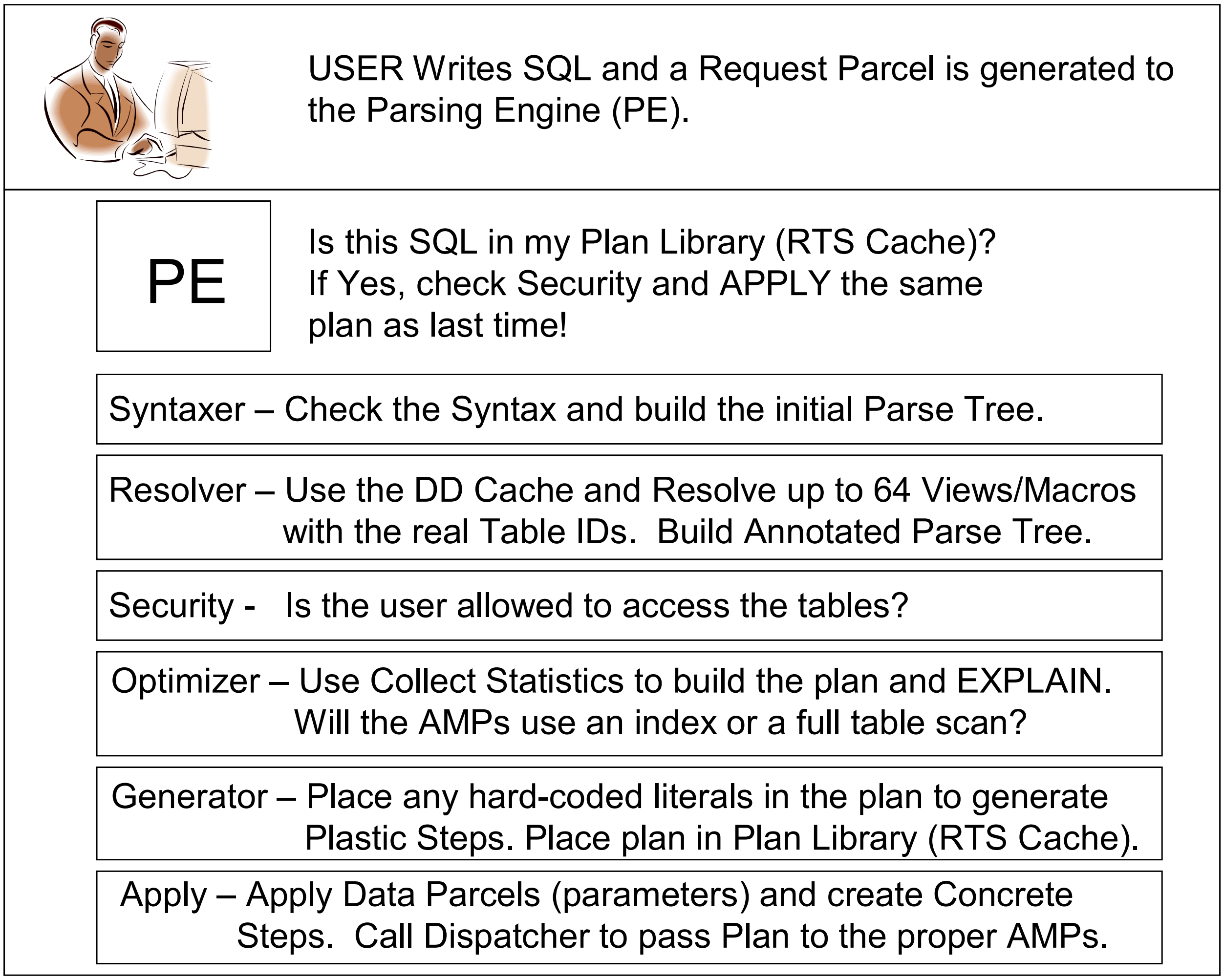
Because Macros execute the same SQL each time they are most likely to be available in the plan library.

The Parsing Engine in Detail

"Mediocre minds usually dismiss anything which reaches beyond their own understanding."

– Francois, Duc de La Rochefoucauld 1670

The slide below describes the PE process.



USER Writes SQL and a Request Parcel is generated to the Parsing Engine (PE).

Is this SQL in my Plan Library (RTS Cache)? If Yes, check Security and APPLY the same plan as last time!

Syntaxer – Check the Syntax and build the initial Parse Tree.

Resolver – Use the DD Cache and Resolve up to 64 Views/Macros with the real Table IDs. Build Annotated Parse Tree.

Security - Is the user allowed to access the tables?

Optimizer – Use Collect Statistics to build the plan and EXPLAIN. Will the AMPs use an index or a full table scan?

Generator – Place any hard-coded literals in the plan to generate Plastic Steps. Place plan in Plan Library (RTS Cache).

Apply – Apply Data Parcels (parameters) and create Concrete Steps. Call Dispatcher to pass Plan to the proper AMPs.

The Parsing Engine Knows All

"Never mistake knowledge for wisdom. One helps you make a living; the other helps you make a life."

– Sandra Carey

The Parsing Engine knows the system hardware and software, the data demographics, and from that information can come up with the best access path.

The Teradata Parsing Engine Knows All

System Information	Data Demographics	Access Path Decisions
Number of Nodes	Number of Rows	What Access Path?
Number of AMPs	Column Information	What Type of Join?
Type of CPUs	Row Size	Join Redistribution of Duplication?
Disk Array Information	Skew Values	What Order of the Table Joins?
BYNET Information	Rows Per Value	Should a Bitmap be used?
Amount of Memory	Index Demographics	Can a Cover Query be Used?

Chapter 13 — Understanding Views and Macros

"The longer I live the more beautiful life becomes."

– Frank Lloyd Wright (1860 – 1959)

If you want to architect your warehouse to perfection then utilize views and macros. They make writing and executing SQL easier for data warehouse users. This chapter will provide insight into maximizing the most out of views and macros in the data warehouse environment. Think of this section as a journey to view a whole new world to experience the wonders of creating, modifying, and dropping views and macros. Data warehouse life truly is beautiful.

Views

Teradata tables can easily contain millions or even billions of rows of data. Companies that I have consulted with don't even blink when they inform me that a typical table in their data warehouse contains 3 billions rows. A table can also have thousands of columns. Would it be surprising if I told you that customers tell me weekly that this is a limitation for them! The enormous size of these tables can really complicate writing SQL especially when the requirement is to join multiple tables. In this case, users would have to know how to code complex SQL when it comes to joining large tables to get the desired results. For these reasons, VIEWS have become a crucial piece in the data warehouse environment today.

In short, a VIEW is a virtual table. It is basically a SELECT statement that is stored in Teradata's Data Dictionary. In regards to a View, think of the Data Dictionary as a reference locater for where the true data resides. In addition, Views are used to provide customized access to data tables. This customization can include the following:

Limiting or restricting access to certain columns
Displaying columns from multiple tables (using joins)
Restricting the number of rows returned from one or more tables.

Views simplify SQL coding for the users and protect the actual tables from the users. Additionally, views allow for easier understanding of large tables by only displaying columns relevant to the user. Views provide the capability of pre-joining multiple tables together in order to simplify user access to information.

The last statement above is one of the biggest benefits for creating Views. The ability to pre-join multiple tables allows users to access data from what could have been a complex task to now querying this information with a simple "SELECT * FROM Viewname". In addition, views can be created to contain complex and sophisticated SQL. These types of views give the power of information back to the user by eliminating the requirement for user to be SQL experts. Knowledge is power!

Views are also used to create “virtual” summary tables, which dynamically utilize the power of Teradata’s parallelism to process data. This concept provides companies the ability to minimize the use of data marts for summary tables and aggregated data.

The other great benefit is that Views require no permanent space and are stored within the Data Dictionary. Therefore views can be customized for the needs of the users. So how do we create these powerful entities? The answer can be found below.

Creating Simple VIEWs and VIEWs that Join Tables

We have discussed a number of powerful reasons for why and how views are utilized. Now it is time to investigate how to create these objects in Teradata. Usually, DBAs have standards for creating views and other objects. In most cases, views are identified within the environment with standard notation (i.e., a view could either start with “v_” or end with “_v”). The syntax for creating a VIEW is as follows:

```
CREATE VIEW <view-name> [(<alias-name>, <alias-name>, … )] AS

          SELECT <column-name> [AS <alias-name> ]
                   [, <column-name> [AS <alias-name> ] ]
                   [, <column-name> [AS <alias-name> ] ]
         FROM <table-name>
       [ WHERE <condition-tests> ]
       [WITH CHECK OPTION];
```

HINT: In Teradata, the command CREATE VIEW may be abbreviated as CV. However, CV is not ANSI compliant.

An example of a VIEW would be as follows. Pay close attention to the syntax and the view name.

```
CREATE VIEW dept_v AS

          SELECT   dept_no  AS ‘Department Number’
                   ,department_name  AS ‘Department Name’
                   ,budget/12  AS  ‘Monthly Budget’
          FROM   department_table ;
```

Once the view has been created, the next step is to SELECT from the view and evaluate the result set. This can be accomplished by doing the following:

Command: **SELECT * FROM dept_v;**

Answer Set:

Department Number	Department Name	Monthly Budget
200	Research and Develop	45833.33
400	Customer Support	41666.67
100	Marketing	41666.67
300	Sales	54166.67
500	Human Resources	37500

The resulting view above displays the monthly budget for each department. Notice that since there is *NO* WHERE clause. This clause adds a condition to test if rows can be eliminated. Now lets create another view that will only display information for Departments 300 and higher.

```
CREATE VIEW  dept_v300 AS

     SELECT   dept_no
              ,dept_name
              ,budget/12 AS Monthly_Budget
     FROM     department_table
     WHERE    dept_no >= 300;
```

Once the view has been created, the next step is to SELECT from the view and evaluate the result set. This can be accomplished by doing the following:

Command: **SELECT * FROM dept_v300;**

Dept_no	Dept_name	Monthly_Budget
400	Customer Support	41666.67
300	Sales	54166.67
500	Human Resources	37500.00

The result set reveals that the WHERE clause for the dept_v300 view eliminates any Department numbers that are less than "300." In addition, no ALIASes were provided for the Dept_no and Dept_name column. So the column headers returned. An ALIAS was still maintained for the derived "Monthly_Budget" column which is required.

