

Altibase Administration

Administrator's Manual

Release 6.1.1

May 11, 2012



Altibase Administration Administrator's Manual

Release 6.1.1

Copyright © 2001~2012 Altibase Corporation. All rights reserved.

This manual contains proprietary information of Altibase Corporation; it is provided under a license agreement containing restrictions on use and disclosure and is also protected by copyright patent and other intellectual property law. Reverse engineering of the software is prohibited.

All trademarks, registered or otherwise, are the property of their respective owners

Altibase Corporation

10F, Daerung PostTower II, 182-13,

Guro-dong Guro-gu Seoul, 152-847, Korea

Telephone: +82-2-2082-1000 Fax: 82-2-2082-1099

E-mail: support@altibase.com [www: http://www.altibase.com](http://www.altibase.com)

Contents

| | |
|--|------------|
| Preface | i |
| About This Manual | ii |
| Audience..... | ii |
| Software Environment..... | ii |
| Organization..... | ii |
| Documentation Conventions | iv |
| Related Documents | vi |
| Online Manual | vi |
| Altibase Welcomes Your Opinions | vi |
| 1. Introduction | 1 |
| 1.1 The Hybrid DBMS Concept..... | 2 |
| 1.2 ALTIBASE HDB Features..... | 8 |
| 1.3 The Structure of ALTIBASE HDB..... | 13 |
| 2. ALTIBASE HDB Components | 19 |
| 2.1 ALTIBASE HDB Directories | 20 |
| 2.2 Executable Binaries | 25 |
| 2.3 ALTIBASE HDB Libraries..... | 28 |
| 3. Creating a Database..... | 29 |
| 3.1 Creating a Database | 30 |
| 4. Startup and Shutdown | 37 |
| 4.1 Startup Procedure | 38 |
| 4.2 Shutting Down ALTIBASE HDB | 40 |
| 5. Objects and Privileges | 43 |
| 5.1 Database Objects..... | 44 |
| 5.2 Tables | 47 |
| 5.3 Queues..... | 53 |
| 5.4 Constraints | 55 |
| 5.5 Indexes | 57 |
| 5.6 Views | 62 |
| 5.7 Sequences | 64 |
| 5.8 Synonyms | 67 |
| 5.9 Stored Procedures and Functions | 68 |
| 5.10 Triggers | 72 |
| 5.11 Database Users | 74 |
| 5.12 Privileges | 76 |
| 6. Managing Tablespaces | 81 |
| 6.1 Tablespaces: Definition and Structure | 82 |
| 6.2 Classifying Tablespaces | 90 |
| 6.3 Disk Tablespace..... | 94 |
| 6.4 The Undo Tablespace..... | 99 |
| 6.5 Tablespace States..... | 103 |
| 6.6 Managing Tablespaces | 104 |
| 6.7 Examples of Tablespace Use..... | 116 |
| 6.8 Managing Space in Tablespaces | 126 |
| 7. Partitioned Objects | 137 |
| 7.1 What is Partitioning? | 138 |
| 7.2 Partitioned Objects | 140 |
| 7.3 Partition Conditions..... | 147 |
| 7.4 Partitioning Methods | 149 |
| 8. Managing Transactions | 165 |
| 8.1 Transactions..... | 166 |
| 8.2 Locking | 169 |
| 8.3 Multi-Version Concurrency Control (MVCC) | 172 |
| 8.4 Transaction Durability..... | 178 |
| 9. Database Buffer | |

| | |
|---|------------|
| Manager..... | 181 |
| 9.1 Structure of the Buffer Manager | 182 |
| 9.2 Managing Database Buffers..... | 188 |
| 9.3 Related Database Properties | 195 |
| 9.4 Statistics for Buffer Management | 196 |
| 10. Backup and Recovery | 197 |
| 10.1 Database Backup | 198 |
| 10.2 Database Recovery | 203 |
| 10.3 Backup and Recovery Examples | 207 |
| 11. SQL Tuning | 219 |
| 11.1 Introduction to SQL Tuning..... | 220 |
| 11.2 Query Optimization Procedures | 227 |
| 11.3 Analyzing Execution Plans..... | 272 |
| 11.4 Using Optimizer Hints | 314 |
| 12. Using Explain Plan | 319 |
| 12.1 Overview | 320 |
| 12.2 Reading Plan Trees | 323 |
| 12.3 Plan Nodes..... | 325 |
| 13. SQL Plan Cache..... | 351 |
| 13.1 Overview | 352 |
| 13.2 Features | 354 |
| 13.3 Related Properties..... | 355 |
| 14. Communication Layer | 357 |
| 14.1 Communication Protocols..... | 358 |
| 15. Securing Data | 363 |
| 15.1 Overview | 364 |
| 15.2 How Security is Organized in ALTIBASE HDB..... | 365 |
| 15.3 Integrating a Security Module..... | 366 |
| 15.4 Starting Security Modules and Encrypting Data..... | 368 |
| 16. Tuning ALTIBASE HDB | 371 |
| 16.1 Log File Groups | 372 |
| 16.2 Group Commit | 377 |
| 17. DB Diagnostic Monitoring | 381 |
| 17.1 Monitoring Database Servers | 382 |
| 17.2 Troubleshooting Procedures | 383 |
| AppendixA. Trace Logs..... | 391 |
| Using Application Trace Logs..... | 391 |
| AppendixB. ALTIBASE HDB Limitations | 393 |
| Maximum ALTIBASE HDB Values | 393 |

Preface

About This Manual

This manual describes the concepts and architecture of ALTIBASE HDB. This manual also explains to administrators how to manage their databases.

Audience

This manual has been prepared for the following ALTIBASE HDB users:

- database administrators
- application developers
- programmers

It is recommended that those reading this manual possess the following background knowledge:

- basic knowledge in the use of computers, operating systems, and operating system utilities
- experience in using relational databases and an understanding of database concepts
- computer programming experience

Software Environment

This manual has been prepared assuming that ALTIBASE HDB 6.1.1 will be used as the database server.

Organization

This manual has been organized as follows:

- [Chapter1: Introduction](#)

This chapter introduces hybrid Database Management Systems and gives the related background.

- [Chapter2: ALTIBASE HDB Components](#)
- [Chapter3: Creating a Database](#)

This chapter describes tablespaces, the primary logical structure with which databases are created.

- [Chapter4: Startup and Shutdown](#)
- [Chapter5: Objects and Privileges](#)

This chapter describes ALTIBASE HDB objects and privileges and explains how to manage them.

- [Chapter6: Managing Tablespaces](#)

This chapter explains the tablespace concept, describes the tablespace structure and supported functions, and explains to administrators how to manage tablespaces efficiently.

- [Chapter7: Partitioned Objects](#)

This chapter describes partitioned tables. “Partitioning” refers to the division of a large database object into several small pieces for efficient management.

- [Chapter8: Managing Transactions](#)

This chapter describes the concepts behind transactions and locking and explains how to manage transactions in an ALTIBASE HDB server.

- [Chapter9: Database Buffer Manager](#)

This chapter describes the structure and functions of the buffer manager.

- [Chapter10: Backup and Recovery](#)

This chapter describes the ALTIBASE HDB backup and recovery features and explains how to manage your database backup and recovery tasks.

- [Chapter11: SQL Tuning](#)

This chapter describes SQL tuning to enhance query performance.

- [Chapter12: Using Explain Plan](#)

- [Chapter13: SQL Plan Cache](#)

This chapter describes the concepts and features of the ALTIBASE HDB SQL Plan Cache.

- [Chapter14: Communication Layer](#)

This chapter describes the methods and protocols involved in establishing a connection between a client application and an Altibase database server.

- [Chapter15: Securing Data](#)

This chapter explains how to use security modules to develop a database encryption strategy.

- [Chapter16: Tuning ALTIBASE HDB](#)

This chapter describes how to use the Log File Group and Group Commit features.

- [Chapter17: DB Diagnostic Monitoring](#)

This chapter explains how to monitor and troubleshoot database servers.

- [Appendix A. Trace Logs](#)

This appendix describes application trace logs and explains how to create and access them.

- [Appendix B. ALTIBASE HDB Limitations](#)

This appendix describes the maximum values of ALTIBASE HDB objects.

Documentation Conventions




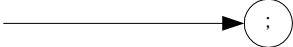

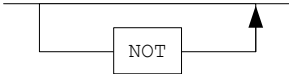
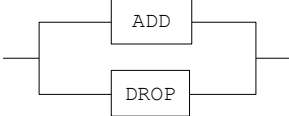
This section describes the conventions used in this manual. Understanding these conventions will make it easier to find information in this manual and other manuals in the series.

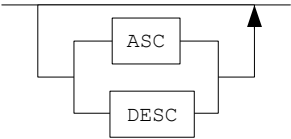
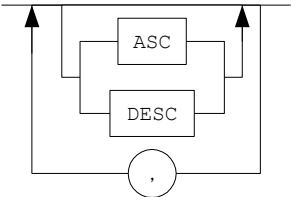
There are two sets of conventions:

- syntax diagram conventions
- sample code conventions

Syntax Diagram Conventions

This manual describes command syntax using diagrams composed of the following elements:

| Elements | Meaning |
|---|--|
|  | Indicates the start of a command. If a syntactic element starts with an arrow, it is not a complete command. |
|  | Indicates that the command continues to the next line. If a syntactic element ends with this symbol, it is not a complete command. |
|  | Indicates that the command continues from the previous line. If a syntactic element starts with this symbol, it is not a complete command. |
|  | Indicates the end of a statement. |
|  | Indicates a mandatory element. |
|  | Indicates an optional element. |
|  | Indicates a mandatory element comprised of options. One, and only one, option must be specified. |

| Elements | Meaning |
|---|--|
|  | Indicates an optional element comprised of options. |
|  | Indicates an optional element in which multiple elements may be specified. A comma must precede all but the first element. |

Sample Code Conventions

The code examples explain SQL statements, stored procedures, iSQL statements, and other command line syntax.

The following table describes the printing conventions used in the code examples.

| Rules | Meaning | Example |
|---------------|---|---|
| [] | Indicates an optional item. | VARCHAR [(size)] [[FIXED] VARIABLE] |
| { } | Indicates a mandatory field for which one or more items must be selected. | { ENABLE DISABLE COMPILE } |
| | A delimiter between optional or mandatory arguments. | { ENABLE DISABLE COMPILE } [ENABLE DISABLE COMPILE] |
| | Indicates that the previous argument is repeated, or that sample code has been omitted. | iSQL> select e_lastname from employees; E_LASTNAME ----- Moon Davenport Kobain . . . 20 rows selected. |
| Other Symbols | Symbols other than those shown above are part of the actual code. | EXEC :p1 := 1; acc NUMBER(11,2) ; |

| Rules | Meaning | Example |
|------------------|--|--|
| Italics | Statement elements in italics indicate variables and special values specified by the user. | <code>SELECT * FROM table_name; CONNECT <i>userID/password</i>;</code> |
| Lower case words | Indicate program elements set by the user, such as table names, column names, file names, etc. | <code>SELECT e_lastname FROM employ- ees;</code> |
| Upper case words | Keywords and all elements provided by the system appear in upper case. | <code>DESC SYSTEM_.SYS_INDEX_;</code> |

Related Documents

For more detailed information, please refer to the following documents:

- Installation Guide
- Getting Started Guide
- SQL Reference
- Stored Procedures Manual
- iSQL User's Manual
- Utilities Manual
- Error Message Reference

Online Manual

Online versions of our manuals (PDF or HTML) are available from the Altibase Download Center (<http://atc.altibase.com/>).

Altibase Welcomes Your Opinions

Please feel free to send us your comments and suggestions regarding this manual. Your comments and suggestions are important to us, and may be used to improve future versions of the manual. When you send your feedback, please make sure to include the following information:

- The name and version of the manual you are using
- Your comments and suggestions regarding the manual
- Your full name, address, and phone number

Please send your e-mail to the following address:

support@altibase.com

In addition to suggestions, this address may also be used to report any errors or omissions discovered in the manual, which we will address promptly.

If you need immediate assistance with technical issues, please contact the Altibase Customer Support Center.

We always appreciate your comments and suggestions.

1 Introduction

This chapter contains the following sections:

- [The Hybrid DBMS Concept](#)
- [ALTIBASE HDB Features](#)
- [The Structure of ALTIBASE HDB](#)

1.1 The Hybrid DBMS Concept

This chapter introduces Hybrid Database Management Systems (Hybrid DBMS), a new concept pioneered by Altibase, and gives the related background.

1.1.1 Background of Hybrid DBMS

The development of the hybrid DBMS is closely related to the characteristics of memory and disks, the two major types of data storage media used by DBMS.

First, memory consists of electronic gates. The time required to access memory is only a few nanoseconds (ns, billionths of a second), and is quite consistent. However, in the event of a power failure, all data in memory are lost. That is, memory is a volatile storage medium.

In contrast, a disk comprises a head arm and a platter. Disk access time is on the order of microseconds (μ s, millionths of a second), which is quite slow compared to random access memory (RAM). Furthermore, access times can be inconsistent. Even SSDs (Solid State Drives), which have recently become more popular, have vastly inferior access times when compared to volatile memory.

However, the data on the disk are stored permanently, even in the event of a power failure.

Second, memory is connected to the main board through the system bus, and its maximum capacity is determined by the specifications of the main board. If the main board has a 32-bit CPU installed, the maximum possible amount of memory is 4GB, whereas, at present, if a 64-bit CPU is installed in the main board, the theoretical maximum amount of memory is on the order of hundreds of gigabytes (that is, hundreds of billions of bytes). In contrast, disks are connected to the main board through the I/O bus, and thus terabytes of disk space can be installed, regardless of the characteristics of the main board.

In conclusion, compared to disks, memory typically enjoys the advantages of access times that are hundreds of times faster and consistent performance, but is still plagued by problems of limited capacity and data loss in the event of a power failure. On the other hand, although disk storage is permanent and virtually unlimited in terms of capacity, it entails problems of low speed and inconsistent access times.

DBMSs can be categorized as one of two kinds, according to the distinct characteristics of the two kinds of memory devices: Disk-Resident DBMSs (hereinafter referred to simply as "DRDBMS"), which store data on disk, and Main Memory DBMSs (hereinafter referred to as "MMDBMS"), which store data in memory. Additionally, hybrid DBMSs have been developed to take advantage of the strong points and compensate for the weak points of both DRDBMS and MMDBMS.

1.1.2 The Reign of DRDBMS

In a DRDBMS, data are stored on disk. DRDBMS are configured such that the data are read from the disk into a memory buffer and sent to client applications.

In this structure, the application typically uses SQL (Structured Query Language) to access the data. One major advantage of DRDBMS is the use of concurrency control and recovery strategies to protect the data, which makes it much easier to develop applications. Moreover, because the data are stored on disk, high-capacity DBMSs can be implemented.

Thanks to these advantages, DRDBMS have been widely adopted in various industrial fields.

However, with the spread of modern society and the sudden increase in demand for improved information handling performance in fields such as the Internet and telecom fields, DBMSs are requested to be highly responsive, scalable, and constantly available. Thus, due to problems of inadequate data processing speed and consistency, DRDBMSs are unusable in an increasing number of fields.

Therefore, in those industrial fields in which high and consistent data processing performance is important, custom-designed memory-based DBMSs have emerged.

However, as such data processing products are not widely used, they have to be individually developed from scratch. This has had the undesirable consequences of increased maintenance and repair costs and decreased performance, integration, and scalability.

In response thereto, MMDBMSs, which are relational databases optimized for use with volatile memory, have been developed with the goal of processing voluminous transactions in real time.

1.1.3 The Advent of MMDBMS

MMDBMS are structured such that data are stored in memory and the data in memory are read and sent directly to client applications.

This structure preserves the main advantages of DRDBMS, namely the ability to access data using standard SQL statements and to protect data via concurrency control and recovery, thereby making it easy to develop applications and share data.

Additionally, because MMDBMS store data in memory, in contrast with DRDBMS, which store data on disk, average processing speed is very fast, and consistent performance, which is an inherent characteristic of memory, is assured. Therefore, MMDBMS are receiving attention in fields in which fast and consistent performance are necessary but development and maintenance issues make it difficult to implement DRDBMS.

An MMDBMS can typically perform an update operation about 10 times as quickly as a DRDBMS, and a search operation about 3 times as quickly.

The reason that an update operation cannot be performed hundreds of times faster than when using a DRDBMS is that, like a DRDBMS, an MMDBMS must also write log files to disk in order to protect the data. However, an update operation is nevertheless faster when using an MMDBMS than when using a DRDBMS, because, in MMDBMS, data protection has been greatly simplified and optimized.

Similarly, the reason that a search operation is not hundreds of times faster than when using a DRDBMS is that a DRDBMS also uses memory buffers to improve data access performance. However, search operations are nevertheless faster using an MMDBMS because data access is simplified and optimized, and access times are consistent when accessing memory (that is, so-called "jitter" is eliminated).

Despite conferring the advantages of high and consistent performance, because MMDBMS must save data in memory, they encounter a limitation when information processing demands are very high in industrial fields in which more than hundreds of GB of data must be saved.

1.1.4 Combining MMDBMS and DRDBMS

At present, the most common solution to this problem is to divide the data and store them separately. MMDBMS and DRDBMS are sometimes combined in systems in which so-called "high-perfor-

1.1 The Hybrid DBMS Concept

mance data”, that is, data that much be accessed quickly, are stored in an MMDBMS, and voluminous data in a DRDBMS.

This structure encounters the following problems: information shared by the MMDBMS and DRDBMS must be synchronized, applications that must bidirectionally communicate with both the MMDBMS and DRDBMS must be connected with both of them at the same time, and error recovery is complicated.

However, to date, because there has been no other way to simultaneously realize high data processing performance and handle large amounts of data, this approach has typically been adopted in fields in which fast data access and voluminous data processing are both required.

1.1.5 The Advent of Hybrid DBMS

Hybrid DBMSs have arisen in response to the need to take advantage of the strong points and overcome the weak points of MMDBMS, DRDBMS, and combined MMDBMS/DRDBMS.

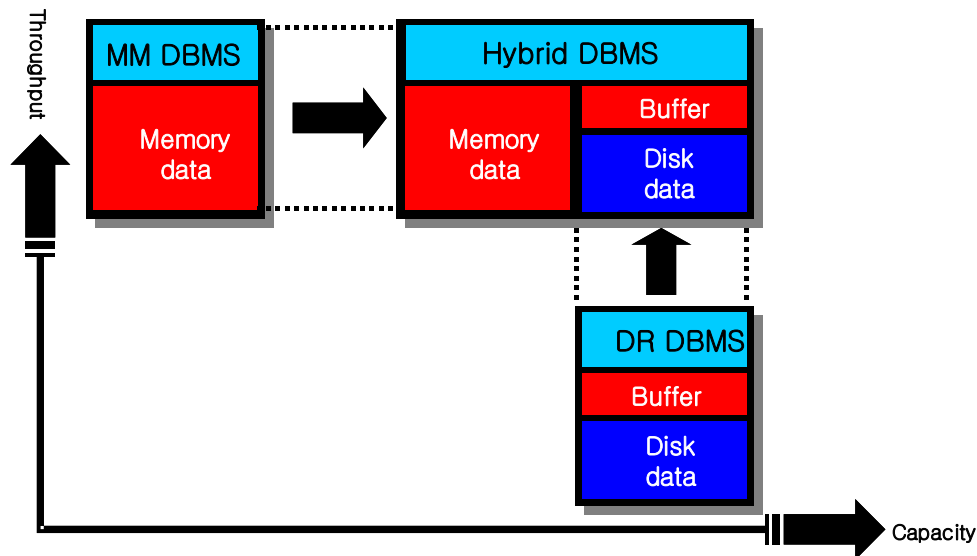
In a Hybrid DBMS, the data are separated, and high-performance data are stored in memory and voluminous data on disk. However, a single DBMS processes both kinds of data in a unified manner.

Because high-performance and voluminous data are both handled by a single DBMS, the aforementioned problems related to combined DBMSs, namely complicated error handling and the requirement for applications to be complicated, are solved. Furthermore, a Hybrid DBMS can be implemented as an MMDBMS, a DRDBMS, or a Hybrid DBMS.

To summarize, the Hybrid DBMS combines the advantages of the MMDBMS, which is optimized for processing high-performance data, and the DRDBMS, which is optimized for processing large amounts of data, because the data are classified and saved according to their characteristics, but handled in an integrated manner.

That is to say, high-performance information processing is possible thanks to the speed of Hybrid DBMS, and large amounts of information can be processed thanks to the efficient resource usage of Hybrid DBMS. Hybrid DBMS can now be adopted in all fields, including those requiring both high performance and the processing of large amounts of data.

Figure 1-1 The Structure of a High Performance / Large capacity DBMS



1.1.6 Characteristics of the ALTIBASE HDB Hybrid DBMS

The ALTIBASE HDB hybrid DBMS consists of the Storage Manager, the Query Processor and the Main Module.

The Storage Manager is the module responsible for data protection via concurrency control and recovery, and the Query Processor is the module responsible for creating execution plans for SQL queries and data access. Finally, the Main Module is responsible for handling sessions and threads, so that requests from multiple clients can be simultaneously handled.

Storage Manager The Storage Manager consists of the Recovery Manager and the Concurrency Controller.

Since the Storage Manager has a layered structure, configuration is simplified and structural stability is assured, and thus it is easy to test individual layers, and the likelihood of errors is minimized.

The Recovery Manager integrates memory table and disk table log files, incorporates an optimized checkpointing algorithm, and provides an Automatic Recovery Test Tool for the development process so that the incidence of errors is reduced.

The Concurrency Controller comprises the Index Layer, the Transaction Layer, the Application Layer, and the Interface Layer, and makes use of both out-place MVCC and in-place MVCC (please refer to the definitions of these terms in the following section, “MVCC”). It implements the “Transaction Status” Slot and “Start SCN” List techniques to improve simultaneous performance. Additionally, when modifying the structure of a B+tree index, the B+tree search process does not require a latch, which improves query speed.

The integration of data interfaces for both disk tablespaces and memory tablespaces has made it easier to use both kinds of tablespaces and improved compatibility therebetween.

Query Processor The Query Processor, which consists of the Query Executor and the Query Optimizer, has a transparent architecture that unites the processing of memory queries, disk queries, and

1.1 The Hybrid DBMS Concept

hybrid queries.