Remember that Views simplify the query process as well and eliminate the need for a user to know how to join muliple tables in SQL. A pre-join view is created below.

```
CREATE VIEW DeptName_v AS

SELECT      First_Name, Last_Name, Dept_Name
FROM        Employee_Table e, Department_Table d
WHERE       e.dept_no = d.dept_no;
```

Once the view has been created, the next step is to SELECT from the view and examine the result set. This can be accomplished by doing the following:

Command: **SELECT * FROM DeptName_v;**

First_name	Last_Name	Dept_name
Billy	Coffing	Research and Develop
John	Smith	Research and Develop
Herbert	Harrison	Customer Support
William	Reilly	Customer Support
Cletus	Strickling	Customer Support
Mandee	Chambers	Marketing
Loraine	Larkins	Sales

Views that join two or more tables can prove to be invaluable for providing easy access to data that is important to users. The result set reveals all the employees that have been assigned to departments. Notice that the WHERE clause establishes a common domain between the tables. That commonality is the dept_no.

When creating VIEWS in Teradata, the following must be considered:

- **Views cannot have indexes** (indexes are established on the underlying tables)
- Column names must use valid characters
- **Aggregates** and **Derived data** must have an **ALIAS**
- Since rows are not in order in views or tables, views **cannot have** an **ORDER BY** clause

How to Change a VIEW Using REPLACE

Technically views are **replaced** and **not modified.** This is done with the **REPLACE VIEW command.** When this command is executed, the named view is replaced with the new SELECT statement(s). The key to remember here is if the view name does NOT exist then a new view will be created.

A really good way to ensure that you are updating the right view is to issue a SHOW VIEW command. Then the user can copy and paste the information into an editor, and make changes to the SELECT portion of the statement. The next step is to change the word CREATE to REPLACE, and then run the statement. This will ensure that the correct view is being replaced.

The syntax to replace a VIEW is as follows:

```
REPLACE VIEW [<database-name>.]<view-name> AS

SELECT <column-name>
            [, <column-name> ]
            [, <column-name> ]
FROM     <table-name>

[ WHERE <condition-tests> ]
[WITH CHECK OPTION];
```

Now that we know the REPLACE syntax, the next step is to go through an example. The view that needs to be updated is the Dept_Name_v. The first step is to conduct a SHOW command to display the current view definition. As you can see, we are taking our own advice!

Command: **SHOW VIEW DeptName_v;**

```
CREATE VIEW DeptName_v AS

SELECT    First_Name, Last_Name, Dept_Name
FROM       Employee_Table e, Department_Table d

WHERE     e.dept_no = d.dept_no;
```

Once we have the View Create Statement, we can copy and paste the current definition into an editor such as Queryman and make the changes to the SELECT statements(s). Remember that you need to change the word CREATE to REPLACE as illustrated below.

```
REPLACE VIEW DeptName_v AS

 SELECT      First_Name, Last_Name, Dept_Name
 FROM        Employee_Table e, Department_Table d
 WHERE       e.dept_no = d.dept_no
 AND         dept_name = 'Sales';
```

Once the view has been recreated, the next step is to SELECT from the view and evaluate the result set. This can be accomplished by doing the following:

Command: **SELECT * FROM DeptName_v ;**

```
First_name Last_Name     Dept_name
Loraine    Larkins       Sales
```

Notice the result set. The change to the view has eliminated all departments except for Sales. This demonstrates the flexibility and power of views for users.

How to Drop a VIEW

"Good teaching is one-fourth preparation and three-fourths theater"

– Gail Godwin

If your Teradata training is not fun and exciting drop your training vendor. If a view is not providing value drop the view. A View can be dropped very easily with one simple command; DROP. This command is shown in the example below.

```
DROP VIEW [<database-name>.]<view-name>;
```

Here's how we would drop the Dept_Name _v:

```
DROP VIEW DeptName_v ;
```

This command appears to be very simple, but that is not the case. Remember that View definitions are stored in the Data Dictionary for reuse. When you DROP a VIEW, you effectively remove the view from the Data Dictionary.

View Aggregation and Nested Views

Utilizing aggregation within a view is an excellent technique. The quandary is whether or not it is better to access a table that stores the aggregation or use a view that performs the aggregation. It is most often best to use a **view that performs the aggregation** because the final result is always **current** with the data in the database **tables**. This technique provides better integrity and can make **application coding requirements** simpler for **users**. Some aggregations are pretty straight forward, however some calculations are extremely complicated. Aggregation views can **simplify maintenance** for many complicated and sensitive calculations. Even better, the calculations are **NOT** effected if **new columns** are **added** to the actual base table.

Below is an example of an Aggregation View.

```
Create VIEW Agg_View_Tera_TOM AS

SELECT Dept_no
           ,Average(salary) as AVGSAL
FROM     Employee_Table
Group by Dept_no;
```

Notice in our example that we have given the aggregation an alias name. You must do this or the create of the view will receive an error.

A **nested view** is a view that calls another view. Teradata SQL has a 1MB limit, but **nested views** can be fully expanded to 2MB. Teradata will **validate** nested view privileges at **execution time** on the user executing the view.

All About Macros

"You're never too old to become younger."

– Mae West (1892 – 1980)

If you want to become younger and simplify your life then learn to create macros. Mae West must have been a Teradata natural because macros will make queries draw fast.

Macros simplify the day-to-day routine driven tasks. Macros are typically utilized by DBAs, but users do get involved with these commands as well. Macros provide the mechanism to store SQL. This allows for the SQL to be reused whenever necessary. Because it is stored inside Teradata, this eliminates the need for users to retype the SQL when the code is needed thus reducing the amount of manual activity to conduct these daily commands. A couple of key facts about macros are as follows:

- Multiple users may share them.
- They can be secured from user access.
- All statements are processed as one transaction – if one fails all fail.
- They can receive and use parameters.

As with views, the CREATE, REPLACE and DROP commands are utilized with Macros. The next thing is to investigate how Macros get created. So let's start, shall we?

Creating a MACRO

When creating Macros, there are a few things to remember. The first thing is that each SQL statement within a Macro must have a semi-colon. The second thing is that all SQL must be enclosed in parentheses.

The syntax for creating a MACRO is as follows:

```
CREATE MACRO <macro-name> AS

            ( [INSERT …;]
              [UPDATE …;]
              [DELETE …;]
              [SELECT …;] ) ;
```

An example of a Create Macro statement would be as follows. Pay close attention to the syntax in regards to the parentheses and semi-colons (;).

```
CREATE MACRO Raisemac AS

        ( SELECT * from Employee_Table;
          UPDATE Employee_Table
          SET   Salary = Salary * 1.5;
          SELECT * from Employee_Table; ) ;
```

The above macro when executed will display the rows and columns in Employee_ Table. The main goal of this Macro is to give everyone a 50% raise. I know this usually does not occur in the real world, but this illustrates the power of the Macro. Back to the macro, make sure when creating a macro, be aware of the placement of semi-colons at the end of the CREATE statement as well as the parentheses.

Macros that Use Parameters

The above Macro example displays a simple way to create a Macro. But what if we need to pass along some input to the Macro before it is executed? The good news is that Macros can be parameterized. This way a value can be set each time the user runs the macro. In order to take advantage of this feature, the MACRO must be created to accept values. So how do we do this? I am sure you may have guessed by now, but read on to find the solution.

To create a Macro that will accept a parameter(s) is as follows:

```
CREATE MACRO <macro-name> (<parameter-name> <data-type> [, …] )AS
          ( [INSERT …;]
            [UPDATE …;]
            [DELETE …;]
            [SELECT …;]
 WHERE <column-name> = :<parameter-name> ) ;
```

When defining a parameter in the macro, parentheses must be used. The parameter name consists of a user-assigned variable and the data type that is recognized by Teradata (e.g., Interger). Once defined, the variable name can be substituted throughout the macro.

In order for the optimizer to recognize the parameter name, it must be preceded by a colon (:) in the WHERE clause as displayed in the syntax above. This lets the PE know that the parameter-name is a variable and not an actual column in a table. Keep in mind that Macros may contain more than one parameter. However, with each added parameter defined, there must be a value assigned to it at execution time.

An example of a Create Macro statement would be as follows. Pay close attention to the syntax in regards to the parentheses and semi-colons (;).

```
CREATE MACRO  ViewDept ( invalue1  INTEGER )  AS

        ( SELECT      *
          FROM        Employee_Table
          WHERE       Dept_no = :invalue1; ) ;
```

The created macro is called ViewDept. The Macro has been created to receive a parameter as (invalue1) with a data type specified (INTEGER). When executed, input will be specified that will hold a department number (Dept_no=400) through out the macro execution. This macro goal is to display all columns for Employees in the department specified.

Changing a MACRO Using REPLACE

Technically Macros are replaced and not modified. This is done with the REPLACE MACRO command. When this command is executed, the named macro is replaced with the new parameters. The key to remember here is if the macro name does NOT exist, then a new macro will be created.

A really good way to ensure that you are updating the right macro is to issue a SHOW Macro command. Once this is done, the user can copy and paste the information into an editor such as Queryman and make changes to the macro. The next step is to change the word CREATE to REPLACE, and then run the statement. This will ensure that the correct macro is being replaced.

The syntax to replace a Macro is as follows:

```
REPLACE MACRO <macro-name> AS
          ( [INSERT …;]
            [UPDATE …;]
            [DELETE …;]
            [SELECT …;] ) ;
```

Now that we know the REPLACE syntax, the next step is to go through an example. The macro that needs to be updated is the Raismac. The first step is to conduct a SHOW command to display the current macro definition. As you can see we are taking our own advice!

Command: **SHOW MACRO Raisemac;**

```
SHOW MACRO Raisemac AS

         (SELECT * from Employee_Table;
          UPDATE Employee_Table
          SET   Salary = Salary * 1.5;
          SELECT * from Employee_Table; ) ;
```

Once we have the MACRO Create Statement, we can copy and paste the current definition into an editor such as Queryman and make the changes to the parameters. Remember that you need to change the word CREATE to REPLACE as illustrated below.

```
REPLACE MACRO Raisemac (deptnum INTEGER) AS

     ( SELECT * from Employee_Table;
        UPDATE Employee_Table
        SET  Salary = Salary * 1.5
       WHERE Dept_no = :deptnum;
       SELECT * from Employee_Table; );
```

The re-created macro is called Raisemac. The Macro has been created to receive a parameter as (invalue1) with a data type specified (INTEGER). When executed, input will be specified that will hold a department number (Dept_no=400) through out the macro execution. This macro goal is to display all columns for Employees in the department specified.