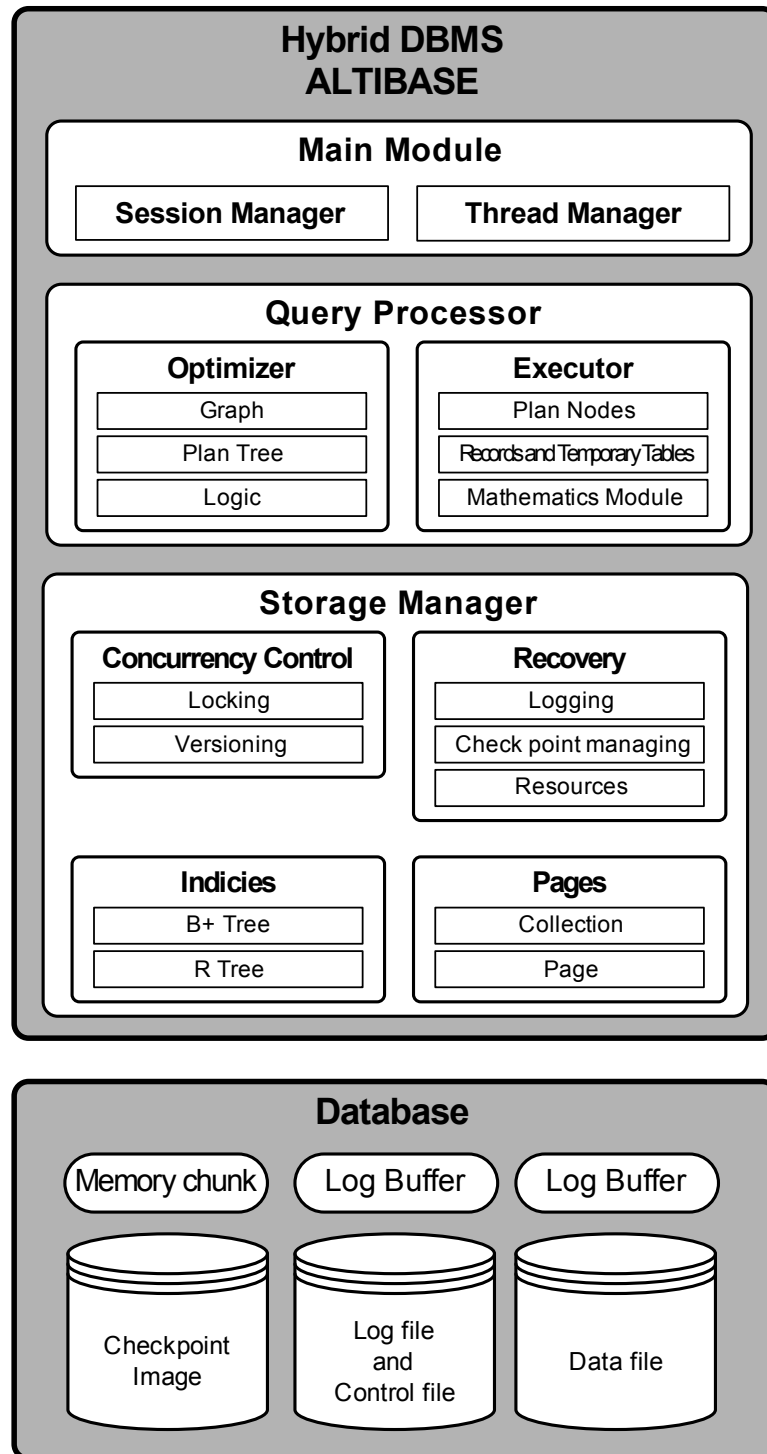
The Query Executor realizes various kinds of joins using the smallest possible number of plan nodes by completely eliminating tuple processing-related interdependencies between plan nodes. For example, even if the number of possible joining methods is increased from 10 to 34, only one plan node is added.

The Query Optimizer uses real-time statistical information that imposes almost no load, and thus enables accurate cost estimation without regard to whether a query is a memory query, a disk query, or a hybrid query. Furthermore, to minimize the amount of space that the optimizer needs in order to conduct a search, a Join Greedy Algorithm is used as the Join Ordering Algorithm.

Main Module The Session Manager plays the role of managing sessions.

The Thread Manager manages threads that process data.

Figure 1-2 The Internal Structure of ALTIBASE HDB



1.2 ALTIBASE HDB Features

This section provides an overview of the components and functionalities of ALTIBASE HDB.

The basic features of ALTIBASE HDB, the high-performance, large-volume hybrid database system, will now be introduced. The characteristics, structure, function, etc., of ALTIBASE HDB are explained briefly here. For more detailed information, please refer to each of the separately published ALTIBASE HDB manuals.

1.2.1 Data Model

The ALTIBASE HDB data model is patterned after the relational model. The relational model includes the three following major concepts:

Database Structures- The objects that databases use to save and access data are referred to as tables, indexes, views, etc. These objects are the basic operational units.

Operations- Operations define what actions users are permitted to conduct on data and database structures. Operations are related to integrity constraints.

Integrity- Integrity constraints are rules pertaining to which operations are permitted on data and structures, and serve to protect data and data structures.

Relational database management systems provide the following benefits:

- physical data management and logical data management are independent of each other
- all data can be accessed easily and in a variety of ways
- databases can be freely designed as desired
- database storage space requirements and data redundancy are reduced

1.2.2 Engine Structure

ALTIBASE HDB supports a client-server architecture. In client-server architecture, the client accesses the server over a communications network, as with a traditional RDBMS.

The ALTIBASE HDB server has a multithreaded internal structure. One client establishes a session with one server thread.

1.2.3 Interfaces

Unlike other real-time database systems, ALTIBASE HDB supports a wide range of industry-standard interfaces for maximum compatibility. the ALTIBASE HDB query language complies fully with the SQL92 standard, and also provides extended features.

Because ALTIBASE HDB supports ODBC,OLE DB, JDBC, and C/C++ Precompiler, you might not even need to change your existing applications in order to use them with our hybrid MMDBMS.

For more about the ALTIBASE HDB SQL specifications, please refer to the *SQL User's Manual*.

1.2.4 Multi-Version Concurrency Control

ALTIBASE HDB manages concurrency using MVCC (Multi-Version Concurrency Control). MVCC is a way of realizing maximum performance by eliminating collisions when read and write operations are performed on multiple versions of a single data item.

In particular, the problem in which a read operation places a lock on data to be changed, thus causing a subsequent modify operation on the data to take a long time, which is the weakness of the conventional row locking method, is overcome, and furthermore, old unnecessary data can be immediately removed, thereby preventing memory from being wasted. MVCC exhibits optimal performance in environments with large numbers of users, and supports “hot backup” systems, that is, databases in which backup operations can be performed at will without first shutting down the database.

In ALTIBASE HDB, the MVCC method is implemented differently for disk tables and memory tables, although this difference is imperceptible to the user. So-called “out-place MVCC”, in which a new version of a record is created every time a record in a memory table is changed, is implemented for memory tables, whereas for disk tables, “in-place MVCC”, in which new data are written over existing records and undo tablespaces are used to refer to previous versions of the data, is implemented.

1.2.5 Transaction Processing

the ALTIBASE HDB Hybrid DBMS architecture provides various features to help you achieve maximum performance. First, the number of transactions that can be simultaneously executed in the database can be controlled by configuring the properties in the altibase.properties file. Additionally, for efficient server operation, AUTOCOMMIT mode can be used. Furthermore, ALTIBASE HDB provides the following transaction isolation levels: “read committed” (0), “repeatable read” (1), and “no phantom read” (2), which can be selected appropriately depending on the user’s requirements.

1.2.6 Logging

For database stability and durability, ALTIBASE HDB maintains logs pertaining to changed database contents. Moreover, ALTIBASE HDB optimizes the way that logs are created in order to achieve the best possible replication performance.

1.2.7 Buffer Pool

To improve the performance of transactions that access disk tablespaces, disk I/O is minimized. This is accomplished using the buffer pool. Pages that have already been read from disk and cached in memory are prevented from being subsequently read from disk again. The buffer pool is managed using the Hot-Cold LRU (Least Recently Used) algorithm.

1.2.8 Double-Write Files

In cases where the page size on the ALTIBASE HDB system and the file system’s physical page size are different, if ALTIBASE HDB is abnormally shut down during disk I/O, the pages may be corrupted.

To prevent this, when ALTIBASE HDB flushes a page, it saves an identical image in what is called a “double-write file” on the disk, and then saves the page again in its original location. Furthermore,

1.2 ALTIBASE HDB Features

when the ALTIBASE HDB server is restarted, it compares the contents of the double-write file with that of the actual page, and restores any corrupted pages.

The double write function compensates for disk errors, but can negatively impact system performance. The user can opt not to implement this system in the interests of performance.

1.2.9 Fuzzy & Ping-Pong Checkpointing

ALTIBASE HDB uses fuzzy & ping-pong checkpointing methods to safely back up the most recent state of the database.

In a main memory database, fuzzy checkpointing stores all changed data pages in a backup database, and because transactions that are currently underway can have an effect thereon, fuzzy and ping-pong checkpointing methods are used together. That is, because two backup databases are maintained, the processing burden associated with the checkpointing process can be decreased, and optimum transaction performance can be realized.

1.2.10 Stored Procedures

Stored procedures are DB procedures that execute multiple SQL statements contained therein in consideration of input arguments, output arguments and input/output arguments and according to conditions specified in the body thereof.

Stored procedures are functionally classified as either procedures or functions depending on whether or not they return a value. Please refer to the *Stored Procedure User's Manual* for more detailed information.

1.2.11 Deadlock Detection

Deadlock is a state in which transactions wait for each other to release locked resources that they require. To deal with such cases, conventional DBMS have a separate thread or process which detects and handles such deadlocks. This kind of detection structure inevitably results in a temporary service interruption. ALTIBASE HDB does not have a separate deadlock detection thread. Instead, a deadlock is detected at the instant it occurs, and ALTIBASE HDB immediately takes steps to prevent service interruption depending on the case, in order to guarantee stable and continuous database operation.

1.2.12 Table Compaction

When a database is running, it is possible for a particular memory table to occupy more memory than it actually requires. This often happens when previously inserted data are updated or deleted. In these cases, it would be more efficient if the memory not needed by the table in question could be returned to the system. To meet this need, ALTIBASE HDB provides a memory table compaction function. Using this function, memory and tables can be more efficiently managed.

1.2.13 Database Replication

ALTIBASE HDB provides log-based database replication to realize both high availability and fault tolerance. This log-based replication system construction, in which database replication is conducted

based on transaction logs, increases the efficiency of ALTIBASE HDB and decreases the load on the system. A replication management thread on a local system, which is currently operating, sends local transaction logs to a replication management thread on a remote system in real time. The replication management thread on the remote system analyzes the received log data and passes them to the ALTIBASE HDB server, which implements the changes in the database. In this way, a system can be provided in which, when normal operation of one of the servers is interrupted, service can be immediately restored without downtime.

ALTIBASE HDB also provides a load-balancing feature. In a replicated ALTIBASE HDB database environment, user transactions can be divided into two or more groups. Each group of transactions is executed on a corresponding server, and changes on one server are reflected on the other servers automatically. In this way, data consistency between the servers is ensured.

1.2.14 Client-Server Structure

When running ALTIBASE HDB in a client-server architecture, the user can select a client-server protocol that is suitable for the system. The communication protocols that ALTIBASE HDB supports are TCP/IP, IPC, and UNIX domain socket.

TCP/IP (Transmission Control Protocol/Internet Protocol) is the protocol that is most commonly used between clients and servers over a network. The IPC (Inter-Process Communication) protocol is typically used by ALTIBASE HDB when shared memory is used to support communication between client applications and the server. Since IPC uses shared memory, marshalling of communication packets is not necessary, and thus faster communication can usually be achieved than when using other protocols.

When the client application and ALTIBASE HDB are on different systems, TCP/IP, which uses Internet sockets, must be used, whereas, when they are on the same system, the domain socket protocol or the IPC protocol can also be used. IPC offers the fastest performance of these communication protocols, followed by domain socket and then TCP/IP.

1.2.15 Database Space

An Altibase database consists of all of the data in the database, stored in one or more tablespaces, and the tablespaces are divided into memory tablespaces and disk tablespaces according to the way in which the data are stored.

Besides the system tablespace, which is created by ALTIBASE HDB, the user can add memory and disk tablespaces.

1.2.16 Direct-Path INSERT

In a direct-path INSERT operation, input data are saved in newly created pages, rather than being saved in free space found in existing pages. In other words, when data are entered, free space in tables is not used, and instead, new extents are allocated from tablespace.

Moreover, because INSERT can be used in the same manner as APPEND, the number of redo and undo operations is reduced, and thus logging expenses are reduced.

1.2 ALTIBASE HDB Features

1.2.17 Database Link

Database Link unites disparate data sources on interconnected servers to produce a single unified result, even if the data are stored in different kinds of data servers that are physically far apart from one another.

1.2.18 iSQL

Users can manage their databases quickly and easily using iSQL (the ALTIBASE HDB interactive SQL command utility).

1.2.19 Audit

With Audit, ALTIBASE HDB provides functionality for comparing and checking individual database tables and outputting mismatched data, and for harmonizing two databases that have been found to contain mismatched data.

1.2.20 iLoader

Using the iLoader utility provided with ALTIBASE HDB, when moving databases or backing up individual tables, users can upload or download data one table at a time.

1.2.21 AdminCenter

Using AdminCenter, a utility provided with ALTIBASE HDB in GUI form, database administrators can access their databases more easily. More than just an ALTIBASE HDB connection and monitoring tool, Admin Center also allows tables and all kinds of objects to be visually managed from Windows systems.

1.3 The Structure of ALTIBASE HDB

In this section, the ALTIBASE HDB internal server processing structure and database structure will be explained, with emphasis on using ALTIBASE HDB in a client-server environment.

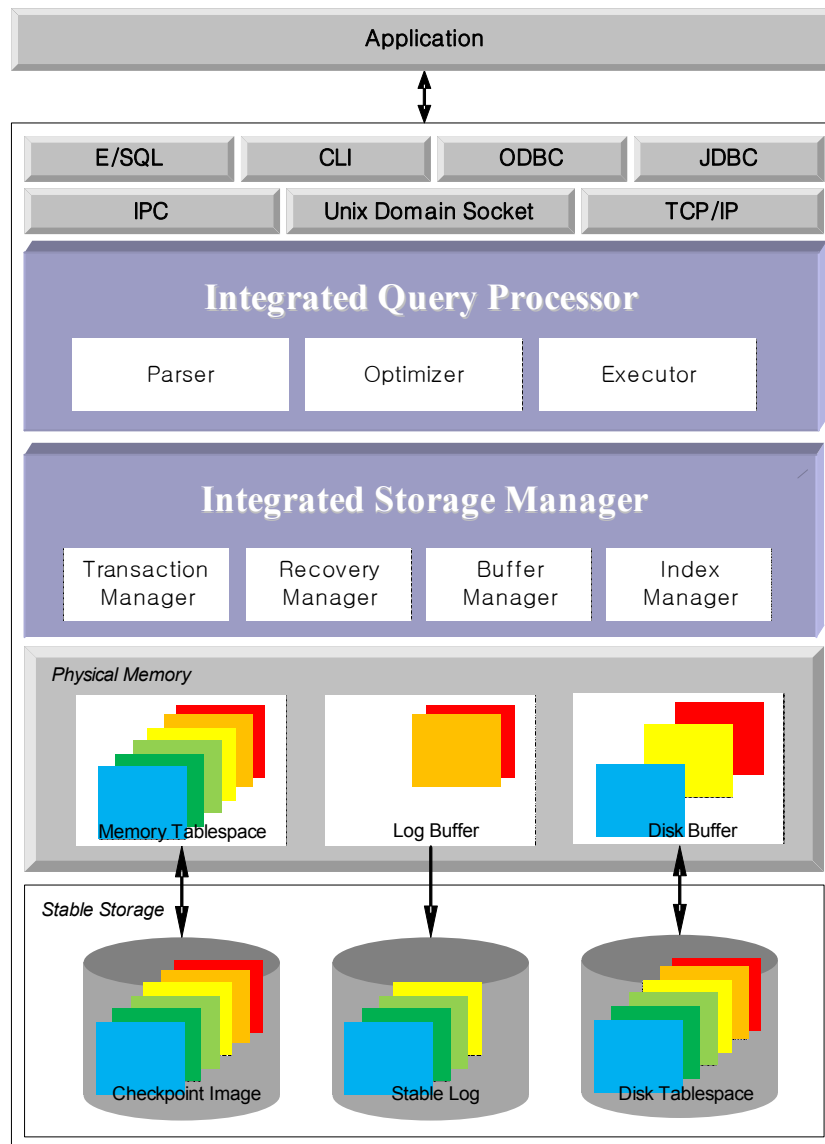
This section is organized as follows:

- [Overall Construction](#)
- [Internal Structure of Server Process](#)
- [Physical Database Structure](#)
- [Logical Database Structure](#)

1.3.1 Overall Construction

The overall construction of ALTIBASE HDB consists of a server component, a client library with which applications are written, and a communications module in between them. The server component includes a Query Processor and a Data Storage Manager.

Figure 1-3 The Structure of the ALTIBASE HDB Server



1.3.2 Internal Structure of Server Process

The internal structure of the ALTIBASE HDB server process consists of the main thread, the service daemon thread, the checkpoint thread, the session management thread, the garbage collection thread, the log flush thread, the buffer flush thread, and the archivelog thread. Each thread performs the function described below:

1.3.2.1 Main Thread

The main thread creates, controls, and terminates all other threads.

1.3.2.2 Service Daemon Thread

The service daemon thread accepts a connection request from a client and connects it with an available service thread in the service thread pool.

1.3.2.3 Service Thread Pool

ALTIBASE HDB creates and manages service threads for query processing and pools them in the Service Thread Pool. The number of service threads that are created corresponds to the user configuration at the time the server was started.

1.3.2.4 Checkpoint Thread

To reduce the amount of work required when recovering from a failure, this thread records information about the current status of the database and the system in data files. Both manual checkpointing and automatic (periodic) checkpointing are available.

1.3.2.5 Session Management Thread

The session management thread monitors the status of the session between the client and the service thread, that is, it monitors whether a given session has been interrupted.

1.3.2.6 Garbage Collection Thread

Using MVCC may cause old and unnecessary data to remain in memory. As soon as some data become unnecessary, the garbage collection thread recovers memory space so that it can be reused, in order to maximize the efficiency of memory usage.

1.3.2.7 Log Flush Thread

The log flush thread maintains the logs that are created in response to every transaction that occurs in a database. It has the function of uniting all of the data in the log buffer and writing the resultant large amount of log data to a log disk. These completely synchronized logs are used to ensure safe recovery in the event that the database system suffers an interruption or disaster.

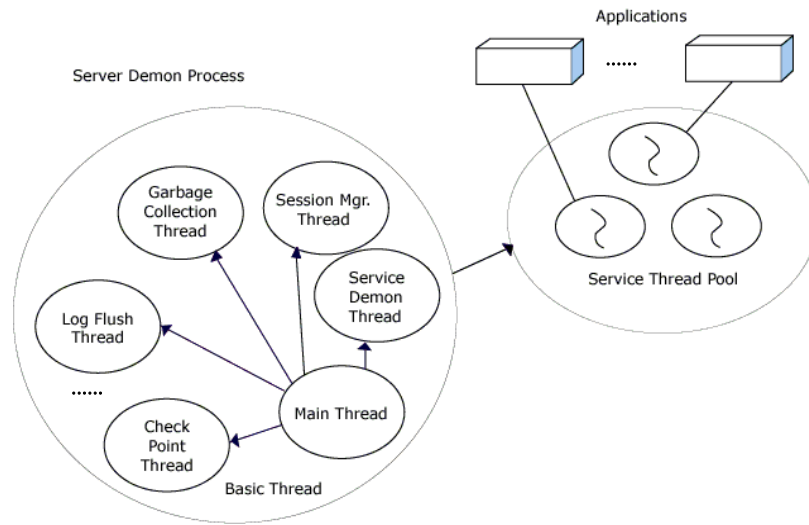
1.3.2.8 Buffer Flush Thread

When all of the memory in the buffer pool is in use, disk I/O is required, which can cause performance inconsistencies ("jitter") on the transactions that are underway. The buffer flush thread regularly checks the buffer, always maintains a certain amount of volume buffer memory, and writes unused pages to disk, so that memory can be reused.

1.3.2.9 Archivelog Thread

This thread regularly copies online log files to a predefined destination for use in recovery from storage media errors. The destination path is specified in the ARCHIVE_DIR property in the alti-base.properties file. This feature works only when the database is in archive log mode.

Figure 1-4 Internal Structure of ALTIBASE HDB Server Process



1.3.3 Physical Database Structure

An Altibase database physically consists of log anchor files, log files, and data files.

1.3.3.1 Log Anchor Files

Log anchor files contain critical information about data files and transaction logs. They contain general information about the state of data files at specific points in time based on transaction log time-stamps. These files must be backed up along with data files in order for database recovery to be possible.

1.3.3.2 Log Files

Log files, also known as “redo log files”, are used to maintain the atomicity and durability of transactions. Atomicity is the ability to return to the state that existed before a transaction by rolling back the transaction. Durability is the ability to restore a database to its original state, which reflects the result of all recently properly committed transactions, in the event of a database fault.

Log files are categorized into prepare log files, active log files and archive log files depending on their contents. A log file which is used to write current transaction logs is called an active log file. A prepare log file is an empty log file which is prepared in advance in order to increase the speed with which logs are written. An archive log file is a backup of a log file which is no longer being written to, but which is kept available for recovery purposes.

Log files are very important because they record the current status of the database. If a current log file should become damaged, the entire database will be damaged, regardless of whether or not a transaction was underway at the time the log file was damaged. Log files are typically used in conjunction with backup files to restore the database in the event that data files become damaged.

1.3.3.3 Data Files

By default, the created system memory tablespace is saved in SYS_TBS_MEM_DATA, whereas the meta tables and the created disk tablespace are saved in SYS_TBS_MEM_DIC and system.dbf, respectively. Moreover, the intermediate results of queries that are currently being executed are saved in temp.dbf, and previous image information, which is used for MVCC (Multi-Version Concurrency Control) is saved in the undo file.

ALTIBASE HDB manages files for storing data on the basis of pages. All data files consist of data pages, which are the smallest unit used by databases.

Pages are categorized into catalog pages, which contain information for managing the database, and data pages, which contain user data. Catalog pages contain detailed information about the current database, and are used to maintain information about changes and consistency checks, which are conducted when ALTIBASE HDB is started up and shut down.

Catalog pages contain lists and information about the use of the other data pages in the database. They are the first pages that can be found in backup databases, and are very important pages.

Actual user data are stored in data pages. Each data page consists of a page header and a page body. A page header consists of link and type information, used to maintain a list of pages, as well as a page identification number. A page body is divided into a number of slots, in which the actual data are ultimately stored.

1.3.4 Logical Database Structure

ALTIBASE HDB logically stores data in memory and disk tablespaces, and physically in data files that correspond to these tablespaces.

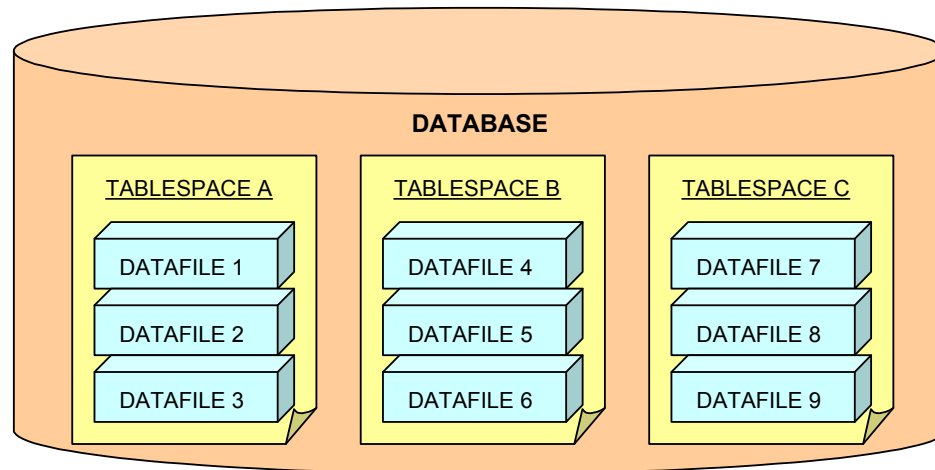
In ALTIBASE HDB, each tablespace consists of one or more data files. However, a data file can only be associated with a single tablespace.

A database and its tablespaces and data files are intimately related as follows:

A database logically consists of one or more storage units known as tablespaces. Tablespaces are logical space in which all of the data in a database are saved. A database physically consists of one or more files called data files. That is to say, data files are the physical space in which all of the data in a database are stored.

The following figure describes the relationship between tablespaces and data files.

Figure 1-5 The Logical Structure of a Database



ALTIBASE HDB allocates tablespaces – logical database areas – to all of the data in a database. The units of allocation of physical database space are pages, extents and segments.

A page is the smallest unit of logical storage. ALTIBASE HDB stores all data in pages.

The next logical step up from a page is an extent. That is, an extent consists of a particular number of consecutive pages.

The next logical database storage area up from an extent is called a segment. Each segment is a set of extents, and all extents in one segment are stored in the same tablespace.

For more information, please refer to the *Administrator's Manual*.

1.3.5 Other Control Files

1.3.5.1 Boot Log File (`altibase_boot.log`)

The ALTIBASE HDB server records information about the booting status in this file. Because this file is written to every time ALTIBASE HDB is started up and shut down, detailed system information is available, and moreover, when ALTIBASE HDB shuts down abnormally, this file provides clues about the error state.

1.3.5.2 Property File (`altibase.properties`)

This file is the ALTIBASE HDB server environment configuration file, and contains all of the information pertaining to how the ALTIBASE HDB server is executed and tweaked.

1.3.5.3 Error Message Files

This file contains error messages related to the data storage management module, the Query Processor, and the ALTIBASE HDB server main module, as well as those related to function execution and data type.

2 ALTIBASE HDB Components

The ALTIBASE HDB product broadly consists of executable binaries and the programming library, as well as sample files, configuration files and include files.

This chapter contains the following sections:

- [2.1 ALTIBASE HDB Directories](#)
- [2.2 Executable Binaries](#)
- [2.3 ALTIBASE HDB Libraries](#)

2.1 ALTIBASE HDB Directories

When ALTIBASE HDB is installed, the following directories are created. The location of the ALTIBASE HDB home directory is saved in the environment variable ALTIBASE_HOME. The bin, conf, lib, include, msg, dbs, logs, sample, install, audit, trc, admin and arch_logs directories can all be found within this directory.

This section describes the purpose and contents of each of these directories.

2.1.1 admin

This directory contains the adminview.sql script file, which creates views related to ALTIBASE HDB system information, as well as other script files for creating stored procedures that are used to view information related to tables, stored procedures, and replication objects.

2.1.2 arch_logs

This is the directory containing backup log files, which are used for recovery. The location and name of this directory must be specified in the altibase.properties file.

2.1.3 audit

This directory contains sample script files for “audit”, which is the ALTIBASE HDB utility for resolving data mismatches that arise in the course of replication.

For more information about the audit utility, please refer to the *Audit User's Manual*.

2.1.4 bin

This directory contains all of the executable support tools that need to be used with ALTIBASE HDB.

The bin directory contains the following files. For detailed explanations of each utility, please refer to the *Utilities User's Manual*.

aexport, altibase, altierr, altimon, altipasswd, altiProfile, audit, check-Server, dumpla, dumplf, iloader, isql, killCheckServer, server, apre, shmutil

2.1.5 conf

This directory contains the ALTIBASE HDB license and a property file (altibase.properties) for setting a wide range of ALTIBASE HDB options. When ALTIBASE HDB is launched, the license is checked, and then the server is initialized with reference to this property file. For more information on ALTIBASE HDB properties, please refer to the *General Reference*.

2.1.6 dbs

When using the default values, database files are created in this directory. The location and name of

this directory must be specified in the altibase.properties file.

By default, system memory tablespace is created and saved in SYS_TBS_MEM_DATA, meta tables are saved in SYS_TBS_MEM_DIC, disk tablespace is created and saved in system001.dbf, and query results that are temporarily needed while queries are being executed are saved in temp001.dbf.

Previous image information that is needed for SQL statement execution and restoration is saved in the undo001.dbf file. Disk pages are temporarily saved in *.dwf files, which are double-write buffer files.

2.1.7 include

This directory contains the header files needed to create applications that use the ODBC library.

2.1.7.1 alaAPI.h

An API header file that is used by the ALTIBASE HDB Log Analyzer.

2.1.7.2 sqlcli.h

A header file that is needed in order to create client applications.

2.1.7.3 sqltypes.h

This file contains information on basic data types that is needed when developing client applications that use ODBC.

2.1.7.4 sqlucode.h

The header file that defines Unicode.

2.1.7.5 ulpLibInterface.h

This file contains information on the structure of error-handling SQL statements for use when developing applications using the C/C++ Precompiler.

2.1.8 install

This directory contains a README file and the altibase_env.mk file, which contains macro settings and other Makefile-related information, which is necessary when creating applications for use with ALTIBASE HDB.

2.1.9 lib

This directory contains an application development library for developing client applications, and contains the following files. For information on how to write applications using these library files, please refer to the *Getting Started Guide*.

2.1 ALTIBASE HDB Directories

2.1.9.1 Altibase.jar

This is the JDBC driver for accessing ALTIBASE HDB via Java applications. This is a Type 4 driver, and is thus a Pure Java driver.

2.1.9.2 libapre.a

This library is needed in order to create embedded SQL programs. For more information on writing embedded SQL programs, please refer to the *Precompiler User's Manual*.

2.1.9.3 libodbccli.a

This library is used when authoring ALTIBASE HDB client-server applications.

2.1.10 logs

This directory contains log anchor files and log files.

The location and name of this directory must be specified in the altibase.properties file. The log anchor file name and the log file name are automatically set by ALTIBASE HDB. However, in order to be prepared in the event of an error in the file system containing the log anchor files, it is nevertheless recommended that the relevant properties be changed, and that individual log anchor files be located on different file systems.

2.1.11 msg

This directory contains the following error message files. Error information is available only in English.

2.1.11.1 E_SM_US7ASCII.msb

This file contains error messages pertaining to the Data Storage Manager.

2.1.11.2 E_QP_US7ASCII.msb

This file contains error messages pertaining to the Query Processor.

2.1.11.3 E_MM_US7ASCII.msb

This file contains error messages pertaining to the ALTIBASE HDB server main module.

2.1.11.4 E_CM_US7ASCII.msb

This file contains error messages pertaining to the ALTIBASE HDB communication module.

2.1.11.5 E_RP_US7ASCII.msb

This file contains error messages pertaining to the ALTIBASE HDB replication module.

2.1.11.6 E_ST_US7ASCII.msb

This file contains error messages pertaining to the ALTIBASE HDB stored procedures module.

2.1.11.7 E_ID_US7ASCII.msb, E_MT_US7ASCII.msb

This file contains error messages pertaining to function execution or data types.

2.1.12 sample

This directory contains sample ALTIBASE HDB applications.

It contains source code and Makefiles for programs written using the JDBC, ODBC, and C/C++ Pre-compiler libraries.

2.1.13 trc

This directory contains trace files in which information about the state of ALTIBASE HDB execution is written. Each internal module writes to a corresponding trace file as follows:

2.1.13.1 altibase_boot.log

The operating state of the ALTIBASE HDB server is recorded in this file. The information recorded in this log file includes system details obtained when ALTIBASE HDB is started up and shut down and the error status when ALTIBASE HDB shuts down abnormally.

2.1.13.2 altibase_id.log

System-level warnings and trace messages are written to this file.

2.1.13.3 altibase_sm.log

Warnings and trace messages pertaining to the storage manger module are written to this file.

2.1.13.4 altibase_mt.log

Warnings and trace messages pertaining to data types or built-in function modules are written to this file.

2.1.13.5 altibase_rp.log

Warnings and trace messages pertaining to the replication module are written to this file.

2.1.13.6 altibase_qp.log

Warnings and trace messages pertaining to the Query Processor are written to this file.

2.1 ALTIBASE HDB Directories

2.1.13.7 altibase_mm.log

Warnings and trace messages pertaining to the ALTIBASE HDB server main module are written to this file.

2.1.13.8 altibase_IPC.log

Information about resources that are created when connecting via IPC is written to this file.

2.2 Executable Binaries

Except where otherwise noted, complete information about the use of these binary files can be found in the *Utilities Manual*.

2.2.1 aexport

This is a tool that is used to achieve a sequence of tasks that must be conducted when upgrading ALTIBASE HDB to a new version. This program automatically creates SQL script files, iSQL executable shell files, and iLoader executable shell files pertaining to all database objects and users.

2.2.2 altibase

This is the server that is used when ALTIBASE HDB is executed in a client-server architecture scheme.

2.2.3 altierr

This tool that finds and outputs detailed information about ALTIBASE HDB error codes.

2.2.4 altimon.sh

This is a shell script program that monitors the ALTIBASE HDB server process.

2.2.5 altiProfile

This is a utility that collects statistical data (e.g. number of executions, execution time) about SQL statements.

2.2.6 altipasswd

This tool is for changing the password of the sys account.

2.2.7 audit

This tool performs two functions: outputting information about mismatched data after comparing and checking databases table by table, and reconciling two databases that contain mismatched data.

For detailed information, please refer to the *Audit User's Manual*.

2.2.8 checkServer

This is a utility for executing script files to check the status of ALTIBASE HDB and perform tasks that must be fulfilled when ALTIBASE HDB terminates abnormally.

2.2 Executable Binaries

2.2.9 dumpla

This tool outputs and examines the contents of ALTIBASE HDB log anchor files.

2.2.10 dumpIf

This tool outputs and examines the contents of ALTIBASE HDB log files.

2.2.11 dump_stack.sh

This tool outputs the process call stack in the event that ALTIBASE HDB shuts down abnormally.

2.2.12 iloader

This tool is for uploading and downloading particular database tables.

For detailed information on this tool, please refer to the *iLoader User's Manual*.

2.2.13 isql

This is a tool for interactively executing database queries.

For detailed information on this tool, please refer to the *iSQL User's Manual*.

2.2.14 killCheckServer

This tool terminates execution of the checkServer utility.

2.2.15 server

This is a shell script program that is used to start up, shut down, or restart the ALTIBASE HDB server.

2.2.16 apre

This application is used for precompiling applications written in C/C++ that contain embedded SQL statements.

For detailed information on this tool, please refer to the *Precompiler User's Manual*.

2.2.17 shmutil

This is a utility for managing databases which are located in shared memory. In the case where ALTIBASE HDB uses shared memory for a main memory database, shmutil is used to perform tasks related to shared memory management.

Note: For more specific details about each utility, please refer to the Utilities Manual.

2.3 ALTIBASE HDB Libraries

In order to develop application programs using ALTIBASE HDB, you will need the following:

- C or C++ programming libraries
- the provided libraries for the ODBC API (libodbccli.a)
- ODBC programming libraries, in order to use the ALTIBASE HDB ODBC driver (altibase_odbc.dll) in an MS Windows environment
- the Java class library (Altibase.jar), in order to program in the Java language
- header files required for programming

This is explained in greater detail in the *Getting Started Guide*.

3 Creating a Database

After ALTIBASE HDB is installed, the database administrator must estimate the amount of user data that is likely to be generated, and create and maintain a database accordingly. This chapter explains the essential points that must be kept in mind when creating a database.

3.1 Creating a Database

An Altibase database consists of one or more logical storage units called tablespaces, which collectively store all of the database's data. ALTIBASE HDB stores data logically in tablespaces and physically in datafiles, which are associated with corresponding tablespaces. Before the database server can be started, it is first necessary to create a database manually using the CREATE DATABASE command.

This section describes tablespaces, the primary logical structure of ALTIBASE HDB, and the kinds of logging systems, and explains how to create a database.

- [Kinds of Tablespaces](#)
- [The ALTIBASE HDB Logging System](#)
- [Preparing to Create a Database](#)
- [Creating a Database](#)
- [Shutting Down a Database Server after Database Creation](#)
- [Database Initialization Properties](#)

3.1.1 Kinds of Tablespaces

An Altibase database consists of several kinds of tablespaces. Tablespaces are classified into several types according to their intended use and the way in which the related data are stored.

By default, the data files in this tablespace have the *.dbf filename extension. Files which are created when the CREATE DATABASE statement is executed are located in the \$ALTIBASE_HOME/dbs/ directory.

Note: There is no limit on the filename extension or location of files specified when users create a tablespace.

ALTIBASE HDB supports the following kinds of tablespaces:

3.1.1.1 Memory Tablespace

Memory tablespace exists in memory. Performance-critical objects are usually located in memory tablespace, along with dictionary tables, system objects such as sequences, and user-created memory tables.

3.1.1.2 Data Tablespace

Data tablespace exists on disk, and usually consists of user tables and indexes. Data tablespace is classified into system tablespace and user data tablespace.

3.1.1.3 Undo Tablespace

This is the tablespace in which images of previous states of data (before update) are stored for a certain period in order to support Multiversion Concurrency Control (MVCC), which is the manage-

ment of multiple versions of records that exist in disk tables.

3.1.1.4 Temporary Tablespace

This is the tablespace for storing temporary tables and indexes, which are created when queries are processed. Analogous to the data tablespace, this tablespace is categorized into system temporary tablespace and user temporary tablespace.

3.1.1.5 Volatile Tablespace

This is a tablespace for saving objects in memory in order to avoid disk input/output and therefore realize better performance. All data objects in volatile tablespace disappear when the database is shut down. The size of volatile tablespace can't exceed the available physical memory space in the system.

3.1.2 The ALTIBASE HDB Logging System

The data in a database must be durable regardless of the circumstances. ALTIBASE HDB guarantees the durability of data using a logging system that comprises the following two kinds of files:

3.1.2.1 Log Files

These are files in which log records are written in order to be ready for use in performing complete system recovery in the event of abnormal shutdown while transactions are underway. ALTIBASE HDB log files are named logfile** (where "***" indicates the sequential number of the log file).

3.1.2.2 Log Anchor Files

Data that are important for database execution, such as information about tablespaces, the location of data files, and checkpointing, are stored in log anchor files. In order for the server to start up correctly, the contents of these files must be valid, otherwise it will be impossible to start the server. Log anchor files are also used for database recovery.

When the database is first created, the log files and the log anchor files are created and saved in \$ALTIBASE_HOME/logs/.

ALTIBASE HDB maintains a set of 3 log anchor files. These log files are created in the same location when the database is created, but it is recommended that the 3 log anchor files be maintained on different file systems. The property for specifying the location of the log anchor files is LOGANCHOR_DIR.

For more about the ALTIBASE HDB properties, please refer to the *General Reference*.

3.1.3 Preparing to Create a Database

After installing the ALTIBASE HDB package, use the iSQL utility, which is provided as part of the package, to manually create a database as follows.

First, execute the iSQL utility with SYSDBA privileges.

3.1 Creating a Database

```
shell> isql -u sys -p manager -sysdba
```

This does not access a database, but establishes an administration session. It will then be possible to see a display like the following:

```
-----  
Altibase Client Query utility.  
Release Version 6.1.1.1  
Copyright 2000, Altibase Corporation or its subsidiaries.  
All Rights Reserved.  
-----  
ISQL CONNECTION = TCP, SERVER = 127.0.0.1, PORT_NO = 20000  
iSQL(sysdba)>
```

Once the preceding steps have been accomplished, it is first necessary to start the server process in order to execute the CREATE DATABASE statement.

It is important to understand the ALTIBASE HDB startup phases. Several important administration tasks are only executable in particular startup phases.

The ALTIBASE HDB server is started according to the following sequence:

1. Phase1: The Pre-Process Phase

Before starting the server process, ALTIBASE HDB initializes database memory in this phase.

A database can be created during the Process phase. In the Pre-process phase, issue the following command to proceed to the Process phase.

```
iSQL> startup process  
Trying Connect to Altibase.. Connected with Altibase.  
TRANSITION TO PHASE: PROCESS  
Command execute success.
```

2. Phase2: The Process Phase

In this phase, you can change and check the ALTIBASE HDB properties and create a database using the CREATE DATABASE command.

3. Phase3: The Control Phase

This is the phase in which ALTIBASE HDB loads database files and prepares for restart recovery. For information about restart recovery, please refer to the section pertaining to [10.2 Database Recovery](#) in Chapter 10.

4. Phase4: The Meta Phase

This is the phase in which recovery is completed and in which it is possible to upgrade meta data and reset online logs.

5. Phase5: The Service Phase

This is the final phase, in which the database is ready to provide service.

3.1.4 Creating a Database

In the Process phase, use the CREATE DATABASE command to create a database as shown below. For

more information on using the CREATE DATABASE statement, please refer to the *SQL Reference*. In the following example, a database is created using the default options.

```
iSQL> create database mydb initsize=50M noarchivelog character set ksc5601
national character set utf16;
DB Info (Page Size = 32768)
  (Page Count = 1537)
  (Total DB Size = 50364416)
  (DB File Size = 1073741824)
Creating MMDB FILES [SUCCESS]
Creating Catalog Tables [SUCCESS]
Creating DRDB FILES [SUCCESS]
[SM] Rebuilding Indices [Total Count:0] [SUCCESS]
DB Writing Completed. All Done.
Create success.
```

3.1.5 Shutting Down a Database Server after Database Creation

Once a database has been created, the server process that was started for that purpose can be shut down, or can proceed to the service phase. To shut down the server, use the shutdown command with the abort option, as follows:

```
iSQL(sysdba)> shutdown abort
iSQL(sysdba)>
```

Once the server has been shut down, iSQL is disconnected from the ALTIBASE HDB server process and returns to the pre-processing phase.

In addition to “abort”, the shutdown command can also be used with the “immediate” and “normal” options, but only when the server is running in the service phase.

A thorough understanding of the ALTIBASE HDB properties related to database initialization, which are listed below, is highly recommended.

3.1.6 Database Initialization Properties

When a database is created using the CREATE DATABASE statement, any properties that are not specified in the CREATE DATABASE statement are set according to the settings made in the `altibase.properties` file, which is located in the `$ALTIBASE_HOME/conf/` directory. The relevant properties are listed below. In the table, a question mark (“?”) indicates the path specified using the ALTIBASE_HOME environment variable.

| Property Name | Description | Default |
|-------------------|---|---------|
| DB_NAME | The name of the database to be created. | mydb |
| MEM_DB_DIR | The directory where the database files will be located. Three values can be specified. | ?/dbs |
| SERVER_MSGLOG_DIR | The directory for storing the files (altibase_boot.log) in which ALTIBASE HDB server messages are recorded. | ?/trc |

3.1 Creating a Database

| Property Name | Description | Default |
|--------------------------|---|--------------|
| SHM_DB_KEY | The key value of shared memory, which is used when ALTIBASE HDB is started using the shared memory feature. | 0 (not used) |
| MEM_MAX_DB_SIZE | The maximum size to which a database can dynamically increase. | 4G |
| LOGANCHOR_DIR | The directory where the log anchor files will be located. Three directories must be specified. | ?/logs |
| LOG_DIR | The directory where the log files will be located. | ?/logs |
| LOG_FILE_SIZE | The size of an individual log file. | 10M |
| STARTUP_SHM_CHUNK_SIZE | The maximum size of a shared memory chunk, which is allocated when ALTIBASE HDB is started. | 1G |
| EXPAND_CHUNK_PAGE_COUNT | The number of memory tablespace pages that are allocated at one time. | 3200 |
| TEMP_PAGE_CHUNK_COUNT | The number of temporary pages in memory tablespace that are allocated at one time. | 128 |
| SYS_DATA_TBS_EXTENT_SIZE | The size of an extent for system data tablespace. | 256K |
| SYS_DATA_FILE_INIT_SIZE | The initial system tablespace size when the CREATE DATABASE statement is executed. | 100M |
| SYS_DATA_FILE_MAX_SIZE | The maximum size of a data file in system tablespace. | 2G |
| SYS_DATA_FILE_NEXT_SIZE | The amount by which a data file in system tablespace is automatically increased when the auto-extend option is used. | 1M |
| SYS_TEMP_TBS_EXTENT_SIZE | The size of an extent in temporary tablespace. | 256K |
| SYS_TEMP_FILE_INIT_SIZE | The initial size of temporary tablespace when the CREATE DATABASE statement is executed. | 100M |
| SYS_TEMP_FILE_MAX_SIZE | The maximum size of a data file in temporary tablespace. | 2G |
| SYS_TEMP_FILE_NEXT_SIZE | The amount by which a data file in temporary tablespace is automatically increased when the auto-extend option is used. | 1M |
| SYS_UNDO_TBS_EXTENT_SIZE | The size of an extent in undo tablespace. | 128K |
| SYS_UNDO_FILE_INIT_SIZE | The initial size of undo tablespace when the CREATE DATABASE statement is executed. | 100M |

| Property Name | Description | Default |
|-----------------------------------|--|---------|
| SYS_UNDO_FILE_MAX_SIZE | The maximum size of a data file in undo tablespace. | 2G |
| SYS_UNDO_FILE_NEXT_SIZE | The amount by which a data file in undo tablespace is automatically increased. | 1M |
| USER_DATA_TBS_EXTENT_SIZE | The size of a single extent in user tablespace. | 256K |
| USER_DATA_FILE_INIT_SIZE | The initial size of user tablespace when the CREATE DATABASE statement is executed. | 100M |
| USER_DATA_FILE_MAX_SIZE | The maximum size of a data file in user tablespace. | 2G |
| USER_DATA_FILE_NEXT_SIZE | The amount by which a data file in user tablespace is automatically increased. | 1M |
| USER_TEMP_TBS_EXTENT_SIZE | The size of an extent in user temporary tablespace. | 256K |
| USER_TEMP_FILE_INIT_SIZE | The initial size of user temporary tablespace when the CREATE DATABASE statement is executed. | 100M |
| USER_TEMP_FILE_MAX_SIZE | The maximum size of a data file in user temporary tablespace. | 2G |
| USER_TEMP_FILE_NEXT_SIZE | The amount by which a temporary data file is increased in size when user tablespace is extended automatically. | 1M |
| ADD_EXTENT_NUM_FROM_TBS_TO_SEG | The number of newly assigned extents when a segment is extended. | 1 |
| ADD_EXTENT_NUM_FROM_SYSTEM_TO_TBS | The number of newly assigned extents when a tablespace is extended. | 4 |
| CHECKSUM_METHOD | The name of the page validation method. | 1 |

For more information about the ALTIBASE HDB properties, please refer to the *General Reference*.

3.1 Creating a Database

4 Startup and Shutdown

In order to provide service after creating a database, it is necessary to start the ALTIBASE HDB server to the service phase. This chapter explains considerations to bear in mind when starting up and shutting down the database.

This chapter comprises the following sections:

- [Startup Procedure](#)
- [Shutting Down ALTIBASE HDB](#)

4.1 Startup Procedure

There are two ways to start up an ALTIBASE HDB server:

- A database administrator logs in to the server using the sys account, accesses the server in -sysdba administrator mode, and then uses the privileges granted to him/her to manually start the server.
- The server is started using a server script.

To start up ALTIBASE HDB, first execute iSQL using the -sysdba option in the same way as when creating a database.

The following demonstrates how to start up an ALTIBASE HDB database using iSQL.

```
$ isql -u sys -p manager -sysdba
-----
Altibase Client Query utility.
Release Version 6.1.1.1
Copyright 2000, Altibase Corporation or its subsidiaries.
All Rights Reserved.
-----
iSQL_CONNECTION = TCP, SERVER = 127.0.0.1, PORT_NO = 20000
iSQL(sysdba) >
```

Note: The ALTIBASE HDB startup command can only be executed using the UNIX account with which ALTIBASE HDB (including iSQL) was installed.

During the startup process, the status of ALTIBASE HDB progresses in sequence through the following phases:

1. PRE-PROCESS
2. PROCESS
3. CONTROL
4. META
5. SERVICE

The startup command can be used with the following phase options:

```
iSQL> STARTUP [PROCESS | CONTROL | META | SERVICE];
```

Users other than the SYS user can access the database only in the SERVICE phase.