Once the Macro has been recreated, the next step is to execute the newly recreated Macro.

How to Execute a MACRO

Now that we have gained valuable knowledge on how to CREATE and REPLACE a MACRO, one question still remains – how is it executed? This next section will demonstrate how to EXECUTE a MACRO. In order for the MACRO to be executed, the EXECUTE (EXEC) command is utilized.

The syntax for executing a MACRO is as follows:

```
EXECUTE <macro-name> [(<parameter-value-list>)];

  or

EXEC <macro-name> [(parameter-value-list>)];
```

If the macro expects a parameter (parameter-value-list) to be included during the executed statement, then the values must be enclosed in parentheses following the macro name. The paramenter-name is optional, but can be listed in the parameter list along with their corresponding parameter. If multiple parameters are required and the parameter-value-names have been listed before MACRO execution (see example below), then the parameters values do not require a specific order. If the parameter-value-names are not listed, then the variables need to be listed in order in which they were declared in the CREATE MACRO statement.

The command to execute a Macro with NO Paramters is as follows:

Command: **EXEC Raisemac;**

The command to executed a MACRO with parameters-names is a follows:

Command #1: **EXEC Raisemac (300);**
Command #2: **EXEC ViewDept (invalue1=400);**
Command #3: **EXEC Samp_macro (34, 'Work');**

Keep in mind that if the parameters are missing when a Macro is executed, a NULL will be inserted for the missing value.

How to Drop a MACRO

A Macro can be dropped very easily with one simple command; DROP. This command is shown in the example below.

```
DROP MACRO <macro-name> ;
```

Here's how we would drop the Raisemac Macro:

```
DROP MACRO Raisemac;
```

This command appears to be very simple, but that is not the case. Remember that Macro definitions are stored in the Data Dictionary for reuse. When you DROP a MACRO, you effectively remove the Macro from the Data Dictionary. If the MACRO contains parameters, they are not needed when removing a MACRO from the system.

It is good practice to save the CREATE syntax for a particular MACRO prior to removing it from the system. This will ensure if the MACRO is removed by mistake, it can easily be recreated. Remember, in order to obtain the CREATE syntax use the SHOW MACRO statement.

Chapter 14 - Locks

"Some birds aren't meant to be caged, their feathers are just too bright. And when they fly away, the part of you that knows it was a sin to lock them up, does rejoice."

– Shawshank Redemption

You don't lock up a bird, but you always lock a query. Teradata uses a lock manager to automatically lock at the **database, table** or **row hash level**. Teradata will lock objects using four types of locks:

Exclusive - Exclusive locks are placed only on a database or table when the object is going through a structural change. An Exclusive lock restricts access to the object by any other user. This lock can also be explicitly placed using the LOCKING modifier.

Write - A Write lock happens on an INSERT, DELETE, or UPDATE request. A Write lock restricts access by other users. The only exception is for users reading data who are not concerned with data consistency and override the applied lock by specifying an Access lock. This lock can also be explicitly placed using the LOCKING modifier.

Read - This is placed in response to a SELECT request. A Read lock restricts access by users who require Exclusive or Write locks. This lock can also be explicitly placed using the LOCKING modifier.

Access - Placed in response to a user-defined LOCKING FOR ACCESS phrase. An ACCESS lock permits the user to READ an object that may already be locked for READ or WRITE. An access lock does not restrict access by another user except when an Exclusive lock is required. A user requesting ACCESS cannot be concerned with data consistency.

When Teradata locks a resource for a user, the **lifespan** of the transaction lock is **forever** or **until the user releases the lock**.

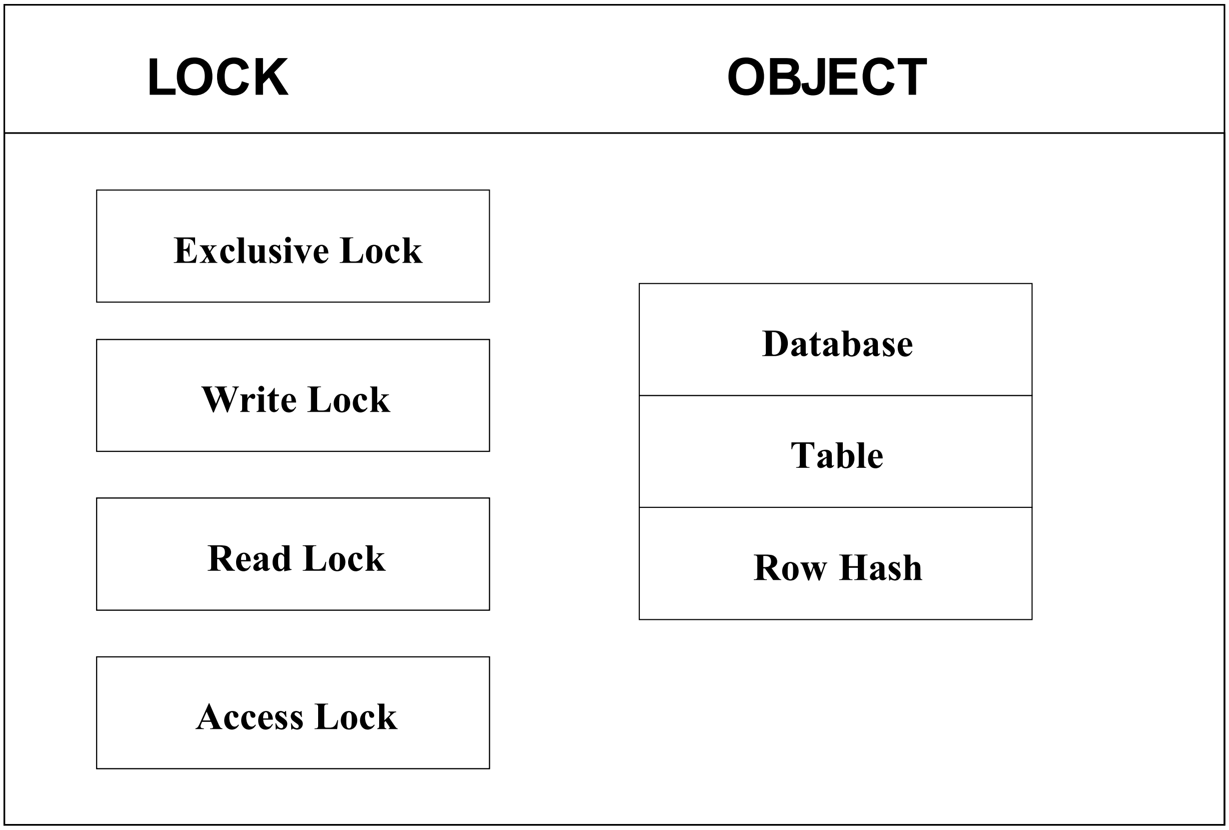
This is different then a **deadlock situation**. If two transactions are **deadlocked** the **youngest** query is always aborted.

Teradata has Four locks at Three levels

"We do not remember days; we remember moments"

– Anonymous

Teradata does not remember days; Teradata remembers data. Teradata has four locks that will lock objects at the Database, Table, or Row Hash Level to protect data integrity.



LOCK

OBJECT

Exclusive Lock

Write Lock

Read Lock

Access Lock

Database

Table

Row Hash

Locks and their compatibility

Locks that are compatible with one another can share access to objects simultaneously. For example it does not matter if one or a thousand users want to read an object. They can all do so at the same time, but Teradata will not allow a user to change a table while others are reading it. This prevents database corruption.

Teradata Lock	Compatible Locks
Exclusive Lock	No Compatibility
Write Lock	Access Lock
Read Lock	Read Lock Access Lock
Access Lock	Read Lock Write Lock Access Lock

An ACCESS Lock is an excellent way to **avoid waiting** for a **write lock** currently on a particular **table**. Two statements allow this:

- **Locking Row for Access**
- **Locking Tablename for Access**

How Teradata Locks Objects

Because tables are spread across all AMPs Teradata must lock a table on every AMP. If your system contains 1,000 AMPs then Teradata must essentially lock 1,000 times. That is because Teradata must lock the table on each AMP. This can be confusing so Teradata ingeniously has come up with a way to do this efficiently and effective.

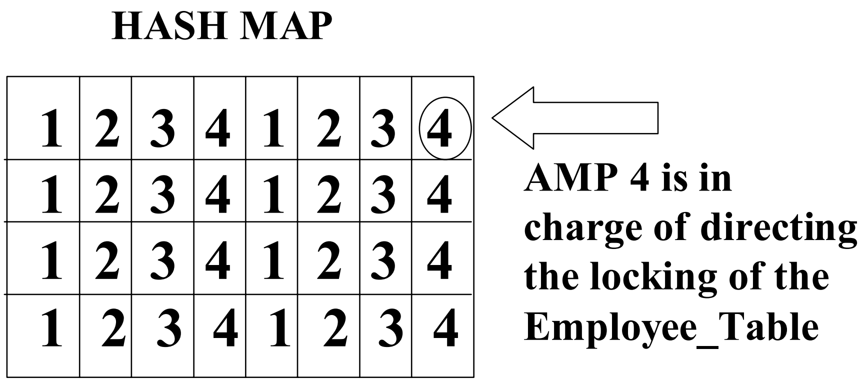
Teradata assigns one AMP per table to direct the locking on all AMPs for that particular table. How does Teradata decide which AMP will lock a particular table? Teradata hashes the table name and a single AMP is selected from the hash map. This ensures that no one AMP is overloaded with locking responsibility. If your system has 1,000 AMPs and 1,000 tables then each AMP will be responsible for locking one table.

If you look at an explain you will often see the phrase, **"Locking a Pseudo Table"**. That means that one AMP is responsible for telling the other AMPs to lock the table. A **Pseudo** table lock is designed to be a **sequential locking** of a table **resource** to avoid two users being **deadlocked** on a table level lock request.

SELECT * from Employee_Table;

HASH → Employee_Table

Row Hash points to a bucket in the Hash Map



HASH MAP

1	2	3	4	1	2	3	4
1	2	3	4	1	2	3	4
1	2	3	4	1	2	3	4
1	2	3	4	1	2	3	4

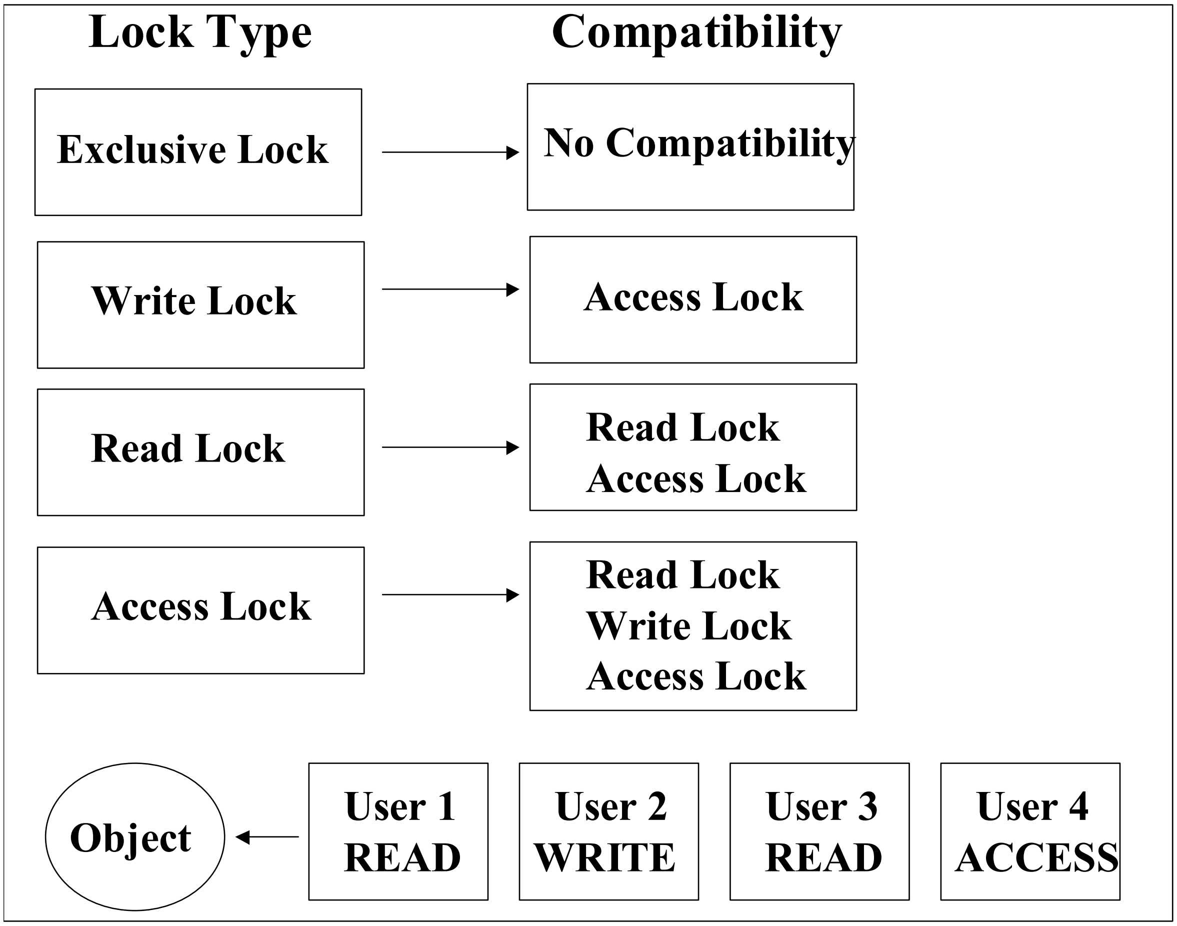
AMP 4 is in charge of directing the locking of the Employee_Table

That is what EXPLAIN means when it states the phrase, "Locking a Pseudo Table"

Teradata Locks – First Come First Serve

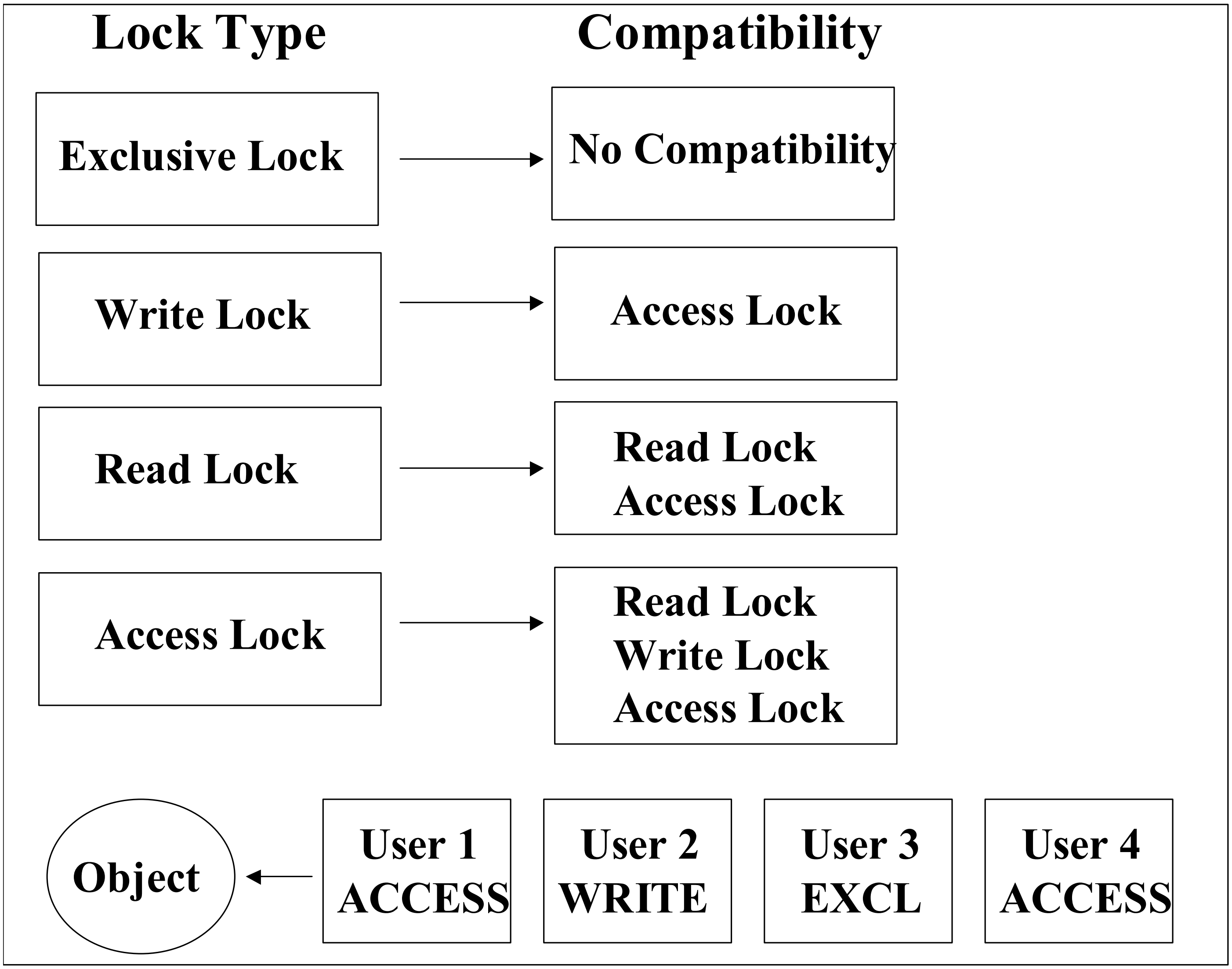
Teradata locks objects on a first come first serve basis. The first user to request an object is the first to lock the object. Teradata will place other users who are accessing the same object in a locking queue. Just as you might stand in line to get a movie ticket Teradata forms a line for users who want to access an object. The only difference is that Teradata allows a user to move up the line if their lock is compatible with the lock in front of them.

This does not mean that a lock in the very back of the line can move right to the front if they are compatible with the first lock. It means that a lock can move up the line – one lock at a time – as long as it is compatible with the lock just in front of it.



USER 1 is first in line and READS the object
USER 2 must wait on USER 1
USER 3 must wait on USER 2
USER 4 moves past USER 3, then USER 2, and simultaneously reads the object with USER 1.