Note: ALTIBASE HDB can only progress from earlier phases to subsequent phases; it is impossible to revert to a previous phase.

The following is an example of starting the server in the service phase:

```
iSQL> startup service;
Trying Connect to Altibase..... Connected with Altibase.
TRANSITION TO PHASE: PROCESS
TRANSITION TO PHASE: CONTROL
TRANSITION TO PHASE: META
[SM] Checking Database Phase: *.*.*[SUCCESS]
[SM] Recovery Phase - 1: Preparing Database...[SUCCESS]
```

```

[SM] Recovery Phase - 2: Loading Database : Dynamic Memory Version
Serial Bulk Loading
. is 8192k: *..[SUCCESS]
[SM] Recovery Phase - 3: Skipping Recovery & Starting Threads...[SUCCESS]
Refining Disk Table [SUCCESS]
[SM] Garbage Collection: ..... [SUCCESS]
[SM] Rebuilding Index [Total Count:61] *****.....
.....
..... [SUCCESS]
TRANSITION TO PHASE: SERVICE
No IPC Initialize: Disabled
--- STARTUP Process SUCCESS ---
Command execute success.

```

The database performs the following tasks in each phase:

| Phase | Tasks |
|-------------|--|
| PRE-PROCESS | The initial phase. The server prepares to advance to the PROCESS phase. |
| PROCESS | In this phase, the CREATE DATABASE and DROP DATABASE statements can be executed, a limited number of performance views can be used to obtain information, and property values can be changed. The server prepares to advance to the CONTROL phase. |
| CONTROL | Media Recovery can be performed in this phase. The server prepares to advance to the META phase. If incomplete recovery is performed during the CONTROL phase, online logs must be reset when proceeding to the META phase. |
| META | Meta data (the dictionary table) can be upgraded in this phase. For information about incomplete recovery, please refer to the section on Complete vs. Incomplete Recovery in Chapter 10. The server prepares to advance to the SERVICE phase. |
| SERVICE | The SERVICE phase is the normal operational state of ALTIBASE HDB. Users other than the SYS user can establish connections to ALTIBASE HDB in this phase. SHUTDOWN NORMAL/IMMEDIATE/ABORT can all be executed. |

4.2 Shutting Down ALTIBASE HDB

The SHUTDOWN command can be used to shut down a currently running ALTIBASE HDB server. The following options are available:

```
iSQL(sysdba)>SHUTDOWN [NORMAL | IMMEDIATE | ABORT];
```

SHUTDOWN NORMAL and SHUTDOWN IMMEDIATE can only be executed while ALTIBASE HDB is running in the SERVICE phase, whereas SHUTDOWN ABORT can be executed in any phase.

Note: The ALTIBASE HDB SHUTDOWN command can only be executed using the UNIX account with which ALTIBASE HDB (including iSQL) was installed.

4.2.1 SHUTDOWN NORMAL

This is the usual method of shutting down the server under normal conditions. Server shutdown tasks must wait until after all clients have disconnected from the server. In order to shut down the server, the following tasks are conducted in sequence:

- Threads that function to detect client-server communication sessions are shut down.
- Service threads are shut down.
- The storage manager is shut down.
- Finally, when all tasks that must wait until after the ALTIBASE HDB server process has completely shut down have been performed, the ALTIBASE HDB server shuts down.

When ALTIBASE HDB is shut down in this way, the following messages are output:

```
iSQL(sysdba)> shutdown normal;  
Ok..Shutdown Proceeding...  
TRANSITION TO PHASE : Shutdown Altibase  
[RP] Finalization : PASS  
shutdown normal success.
```

4.2.2 SHUTDOWN IMMEDIATE

When SHUTDOWN IMMEDIATE is executed, the ALTIBASE HDB server first forcibly disconnects currently connected sessions, then rolls back pending transactions and shuts down.

When ALTIBASE HDB is shut down in this way, the following is output:

```
iSQL(sysdba)> shutdown immediate  
Ok..Shutdown Proceeding...  
TRANSITION TO PHASE : Shutdown Altibase  
[RP] Finalization : PASS  
shutdown immediate success.
```

The server can also be forcibly shut down using a server script.

```
$ server stop  
-----  
Altibase Client Query utility.  
Release Version 6.1.1.1
```

```
Copyright 2000, ALTIBASE Corporation or its subsidiaries.  
All Rights Reserved.  
-----  
ISQL_CONNECTION = TCP, SERVER = 127.0.0.1, PORT_NO = 20300  
Alter success.  
Alter success.  
Alter success.  
Ok..Shutdown Proceeding....  
TRANSITION TO PHASE : Shutdown Altibase  
[RP] Finalization : PASS  
shutdown immediate success.
```

4.2.3 SHUTDOWN ABORT

SHUTDOWN ABORT forcibly kills the ALTIBASE HDB server. When ALTIBASE HDB is shut down in this way, the contents of the database will likely be invalid, and thus the next time the server is executed, recovery tasks will have to be performed.

When ALTIBASE HDB is shut down in this way, the following is output:

```
iSQL(sysdba)> shutdown abort  
iSQL(sysdba)>
```

The server can also be forcibly shut down using a server script.

```
$ server kill  
-----  
Altibase Client Query utility.  
Release Version 6.1.1.1  
Copyright 2000, ALTIBASE Corporation or its subsidiaries.  
All Rights Reserved.  
-----  
ISQL_CONNECTION = TCP, SERVER = 127.0.0.1, PORT_NO = 20300  
$
```

4.2 Shutting Down ALTIBASE HDB

5 Objects and Privileges

This chapter describes how to manage objects and privileges in ALTIBASE HDB.

This chapter comprises the following sections:

- [Database Objects](#)
- [Tables](#)
- [Queues](#)
- [Constraints](#)
- [Indexes](#)
- [Views](#)
- [Sequences](#)
- [Synonyms](#)
- [Stored Procedures and Functions](#)
- [Triggers](#)
- [Database Users](#)
- [Privileges](#)

5.1 Database Objects

Database objects are classified as either schema objects, which are managed within particular schema, or non-schema objects, which are managed by ALTIBASE HDB. This chapter describes the characteristics of both schema and non-schema database objects.

5.1.1 Schema Objects

A schema is a logical group of database objects that are owned by a single user and managed using SQL statements. In ALTIBASE HDB, a schema has the name of a user who controls it. The objects contained in a schema are called schema objects. ALTIBASE HDB provides the following kinds of schema objects:

5.1.1.1 Tables

Tables are the basic units of data storage in an Altibase database. Tables are groups of records, each comprising the same number of columns. In ALTIBASE HDB, tables are classified as either memory tables or disk tables, depending on the location in which data are stored. Additionally, tables are classified as either system tables, which are created and managed by the system, or user tables, which are created and managed by general users.

Moreover, special consideration must be given to the management of tables which are to be replicated and tables containing very large amounts of data.

5.1.1.2 Constraints

Constraints are restrictions that are imposed when data are inserted or modified in order to maintain the integrity of data. Constraints are classified as either column constraints or table constraints depending on the target of the constraint, and are also classified as one of the following kinds of constraints depending on the nature of the restrictions:

- NOT NULL / NULL
- UNIQUE KEY
- PRIMARY KEY
- FOREIGN KEY
- TIMESTAMP

5.1.1.3 Indexes

Indexes are optionally created in association with specific tables so that the data in those tables can be accessed more quickly, thereby improving the performance of DML statements.

5.1.1.4 Views

Views do not themselves actually contain any data, but are logical tables that are constructed on the basis of one or more tables or other views. Updatable views and materialized views are not sup-

ported in ALTIBASE HDB.

5.1.1.5 Sequences

ALTIBASE HDB provides sequences for generating unique sequential numerical values.

5.1.1.6 Synonyms

Synonyms are provided as aliases for tables, sequences, views, stored procedures and stored functions so that they can be used without being accessed directly the object name.

5.1.1.7 Stored Procedures and Functions

A procedure or function is a schema object that consists of a set of SQL statements, flow control statements, allocation statements, and error-handling routines. Procedures and functions are permanently saved in the database and allow a complete set of operations corresponding to a single task to be executed merely by calling the relevant procedure or function by name.

Procedures and functions are distinguished from each other in that a function returns a value, while a procedure does not.

5.1.1.8 Database Triggers

A trigger is a special kind of stored procedure that is executed by the system when data are inserted into a table, deleted from a table, or updated, so that a particular task can be automatically executed. Along with constraints, user-defined triggers can help maintain the consistency of the data stored in tables.

5.1.1.9 Database Link

Database Link allows data servers that are in disparate locations but are connected by a network to be integrated so that their data can be combined to output a single desired result.

It is described in details in the *Database Link User's Manual*.

5.1.2 Non-schema Objects

Non-schema objects are objects that are not assigned to any specific schema, but are managed at the database level. ALTIBASE HDB supports the following non-schema objects:

5.1.2.1 Directories

Stored procedures are able to control files, which allows them to read from and write to text files in the file system managed by the operating system. Thanks to this functionality, the user can perform various kinds of tasks using stored procedures, for example, leaving messages in files, reporting the results of files or reading data from files for insertion into tables. The directory object is used to manage information about the directories accessed by stored procedures.

Please refer to the *SQL Reference* for a detailed description of the directory object.

5.1 Database Objects

Please refer to the *Stored Procedures Manual* for a detailed explanation of how to handle files using stored procedures.

5.1.2.2 Replications

A replication is an object that maintains the consistency of the data in tables on different servers by automatically transferring data from a local server to a remote server.

Please refer to the *Replication Manual* for a detailed explanation of how to manage replication.

5.1.2.3 Tablespaces

A database is divided into multiple tablespaces, which are the largest logical data storage unit.

Tablespaces are broadly classified as memory tablespaces and disk tablespaces based on the location in which data are stored. Every database contains the system tablespaces, which are created automatically when a database is created and cannot be deleted. Additionally, users are free to create or delete user tablespaces as required.

For more detailed information on how to manage tablespaces, please refer to [Chapter6: Managing Tablespaces](#).

5.1.2.4 Users

User accounts are necessary in order to connect to ALTIBASE HDB and to function as the owners of schema. Users are created using the system, and are classified either as system users, who manage the system, or as general users.

General users require suitable privileges in order for them to connect to the database and perform operations on data.

5.2 Tables

Tables are the basic units of data storage in an Altibase database. They are constructed of columns and contain multiple rows. This section defines the terminology related to tables and explains the concepts and methods pertaining to table management.

5.2.1 Memory Tables and Disk Tables

Tables are classified as either memory tables or disk tables depending on where the data they contain are stored. When a table is created, whether it is a memory table or a disk table, that is, whether it is to be maintained in memory tablespace or disk tablespace, must be specified.

5.2.2 System Tables and User Tables

Tables are additionally classified as either system tables, which are internally created and managed by the system, and user tables, which are created and managed by users.

System tables, which are also known as the data dictionary, are further classified as either meta tables, in which information about database objects is stored, and process tables, in which information about processes is stored. Process tables are still further classified as either static tables or performance views.

For more information on the data dictionary, please refer to the *General Reference*.

5.2.3 Large Memory Tables

Before executing SQL statements on large memory tables, it is important to understand the following:

5.2.3.1 Altering the Specifications of Large Memory Tables Using DDL

When it is desired to execute a DDL statement on a table containing a large amount of data, rather than executing an ADD COLUMN or DROP COLUMN statement directly on the table, it is preferable to use the iLoader utility to download the data from the table, drop the table, create the table again with the new schema, and then use the iLoader utility to populate the table with the downloaded data.

5.2.3.2 Manipulating Data in Large Memory Tables Using DML

Executing DML statements on tables that do not contain much data does not cause a big problem from the viewpoint of ALTIBASE HDB performance or usage, as long as the data are properly managed. However, when even a single UPDATE or DELETE DML statement affects a large number of records in a table, the transaction associated with this DML statement can take a long time to execute. The occurrence of such slow transactions can cause the following serious problems, which negatively affect the use of ALTIBASE HDB:

5.2 Tables

Exclusive Access to the Table

If a transaction takes a long time to process, other transactions attempting to access the table will be suspended because of the lock held by the transaction that is taking a long time. Moreover, if the size of records being changed exceeds the size specified by the `LOCK_ESCALATION_MEMORY_SIZE` property in the `altibase.properties` file, lock escalation can occur, in which case even other transactions that merely intend to read data can fail to gain access to the table.

Increased ALTIBASE HDB Memory Usage

In ALTIBASE HDB, an SCN (System Commit Number) is used so that the garbage collector can determine which versions of records are to be deleted. The garbage collector only deletes records that have SCNs lower than the SCNs that are being used by transactions that have not been committed. Therefore, transactions that are taking a long time to execute can fool the garbage collector into believing that there are no records to delete, and thus unnecessary records will not be deleted.

Thus, when bulk update/delete transactions take a long time to execute, the garbage collector stops working, and unnecessary versions of records accumulate, which increases the size of the database and thus the amount of memory consumed by ALTIBASE HDB.

Accumulation of Log Files

Log files created by transactions, aside from those logs that are necessary for replication or for restart recovery, are deleted from disk when checkpointing occurs. The log file that is necessary for restart recovery is the oldest of the log files created by transactions that were underway at the time that checkpointing occurred.

Therefore, even after checkpointing has occurred, a transaction that takes a long time to execute can prevent the removal of log files that are necessary for restart recovery, and thus the file system in which the log files are saved may become incapable of storing any additional log files.

5.2.4 Multiplexed Page Lists

In the case of memory tables, when the log file group feature is enabled, the number of page lists that is created is the same as the number of LFGs.

For more detailed information about the Log File Group functionality, please refer to [Chapter 16: Tuning ALTIBASE HDB](#) in this manual.

5.2.5 Replicated Tables

In ALTIBASE HDB, DDL statements can be executed on tables that are to be replicated, but the following properties must first be set as shown:

- Set `REPLICATION_DDL_ENABLE` to 1.
- Set the `REPLICATION` property of the session, which is set using the `ALTER SESSION SET REPLICATION` statement, to some value other than `NONE`.

For more information about managing replicated tables, please refer to the *Replication Manual*.

5.2.6 Creating Tables

Tables can be created using the CREATE TABLE statement.

When creating a table, the following can be specified: column definitions, constraints, the tablespace in which the table is to be saved, the maximum number of records that the table can contain, the rule governing the use of page space in the storage manager for the table, etc.

For more detailed information about the CREATE TABLE statement, please refer to the *SQL Reference*.

5.2.6.1 Example

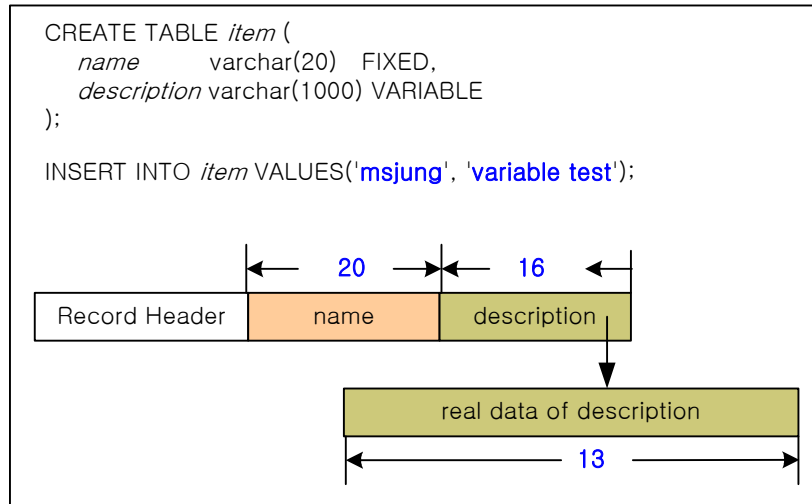
```
CREATE TABLE book(
  isbn CHAR(10) CONSTRAINT const1 PRIMARY KEY SET PERSISTENT = ON,
  title VARCHAR(50),
  author VARCHAR(30),
  edition INTEGER DEFAULT 1,
  publishingyear INTEGER,
  price NUMBER(10,2),
  pubcode CHAR(4)) MAXROWS 2 TABLESPACE user_data;

CREATE TABLE dept_4002
AS SELECT * FROM employees
WHERE dno = 4002;
```

5.2.6.2 Directions for Defining a Column on Memory Table

The user can specify a column of the VARCHAR data type as either FIXED or VARIABLE. If this is not set by the user and the length of the data is shorter than the value set in the MEMORY_VARIABLE_COLUMN_IN_ROW_SIZE property, the data are automatically stored in the FIXED area, otherwise, the data are stored in the VARIABLE area. For a FIXED type column, even though the type is VARCHAR, space for saving data up to the specified length is pre-allocated as it would be for a CHAR data type column, whereas for a VARIABLE type column, the amount of space that is allocated corresponds to the actual length of the data. When VARCHAR type data are compared, the columns are not padded with blank spaces, regardless of whether they are FIXED or VARIABLE type columns.

The following diagram shows how data are saved in columns declared as FIXED or VARIABLE. For a FIXED type column, even though the data type is VARCHAR, space in memory is pre-allocated just as for the CHAR data type, whereas for a VARIABLE type column, memory space corresponding to the actual length of the data is allocated.

Figure 5-1 VARCHAR Column Structure

Because the “name” column in the “item” table was declared as VARCHAR(20) of type FIXED, even though the actual data that are inserted (“msjung”) are only 6 characters long, 20 bytes of space are allocated within the record.

In contrast, because the “description” column in the “item” table was declared as VARCHAR(1000) of type VARIABLE, the amount of space that is allocated to store the value is equal to the actual length of the value that is inserted (“variable test”), which is 13 characters long. However, this is not continuous space within the record, rather, it is a separately allocated space¹. When a VARCHAR type column is declared as VARIABLE type, 16 bytes of space are allocated within the record for storing information about the actual storage location of the data. Therefore, as can be seen in the preceding diagram, the actual amount of space required to hold the value in the “description” column is 29 bytes.

5.2.7 Altering Tables

Using the ALTER TABLE and RENAME statements, table definitions can be altered in the following ways:

- Changing table names
- Adding new columns
- Dropping existing columns
- Specifying column default values
- Changing column names

1. If an amount of memory corresponding to the size of data were allocated every time actual data were stored in a VARCHAR column of type VARIABLE, performance would be affected. Therefore, in ALTIBASE HDB, slots having internally determined sizes, such as 4kB, 8kB and 16kB, are set aside in advance, and the server selects the optimal slot size for saving the data when entering data into a VARIABLE type VARCHAR column.

- Adding constraints
- Dropping constraints
- Compacting memory tables
- Increasing the maximum allowable record count
- Enabling and disabling indexes

5.2.7.1 Examples

```
ALTER TABLE book
  ADD COLUMN (isbn CHAR(10) PRIMARY KEY,
             edition INTEGER DEFAULT 1);
```

```
ALTER TABLE book
  DROP COLUMN isbn;
```

```
ALTER TABLE department
  RENAME COLUMN dno TO dcode;
```

For more information about the ALTER TABLE statement, please refer to the *SQL Reference*.

5.2.8 Dropping Tables

Tables can be dropped (removed) using the DROP TABLE statement.

5.2.8.1 Example

```
DROP TABLE employees;
```

5.2.9 Truncating Tables

Table records can be deleted not only using the DELETE statement, but also using the TRUNCATE TABLE statement. Internally, when the DELETE statement is executed, records are deleted one by one, whereas the TRUNCATE TABLE statement executes the DROP TABLE DDL statement and creates a new table having the identical specification.

Therefore, when executing the TRUNCATE TABLE statement, the entire table is first locked, and, after the TRUNCATE TABLE statement is successfully executed, the data cannot be restored using the ROLLBACK statement.

5.2.10 Data Manipulation

The records in tables can be manipulated using the following DML statements:

- INSERT
- DELETE
- UPDATE

5.2 Tables

- SELECT

Because it is dangerous to perform bulk UPDATE/DELETE operations on large data while ALTIBASE HDB is running, when writing applications using ODBC or APRE, it is advisable to do so in such a way that UPDATE/DELETE operations are first performed on individual records and then committed.

The following is an example of a program authored using the C/C++ Precompiler that avoids bulk UPDATE/DELETE operations and UPDATES records individually.

| | |
|--|---|
| <p>(a) using iSQL to perform a bulk update operation</p> <pre>iSQL>update t1 set coll=2 where coll > 1000;</pre> | <p>(b) using the ALTIBASE HDB C/C++ Precompiler to update individual records</p> <pre>..... EXEC SQL DECLARE update_cursor CURSOR FOR select coll from t1 where coll > 1000; EXEC SQL OPEN update_cursor; while (1) { EXEC SQL FETCH update_cursor INTO :t1_col; if (sqlca.sqlcode == SQL_NO_DATA) break; EXEC SQL update t1 set coll=2 where coll=:t1_col; }.....</pre> |
|--|---|

5.2.11 Related SQL Statements

The following SQL statements are supported for use with tables. For more detailed information, please refer to the *SQL Reference*.

- CREATE TABLE
- ALTER TABLE
- RENAME TABLE
- TRUNCATE TABLE
- LOCK TABLE
- INSERT
- DELETE
- UPDATE
- SELECT

5.3 Queues

The ALTIBASE HDB message queuing function supports asynchronous data transfer between the database and client applications. Queue tables are database objects, just like other database tables, and thus can be controlled using DDL and DML statements.

5.3.1 Creating Queues

When the user uses the CREATE QUEUE statement to create a queue, the database creates a table having the name specified by the user. This is called a queue table. Queue tables have the following structure:

| Column | Data Type | Length | Default | Description |
|--------------|-----------|----------------|---------|---|
| MSGID | BIGINT | 8 | - | A message identifier, set by ALTIBASE HDB automatically |
| CORRID | INTEGER | 4 | 0 | A message identifier set by the user |
| MESSAGE | VARCHAR | Message length | - | The message text |
| ENQUEUE_TIME | DATE | 8 | SYSDATE | The time that the message was added to the queue |

The user cannot freely change the name of the queue table or the names of its columns. A primary key is automatically created in the MSGID column.

The database internally creates a sequence called *queue_name_NEXT_MSG_ID* to generate unique MSGID values. The user can use the SYSTEM_.SYS_TABLES_ meta table to view information about this sequence.

Because the sequence must be maintained until the queue table is deleted, the DROP SEQUENCE statement cannot be used to expressly remove the sequence.

Queue tables are saved as type 'Q' in the SYSTEM_.SYS_TABLES meta table. Indexes can be created for queue tables as desired using the CREATE INDEX statement.

5.3.1.1 Example

```
CREATE QUEUE Q1(40);
```

5.3.2 Alter Queues

The structure of queue tables, which are created using the CREATE QUEUE statement, cannot be changed using an ALTER TABLE statement or the like. Queues can only be removed using the DROP QUEUE statement. However, the user can manipulate the data in queues using statements such as ENQUEUE/DEQUEUE, DELETE, SELECT, etc.

5.3 Queues

5.3.3 Removing Queues

Queue tables can be removed from the database using the DROP QUEUE statement.

5.3.3.1 Example

```
iSQL> DROP QUEUE Q1;
```

5.3.4 Deleting Data

The TRUNCATE TABLE statement can be used when it is desired only to delete all of the messages loaded into a queue.

5.3.4.1 Example

```
iSQL> TRUNCATE TABLE Q1;
```

5.3.5 Data Manipulation

The records in queue tables can be manipulated using the following SQL statements:

- ENQUEUE
- DEQUEUE
- DELETE
- SELECT

5.3.6 Related SQL Statements

The following SQL statements are provided for use with queue tables. For more detailed information about queues, please refer to the *SQL Reference*.