Locking Queue Example 2



Lock Type

Compatibility

Exclusive Lock → No Compatibility

Write Lock → Access Lock

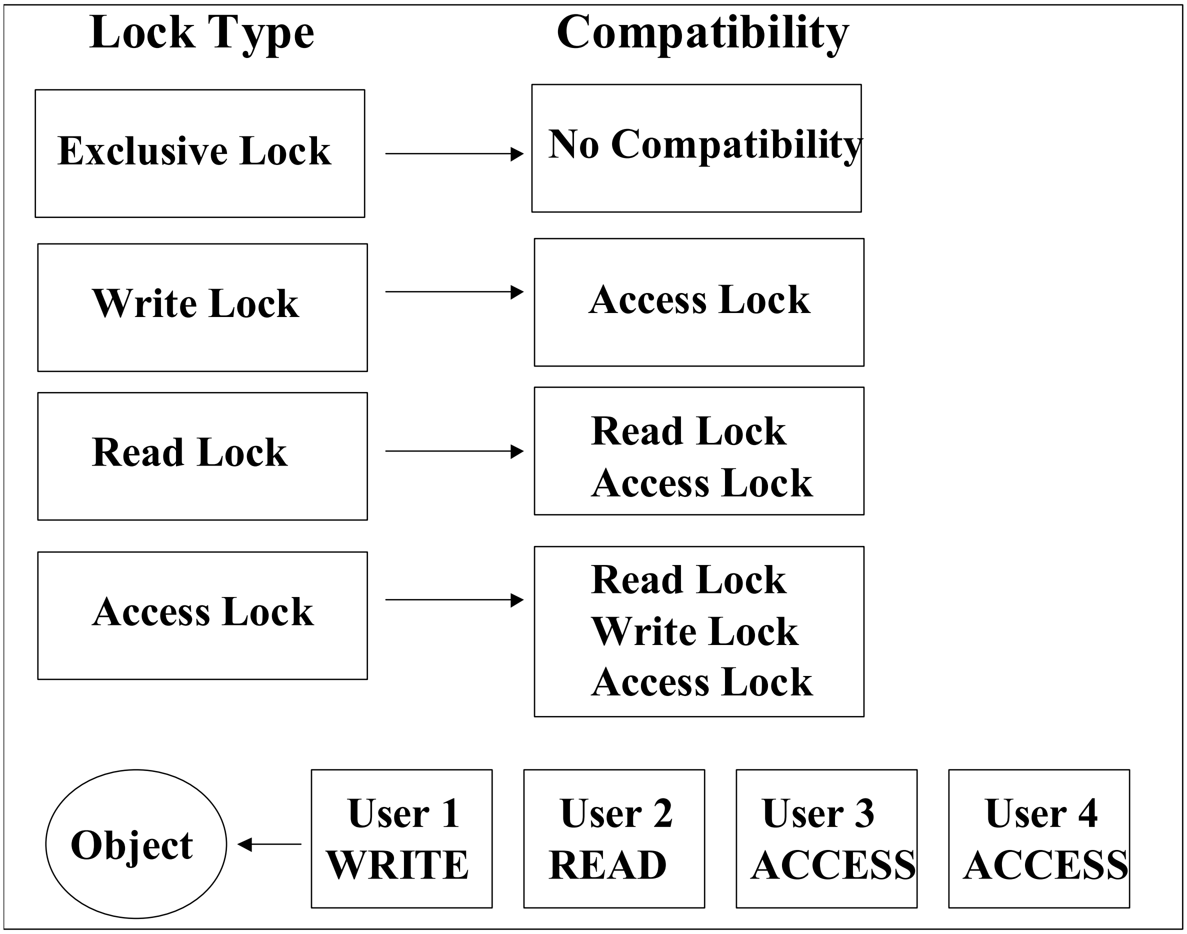
Read Lock → Read Lock
Access Lock

Access Lock → Read Lock
Write Lock
Access Lock

Object ← User 1 ACCESS | User 2 WRITE | User 3 EXCL | User 4 ACCESS

USER 1 READS the Object immediately
USER 2 is compatible and WRITES on the object also
USER 3 must wait on both USER 1 and USER 2
USER 4 must wait until USER 3 is done. It is not compatible with the EXCL and can't move up.

Locking Queue Example 3



Lock Type	Compatibility
Exclusive Lock	No Compatibility
Write Lock	Access Lock
Read Lock	Read Lock Access Lock
Access Lock	Read Lock Write Lock Access Lock

USER 1 WRITES the Object immediately
USER 2 waits until USER 1 is finished
USER 3 moves past USER 2 and ACCESSES the object
USER 4 moves with USER 3 and moves past USER 2 and then ACCESSES the object. USER 2 will READ the object when and only when USER 1 releases the WRITE lock.

Locking Modifier

Teradata locks objects automatically based on the user's SQL. If the user is performing a SELECT, a READ lock is placed on the object. If the user is performing an UPDATE then a WRITE lock is placed on the object. If the user is ALTERING a table then an EXCLUSIVE lock is placed on the table.

Users can however place locks themselves and this is called a Locking Modifier. Locking Modifiers are usually used for ACCESS LOCKS.

When users decide they want to read data and they don't want to wait on any WRITE LOCKS then they can specifically state in their SQL that they want to LOCK the TABLE for ACCESS. If this statement is in their SQL like on the example on the following page the SQL will not have to wait on any WRITE locks. The user should only request an ACCESS LOCK when they don't need their data to be 100% exact. For example, a user may need to see the average order total placed by customers in their region. If they need the information because they are curious they might want to place an ACCESS Lock on it. If the information is not perfect because one of the orders was in the middle of an UPDATE it probably won't dramatically affect the total. If however, these totals are being summed up for the Internal Revenue Service (IRS) then the user may not want to place the ACCESS LOCK.

Any lock may be **upgraded** by a user, but only a **READ lock** can be **downgraded** to an **ACCESS** Lock. To upgrade a READ lock to an Exclusive lock you must have **DROP Table privileges**.

```
Locking Employee_Table for ACCESS
SELECT     Emp_No
           ,Dept_No
           ,Last_name
           ,Salary
FROM       Employee_Table
ORDER BY 4;
```

A Locking Modifier allows a user to specify in their SQL what type of lock to place on one or more objects.

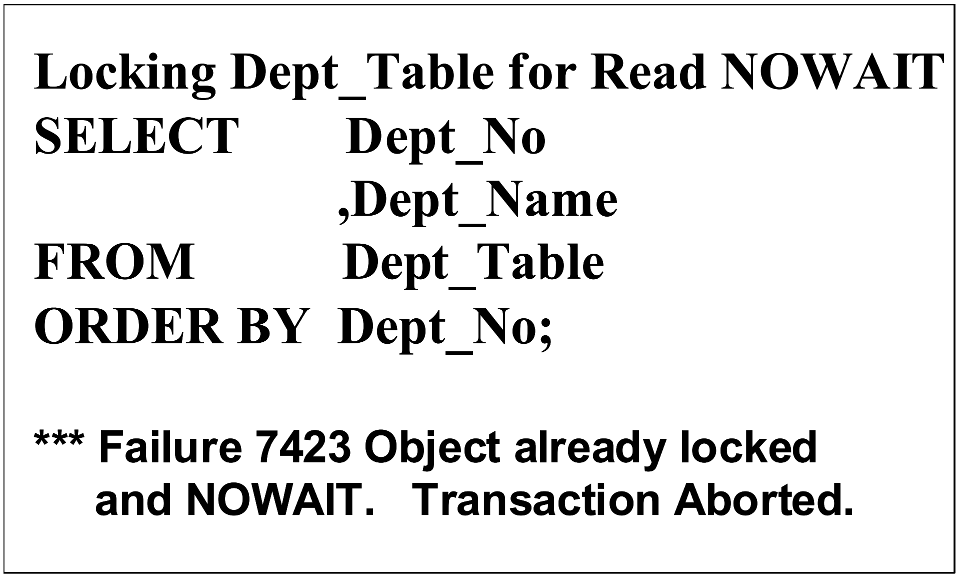
The NOWAIT Option

When the **NOWAIT** is used, if a lock request **cannot** be responded to immediately the transaction **will abort**. The NOWAIT option is used when it is not desirable to have a request wait for resources, or cause resources to be tied up while waiting. The **NOWAIT** option is an excellent way to **avoid** waiting on conflicting locks.

Don't make the mistake of thinking the NOWAIT option means you have a free dash to the front of the lock line.

The NOWAIT option dictates that a transaction is to ABORT immediately if the LOCK MANAGER cannot immediately place the lock.

Use a LOCKING modifier with the NOWAIT option when you don't want requests waiting in the queue. A 7423 return code informs the user that the lock could not be placed due to an existing, conflicting, lock.



```
Locking Dept_Table for Read NOWAIT
SELECT       Dept_No
            ,Dept_Name
FROM         Dept_Table
ORDER BY     Dept_No;

*** Failure 7423 Object already locked
    and NOWAIT.   Transaction Aborted.
```

Chapter 15 - Collect Statistics

"Measure a thousand times – cut once"

– Turkish Proverb

Teradata Collects Statistics by measuring thousands of rows and then placing the statistics in USER DBC. Collecting Statistics is extremely important to the Optimizer. It helps the optimizer predict the **size of spool files** and the **number of occurrences** of a value. Collecting Statistics helps the optimizer determine the **best join types** and **sequences** also. It is also important because **statistics** are used to estimate the **cardinalities** of spool files, and must be utilized if the Optimizer wants to enable **NUSI Bitmapping**.

Teradata allows you to measure your data even better than before with limit increases and clever flexibility in the COLLECT STATISTICS category. **Collect statistics** will now allow you to **collect** using a **sample** of the table.

The number of Columns on a table that you can COLLECT STATISTICS has gone from 40 to **512 columns** and **indexes** per table.

You can still collect at either the column or index level, but **multi-column collection** can now be done at the **column level** without **creating** an **index**. You should usually collect at the column level because indexes have a tendency to get dropped and recreated quite often.

Teradata has made some new changes to collect statistics and kept some things the same. For example, you still need at least **one proper privilege** on an **object** to be allowed to **collect statistics**. Statistics can be collected on **global temporary tables**, but remember that these tables delete the data when users log off of session so collecting at the right time is important. You can **view statistics** with the **Help Statistics Command** or utilize the **stats module** of **Teradata Manager.**

COLLECT STATISTICS at the INDEX LEVEL when:

> You have a multi-column index in which the columns are frequently used together in an equality WHERE condition or JOIN condition.

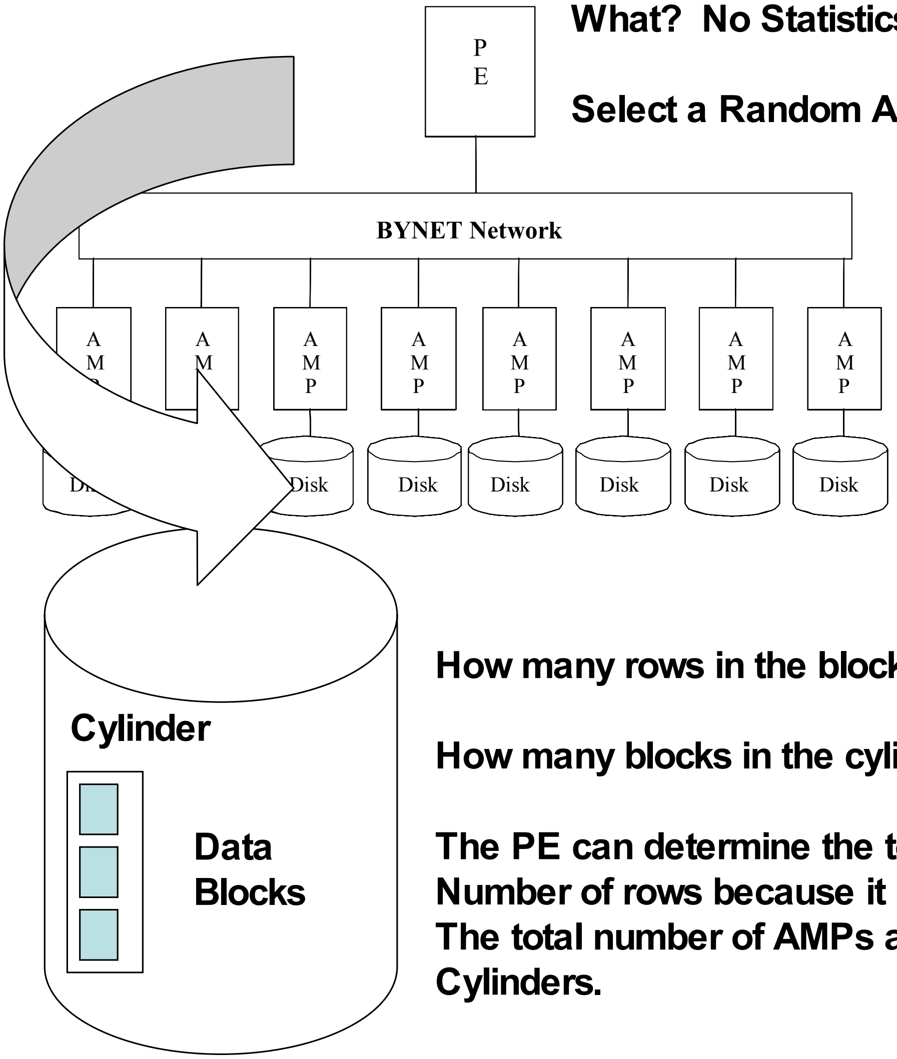
COLLECT STATISTICS at the COLUMN LEVEL for INDEXES when:

> You have a multi-column index in which the columns are NOT used together frequently for an equality WHERE condition or JOIN condition.