- CREATE QUEUE
- DROP QUEUE
- ENQUEUE
- DEQUEUE

5.4 Constraints

Constraints are limitations that govern the insertion of data into tables and the changes that can be made to existing data. This section explains the kinds of constraints and how to use them to ensure data consistency.

5.4.1 Types

The following kinds of constraints are supported in ALTIBASE HDB:

5.4.1.1 NOT NULL

This constraint prevents NULL values from being inserted into columns. It can be set for individual columns. If it is set to NULL, then NULL values are permitted. If NOT NULL is not expressly set for a column, the default is to allow NULL values.

5.4.1.2 UNIQUE KEY

This constraint, which can be defined for one or more columns, prevents the insertion of duplicate values into one or more columns. A unique index is created when a unique key constraint is defined.

5.4.1.3 PRIMARY KEY

The Primary Key constraint can be thought of as a combination of the Unique Key constraint and the NOT NULL constraint. A primary key constraint can be defined for one or more columns. When it is created, a unique index is created internally. NULL values cannot be entered in any of the columns included in the primary key constraint.

5.4.1.4 FOREIGN KEY

A FOREIGN KEY constraint requires each value in a column or set of columns to match a value in an associated table's UNIQUE or PRIMARY KEY. A FOREIGN KEY constraint helps protect referential integrity.

5.4.1.5 TIMESTAMP

This constraint sets the value of a column to the system time when a new record is inserted or an existing record is updated. A TIMESTAMP constraint is usually set for one column of a table that is replicated.

5.4.2 Column Constraints and Table Constraints

A column constraint is a constraint that is set for a single column, whereas a table constraint is a single constraint that is set for the entire table, and applies to multiple columns in the table.

The NOT NULL and TIMESTAMP constraints can only be used as column constraints, but the other kinds of constraints can be set as either column constraints or table constraints.

5.4 Constraints

5.4.3 Creating Constraints

The user can define a constraint when creating a table using the CREATE TABLE statement or altering a table using the ALTER TABLE statement. The user can specify the name of a constraint when defining the constraint. If the user does not set the name of the constraint, the system will automatically assign a name. If the constraint is of a type that requires an index, the system automatically creates the index and assigns a name to it.

5.4.3.1 Example

```
CREATE TABLE inventory(  
    subscriptionid CHAR(10),  
    isbn CHAR(10),  
    storecode CHAR(4),  
    purchasedate DATE NOT NULL,  
    quantity INTEGER,  
    paid CHAR(1),  
    PRIMARY KEY(subscriptionid, isbn),  
    CONSTRAINT fk_isbn FOREIGN KEY(isbn, storecode) REFERENCES book(isbn, store-  
code))  
TABLESPACE user_data;  
  
ALTER TABLE book  
ADD CONSTRAINT const1 UNIQUE(bno);
```

5.4.4 Drop Constraints

A constraint can be removed using the ALTER TABLE statement.

5.4.4.1 Example

```
ALTER TABLE book DROP UNIQUE(bno);
```

5.4.5 Related SQL Statements

The following SQL statements are supported for use with constraints. For more information, please refer to the *SQL Reference*.

- CREATE TABLE
- ALTER TABLE

5.5 Indexes

Indexes allow the records in tables to be accessed more quickly. This section describes the types of indexes that are supported in ALTIBASE HDB and how to manage and use index objects.

5.5.1 Index Types

ALTIBASE HDB supports two types of indexes: B-tree indexes and R-tree indexes. The R-tree index is a multi-dimensional index type for use with spatial queries.

5.5.1.1 B-Tree Indexes

B-tree indexes are used with all data types except the GEOMETRY data type, which is a spatial data type. B-tree indexes have historically been used with DBMSs, and many variants thereof have arisen over the years due to the large amount of research that has been conducted to date. Of these variants, ALTIBASE HDB supports the B⁺-tree index type.

A B⁺-tree index comprises leaf nodes at the lowest index level, a root node at the highest level, and internal nodes in between the root and leaf nodes. Key values exist only for all leaf nodes, and root and index nodes comprise separator keys between left child nodes and right child nodes.

5.5.1.2 R-Tree Indexes

R-Tree Indexes are used with the GEOMETRY spatial data type.

When finding target objects using R-tree indexes, the following procedure is used:

1. Conditional filtering is conducted using the MBR (Minimum Bounding Rectangle) that covers each spatial object.
2. "Refinement", which is checking for accurate index search conditions about objects that remain after step 1, is conducted.

The algorithms for adding, deleting, splitting and merging nodes in R-Tree Indexes are similar to those for B-tree Indexes, except that they are based on MBRs.

5.5.2 Index Attributes

Based on how a key column is configured when an index is created and on the attributes of the key column, an index has the following attributes. The use of persistent indexes as memory table indexes is supported in order to reduce the boot-up time when the system is started.

5.5.2.1 Unique Index

This index prevents the use of duplicate values in indexed columns.

5.5.2.2 Unique Keys vs. Primary Keys

Unique Keys and Primary Keys are alike in that neither of them permits the existence of duplicate

5.5 Indexes

values. However, they differ in whether they permit NULL values. Primary Keys do not permit NULL values.

5.5.2.3 Non-Unique Index

This index type permits duplicate values in index columns. If the UNIQUE KEY option is not set when an index is created, the default is to allow duplicate values.

5.5.2.4 Non-Composite Index

This kind of index is based on only one column.

5.5.2.5 Composite Index

When a single index is created based on multiple columns, it is called a Composite Index.

5.5.2.6 Persistent Index

A memory table index exists not in a persistent database area but in a temporary memory area. As a result, all memory table indexes need to be rebuilt whenever the system is restarted. For exceedingly large databases, the time taken to rebuild indexes increases in proportion to the size of the database.

To prevent this, indexes that exist in temporary memory space are stored as permanent indexes in files when the system is shut down normally. Then, when the system is restarted, the indexes are restored from these files, thereby reducing boot-up time. However, in this case, ALTIBASE HDB builds the index based on key pointers, which may take a longer time than building the index based on key values. Furthermore, additional disk space for storing the index files is required. Therefore, it is to be understood that persistent indexes are especially intended for use in the following cases:

- in cases where the index key value size is so large that no performance benefit results from building indexes based on key values
- in cases where fast startup times are highly desirable

5.5.2.7 Non-Persistent Index

This is a normal index, and is not stored when the system is shut down normally. If the PERSISTENT=ON option is not specified when the index is created, an index is a non-persistent index by default.

5.5.3 Index Management

Indexes are used to enable quicker access to the records in tables. Because an index is an object that is physical and logically independent from a table, it can be built, deleted or changed without consideration for the table on which it is based.

If table records are changed, the corresponding indexes are also changed. Therefore, the user should create indexes only when necessary, and should modify or delete them so that they are managed optimally based on the way in which the associated tables are accessed.

5.5.3.1 Creating Indexes

An index is created for one or more columns in the table. Indexes are created automatically when constraints are defined, or users can explicitly create indexes using the CREATE INDEX statement.

Example

- Creating an index by defining a table constraint:

```
CREATE TABLE TB1 (C1 INTEGER PRIMARY KEY, C2 INTEGER UNIQUE);
```

- Creating an index by changing a table constraint:

```
ALTER TABLE TB1 ADD PRIMARY KEY (C1);
ALTER TABLE TB1 ADD UNIQUE (C2);
```

- Specifying the order of columns when creating a composite index:

```
CREATE INDEX TB1_IDX1 ON TB1 (C1 ASC, C2 DESC);
```

- Creating a index using the INDEXTYPE option to specify the type of index:

```
CREATE INDEX TB1_IDX1 ON TB1 (C1) INDEXTYPE IS BTREE;
```

- Creating a unique index using the UNIQUE option:

```
CREATE UNIQUE INDEX TB1_IDX ON TB1 (C1);
```

- Creating a PERSISTENT index:

```
CREATE INDEX TB1_IDX1 ON TB1 (C1) SET PERSISTENT=ON;
```

5.5.3.2 Options for Creating Disk B-Tree Indexes (NOLOGGING, NOFORCE)

When a disk B-tree index is built, a log is recorded so that it can be used to recover the index in the event of a system error. In order to reduce the size of logs and the amount of time taken to build an index, the NOLOGGING option can be specified when the index is created.

When the NOLOGGING option is used, all pages of an index are written to disk immediately after the index is built, thus ensuring the consistency of the index after it is built, even if a system fault occurs.

However, when indexes are created with the NOLOGGING option, if the NOFORCE option, which specifies that index pages are not to be written to disk immediately, is also specified, although the time required to build the index is reduced, index consistency may be lost if a system or media fault occurs. Media backup should be conducted in order to ensure the durability of indexes that are created with both the NOLOGGING and NOFORCE options.

| | Total Index Build Time | Consistency & Durability |
|-----------------|---|--|
| LOGGING | Index Building Time + Logging Time | Recoverable when a system or media fault occurs |
| NOLOGGING FORCE | Index Building Time + Time Taken to Write Index to Disk | Recoverable when a system fault occurs, but consistency may be lost when a media fault occurs. |

5.5 Indexes

| | Total Index Build Time | Consistency & Durability |
|-------------------|------------------------|--|
| NOLOGGING NOFORCE | Index Building Time | Consistency may be lost when a system or media fault occurs. |

Example

- Creating an index that is not logged and write the index to disk:

```
CREATE INDEX TB1_IDX1 ON TB1 (C1) NOLOGGING;  
or  
CREATE INDEX TB1_IDX1 ON TB1 (C1) NOLOGGING FORCE;
```

- Creating an index that is not logged (NOLOGGING) and that is not written to disk after being built (NOFORCE):

```
CREATE INDEX TB1_IDX1 ON TB1 (C1) NOLOGGING NOFORCE;
```

5.5.3.3 Modifying Indexes

The attributes of an index, such as whether it is active and whether it is permanent, can be changed using the ALTER INDEX statement.

Example

```
ALTER INDEX EMP_IDX1 SET PERSISTENT = ON;
```

5.5.3.4 Dropping Indexes

An index can be removed explicitly using the DROP INDEX statement, or implicitly by removing the associated constraint.

Example

```
DROP INDEX emp_idx1;
```

5.5.4 Using Indexes

5.5.4.1 Bottom-Up Index Building

In ALTIBASE HDB, indexes are built from the bottom up. Therefore, it is more efficient to build indexes after data have been uploaded. If a large volume of data is inserted into a table for which an index has been built, slow performance may result, because each time a record is inserted, the index will need to be changed to reflect this.

5.5.4.2 Disk Index Consistency

For disk table indexes created with the NOLOGGING option, index consistency cannot be guaranteed in the event of a system or media fault. If such a fault occurs, use the V\$DISK_BTREE_HEADER performance view to check the consistency of disk indexes. If an index for which IS_CONSISTENT is

set to 'F' is found, delete the index and rebuild it when it is needed.

5.5.5 Related SQL Statements

The following SQL statements are supported for use with indexes. For more information, please refer to the *SQL Reference*.

- CREATE TABLE
- ALTER TABLE
- CREATE INDEX
- ALTER INDEX
- DROP INDEX

5.6 Views

A view is a presentation of data from one or more tables. A view contains no actual data, but rather presents data from the tables and views on which it is based. Views can be thought of as logical tables. This section describes how to manage views.

5.6.1 Base Tables and Views

So-called “base tables” are just the objects (tables or other views) that views access, and from which they read data. More than one base table can be associated with a single view.

In ALTIBASE HDB, only read-only views are supported. Updatable views and materialized views are not supported.

5.6.2 Creating Views

Views can be created using the CREATE VIEW statement.

5.6.2.1 Example

```
CREATE VIEW avg_sal AS
  SELECT DNO, AVG(salary) emp_avg_sal
  -- salary average of each department
  FROM employees
  GROUP BY dno;
```

5.6.3 Modifying Views

Use the CREATE OR REPLACE VIEW statement to change the contents of an existing view, that is, change its underlying SELECT query statement.

5.6.3.1 Example

```
CREATE OR REPLACE VIEW emp_cus AS
  SELECT DISTINCT o.eno, e.ename, c.cname
  FROM employees e, customers c, orders o
  WHERE e.eno = o.eno AND o.cno = c.cno;
```

Because views are based on base tables, when the definition of a base table is changed using a DDL statement, any views based on the table may become invalid, that is, unable to be viewed. In such cases, the ALTER VIEW statement can be used with the COMPILE option to recompile the view so that it is valid.

Additionally, the RENAME statement can be used in cases where it is desired only to change the name of a view.

5.6.3.2 Example

```
ALTER VIEW avg_sal COMPILE;
```

5.6.4 Dropping Views

Views can be removed using the DROP VIEW statement.

5.6.4.1 Example

```
DROP VIEW avg_sal;
```

5.6.5 Related SQL Statements

The following SQL statements are supported for use with views. For more information on these statements, please refer to the *SQL Reference*.

- CREATE VIEW
- ALTER VIEW
- DROP VIEW
- SELECT

5.7 Sequences

In ALTIBASE HDB, the Sequence object is provided for use as a generator of sequences of unique numbers. Next sequence values can be cached to ensure consistent performance.

5.7.1 Using Sequences

The sequence generator is particularly useful in multiuser environments for generating sequences of unique numbers without the overhead of disk I/O or transaction locking. For example, assume two users are simultaneously inserting new records into a table called "orders". By using a sequence to generate unique order numbers for the `order_id` column, neither user has to wait for the other to enter the next available order number. The sequence automatically generates a unique value for each user.

A sequence is generally used to generate a key value that is set in a desired column using a DML statement. The expressions `sequence_name.NEXTVAL` and `sequence_name.CURRVAL` are used to access the sequence.

- `sequence_name.NEXTVAL` is used to obtain the next value in the sequence
- `sequence_name.CURRVAL` is used to obtain the current value in the sequence

After a sequence is created, the first time it is executed, the sequence's `sequence_name.CURRVAL` value cannot be used. In order to use the `sequence_name.CURRVAL` value for a newly created sequence, the `sequence_name.NEXTVAL` value must first be accessed.

Every time the sequence's nextval value is accessed, the value of the sequence increments internally by the amount specified. The increment of the sequence is explicitly specified using the `INCREMENT BY` option when the index is created, and defaults to 1 if not specified.

5.7.2 Using Sequences in INSERT Statements

The following example shows how to generate a key value using a sequence and insert it into a table:

5.7.2.1 Example

```
iSQL> create sequence seq1;  
iSQL> insert into t1 values (seq1.nextval);
```

In the above example, supposing that the sequence has been newly created, its initial value of 1 will be entered into table t1, and `seq1.nextval` will increase from 1 to 2.

5.7.3 Creating Sequences

The `CREATE SEQUENCE` statement is used to create a sequence. The following options can be used when creating a sequence:

- `START WITH`

This is the starting value of the sequence.

- INCREMENT BY

This is the amount by which the sequence increases or decreases.

- MAXVALUE

This is the maximum value of the sequence.

- MINVALUE

This is the minimum value of the sequence.

- CYCLE

This option is specified to ensure that the sequence will continue to generate values when it reaches its maximum or minimum value. The sequence cycles again from the minimum value in the case of an ascending sequence, or from the maximum value in the case of a descending sequence.

- CACHE

Sequence values can be created in advance and cached in memory so that they can be returned more quickly. The number of sequence values cached in this way is equal to the value specified using the CACHE option. The cache is populated when a key value is first requested from a new sequence, and is accessed every time the next key value is subsequently requested from the sequence. After the last sequence value in the cache has been used, the next request for a key value from the sequence causes new sequence values to be created and cached in memory. Then the first value is returned from this new cache. When a sequence is created, the default CACHE value is 20.

5.7.3.1 Example

- Creating a basic sequence (starting from 1 and incrementing by 1):

```
CREATE SEQUENCE seq1;
```

- Creating a sequence that generates even numbers and cycles from 0 to 100:

```
CREATE SEQUENCE seq1
START WITH 0
INCREMENT BY 2
MINVALUE 0
MAXVALUE 100
CYCLE;
```

5.7.4 Modifying Sequences

All sequence options except for the START WITH value can be modified using the ALTER SEQUENCE statement.

5.7.4.1 Example

```
ALTER SEQUENCE seq1
```

5.7 Sequences

```
INCREMENT BY 1  
MINVALUE 0  
MAXVALUE 100;
```

5.7.5 Dropping Sequences

Sequences can be removed as desired using the DROP SEQUENCE statement.

5.7.5.1 Example

```
DROP SEQUENCE seq1;
```

5.7.6 Related SQL Statements

The following SQL statements are provided for use with sequences. For more information, please refer to the *SQL Reference*.

- CREATE SEQUENCE
- ALTER SEQUENCE
- DROP SEQUENCE

5.8 Synonyms

ALTIBASE HDB supports the use of synonyms as aliases for tables, views, sequences, stored procedures, or stored functions. Synonyms require no storage space other than that used to store their definitions in the data dictionary.

5.8.1 Using Synonyms

Database synonyms are advantageously used in the following cases:

- when it is desired to conceal the original name of a particular object or the identity of the user who created it,
- to simplify the use of a SQL statement, and
- to minimize the changes that must be made to applications in order for them to be used by various users.

5.8.2 Creating Synonyms

Synonyms are created using the CREATE SYNONYM statement.

5.8.2.1 Example

To create the synonym "my_dept" as an alias for the table "dept":

```
CREATE SYNONYM my_dept FOR dept;
```

5.8.3 Dropping Synonyms

Synonyms can be removed as desired using the DROP SYNONYM statement.

5.8.3.1 Example

To remove the synonym "my_dept":

```
DROP SYNONYM my_dept;
```

5.8.4 Related SQL Statements

The following SQL statements are provided for use with synonyms. For more information, please refer to the *SQL Reference*.

- CREATE SYNONYM
- DROP SYNONYM

5.9 Stored Procedures and Functions

A stored procedure is a set of SQL statements, flow control statements, assignment statements, error handling routines etc. that are programmed in a single module that corresponds to a complete business task. The module is permanently stored in the database as a database object so that the entire business task can be conducted merely via the single action of calling the module on the server by name. This chapter describes how to manage stored procedures.

Stored procedures and stored functions differ in that stored functions return a value to the caller, while stored procedures do not. Because they are identical in all other respects, explanations of stored procedures can also be understood to apply to stored functions unless otherwise noted.

This chapter provides simple examples that illustrate how to manage stored procedures. For a more detailed explanation of the terminology, concepts, and management of stored procedures and stored functions, please refer to the *Stored Procedures Manual*.

5.9.1 Categories

5.9.1.1 Stored Procedures

A stored procedure is a database object that executes multiple SQL statements at one time in consideration of input, output, and input/output parameters according to conditions defined in its body. It does not have a return value, and thus sends values to the client using output or input/output parameters. Because it does not have a single return value, it cannot be used as an operand within an expression in another SQL statement.

5.9.1.2 Stored Functions

A stored function is identical to a stored procedure except that it has a return value. Because it differs from a stored procedure in this way, it can be used as an operand in an expression within another SQL statement, just like the functions provided by the system.

5.9.1.3 Typesets

A typeset is a set of user-defined types used within a stored procedure. Typesets are usually used when stored procedures exchange user-defined types, that is, parameters and return values, with each other.

5.9.2 SQL Statements Related to Stored Procedures

The following table shows the kinds of SQL statements that can be used with stored procedures.

| Task | Statement | Description |
|-----------|-------------------------------|---|
| CREATE | CREATE [OR REPLACE] PROCEDURE | Creates a new stored procedure or redefines an existing stored procedure |
| | CREATE [OR REPLACE] FUNCTION | Creates a new stored function or redefines an existing stored function |
| | CREATE [OR REPLACE] TYPESET | Creates or modifies a typeset |
| ALTER | ALTER PROCEDURE | If the definitions of objects referred to in a stored procedure are changed after the stored procedure has been created, the current stored procedure execution plan tree may not be optimized. In such cases, this statement recompiles the stored procedure to create an optimized execution plan tree. |
| | ALTER FUNCTION | If the definitions of objects referred to in a stored function are changed after the stored function has been created, the current stored function execution plan tree may not be optimized. In such cases, this statement recompiles the stored function to create an optimized execution plan tree. |
| DROP | DROP PROCEDURE | Removes a previously created stored procedure |
| | DROP FUNCTION | Removes a previously created stored function |
| | DROP TYPESET | Removes a previously created typeset |
| EXECUTION | EXECUTE | Executes a stored procedure or stored function |
| | <i>function_name</i> | Executes a stored function within an SQL statement, just like a built-in function |

5.9.3 Creating Stored Procedures

A stored procedure can be created using the CREATE PROCEDURE statement.

5.9.3.1 Example

```
CREATE PROCEDURE proc1
(p1 IN INTEGER, p2 IN INTEGER, p3 IN INTEGER)
AS
  v1 INTEGER;
  v2 t1.i2%type;
  v3 INTEGER;
BEGIN
  SELECT *
  INTO v1, v2, v3
  FROM t1
  WHERE i1 = p1 AND i2 = p2 AND i3 = p3;
```

5.9 Stored Procedures and Functions

```
IF v1 = 1 AND v2 = 1 AND v3 = 1 THEN
  UPDATE t1 SET i2 = 7 WHERE i1 = v1;
ELSIF v1 = 2 AND v2 = 2 AND v3 = 2 THEN
  UPDATE t1 SET i2 = 7 WHERE i1 = v1;
ELSIF v1 = 3 AND v2 = 3 AND v3 = 3 THEN
  UPDATE t1 SET i2 = 7 WHERE i1 = v1;
ELSIF v1 = 4 AND v2 = 4 AND v3 = 4 THEN
  UPDATE t1 SET i2 = 7 WHERE i1 = v1;
ELSE -- ELSIF v1 = 5 AND v2 = 5 AND v3 = 5 THEN
  DELETE FROM t1;
END IF;

INSERT INTO t1 VALUES (p1+10, p2+10, p3+10);
END;
/
```

5.9.4 Modifying Stored Procedures

- To alter the parameter or main body of a stored procedure without changing the name of the stored procedure, use the CREATE OR REPLACE PROCEDURE statement to re-create the stored procedure.

Ex)

```
CREATE OR REPLACE PROCEDURE proc1
(p1 IN INTEGER, p2 IN INTEGER, p3 IN INTEGER)
AS
  v1 INTEGER;
  v2 t1.i2%type;
  v3 INTEGER;
BEGIN
  .
  .
  .
END;
/
```

- When tables and sequences referred to by a stored procedure, or other stored procedures or stored functions called by the stored procedure, are changed so that they differ from their definitions at the time that they were created, it may become impossible to execute the stored procedure according to its execution plan tree. In such cases, the stored procedure is said to be in an invalid state.

For example, in the case where an index that existed when a stored procedure was first created is deleted, it will become impossible to use the old execution plan to access the table if the old execution plan tree used the deleted index to access the table.

The ALTER PROCEDURE statement is used to recompile the invalid stored procedure and thereby create a new, valid execution plan.

Ex)

```
ALTER PROCEDURE proc1 COMPILE;
```

5.9.5 Dropping Stored Procedures

Stored procedures can be removed using the DROP PROCEDURE statement.

5.9.5.1 Example

```
DROP PROCEDURE proc1;
```

5.9.6 Related SQL Statements

The following SQL statements are supported for use with stored procedures and stored functions. For more information, please refer to the *SQL Reference*.

- CREATE PROCEDURE
- CREATE FUNCTION
- CREATE TYPESET
- ALTER PROCEDURE
- ALTER FUNCTION
- DROP PROCEDURE
- DROP FUNCTION
- DROP TYPESET
- EXECUTE
- *function_name*

5.10 Triggers

A trigger is a special kind of stored procedure that is automatically executed (or “fired”) by the system in order to accomplish a particular task when data are inserted into a table, deleted from a table, or modified. This section describes how to manage triggers.

5.10.1 The Constituents of Triggers

The following trigger constituents determine when a trigger fires, whether it fires, and what it executes.

- **Trigger Event:** This is the SQL statement that causes the trigger to fire when executed.
- **Trigger Condition (WHEN clause):** This is a SQL condition that must be satisfied to fire the trigger.
- **Trigger Action:** This is the body of the stored procedure that the trigger executes when the trigger condition is TRUE.

5.10.2 Trigger Events

One of three DML statements can be specified as the event that causes the trigger to fire.

- **DELETE:** Specify DELETE to tell the trigger to fire whenever a row is removed from the table using a DELETE statement.
- **INSERT:** Specify INSERT to tell the trigger to fire whenever a row is added to the table using an INSERT statement.
- **UPDATE:** Specify UPDATE to tell the trigger to fire whenever data in the table are changed using an UPDATE statement. If an OF clause is present in the UPDATE trigger event, the trigger fires only if data in the columns explicitly named in the OF clause are changed.

Note: In order to maintain the integrity of the database, changes made to tables by replication will not be processed as trigger events.

5.10.3 Creating Triggers

Triggers can be created using the CREATE TRIGGER statement.

5.10.3.1 Example

```
CREATE TRIGGER del_trigger
  AFTER DELETE ON orders
  REFERENCING OLD ROW old_row
  FOR EACH ROW
  AS BEGIN
    INSERT INTO log_tbl VALUES(old_row.ono, old_row.cno, old_row.qty,
    old_row.arrival_date, sysdate);
  END;
/
```

5.10.4 Modifying Triggers

The ALTER TRIGGER statement can be used to disable the execution of an existing trigger or recompile an invalid trigger. When a trigger is first created, it is automatically enabled by default. It can be subsequently disabled and enabled using the DISABLE and ENABLE clause with the ALTER TRIGGER statement.

5.10.4.1 Example

```
ALTER TRIGGER del_trigger DISABLE;
```

5.10.5 Dropping Triggers

The DROP TRIGGER statement can be used to remove a trigger from the database.

5.10.5.1 Example

```
DROP TRIGGER del_trigger;
```

5.10.6 Related SQL Statements

The following SQL statements are supported for use with triggers. For more information, please refer to the *SQL Reference*.

- CREATE TRIGGER
- ALTER TRIGGER
- DROP TRIGGER

Additionally, because a trigger is a kind of stored procedure, for a detailed description of the trigger body, please refer to the *Stored Procedures Manual*.

5.11 Database Users

In a newly created database, only the system administrators, that is, the SYSTEM_ and SYS users, exist. Because these users are DBAs (database administrators), in order to construct general schema, it is necessary to create general users to manage schema objects. This section explains how to create and manage users.

5.11.1 The SYSTEM_ and SYS Users

Database users can be classified into system administrators, who are created by the system when the database is created, and general users.

The system administrators comprise the SYSTEM_ user, who is the owner of meta tables and thus has the right to execute DDL and DML statements on meta tables, and the SYS user, a DBA (database administrator) who possesses all rights for normal tables and the right to conduct all tasks at the system level. These users cannot be modified or removed using the DDL statements.

5.11.2 Creating Users

Users can be created using the CREATE USER statement. It is necessary to have the CREATE USER system privilege in order to execute this statement. When a user is created using the CREATE USER statement, a password must be specified. Additionally, the default tablespace for user-created objects can also be specified.

5.11.2.1 Example

```
CREATE USER dlr IDENTIFIED BY dlr123
DEFAULT TABLESPACE user_data
TEMPORARY TABLESPACE temp_data
ACCESS sys_tbs_memory ON;
```

5.11.3 Modifying Users

The ALTER USER statement can be used to change a user's password or modify tablespace settings.

5.11.3.1 Example

- To change a user's password:

```
ALTER USER dlr IDENTIFIED BY dlr12345;
```

- To change a user's default tablespace:

```
ALTER USER dlr DEFAULT TABLESPACE dlr1_data;
```

- To change a user's temporary tablespace:

```
ALTER USER dlr TEMPORARY TABLESPACE dlr1_tmp;
```

- To change a user's access rights for a particular tablespace:

```
ALTER USER dlr ACCESS dlr2_data ON;
```

5.11.4 Dropping Users

To remove a user, use the DROP USER statement. Additionally, to simultaneously delete all of the objects owned by the user, use the CASCADE option. Executing the DROP USER statement without the CASCADE option while there are objects remaining in the user's schema will result in an error.

5.11.4.1 Example

```
DROP USER dlr CASCADE;
```

5.11.5 Related SQL Statements

The following SQL statements are provided for use on users. For more information, please refer to the *SQL Reference*.

- CREATE USER
- ALTER USER
- DROP USER

5.12 Privileges

In order to access the objects or data in a database, the user must have appropriate privileges. This chapter describes user and object privileges in ALTIBASE HDB and how to manage these privileges.

5.12.1 The Kinds of Privileges

Although ALTIBASE HDB provides functionality to support the management of user privileges, there is no provision for managing so-called “roles”, which are sets of privileges. This section describes the kinds of privileges supported by ALTIBASE HDB.

5.12.2 System Privileges

System privileges are usually managed by the DBA. Users with system privileges can perform all database tasks and access all objects in all schemas.

A complete list of the system access privileges supported in ALTIBASE HDB is provided in the following table. For more information about each privilege, please refer to the *SQL Reference*.

| System Privilege | SQL Statement |
|------------------|-----------------------|
| DATABASE | ALTER SYSTEM |
| INDEXES | CREATE ANY INDEX |
| | ALTER ANY INDEX |
| | DROP ANY INDEX |
| PROCEDURES | CREATE PROCEDURE |
| | CREATE ANY PROCEDURE |
| | ALTER ANY PROCEDURE |
| | DROP ANY PROCEDURE |
| | EXECUTE ANY PROCEDURE |
| SEQUENCES | CREATE SEQUENCE |
| | CREATE ANY SEQUENCE |
| | ALTER ANY SEQUENCE |
| | DROP ANY SEQUENCE |
| | SELECT ANY SEQUENCE |
| SESSIONS | CREATE SESSION |

| System Privilege | SQL Statement |
|------------------|----------------------|
| TABLES | CREATE TABLE |
| | CREATE ANY TABLE |
| | ALTER ANY TABLE |
| | DELETE ANY TABLE |
| | DROP ANY TABLE |
| | INSERT ANY TABLE |
| | LOCK ANY TABLE |
| | SELECT ANY TABLE |
| | UPDATE ANY TABLE |
| TABLESPACES | CREATE TABLESPACE |
| | ALTER TABLESPACE |
| | DROP TABLESPACE |
| USERS | CREATE USER |
| | ALTER USER |
| | DROP USER |
| VIEWS | CREATE VIEW |
| | CREATE ANY VIEW |
| | DROP ANY VIEW |
| MISCELLANEOUS | GRANT ANY PRIVILEGES |
| TRIGGER | CREATE TRIGGER |
| | CREATE ANY TRIGGER |
| | ALTER ANY TRIGGER |
| | DROP ANY TRIGGER |

5.12.3 Object Privileges

The privileges for access to objects are managed by the owner of the object. These privileges govern access to, and manipulation of, objects.

The object access privileges supported in ALTIBASE HDB are shown in the following table.

| Object Privilege | Table | Sequence | PSM | View |
|------------------|-------|----------|-----|------|
| ALTER | O | O | | |

5.12 Privileges

| Object Privilege | Table | Sequence | PSM | View |
|------------------|-------|----------|-----|------|
| DELETE | O | | | |
| EXECUTE | | | O | |
| INDEX | O | | | |
| INSERT | O | | | |
| REFERENCES | O | | | |
| SELECT | O | O | | O |
| UPDATE | O | | | |

5.12.4 Granting Privileges

The GRANT statement is used to explicitly grant rights to particular users.

The SYSTEM_ and SYS users are DBAs (database administrators), and thus possess all rights. In contrast, rights must be explicitly granted to general users as desired.

When general users are created using the CREATE USER statement, the system automatically grants them the following rights:

- CREATE SESSION
- CREATE TABLE
- CREATE SEQUENCE
- CREATE PROCEDURE
- CREATE VIEW

5.12.4.1 Example

- Granting system privileges:

```
GRANT ALTER ANY SEQUENCE, INSERT ANY TABLE, SELECT ANY SEQUENCE TO uare5;
```

- Granting object privileges:

```
GRANT SELECT, DELETE ON sys.employees TO uare8;
```

5.12.5 Revoking Privileges

Privileges that have previously been granted to users can be removed using the REVOKE statement.

5.12.5.1 Example

- Revoking system privileges:

```
REVOKE ALTER ANY TABLE, INSERT ANY TABLE, SELECT ANY TABLE, DELETE ANY  
TABLE FROM uare10;
```

- Revoking object privileges:

```
REVOKE SELECT, DELETE ON sys.employees FROM uare7, uare8;
```

5.12.6 Related SQL Statements

The following SQL statements are provided for use in managing privileges. For more information, please refer to the *SQL Reference*.

- GRANT
- REVOKE

5.12 Privileges

6 Managing Tablespaces

This chapter explains the tablespace concept, describes the structure of tablespaces and functions supported for use with them, and presents information that administrators should be familiar with in order to manage tablespaces efficiently.

This chapter contains the following sections:

- [Tablespaces: Definition and Structure](#)
- [Classifying Tablespaces](#)
- [Disk Tablespace](#)
- [The Undo Tablespace](#)
- [Tablespace States](#)
- [Managing Tablespaces](#)
- [Examples of Tablespace Use](#)
- [Managing Space in Tablespaces](#)

6.1 Tablespaces: Definition and Structure

This section describes what a tablespace is. It also examines the relationship between tablespaces and databases, and describes the respective structures of disk tablespaces, memory tablespaces and volatile tablespaces.

6.1.1 What Is a Tablespace?

A tablespace is a logical storage space for storing tables, indexes and other database objects. A database usually requires at least one tablespace in order to operate correctly. The system tablespaces are created automatically when a database is created. Additionally, the user can also create user-defined tablespaces as desired.

In ALTIBASE HDB, user-defined tablespaces are classified as disk tablespaces, in which database objects reside on disk, memory tablespaces, in which the objects reside in memory, and volatile tablespaces, in which the objects reside in memory and for which logging is not performed. The user determines which kind of tablespace to use depending on the characteristics of the data to be stored in the tablespace.

Disk tablespaces are an appropriate choice for large volumes of data, such as for example historical data, while memory tablespaces are suitable for small volumes of data that are accessed frequently, and volatile tablespaces are appropriate for storing data temporarily so that they can be processed quickly.

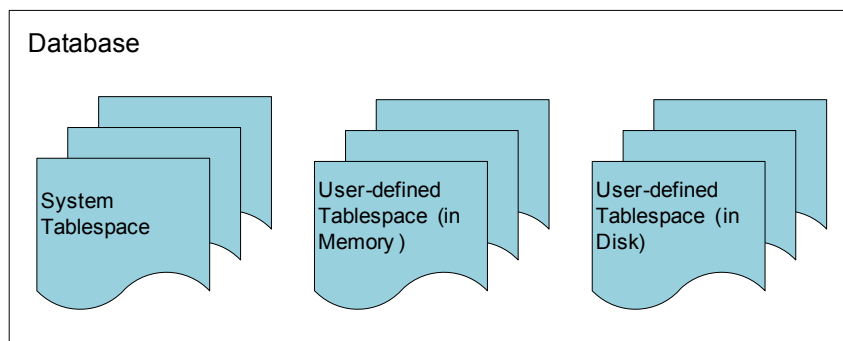
6.1.2 The Relationship between a Database and Tablespaces

When a database is created, 4 system tablespaces (the system dictionary tablespace, the system data tablespace, the system undo tablespace and the system temporary tablespace) are automatically created.

In addition, the user can create user-defined tablespaces (disk, memory or volatile tablespaces) as needed. The user can create user-defined tablespaces either on disk or in memory based on the characteristics of the data.

[Figure 5-1] shows the relationship between a database and tablespaces.

Figure 6-1 The Relationship between a database and tablespaces



6.1.3 The Structure of Disk Tablespace

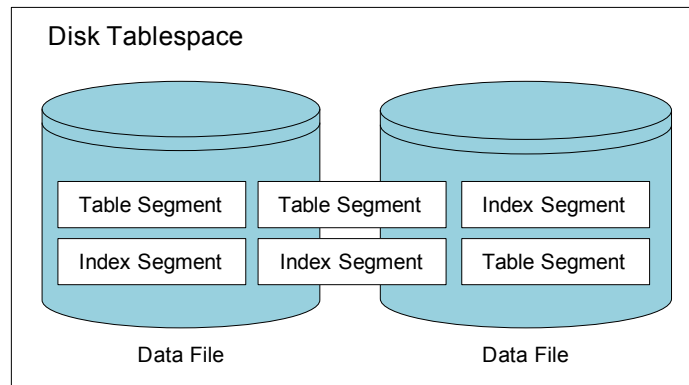
A disk tablespace is a tablespace in which all data are stored on disk. Physically it consists of data files, whereas logically it consists of segments, extents and pages.

6.1.3.1 Disk Tablespace Physical Structure

Disk tablespaces are closely related to data files and segments. [Figure 5-2] shows the relationship between a disk tablespace and data files and segments.

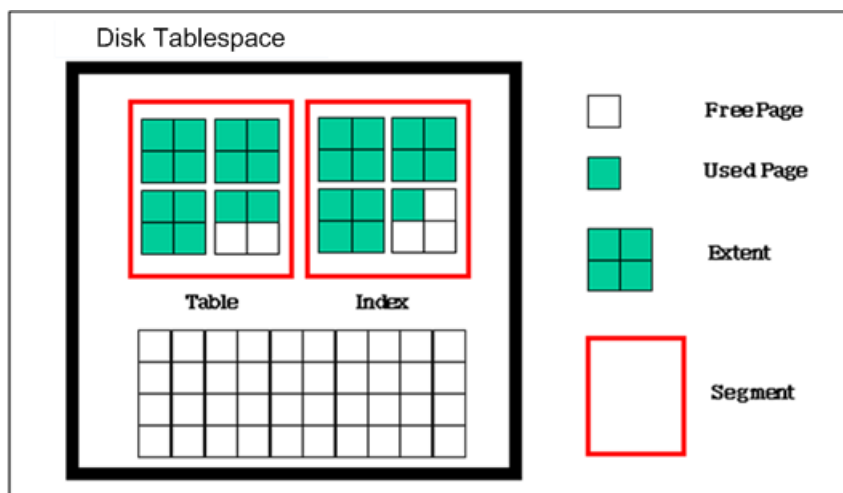
Disk tablespaces, data files and segments have the following characteristics: A disk tablespace consists of one or more data files, which exist in the form of files supported by the operating system. A segment is stored in a tablespace logically and in a data file physically. A segment is wholly contained within a particular disk tablespace, but segments can refer to segments that are stored in other disk tablespaces.

Figure 6-2 The Relationships between Disk Tablespaces, Data Files and Segments



6.1.3.2 Disk Tablespace Logical Structure

A disk tablespace logically consists of segments, extents and pages. The relationships between them are shown in [Figure 5-3].

Figure 6-3 The Logical Structure of a Disk Tablespace

Segment

A segment is a set of extents that contains all of the data for an object within a tablespace. Segments are the units within a tablespace by which tables or indexes are allocated. A single table or index is logically the same as one segment. The kinds of segments used in ALTIBASE HDB are as follows:

Table 6-1 Segment Type

| Segment Type | Description |
|---------------|--|
| Table Segment | This segment type is the most basic means of storing data within a database. All of the data in a table, or, in the case of a partitioned table, all of the data in a partition, are stored in a single table segment. When a table is created, ALTIBASE HDB allocates this table segment in a tablespace. |
| Index Segment | A single index segment contains all of the data for one index, or for one partition of a partitioned index. The purpose of an index is to assist in locating data in a table based on some particular key. When an index is created, ALTIBASE HDB allocates an index segment to a tablespace. |
| Undo Segment | Undo segments are used by transactions that change the database. Before a table or index is changed, the value before the change (i.e. the "before-image") is stored in an undo segment so that the change can be undone if the associated transaction is rolled back. |
| TSS Segment | These are used for managing Transaction Status Slots, which are managed internally within ALTIBASE HDB. They are allocated within system undo tablespace. |

Each segment internally maintains a free extent list and a full extent list. When there are not enough free extents, a request is made to add one or more additional extents to a tablespace.

Extent

In disk tablespace, an extent is the unit by which contiguous pages, which are the resource required for storing data objects, are assigned. When saving data, if there are not enough free pages available to save the data, additional pages are allocated in extent units.

By default, a single disk tablespace extent consists of 64 pages (512KB). In ALTIBASE HDB, the extent size can be specified individually for each tablespace.

Page

The smallest unit for storing records in tables and indexes is called the page, which is also the smallest unit for performing I/O. In ALTIBASE HDB, the page size is 8KB. (The simultaneous use of multiple page sizes isn't supported in ALTIBASE HDB.)

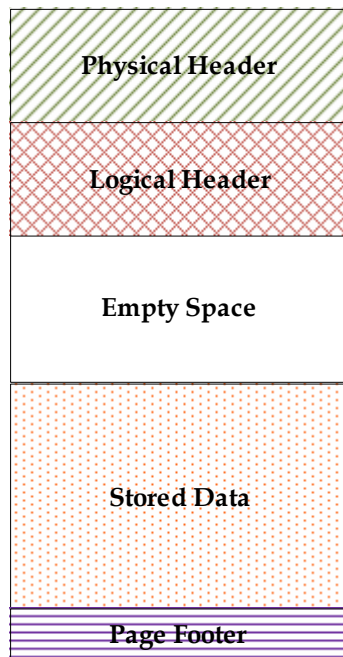
There are several kinds of pages, including data pages, index pages and undo pages, corresponding to the kind of data that are stored in the pages.

The basic structure of pages, as well as how to store data in them, are described below.

Page Structure

A page has a header for storing basic information about the page, free slots (this is the only instance of the term "Free slot" in this document), and the like. Records are stored in the remaining space. A page is internally divided into 5 areas, as shown below:

Figure 6-4 The Structure of a Page in Disk Tablespace



- Physical Header

This area contains information that is common to all data pages, regardless of type.

- Logical Header

6.1 Tablespaces: Definition and Structure

This area contains information that is necessary depending on the type of page.

- Free Space

This area is used to save new data.

- Stored Data

Rows, indexes or undo records are stored here depending on the type of page.

- Page Footer

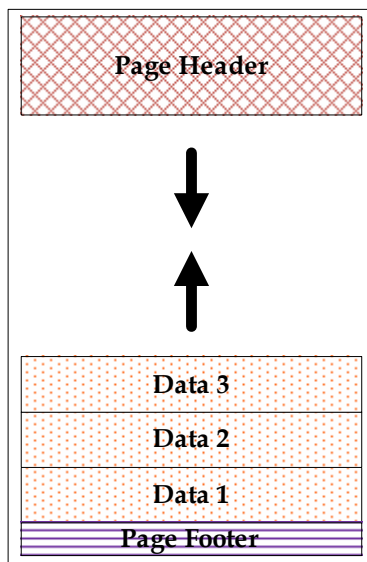
This area is located at the end of the page, and contains information that is used to check page integrity.

How Records are Stored in a Page

The records in a page are stored in free space, starting from the end of the page and working towards the beginning of the page.

The logical header of the page is saved extending toward the end of the page. Its size is variable.

Figure 6-5 How Records are Stored in a Page

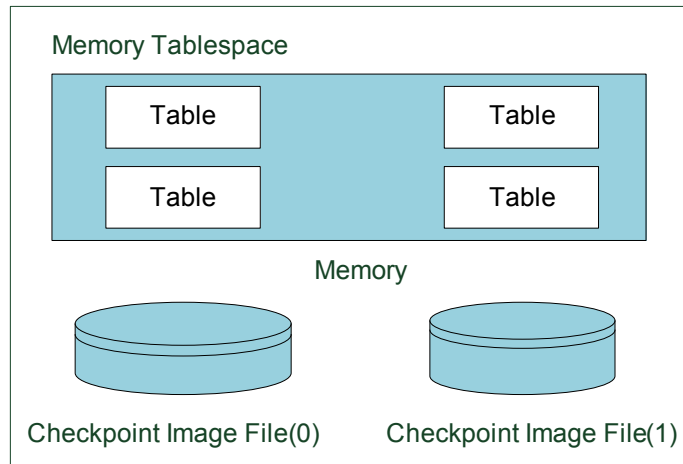


6.1.4 The Structure of Memory Tablespace

A memory tablespace is a tablespace in which all data are stored in memory. Its physical structure consists of checkpoint image files, whereas its logical structure consists of page lists and pages.

6.1.4.1 Memory Tablespace Physical Structure

Memory tablespaces are closely related to checkpoint image files. [Figure 5-6] shows the relationship between a memory tablespace and tables and checkpoint image files.

Figure 6-6 The Relationship between a Memory Tablespace , Tables and Checkpoint Image Files

Memory tablespaces, tables and checkpoint image files have the characteristics described below.

Unlike a disk tablespace, in a memory tablespace, data are stored not in a data file but sequentially in memory.

Because a continuous memory space is divided into pages, a table can be thought of as a list of pages. In the interests of managing disk I/O expense and tables containing large amounts of data, disk tablespaces are managed in units of extents, not pages. A segment is, conceptually, a way of managing a list of extents.

However, because the purpose of memory tablespace is to provide faster access to data, rather than to manage large volumes of data, the concept of segments and extents is not necessary. Consequently, tables in a memory tablespace are managed using lists of pages.

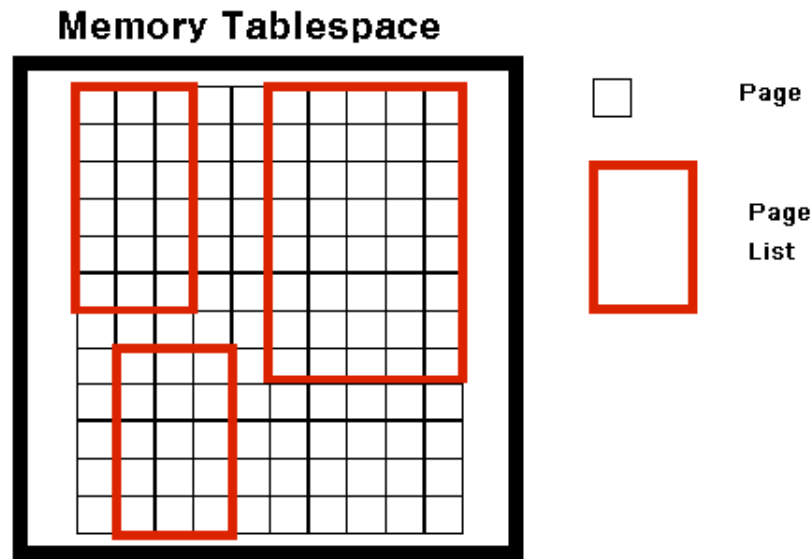
Memory tables are physically backed up in checkpoint image files when checkpointing occurs. The purpose of checkpoint image files is different from that of data files in a disk tablespace. Data files in a disk tablespace are for storing objects, whereas checkpoint image files are for backing up objects in a memory tablespace. Checkpoint image files are not directly required for the operation of the database. However, they are required in order to reduce the amount of time taken to perform backup and recovery.