Dynamic AMP Sampling

When statistics are not collected on a table, the optimizer will state in the explain that it has **low confidence**. This is because it is not confident with exactly what is going on with the table. To come up with a plan, Teradata must perform **dynamic AMP sampling**. To perform dynamic sampling, the optimizer will choose an AMP to **represent** an **estimation** of **population demographics**. It will instruct the AMP to choose a **random block** from a **random cylinder** and **count the rows** in that **cylinder**.

The optimizer knows **how many AMPs** are in the configuration, the **number of cylinders**, the **number of data blocks** inside a **cylinder**, and the **number of rows per block** in the **sampled** cylinder. It then does the math and estimates (**with low confidence**) the statistics to come up with a plan.



What? No Statistics?!

Select a Random AMP

How many rows in the block?

How many blocks in the cylinder?

The PE can determine the total Number of rows because it knows The total number of AMPs and Cylinders.

How Collect Statistics Works

COLLECT STATISTICS is remarkable. Here is how it works. Lets say you have an employee table with 1,000,000 employees. You decide to collect statistics on the column Last_name.

The AMPs will now either do a full table scan or a percentage COLLECT STATISTICS based on which option you choose.

The system will sort the last_name values in the employee table and place the names in 100 intervals. Because the last_name values are sorted and statistics are gathered about all 100 intervals the Parsing Engine can estimate the rows it will need to have the AMPs retrieve.

Interval 1	Interval 2	Interval 3	Interval 4
Arden	Brubaker	Davis	Grace
Allen	Bell	Davis	Fryer
55	40	90	44
100	120	100	75
800	790	900	750

● ●

Statistics Kept in interval

Highest Value in Interval:
Most Frequent Value in Interval:
Rows with Most Frequent Value:
Other Values in the Interval:
Number of Rows of Other Values:

COLLECT STATISTICS sorts the values and places them in 100 different intervals. The PE can interrogate the intervals to Estimate ranges of values as well as frequent values.

Sample Statistics

"A true DBA magician is someone who pulls habits out of Stats"

– Tera-Tom Coffing

Teradata can now predict the habits of data by collecting only a sample of the statistics. COLLECT STATISTICS can be quite time consuming to the system so Teradata has invented the ability to COLLECT STATISTICS USING SAMPLE. It is no magic trick!

The system will collect on only a percentage of the data. This saves time and resources, but you could pay a price in accuracy. The better the data is distributed the better the sample will work. You should consider sampling your COLLECT STATISTICS when:

Collecting statistics on very large tables
You need to COLLECT STATISTICS but it is in the middle of the day

Sampling will generally be better than not collecting statistics.

Remember these fundamentals:

System determines appropriate sample size to generate accurate statistics.

Statistics on a column or index are either sampled or full, but not both. The most recent COLLECT STATISTICS determines if it is sampled or full.

No not use sampling on highly skewed data.

Here is an example of the SQL:
COLLECT STATISTICS using sample on Employee_table Column social_security;

What You Should COLLECT STATISTICS On

"Diplomacy is the art of saying "Nice Doggie" until you can find a rock."

– Will Rogers

If you want to find yourself between a rock and a hard place then don't collect statistics.

Here are some excellent guidelines on what you should collect statistics on:

- **All Non-Unique indices**
- **Non-index join columns**
- **The Primary Index of small tables**
- **Primary Index of a Join Index**
- Secondary Indices defined on any join index
- Join index columns that frequently appear on any additional join index columns that frequently appear in WHERE search conditions
- Columns that frequently appear in **WHERE search conditions** or in the **WHERE clause of joins**.

The two key words to collect and drop statistics are:

COLLECT STATISTICS
DROP STATISTICS

Release V2R5 allows the collection of statistics on multiple non-indexed columns in a table. This feature enhances the ability of the optimizer to better estimate the number of qualifying rows based on the use of these columns in the WHERE clause. Prior to V2R5 you can only collect statistics on a single column in a single COLLECT STATISTICS command.

COLLECT STATISTICS DETAILED SYNTAX

Here is the syntax to collect statistics for V2R5:

```
COLLECT STATISTICS  [ USING SAMPLE ]
      ON [ TEMPORARY ]  { <table-name> | <join-index-name> | <hash-index-name> }
 [ COLUMN { <column-name> | (<column-list>) }
   | [ UNIQUE ] INDEX { <index-name>  [ ALL ]  | (<column-list>) }
                             [ ORDER BY {  HASH | VALUES } [ <column-name> ] ]      ]
;
```

or CREATE INDEX Style

```
COLLECT STATISTICS  [ USING SAMPLE ]
 [ COLUMN { <column-name> | (<column-list>) }
   | [ UNIQUE ] INDEX { <index-name>  [ ALL ]  | (<column-list>) }
                             [ ORDER BY {  HASH | VALUES } [ <column-name> ] ]      ]
ON [ TEMPORARY ]  { <table-name> | <join-index-name> | <hash-index-name> }
;
```

Note: 1. STATISTICS may be abbreviated as STATS
2. USING SAMPLE, new in V2R5, does not do a full table scan, so it is faster, but less accurate
3. INDEX specifications
ALL –used when ALL used in CREATE INDEX or for NUSIs on join index
ORDER BY - when VALUES used in CREATE INDEX
4. COLUMN prior to V2R5 allowed only a single column

COLLECT STATISTICS Examples

Here is an example of collecting statistics on the columns dept and emp_no and the multi-column index of lname, fname columns in the employee table (V2R4).

```
COLLECT STATISTICS on TomC.Employee column dept;
COLLECT STATISTICS on TomC.Employee column emp_no;
COLLECT STATISTICS on TomC.Employee Index(lname, fname);
```

The above commands still work, however in V2R5 they can be run as:

```
COLLECT STATISTICS column (dept,emp_no) on TomC.Employee;
COLLECT STATISTICS Index(lname, fname) on TomC.Employee;
```

Notice that with V2R5, you can now collect statistics on more than one column in a single statement. This saves time because the statistics for all columns are collected in a single pass of the file, no longer one pass per column. Therefore, this type of statistics collection will be faster. Additionally, only in V2R5 can you get a sample.

The next example of COLLECTING STATISTICS at the table level is for V2R5 and later:

```
COLLECT STATISTICS USING SAMPLE  column lname on TomC.Employee;
```

You will always COLLECT STATISTICS on a column or index one at a time initially. You must use the COLLECT STATISTICS command for each column or index you want to collect in a table. In the above examples, we collected statistics on the column dept and the index(lname, fname). **You can collect statistics at either the column or index level.** It is best to COLLECT STATISTICS at the column level unless you are dealing with a **multi-column index**. COLLECT at the index level only for indices that are multicolumn indices. Otherwise collect columns and single column indices at the column level. Single column indices actually perform the same COLLECT STATISTICS functions as if they were collected at the column level. Plus, if you drop an index, you lose the statistics.

The table Employee now has COLLECTED STATISTICS defined within the table. Although you must collect statistics the first time at the column or index level you only collect statistics at the TABLE LEVEL for all refreshing of STATISTICS. Here is an example of COLLECTING STATISTICS at the table level (V2R5 and prior).

```
COLLECT STATISTICS on TomC.Employee;
```

The system will refresh the COLLECT STATISTICS on the columns and indices it had previously collected on the table. You cannot request sampling at the table level, only at the column or index request.

Chapter 16 - MISC

Identity Columns

"When I was a boy I was told that anybody could become President; I'm beginning to believe it."

– Clarence Darrow

A data warehouse is not made by God, but instead a product of evolution. Not even Clarence Darrow could defend a data warehouse that has no Identity Columns.

Identity Columns generate a **system-generated number for every row** as it is **inserted** in the table. It is also knows as a DBS Generated Unique Primary Index. Identity Columns are used if you want to guarantee row uniqueness in a table and they can **simplify the initial port** from databases other than Teradata. They are often used as the Primary Index so they can guarantee even distribution. In some cases they are utilized to optimize and simplify the first port from another database that uses generated keys.

Identity columns do NOT need to be part of the **Primary Index**, but they are often used for that purpose because they **provide even data distribution**. The **biggest rule** is that you can **only specify one Identity Column per table**.