When checkpointing occurs, pages in memory are stored in files of a type supported by the operating system. In ALTIBASE HDB, so-called "ping-pong checkpointing" is implemented, which means that two sets of checkpoint image files (namely, #0 and #1) are maintained, and used alternately when checkpointing takes place. In addition, each checkpoint image can be divided into several small files, with the goal of distributing disk I/O expense.

6.1.4.2 Memory Tablespace Logical Structure

The elements that logically constitute memory tablespace are page lists and pages. The relationship between these elements is shown in [Figure 5-7].

Figure 6-7 The Logical Structure of Memory Tablespace



6.1.4.3 Page List

The concept of a page list explains how tables are logically configured in a memory tablespace. A page list is a list of pages, which are the unit into which the memory occupied by a memory tablespace is divided.

Tables are the only memory tablespace objects for which page lists are maintained. Because the object of indexes is not to maintain database consistency, they do not use page lists. When the system is restarted, memory table indexes are rebuilt, which eliminates the load that would otherwise be imposed by performing index logging while the database is running.

6.1.4.4 Page

The characteristics of memory tablespaces are different from those of disk tablespaces with respect to the structure of pages and the way that data are stored.

Unlike disk tablespaces, when dealing with memory tablespaces, there is no need to consider disk I/O expense, and thus the method of updating records is so called "out-place update".

In out-place update, the existing record image is not directly changed; rather, record space is allocated for the new version of the record. This update process consists of deleting the existing record and inserting a new record, thereby eliminating the cost of rearranging the record that currently exists. In addition, it allows direct access to existing data, ensuring fast performance in highly concurrent applications.

6.1.5 The Structure of Volatile Tablespace

The structure of a volatile tablespace is identical with that of a memory tablespace in that all data are stored in memory. The difference between them is that, in a volatile tablespace, there is no image file on disk. The data in a volatile tablespace reside only in memory.

Since tasks that are conducted in a volatile tablespace are not accompanied by disk logging and are not subjected to checkpointing, they entail absolutely no disk I/O. As a result, volatile tablespaces are useful in applications requiring fast performance. Logically, they consist of page lists and pages.

6.1.5.1 Volatile Tablespace Physical Structure

The structure of volatile tablespaces is identical with that of memory tablespaces in that database objects reside in memory. However, volatile tablespaces do not have checkpoint image files.

6.1.5.2 Volatile Tablespace Logical Structure

Just like memory tablespaces, the constituent elements of volatile tablespaces are page lists and pages.

6.2 Classifying Tablespaces

The tablespaces provided in ALTIBASE HDB can be classified into three kinds based on the three following criteria. A single type of tablespace can have more than one of the properties listed below.

- [By Where the Data are Stored](#)
- [By What Data are Stored](#)
- [By Who the Creator Is](#)

6.2.1 By Where the Data are Stored

ALTIBASE HDB tablespaces can be classified according to where the data are stored as follows:

- [Memory-Resident Tablespace](#)
- [Disk Tablespace](#)

6.2.1.1 Memory-Resident Tablespace

Memory-resident tablespaces are classified as either memory tablespace or volatile tablespace based on whether logging is performed and on whether disk image files exist.

Memory tablespaces are tablespaces for storing objects in memory. Because all objects stored in memory tablespaces use memory-based database technology, the user can access data in real time. However, the size of memory tablespaces is limited by the amount of physical memory space that is available in the system.

Volatile tablespaces are tablespaces for storing objects in memory without disk I/O operations. Because all objects stored in volatile tablespaces use memory-based database technology and additional technologies, the user can access data in real time. However, the size of memory tablespaces is limited by the amount of physical memory space that is available in the system, and additionally, when the database is shut down, all volatile data objects are lost.

6.2.1.2 Disk Tablespace

A disk tablespace is a tablespace for storing disk-based objects. It is intended for the management of large amounts of data rather than for rapid data access. Accessing objects that are stored in disk tablespaces entails disk I/O. Since this disk I/O expense accounts for the biggest share of data access time, disk tablespaces use memory buffers to reduce disk I/O expenses.

6.2.2 By What Data are Stored

ALTIBASE HDB tablespaces can also be classified according to what data are stored in them as follows:

- [The Dictionary Tablespace](#)
- [The Undo Tablespace](#)

- [Temporary Tablespace](#)
- [Data Tablespace](#)

6.2.2.1 The Dictionary Tablespace

The dictionary tablespace is the tablespace for storing the meta data that are required for system operation. A database can have only one dictionary tablespace, which is automatically created by the system when a database is created. Users cannot create objects in dictionary tablespace; only the system can create system objects for managing meta data. In order to ensure fast access to meta data, the dictionary tablespace exists in memory. If the dictionary tablespace crashes, the entire database becomes inoperable, in which case the database will need to be restored through backup and media recovery.

6.2.2.2 The Undo Tablespace

The undo tablespace is for storing undo images that remain after operations are conducted on disk objects. Since ALTIBASE HDB uses Multi-Version Concurrency Control, it requires space in which to store images that show the state of data before changes were made. These so-called "pre-change images" are stored in the undo tablespace. Using the undo tablespace helps prevent the unlimited expansion of data tablespaces due to the storage of many pre-change images.

Only one undo tablespace can exist in the database, and it is shared by all disk tablespaces in the database. This makes the undo tablespace essential for system operation, like the dictionary tablespace. It can be backed up, but the entire tablespace must be backed up at one time.

6.2.2.3 Temporary Tablespace

A temporary tablespace is a tablespace for storing temporary results that are generated while a query is being executed. As a result, all data in the temporary tablespace pertaining to the query disappear when the associated transaction is completed.

With this type of tablespace, concurrency control, logging for recovery and the like are not conducted, enabling fast read and write speeds. There can be more than one temporary tablespace in the database, and users can create user-defined temporary tablespaces as desired. Note that temporary tablespaces cannot be backed up.

6.2.2.4 Data Tablespace

Data tablespaces are for storing user-defined objects. There can be more than one data tablespace in the database, and the user can create a data tablespace as a disk, memory or volatile tablespace based on the characteristics of the data to be stored therein.

6.2.3 By Who the Creator Is

Additionally, ALTIBASE HDB tablespaces can be classified by who created them as follows:

- [System Tablespace](#)
- [User-defined Tablespace](#)

6.2 Classifying Tablespaces

6.2.3.1 System Tablespace

A system tablespace is a tablespace for storing data required for the operation of the system. The system tablespaces include the system dictionary tablespace, the system undo tablespace, the system data tablespace and the system temporary tablespace. System tablespaces are created when the database is created, and cannot be deleted or renamed by users. Backup and media recovery can be performed for system tablespaces in their entirety.

6.2.3.2 User-defined Tablespace

A user-defined tablespace is a tablespace for storing the content of user-defined objects. The meta data pertaining to objects defined in user-defined tablespaces are stored in the dictionary tablespace. User-defined tablespaces can be explicitly deleted or renamed by users. Additionally, backup and media recovery can be performed for entire tablespaces.

6.2.4 Tablespace List

Several tablespaces are created when a database is created.

As shown in [Table 5-2], these include the system tablespaces, temporary tablespaces and basic memory and disk tablespaces for direct use by users.

In addition, users can add more tablespaces using the 'CREATE TABLESPACE' statement.

Table 6-2 Tablespace List

| ID | Tablespace Type | Data Storage Location | Tablespace Name | Time of Creation |
|-----|----------------------------------|-----------------------|-------------------|-------------------------------|
| 0 | SYSTEM DICTIONARY TABLESPACE | Memory | SYS_TBS_MEM_DIC | CREATE DATABASE |
| 1 | SYSTEM MEMORY DEFAULT TABLESPACE | Memory | SYS_TBS_MEM_DATA | CREATE DATABASE |
| 2 | SYSTEM DISK DEFAULT TABLESPACE | Disk | SYS_TBS_DISK_DATA | CREATE DATABASE |
| 3 | SYSTEM UNDO TABLESPACE | Disk | SYS_TBS_DISK_UNDO | CREATE DATABASE |
| 4 | SYSTEM DISK TEMPORARY TABLESPACE | Disk | SYS_TBS_DISK_TEMP | CREATE DATABASE |
| >=5 | USER MEMORY DATA TABLESPACE | Memory | User-Defined | CREATE MEMORY DATA TABLESPACE |
| >=5 | USER DISK DATA TABLESPACE | Disk | User-Defined | CREATE DISK DATA TABLESPACE |

| ID | Tablespace Type | Data Storage Location | Tablespace Name | Time of Creation |
|-----|--------------------------------|-----------------------|-----------------|----------------------------------|
| >=5 | USER DISK TEMPORARY TABLESPACE | Disk | User-Defined | CREATE DISK TEMPORARY TABLESPACE |
| >=5 | USER VOLATILE DATA TABLESPACE | Memory | User-Defined | CREATE VOLATILE DATA TABLESPACE |

6.3 Disk Tablespace

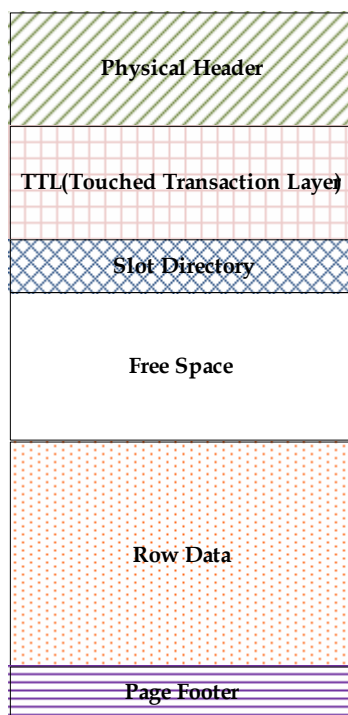
A disk tablespace is a tablespace in which all of the data therein are stored on disk. This section discusses the structure of disk tablespaces, which is based on disk data pages, and how row data are input into disk pages.

6.3.1 Data Page Structure

In ALTIBASE HDB, the smallest unit of database storage space management is the page. The size of a page is 8KB. Multiple page sizes are not supported.

A data page is one of several kinds of pages, and stores row data. Row data are stored in free space starting from the end of the page. If there is not enough free space, it is advisable to create larger regions of free space using compaction to turn fragmented space into contiguous space.

Figure 6-8 The Structure of a Data Page in Disk Tablespace



A data page consists of six different areas, as shown in [Figure 5-8].

- **Physical Header**
This area contains information that is common to all pages, regardless of the type of page.
- **TTL (Touched Transaction Layer)**
This area contains information related to MVCC (Multi-Version Concurrency Control).
- **Slot Directory**

This area contains information about so-called “row offset”, that is, the location within the page at which the Row Data area is saved.

- Free Space

This area is available space that is used for saving the results of operations such as insert and update operations.

- Row Data
- Page Footer

This area is located at the end of the page, and contains information that is used for checking page integrity.

6.3.2 Managing Space in Disk Tablespace

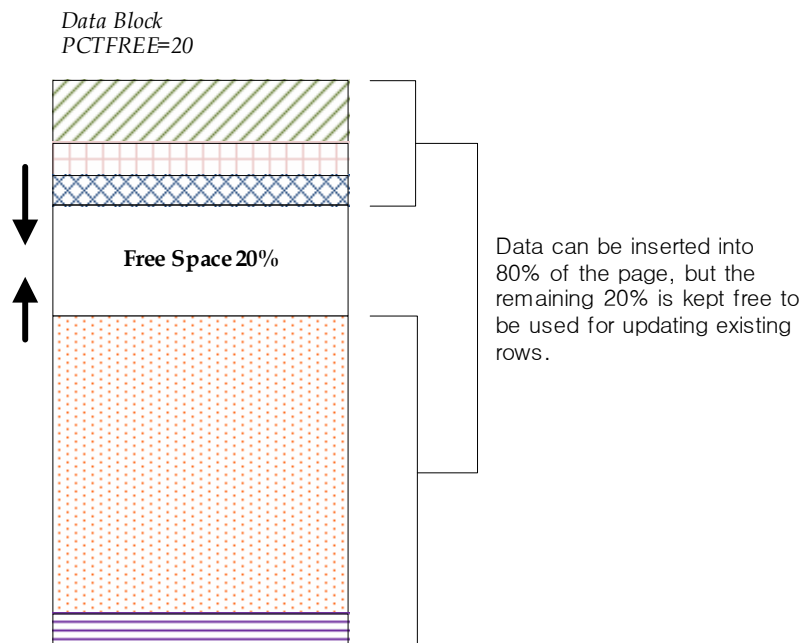
Disk tablespace can be manually managed using the PCTFREE and PCTUSED parameters.

How free space is used when new data are entered into rows or existing data are updated can be controlled using the PCTFREE and PCTUSED parameters. These two parameters are set using the PCTFREE and PCTUSED properties in the `altibase.properties` file. They can also be explicitly specified when a table is created using the `CREATE TABLE` statement or changed using the `ALTER TABLE` statement.

6.3.2.1 PCTFREE

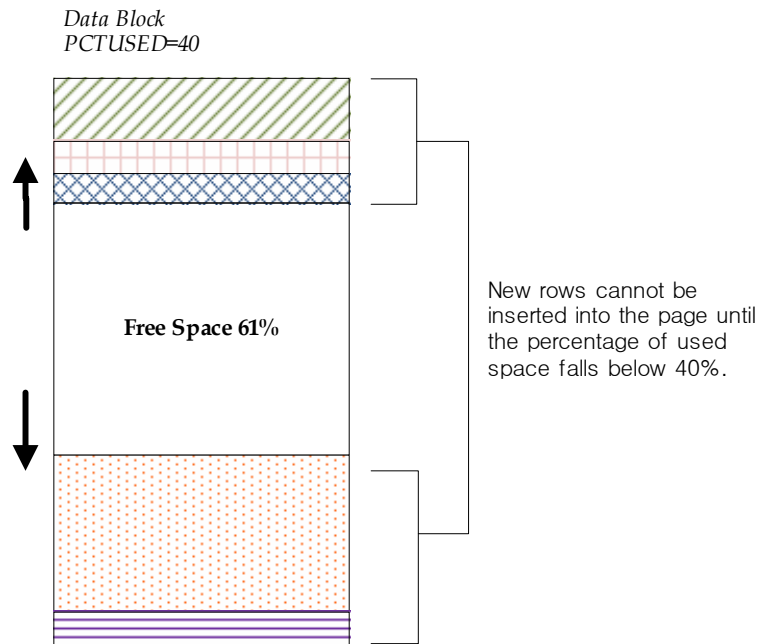
PCTFREE is the minimum amount of free space, expressed as a percentage, that is reserved for updating rows that have already been stored in a page.

For example, if PCTFREE is set to 20, data can be inserted into the page until it is 80% full, and the remaining 20% of the page will be set aside for use in updating existing rows.

Figure 6-9 PCTFREE and Page Structure**6.3.2.2 PCTUSED**

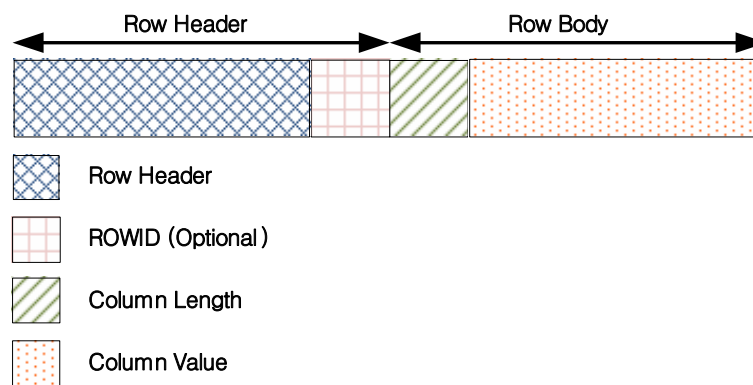
PCTUSED is the threshold percentage below which the amount of used space in a page must decrease in order for the page to change from the state in which only update operations are possible to the state in which records can be inserted.

If the amount of free space falls below the limit specified in PCTFREE, it becomes impossible to insert new records into the page, and free space in the page can only be used to update existing rows. This state persists until the percentage of used space falls below the threshold specified by PCTUSED.

Figure 6-10 PCTUSED and Page Structure

6.3.3 Row Structure

Rows can be divided into one or more pieces. If it is possible to store an entire row in a page, it is saved as one row piece. However, if it is not possible to store the row in a single page, the row is divided into several pieces and then saved. These row pieces are "chained", that is, they are associated with each other, via a common ROWID value.

Figure 6-11 The Structure of a Row Piece

A row piece comprises a row header and a row body.

A row header contains 18 bytes of header information. Additionally, if the row piece is a chained row piece, the row header contains 6 more bytes for storing the value of ROWID.

In the row body, pairs comprising the length of a column and the value stored in the column are

6.3 Disk Tablespace

stored sequentially. If the value stored in the column is less than 250 bytes, only 1 byte is necessary for storing the column length, whereas 3 bytes are used to store the column length if the value stored in the column exceeds 250 bytes.

In order to conserve space, if the value of the column is NULL, only the length of the column, which is 0, is saved. No column value is saved. Additionally, for columns that contain NULL values and are located at the end of the row, neither the column value nor the length is saved.

Columns are saved in the order in which they are specified in the CREATE TABLE statement. Therefore, when executing the CREATE TABLE statement, locating columns that are expected to frequently contain NULL values at the end of the row is good practice, because it can reduce the amount of space required to store rows.

6.3.4 Row Chaining and Migration

When row data are too large to be saved in a single page, row chaining and row migration will occur.

Row chaining occurs when data that are being inserted are so large that the row containing them cannot fit in a single page. When row chaining occurs, long rows are divided into pieces and saved in multiple pages. These pieces are associated with one another by their common ROWID value.

Row migration occurs when a row that was saved in a single page is increased in size by an update operation and thus can no longer fit in a single page. In this case, the entire row is migrated to new pages. The original row becomes a pointer that indicates the new location where the row is saved. However, when a row is migrated, the ROWIDs of its pieces do not change.

When row chaining or migration occurs, the execution of DML statements requires more than one page to be read. This will degrade performance due to the increased amount of disk I/O.

6.4 The Undo Tablespace

The undo tablespace is for storing information that is used to cancel update operations. Because ALTIBASE HDB uses Multi-Version Concurrency Control (MVCC), free space is required for saving images of data as they existed before update operations.

The database has only one undo tablespace, which is shared by all disk tablespaces in the database.

This section discusses the features of the undo tablespace and how to manage it, for example, how to determine its size.

- [Undo Records](#)
- [Features of the Undo Tablespace](#)
- [Transaction Segment Management](#)
- [Reusing Segment Space](#)
- [Modifying the Undo Tablespace](#)

6.4.1 Undo Records

In order to rollback or undo the results of update transactions, related information must be stored in the database. This information is usually saved in undo records before transactions are committed.

Undo records are used for the following purposes:

- rolling back transactions
- recovering the database
- guaranteeing read consistency

When a ROLLBACK statement is executed, undo records are used to cancel changes that have been made to the database by transactions that have not yet been committed.

Undo records are also used during database recovery. After the database has been restored by re-executing transactions ("REDO") on the basis of log files, undo records are used to cancel changes that were not committed.

Additionally, when a record that is in the process of being changed by a transaction is read by another transaction, even though the transactions access the record simultaneously, read consistency is guaranteed because an image of the record before the change is stored in the undo record.

6.4.2 Features of the Undo Tablespace

The features of the undo tablespace are as follows:

- The undo tablespace is automatically managed by the system.
- The default undo tablespace file is undo001.dbf, which is in auto extension mode. Data files can be added to the undo tablespace, and their sizes can be changed.

6.4 The Undo Tablespace

- The undo tablespace can be backed up online.
- Database objects other than TSS segments and undo segments cannot be created in the undo tablespace.
- Because the undo tablespace is a system tablespace, it cannot be taken offline or discarded.
- The undo tablespace is reset whenever the server is restarted.

In ALTIBASE HDB, information about the undo tablespace, as well as the space in the undo tablespace, is managed by the system. In other words, the server automatically manages the space and segments in the undo tablespace.

The undo tablespace is created when the database is created. Because it is a system tablespace, only one undo tablespace can exist. If the undo tablespace does not exist, the server will fail to start up, and an error message will be written to the boot log.

In the undo tablespace, transaction segments (TSS segments and undo segments) are managed. The user can change the number of transaction segments using the TRANSACTION_SEGMENT_COUNT property. The numbers of TSS segments and undo segments that are created, respectively, equal the number specified by the user in this property. If the TRANSACTION_SEGMENT_COUNT property is set to 255, 255 TSS segments and 255 undo segments are created every time the server is started up.

If this property is changed to some other value to specify a different number of transaction segments, that number of segments will be created the next time the server is restarted.

6.4.3 Transaction Segment Management

A transaction segment consists of one TSS segment and one undo segment, which are essential for update transactions in disk tablespaces. A transaction segment can't be simultaneously shared by multiple transactions, because one transaction segment is bound to one update transaction, and is not unbound until the disk update transaction has been completed.

The transaction segments that are currently bound can be checked by querying the V\$TXSEGS performance view. When a transaction segment is bound to an update transaction that takes place in a disk tablespace, a record indicating the segment ID and the transaction ID is created in V\$TXSEGS. When the segment is unbound, the record is deleted.

Additionally, space allocated for TSS segments and undo segments can be reused by other transactions after the segments expire. Therefore, when space is required for undo transactions, the undo tablespace does not necessarily need to be expanded by creating segments; instead, segments that have expired can be reused.

TSS segments are allocated for reuse in units of one megabyte (1MB), and undo segments in units of two megabytes (2MB).

The following user properties pertain to undo tablespace:

- SYS_UNDO_FILE_INIT_SIZE
The initial size of the undo tablespace data files at the time of creation
- SYS_UNDO_FILE_MAX_SIZE

The maximum size of the undo tablespace data files

- `SYS_UNDO_TBS_NEXT_SIZE`

The amount by which the size of the undo tablespace data files is automatically extended

- `SYS_UNDO_TBS_EXTENT_SIZE`

The number of pages in one extent in undo tablespace

- `TRANSACTION_SEGMENT_COUNT`

The number of transaction segments

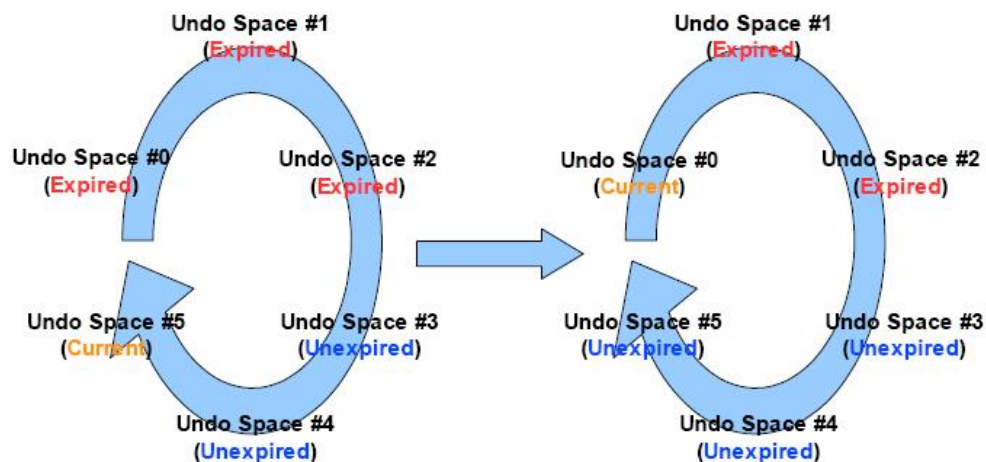
6.4.4 Reusing Segment Space

After a transaction has been committed, undo data are no longer necessary for the purposes of roll-back or recovery. However, so-called "long-term transactions", meaning transactions that take a long time to commit, require previous versions of records, which depend on undo data, in order to ensure read consistency. However, after sufficient time has passed, undo data are not required for the purpose of ensuring read consistency either.