Identity Keys are ANSI Compliant and **work for:**

Single Inserts
Multi-session concurrent insert requests
INSERT/SELECTs
BTEQ Import
TPump

They do not work for:

MultiLoad or FastLoad INSERTs
Part of a composite Primary or composite secondary index
Temporary or Volatile tables
Part of a Join Index, Hash Index, PPI or Value-Ordered index
Atomic UPSERTs on a table with the Identity Column as the Primary Index
ALTER TABLE statements that add an Identity Column to an existing table

Identity Columns Example

Identity Columns are often used as the Primary Index, but should only be defined for the Primary Index if it is the primary path to the table. You might also consider using Identity Columns as the Primary Index if you don't have a column or columns that will provide enough uniqueness for even distribution.

```
Create Table Employee_Table
( Employee_No INTEGER
          GENERATED ALWAYS AS IDENTITY
          (START WITH 1 INCREMENT BY 5
           MAXVALUE 1000000
           NO CYCLE)
, Dept_No        INTEGER
,First_Name      CHAR(20)
,Last_Name       CHAR(20)
,Salary          DECIMAL (10,2)
) PRIMARY INDEX (Employee_No);
```

Identity Columns are implemented in the CREATE TABLE Statement

Data Types can be any numeric type

GENERATED ALWAYS will always generate a value

GENERATED BY DEFAULT generates a value only when no value is specified

CYCLE restarts numbering after the maximum/minimum number is generated

GENERATED ALWAYS + NO CYCLE implies uniqueness

Exact incrementing is NOT Guaranteed

Generated Numbers don't necessarily reflect row insertion sequence

Uniqueness is guaranteed

LIKE Clause

The LIKE Command **searches character data strings** and can **use wildcards** to find data that is similar. For example, a user can use the LIKE command to request rows where the last_name starts with ‘SM’. **LIKE** and **SUBSTRING** are great for **decomposable** data.

```
SELECT  <column-list>
FROM    <table-name>
WHERE <column-name> LIKE  ‘[<wildcard(s)>]<search-string>[<wildcard(s)>]’;
```

The wildcard characters are:

Wildcard symbol	What it does
_ (underscore)	matches any single character, but a character must be present
% (percent sign)	matches any single character, a series of characters or the absence of characters

```
SELECT Course_Name, Course_Id FROM Course_Table
WHERE Course_Name LIKE ‘%SQL’;
```

English: Select two columns from the rows in the course table if Course_Name has SQL at the end of the string.

```
SELECT * FROM Student_Table
WHERE Last_Name LIKE ‘_m%’;
```

English: Select all columns from the rows in the Student table if Last_Name has an m as the second character.

Hint - Man

LIKE works on Character Data. If I have a name that is CHAR(10) and a name such as Smith then the system sees the name as ‘Smith   ‘. Notice the 5 extra spaces. You can use the TRIM function with the LIKE function to find anyone whose name ended in h.

```
SELECT * FROM Student_Table
WHERE TRIM(Last_Name) LIKE ‘% h’;
```

Issues: The LIKE statement is not returning all the rows expected to be in the result set. More than likely you are dealing with Character strings of CHAR data type. Remember ‘Smith’ in the eyes of Teradata is ‘Smith+ empty space’. This empty space resides in the rest of the fixed data space on the system.

SUBSTRING and SUBSTR Functions

Both of these functions serve the purpose of returning a **desired section of a character string**. A user chooses the starting position and then how many characters to pull out of the string. The <length> specified must be a **positive number**, otherwise an error will occur.

SELECT **SUBSTRING(**<column-name> **FROM** <start-location> **[FOR <length>])**
FROM <table-name> ;

```
SELECT SUBSTRING(Course_Name FROM 3 FOR 4)
FROM Course_Table ;


SELECT SUBSTRING('Teradata' FROM 2 FOR 5) ;
```

English: The last example will return a result of: erada

SELECT **SUBSTR (** <column-name>, <start-location> [, <length>] **)**
FROM <table-name> **;**

```
SELECT SUBSTR(Course_Name, 3 , 4)
FROM Course_Table ;


SELECT SUBSTR('Teradata', 2 , 5 ) ;
```

English: The last example will return a result of: erada

LIKE and **SUBSTRING** are great for **decomposable** data.

Referential Integrity

"What lies behind us and what lies before us are tiny matters compared to what lies within us."

– Ralph Waldo Emerson

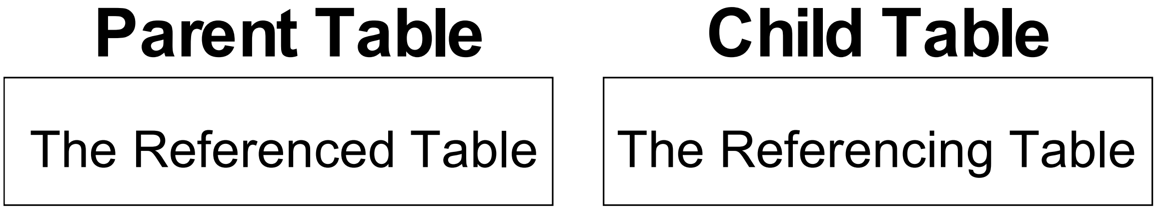
Referential integrity believes that what lies within a table is extremely important. Referential Integrity (RI) is designed to protect referencing data from being deleted by accident. It maintains the integrity and consistency of data in the database by ensuring that data corruption does not occur during Inserts, Updates or Deletes. Certain **Business Applications require Referential Integrity to run**.

Referential Integrity is based upon how **Primary** and **Foreign keys** are defined and **ensures data integrity** and **consistency**. It says that you cannot have a table row with a non-null value for a referencing (FK) column where there is no equal value in its referenced (PK) column. Or, positively, you may designate columns in a referencing table that are FKs in other referenced tables. Because of this rule, **a referenced column** must be a **Primary Key** and must be **Unique** and **Not Null**. That is why a **NUPI cannot** be **referenced** by a referencing column. Even if the NUPI is defined as **NOT NULL** it is still **missing the uniqueness requirement**.

When inserts are made into a referencing table in a non-null FK column, RI will check to be sure that the referenced table has a row with that same value as its Primary key. If **no columns are specified** for the **referenced table** than that table must have a **single column Primary Key** or single column defined with a **UNIQUE constraint**.

Referential Integrity allows up to **64 columns** on a **Foreign Key**. It can be defined with **NO Check Option**. That is covered over the next couple of pages. Referential Integrity actually helps the **PE Optimizer** develop **better** and quicker **query plans**.

The **ALTER Table** statement allows **WITH CHECK Option** and **DROP INCONSISTENT REFERENCES** as Referential Integrity options.



Parent Table

The Referenced Table

Child Table

The Referencing Table

Soft Referential Integrity

Soft Referential Integrity is Referential Integrity that is understood by the PE Optimizer, but not checked by the system. Why would you ever want referential integrity that does not force integrity? Because in today's data warehouse environment more and more joins are being done. Developers are creating more and more Join Views and applications are creating a great deal of joins and in some cases bad joins at that.

Soft Referential Integrity ties tables together in the mind of the Optimizer, but does not have the heavy load of checking each time an insert or an update is done. Soft Referential Integrity will help the Optimizer join tables together without mistakes or redundancy. Real Referential Integrity is set up on tables so things that should not happen can't. For example, a company might set up real referential integrity so nobody could be deleted from the Employee_Table unless they were previously deleted from the Payroll_Table. This would prevent someone from being fired and yet still receiving his or her paychecks by mistake.

FastLoad and MulitLoad can INSERT or UPDATE into a table specified with Soft Referential Integrity. Understand that Soft Referential Integrity is slightly different than real Referential Integrity. With Real Referential Integrity you won't be able to delete someone from the Employee_Table until they are deleted from the Payroll_Table. With Soft Referential Integrity you could.

You create Soft Referential Integrity by stating **NO CHECK OPTION** on referential integrity you place on tables. This can enhance joins immensely. When the optimizer recognizes **Soft Referential Integrity** on two joining tables, it will act **accordingly**. If data is **expected** from **both tables**, Teradata will **do** the **join** like always. If data is **requested** from the **referenced table**, Teradata will **do** the **join** like always. However, if data is **requested** from just the **referencing table**, Teradata will **return data directly** from the **referencing table without joinin**g the other table.

```
Create TABLE Employee_Table
(
Employee_Number   INTEGER
 ,Dept_No         INTEGER
 ,Last_Name       Char(20)
 ,First_Name      Char(20)
 ,Salary          Decimal (10,2)

  ,FOREIGN KEY (Dept_No)
   REFERENCES WITH NO CHECK OPTION
   Department_Table (Dept_No)
) Unique Primary Index (Employee_No);
```

Materialized Views

Materialized Views are implemented via Join Indexes. You merely create a Join Index between two tables. Then create a view that joins the tables together. The expense of a Join Index is additional storage and the fact that UPDATES to the base tables slow down because the Join Index table needs to be UPDATED also.

CREATE the Join Index first.

```
Create Join Index Cust_Ord_SPI
   AS
SELECT O.Order_Number
           ,O.Customer_No
           ,O.Order_Total
           ,O.Order_Date
           ,C. Customer_Name
FROM    Order_Table as O
INNER JOIN
           Customer_Table as C
ON         O.Customer_No
=          C.Customer_No;
```

Then CREATE the Materialized view.

```
Create View ORD_CUST_V
   AS
SELECT O.Order_Number
           ,O.Customer_No
           ,O.Order_Total
           ,O.Order_Date
           ,C. Customer_Name
FROM    Order_Table as O
INNER JOIN
           Customer_Table as C
ON         O.Customer_No
=          C.Customer_No;
```

Compression

"To succeed... you need to find something to hold on to, something to motivate you, something to inspire you."
--Tony Dorsett

"To succeed… you need to find some things in your tables to compress. Saving a half Tera-byte will motivate and inspire management."
--Tera-Tom

Compression is fantastic! Did you know that you could now compress up to 255 distinct values (plus NULLs) per fixed width column? You can. This can significantly enhance system costs and performance. Incorporating multi-value compression on existing tables is perfect for Capacity Planning. You can save quite a bit of space.

- Up to 255 distinct values (plus NULL) can be compressed per column.

- Only fixed-width columns can be compressed. No VarChar capabilities yet.

- Primary Index columns can't be compressed.

- Alter table does not support compression on existing columns at this time.

- **Compression** works with **Multi-Set Tables**

If you want to improve overall Teradata system **performance** we suggest **four areas.**

- **Multi-Value compression** – Saves space and adds speed
- **Partition Primary Index** – Adds speed
- **Multi-statement requests** – Lowers overhead
- **Hire a CoffingDW Consultant** – Saves money and adds tremendous knowledge

Implementing Compression

You can implement compression when you create the table with the CREATE statement or in some cases the ALTER table statement. Here is the Syntax:

```
CREATE TABLE Employee
(
Employee_No      Integer
,Dept            Integer
,First_Name      Varchar(20)
,Last_Name       Char(20) COMPRESS
                  (‘Smith’, ‘Wilson’,
                   ‘Davis’, ‘Jones’)
) Unique Primary Index(Employee_No)
;
```

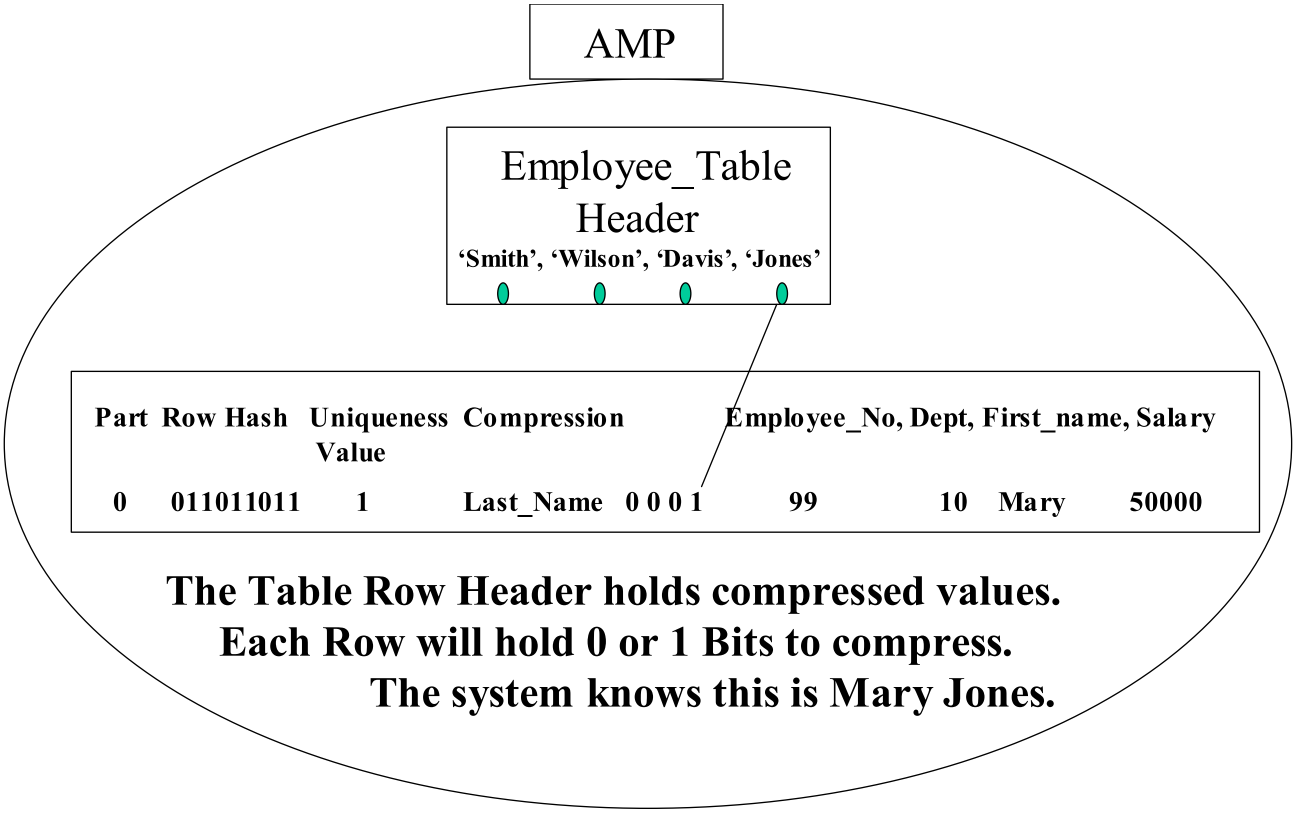
The ALTER statement can’t compress existing columns. It can, however, add new columns and compress them.

```
ALTER TABLE Employee
  ADD Department_Name    Char(20)
        COMPRESS
          (‘Sales’, ‘Marketing’,
            ‘Campaigns’, ‘HR’)
;
```

How Compression Works

Compression works in conjunction with both the Table Header and each data row that has compressed columns. When you create a table, a table header is created on each AMP. It will hold a grid of the compressed columns and the compressed value options. Then, when a row holds a compressed value it will flip a bit in the row overhead to indicate the value to represent.

```
CREATE TABLE Employee
(
Employee_No      Integer
,Dept            Integer
,First_Name      Varchar(20)
,Last_Name        Char(20) COMPRESS
                  ('Smith', 'Wilson',
                   'Davis', 'Jones')
) Unique Primary Index(Employee_No)
;
```



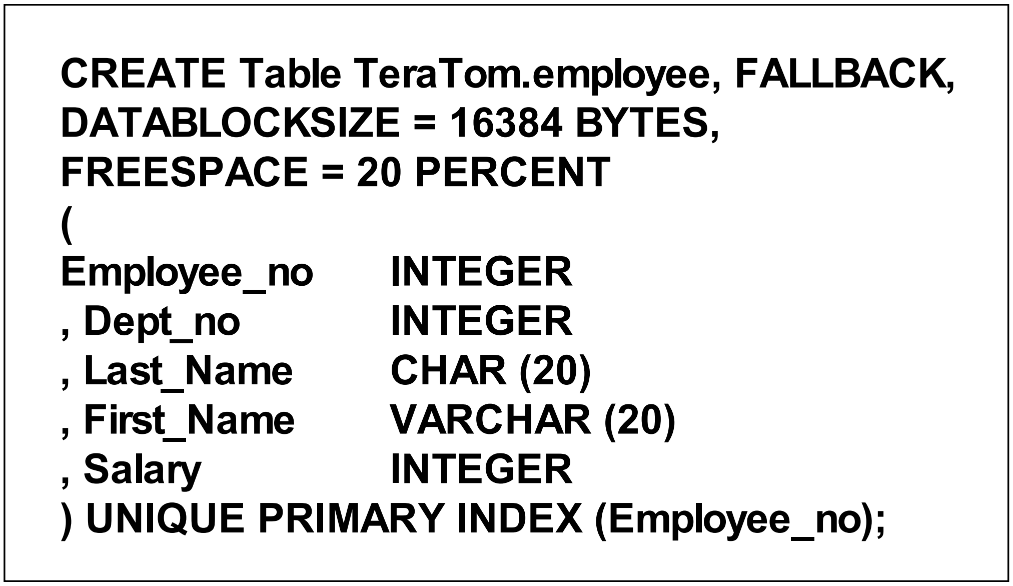
Part	Row Hash	Uniqueness Value	Compression	Employee_No,	Dept,	First_name,	Salary
0	011011011	1	Last_Name 0 0 0 1	99	10	Mary	50000

The Table Row Header holds compressed values.
Each Row will hold 0 or 1 Bits to compress.
The system knows this is Mary Jones.

Creating a Table With DATABLOCKSIZE

Teradata produces some of the largest systems in the world. What is amazing is how easy the systems are to administer for the DBA. In other databases, the DBA must create table space and place the table on a certain disk. Teradata just requires you to create the table and Teradata handles the complexity.

The picture below is an example of creating a table.



```
CREATE Table TeraTom.employee, FALLBACK,
DATABLOCKSIZE = 16384 BYTES,
FREESPACE = 20 PERCENT
(
Employee_no      INTEGER
, Dept_no        INTEGER
, Last_Name      CHAR (20)
, First_Name     VARCHAR (20)
, Salary         INTEGER
) UNIQUE PRIMARY INDEX (Employee_no);
```

Teradata allows the user some internal storage options when creating a table. The keywords are **DATABLOCKSIZE** and **FREESPACE PERCENTAGE**. The FREESPACE PERCENTAGE tells the system at what percentage that Teradata should keep a **cylinder free of rows** on data loads when using **Fastload** and **Multiload** utilities.

DATABLOCKSIZE determines a **maximum block** size for multiple row storage on a disk. The data block is the physical I/O unit for the Teradata file system. Larger block sizes enhance full table scan operations by retrieving more rows in a single I/O. Smaller blocks sizes are best for online transaction oriented tables. You specify this number in **BYTES**, **KILOBYTES**, or **KBYTES**. BYTES specifications are rounded to the nearest sector of **512 bytes**. KBYTES, or KILOBYTES, are in set increments of **1024 BYTES**. If DATABLOCKSIZE is not specified, the size used is the default of either **32,256 (63 sectors)** or **65,024 (127 sectors)**.

The ALTER TABLE Command

> *"Forgiveness does not change the past, but it does enlarge the future."*
>
> **– Paul Boese**

The only thing in life that we can consistently count on is change. This is especially true in a data warehouse environment. As business requirements change, it is sometimes necessary to reflect those changes into the tables. Teradata allows for modifications of a table at either the table or column level, using the ALTER command. Here is a list of table changes available to the ALTER table:

- Add one or more new columns to an existing table (**up to 2048 columns**)
- Add new attributes for one or more columns in a table
- Add a **new column** with **Multi-Value compression**
- Drop one or more columns from an existing table
- Modify constraints on an existing table at the column or table level
- Modify a constraint for a **Name** and **Check Constraint**
- Add or remove Fallback or Journaling
- Modify the **DATABLOCKSIZE** or **FREESPACE PERCENT**
- Change the name of a column in an existing table

```
ALTER Table TeraTom.employee,
DUAL AFTER JOURNAL
ADD Soc_Sec      INTEGER;
```

Index Wizard

"As I would not be a slave, so I would not be a master."

– Abraham Lincoln

A warehouse divided can not stand poor index selections. This is why the Index Wizard was created. The Index Wizard is designed to help with Index recommendations and comes with a beautiful Graphical User Interface (GUI). The wizard works by **analyzing SQL statements** in a **defined workload** and then recommends the best **indexes** to utilize. The Index Wizard even creates the **DDL statements** for **the indexes**.

Index Wizard analyzes a workload of SQL and then creates a series of reports and index recommendations describing the costs and statistics associated with the recommendations. The reports are designed to help you back up your decision to apply an index. Index wizard also **recommends dropping** of certain **indexes** that it deems unnecessary

Both Index Wizard and Statistics Wizard allow a user to import workloads from other Teradata tools such as Database Query Log (DBQL) or Query Capture Database (QCD).

Here are the steps to using the Index Wizard:

- Define a workload
- The workload is analyzed
- The Wizard recommends indexes
- Reports are generated
- Indexes are validated
- Indexes can be applied

You can define a workload:

- Using DBQL Statements
- Using QCD
- Entering SQL Text
- Importing a Workload
- Creating a new workload from an existing one