Therefore, ALTIBASE HDB maintains undo records pertaining to committed transactions only as long as necessary, after which the space occupied by the undo data is made available for use by other transactions.

If there are no active transactions accessing the space containing the undo data for transactions that have been committed, the so-called "undo space" is said to have expired. Conversely, if active transactions that might need to access the undo space still exist, the space is considered valid, or unexpired. Expired undo space can be reused by other transactions, whereas unexpired space cannot.

Figure 6-12 Reusing Undo Spaces in an Undo Segment



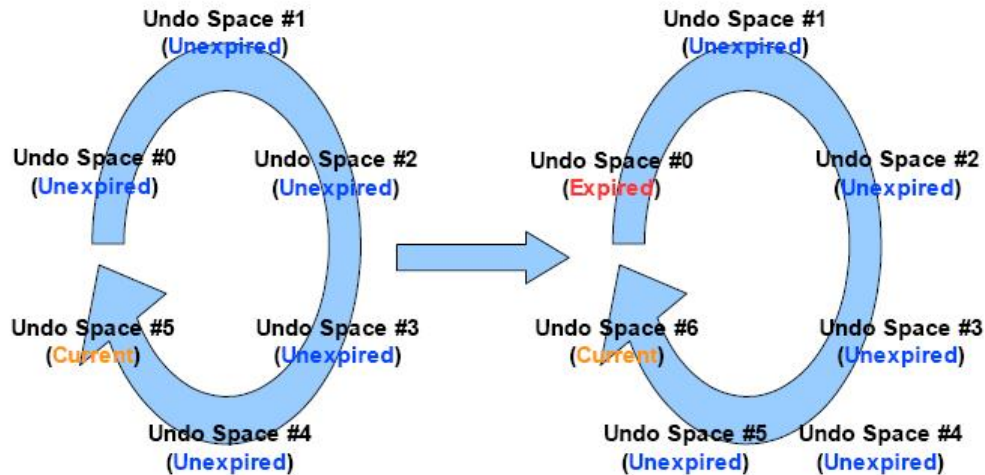
[Figure 5-12] shows how the cyclical structure of undo segments allows undo space to be reused.

Undo spaces are used sequentially starting with undo space #0, until undo space #5 is the space that is currently being used. Then, if undo space #0, which is the next undo space to be used, is confirmed

6.4 The Undo Tablespace

to have expired, then when all of undo space #5 has been used up, undo space #0 is reused without requiring expansion of the undo segment.

Figure 6-13 Undo Segment Expansion



However, if undo space #0 has not expired, extents are added to the undo segment, thus creating undo space #6, as can be seen in [Figure 5-13].

The space in TSS segments is also reusable in this fashion.

6.4.5 Modifying the Undo Tablespace

The undo tablespace can be modified using the ALTER TABLESPACE statement. However, because the undo tablespace is usually managed by the system, only the following operations can be conducted by the user:

- adding or dropping data files
- increasing or reducing the size of data files
- starting or completing the online backup of data files

If the undo tablespace has insufficient space, or in order to prevent errors related to insufficient space, either add data files to the undo tablespace or increase the size of the existing data files.

The following example illustrates how to add a data file to the undo tablespace.

```
ALTER TABLESPACE SYS_TBS_DISK_UNDO  
ADD DATAFILE 'undo002.dbf' AUTOEXTEND ON NEXT 1M MAXSIZE 2G;
```

A data file can be dropped using the ALTER TABLESPACE ... DROP DATAFILE statement, and can be increased or decreased in size using the ALTER TABLESPACE ... ALTER DATAFILE... statement. Additionally, data file backup can be commenced using the ALTER TABLESPACE ... BEGIN BACKUP statement, and can be completed using the ALTER TABLESPACE ... END BACKUP statement.

6.5 Tablespace States

A tablespace is said to be in one of an "online", "offline" or "discard" state.

The state of user-defined disk and memory tablespaces can be changed from online to offline or vice-versa. However, the state of volatile and temporary tablespaces cannot be changed. Additionally, the state of tablespaces that contain tables to be replicated cannot be changed.

The `ALTER TABLESPACE ONLINE` and `ALTER TABLESPACE OFFLINE` statements can be used to change the state of a tablespace. Please note, however, that this can only be achieved during the Meta and Service startup phases.

6.5.1 Online

In this state, all resources related to the tablespace are allocated, and the tablespace is ready to be used in the database. DML and DDL statements can be executed on the tablespace and on the tables and indexes it contains. If it is desired to temporarily prevent a tablespace that is currently online, along with all of the tables and indexes it contains, from being used, all that is required is to take it offline by executing the `ALTER TABLESPACE OFFLINE` statement.

6.5.2 Offline

It is impossible to execute DML and DDL statements on the tables or indexes that exist in an offline tablespace. Additionally, the only DDL statements that can be executed on the tablespace are the `DROP TABLESPACE`, `ALTER TABLESPACE DISCARD` and `ALTER TABLESPACE ONLINE` statements.

The `ALTER TABLESPACE ONLINE` statement is used to bring an offline tablespace back online so that it can be accessed, along with the tables and indexes it contains.

Because the objects in a memory tablespace are not loaded into memory when the memory tablespace is offline, in situations where the amount of memory is limiting (that is, insufficient), the user is advised to take memory tablespaces offline.

6.5.3 Discard

If ALTIBASE HDB fails to start normally, and the reason is known to be because the data consistency of a particular tablespace has been broken, it will be necessary to start ALTIBASE HDB without the offending tablespace. In order to accomplish this, the tablespace must be discarded.

Tablespaces are discarded using the `ALTER TABLESPACE DISCARD` statement, which can be executed only during the control startup phase.

Please bear in mind that the only statement that can be executed on a discarded tablespace is the `DROP TABLESPACE` statement, and thus care should be taken when executing the `ALTER TABLESPACE DISCARD` statement.

6.6 Managing Tablespaces

This section describes how to manage tablespaces in ALTIBASE HDB.

6.6.1 CREATE

A tablespace can be created only by the SYS user or by a user to whom the Create Tablespace authority has been granted. To create a tablespace, use the `CREATE TABLESPACE ...` SQL statement. Only user-defined data tablespaces can be created. That is, system tablespaces cannot be created by the user.

Disk tablespaces are classified as either disk data tablespaces or disk temporary tablespaces.

Memory tablespaces are memory data tablespaces. There is no such thing as a “memory temporary tablespace”.

Similarly, all volatile tablespaces are volatile data tablespaces. There are no “volatile temporary tablespaces”.

The SQL statement that is used to create tablespaces is described below:

```
CREATE [DISK/MEMORY/VOLATILE] [DATA/TEMPORARY] TABLESPACE
  Tablespace Name
  Disk Data File Attributes
  Disk Temporary File Attributes
  Memory Tablespace Attributes
  Volatile Tablespace Attributes;
```

When determining whether to create a memory, disk or volatile tablespace, the user should consider the characteristics of the objects to be stored in the tablespace, such as their size and how often it is expected that they will be accessed.

The tablespace attributes that can be specified when a tablespace is created vary depending on whether the tablespace is a disk, memory or volatile tablespace. Unlike a disk tablespace, in which multiple data files are managed, in a memory tablespace, the objects are stored in a single continuous memory space. Therefore, when a disk tablespace is created, some of the attributes that are specified apply to individual data files, whereas when a memory tablespace is created, all of the attributes apply to the entire memory tablespace. In other words, attributes such as the initial size and the size to which the tablespace can expand are set for a memory tablespace, whereas the attributes that are set for a disk tablespace apply to its data files.

6.6.1.1 Tablespace Name

A tablespace name must be unique. More than one tablespace having the same name cannot be created. While the names of data files can be set in a disk tablespace, for a memory tablespace, only the path where the checkpoint image will be stored can be specified. The name of the checkpoint image is automatically generated based on the tablespace name.

6.6.1.2 Disk Data File Attributes

Data file attributes can only be set for disk data tablespaces. The `DATAFILE` clause has the following form:

```

DATAFILE [DATAFILE Clause
        AUTOEXTEND [AUTOEXTEND Clause
        MAXSIZE [MAXSIZE Clause] ] ]
EXTENTSIZE [EXTENTSIZE Clause]

```

Each data file can have the following attributes:

DATAFILE Clause

```
DATAFILE {datafile path and name} SIZE integer [K/M/G] [REUSE]
```

This is used to specify the data file path and name. The SIZE clause and the REUSE clause can be omitted. The SIZE clause is used to specify the initial size of a data file when it is created. Each data file includes a file header. SIZE is the total size of all pages, excluding the page for the file header (1 page). As a result, the specified initial size of a data file and its actual size are not the same. If the maximum file size supported by the operating system is smaller than the specified initial size, an error will be returned.

AUTOEXTEND Clause

```
AUTOEXTEND [{ON NEXT integer [K/M/G]} / {OFF}]
```

This attribute determines whether a disk data file will increase in size. If it is set to ON, the size of the data file is automatically increased by the system. If it is set to OFF, the user must explicitly increase the file size. The increment by which the data file is extended can be specified by the user in the NEXT clause.

When a data file is being extended, that is, when it is being increased in size, all operations that are underway in the tablespace to which the data file belongs must wait until the operation to increase the size of the data file is complete.

MAXSIZE Clause

```
MAXSIZE [{integer [K/M/G]} / {UNLIMITED}]
```

This clause is a subclause of the AUTOEXTEND clause, and indicates the maximum size to which the data file can be increased. Like the initial size, if the maximum file size supported by the operating system is smaller than the maximum size specified for a data file, the maximum size is set to the maximum file size of the operating system. If MAXSIZE is set to UNLIMITED, the data file is increased in size until all of the available disk space has been used up.

EXTENTSIZE Clause

```
EXTENTSIZE [{integer [K/M/G]} / {UNLIMITED}]
```

This clause defines the size of an extent, that is, (the size of a page) * (the number of pages that are allocated to a table segment or index segment in disk tablespace at one time). If the extent size is not specified, it defaults to 512kB (64 pages).

6.6.1.3 Disk Temporary File Attributes

Temporary file attributes can only be set for disk temporary tablespaces. The TEMPFILE clause has the following form:

6.6 Managing Tablespaces

```
DATAFILE [TEMPFILE Clause  
        AUTOEXTEND [AUTOEXTEND Clause  
        MAXSIZE [MAXSIZE Clause] ] ]  
EXTENTSIZE [EXTENTSIZE Clause]
```

Each temporary file can have the following attributes:

TEMPFILE Clause

```
TEMPFILE {datafile path and name} SIZE integer [K/M/G] [REUSE]
```

This specifies the path and name of a temporary file. The SIZE clause and the REUSE clause can be omitted. The SIZE clause is used to specify the initial size of a temporary file when it is created. Each temporary file includes a file header. SIZE is the total size of all pages, excluding the page for the file header (1 page). As a result, the specified initial size of a temporary file and its actual size are not the same. If the maximum file size supported by an operating system is smaller than the specified initial size, an error will be returned.

AUTOEXTEND Clause

```
AUTOEXTEND [{ON NEXT integer [K/M/G]} / {OFF}]
```

This attribute determines whether a disk temporary file will increase in size. If it is set to ON, the size of the data file is automatically increased by the system. If it is set to OFF, the user must explicitly increase the file size. The increment by which the data file is extended can be specified by the user in the NEXT clause.

MAXSIZE Clause

```
MAXSIZE [{integer [K/M/G]} / {UNLIMITED}]
```

This clause is a subclause of the AUTOEXTEND clause, and indicates the maximum size to which the temporary file can be increased. Like the initial size, if the maximum file size supported by the operating system is smaller than the maximum size specified for a temporary file, the maximum size is set to the maximum file size of the operating system. If MAXSIZE is set to UNLIMITED, the temporary file is increased in size until all of the available disk space has been used up.

EXTENTSIZE Clause

```
EXTENTSIZE integer [K/M/G]
```

This clause defines the size of an extent, that is, (the size of a page) * (the number of pages that are allocated to a table segment or index segment in temporary tablespace at one time). If the extent size is not specified, it defaults to 256kB (32 pages).

6.6.1.4 Memory Tablespace Attributes

The attributes for memory tablespace are similar to those for disk tablespaces, but additionally include a checkpoint image path attribute. Their syntax is as follows:

```
DATAFILE [SIZE Clause  
        AUTOEXTEND [AUTOEXTEND Clause  
        MAXSIZE [MAXSIZE Clause] ] ]  
EXTENTSIZE [CHECKPOINT PATH]
```

Memory tablespaces can have the following attributes:

SIZE Clause

`SIZE integer [K/M/G]`

This is the amount of memory that must be initially allocated when a memory tablespace is created. This value must be a multiple of the default extension increment size for memory tablespaces. (This increment size is equal to the number of page(s) specified in the EXPAND_CHUNK_PAGE_COUNT property multiplied by the size of a memory tablespace page (32kB).)¹⁾

The size can be specified in kilobytes ("K"), megabytes ("M") or gigabytes ("G"). If no units are specified, the default unit is kilobytes ("K").

AUTOEXTEND Clause

`AUTOEXTEND [{ON NEXT integer [K/M/G]}/{OFF}]`

This determines whether the size of memory tablespace will be increased automatically. If it is set to ON, the tablespace is automatically increased in size by the system, whereas if it is set to OFF, the user must explicitly increase the size of the tablespace. The extension increment size, that is, the amount by which the size is increased, can be specified by the user. The NEXT clause indicates the extension increment size.

Like the initial size, the extension size must be set to a multiple of the page size specified in the EXPAND_CHUNK_PAGE_COUNT property.

If the automatic extension size is too small, automatic extension can occur too often. When ALTIBASE HDB performs automatic extension, it adds up the size of all memory tablespaces and compares the total size with the size specified in the MEM_MAX_DB_SIZE property. If this operation occurs too often, system performance can suffer.

MAXSIZE Clause

`MAXSIZE [{integer [K/M/G]}/{UNLIMITED}]`

This is a subclause of the AUTOEXTEND clause, and indicates the maximum size to which a memory tablespace can be extended. Like the initial size, it cannot exceed the amount of memory space available in the system. If it is set to UNLIMITED, the tablespace is automatically increased in size until the total size of all memory tablespaces in the system reaches the limit specified in the MEM_MAX_DB_SIZE property.

CHECKPOINT PATH

`CHECKPOINT PATH 'Checkpoint Image Path List'`
`SPLIT EACH integer [K/M/G]`

The checkpoint image path attribute only applies to memory tablespaces. ALTIBASE HDB uses ping-pong checkpointing for high-performance transaction processing in memory tablespaces. For ping-pong checkpointing, at least two sets of checkpoint images are created on disk. Each checkpoint

-
1. For example, if EXPAND_CHUNK_PAGE_COUNT is set to 128, the default extension increment size of memory tablespaces would be 128 * 32kB = 4 MB. Therefore, SIZE can be a multiple of 4 MB.

6.6 Managing Tablespaces

image can be divided into several files and saved in that form. The size of the files into which the checkpoint image is divided can be specified using the SPLIT EACH clause. These files can be stored in different paths in order to distribute the expense of disk I/O. The user can freely specify the size of the files into which the checkpoint image is divided and the path where the checkpoint images are saved. The user can add or change paths for saving checkpoint image files, but cannot change the size of the files into which the checkpoint image is divided once it has been set.

6.6.1.5 Volatile Tablespace Attributes

The attributes that are applicable to volatile tablespaces are similar to those for memory tablespaces, with the exception that the checkpoint image path attribute is not supported.

```
SIZE {SIZE Clause}  
AUTOEXTEND [AUTOEXTEND Clause  
MAXSIZE [MAXSIZE Clause] ]
```

Volatile tablespaces can have the following attributes:

SIZE Clause

```
SIZE integer [K/M/G]
```

This specifies the initial memory size that is allocated when a volatile tablespace is created. This value must be a multiple of the default extension increment size for memory tablespaces. (This increment size is equal to the number of page(s) specified in the EXPAND_CHUNK_PAGE_COUNT property multiplied by the size of a memory tablespace page (32kB).)

The size can be specified in kilobytes ("K"), megabytes ("M") or gigabytes ("G"). If no units are specified, the default unit is kilobytes ("K").

AUTOEXTEND Clause

```
AUTOEXTEND [{ON NEXT integer [K/M/G] }/{OFF}]
```

This determines whether the size of volatile tablespace will be increased automatically. If it is set to ON, the tablespace is automatically increased in size by the system, whereas if it is set to OFF, the user must explicitly increase the size of the tablespace. The extension increment size, that is, the amount by which the size is increased, can be specified by the user in The NEXT clause.

Like the initial size, the extension size must be set to a multiple of the page size specified in the EXPAND_CHUNK_PAGE_COUNT property.

If the automatic extension size is too small, automatic extension can occur too often. When ALTI-BASE HDB performs automatic extension, it adds up the size of all volatile tablespaces and compares the total size with the size specified in the VOL_MAX_DB_SIZE property. If this operation occurs too often, system performance can suffer.

MAXSIZE Clause

```
MAXSIZE [{integer [K/M/G] }/{UNLIMITED}]
```

This is a subclause of the AUTOEXTEND clause, and indicates the maximum size to which a volatile tablespace can be extended. Like the initial size, it cannot exceed the memory space available in the system. If it is set to UNLIMITED, the tablespace is automatically increased in size until the total size

of all memory tablespaces in the system reaches the limit specified in the `VOLATILE_MAX_DB_SIZE` property.

6.6.1.6 Examples

Ex. 1) To create a disk data tablespace comprising 3 data files:

```
iSQL> CREATE DISK DATA TABLESPACE user_data DATAFILE
'/tmp/tbs1.user' SIZE 10M AUTOEXTEND ON NEXT 1M MAXSIZE 1G,
'/tmp/tbs2.user' SIZE 10M AUTOEXTEND ON NEXT 1M MAXSIZE 500M,
'/tmp/tbs3.user' SIZE 10M AUTOEXTEND ON NEXT 1M MAXSIZE 1G;
Create success.
```

Ex. 2) To create a memory data tablespace:

```
iSQL> CREATE MEMORY DATA TABLESPACE user_data SIZE 12M
AUTOEXTEND ON NEXT 4M MAXSIZE 500M
CHECKPOINT PATH '/tmp/checkpoint_image_path1',
'/tmp/checkpoint_image_path2' SPLIT EACH 12M;
Create success.
```

Ex. 3) To create a volatile data tablespace:

```
iSQL> CREATE VOLATILE DATA TABLESPACE user_data SIZE 12M
AUTOEXTEND ON NEXT 4M MAXSIZE 500M;
Create success.
```

6.6.2 Dropping Tablespaces

A tablespace can be deleted only by the SYS user or by a user who has been granted the `DROP TABLESPACE` privilege. To delete a tablespace, use the `'DROP TABLESPACE ...'` SQL statement. System tablespaces cannot be deleted by general users. Memory, disk and volatile tablespaces are all deleted the same way, using the following command:

```
DROP TABLESPACE {Tablespace Name}
[ {INCLUDING CONTENTS} [AND DATAFILES]
[ [CASCADE CONSTRAINTS] ] ] ;
```

The tablespace to be deleted is identified by name. The available options are described below. If the following options are not specified, the only thing that is deleted from the log anchor is the tablespace schema.

6.6.2.1 The INCLUDING CONTENTS Clause

This is used to specify that the objects (that is, the tables and indexes) in the tablespace are also to be deleted. If any objects are present in the tablespace, this option must be set, otherwise the `DROP TABLESPACE` operation will fail.

6.6.2.2 The AND DATAFILES Clause

Specifying the `INCLUDING CONTENTS` clause deletes the records and keys of an object, but not the data files themselves. Therefore, in order to delete the data files, the `AND DATAFILES` clause must also be used. The `AND DATAFILES` clause is a subclause of the `INCLUDING CONTENTS` clause. If it is used, all of the data files in the tablespace are physically deleted. The `AND DATAFILES` option is only

6.6 Managing Tablespaces

useful with disk tablespaces. If it is specified when dropping a memory tablespace, it is ignored, whereas an error will occur if it is specified when dropping a volatile tablespace.

6.6.2.3 The CASCADE CONSTRAINTS Clause

This is also a subclause of the INCLUDING CONTENTS clause. If an attempt is made to drop a tablespace when there are constraints in other tablespaces that refer to objects in the tablespace to be dropped, the drop operation will fail, and an error indicating that objects remain in the tablespace will be raised. In this case, the CASCADE CONSTRAINTS clause should be used to additionally delete all external references to objects in the tablespace.

6.6.3 Modifying Tablespaces

A tablespace can be modified only by the SYS user or by a user to whom the Alter Tablespace authority has been granted. Tablespaces are modified using the 'ALTER TABLESPACE ...' SQL statement. This command can be used to change the definition of an existing tablespace, the attributes of one or more data files or temporary files, or the attributes of a memory or volatile tablespace. The related SQL syntax is as follows:

```
ALTER TABLESPACE {Tablespace Name}
{ {ALTER Disk Data File Clause}
  {ALTER Temporary File Clause}
  {ALTER Memory Tablespace Clause}
  {ALTER Volatile Tablespace Clause}
  {ALTER Tablespace State Clause} };
```

6.6.3.1 ALTER Disk Data File Clause

This clause can be used on a disk system tablespace or a disk data tablespace, and has the following options:

```
ALTER TABLESPACE {Tablespace Name}
{ADD Data File Clause
  DROP Data File Clause
  ALTER Data File Size Clause
  RENAME Data File Clause}
```

ADD Data File Clause

```
ADD {DATAFILE} {Data File Clause}
[AUTOEXTEND [AUTOEXTEND Clause
  MAXSIZE [MAXSIZE Clause]]
```

This clause is used to increase the amount of data storage space in a disk tablespace. The available options are the same as the data file options for the CREATE TABLESPACE statement.

DROP Data File Clause

```
DROP {DATAFILE} {Data File Name}
```

This is used to reduce the amount of data storage space for a disk tablespace. While the data storage space can be freely increased by adding more data files, a data file can be deleted only when it is not

in use, that is, when no extents have been allocated to the data file.

ALTER Data File Size Clause

```
ALTER {DATAFILE} {Data File Name}
    [{AUTOEXTEND [AUTOEXTEND Clause]}
    {SIZE [SIZE Clause]}]
```

Various attributes of each data file in a disk data tablespace, including its current size, maximum size, the extension increment size and whether it is automatically increased in size, can be changed.

The specified current size and maximum size must be greater than the amount that is currently being used.

RENAME Data File Clause

```
RENAME {DATAFILE} {The path and name of the existing data file}
    TO {The path and name of a new data file}
```

This command can be used to change the location of a data file and thereby change the file system in which the data in a tablespace are stored. This clause can be used in any startup phase, regardless of whether the applicable tablespace is online or offline. However, it can only be used with offline tablespaces in the service phase.

6.6.3.2 ALTER Temporary File Clause

This can be used only with disk temporary tablespaces. It has the following options:

```
ALTER TABLESPACE {Tablespace Name}
    {ADD Temporary File Clause
    DROP Temporary File Clause
    ALTER Temporary File Size Clause
    RENAME Temporary File Clause}
```

ADD Temporary File Clause

```
ADD {TEMPFILE} {Temporary File Clause}
    AUTOEXTEND [AUTOEXTEND Clause]
    MAXSIZE [MAXSIZE Clause]
```

This is used to extend the data storage space in a disk temporary tablespace. The available options are the same as the temporary file options that are available when a disk tablespace is created.

DROP Temporary File Clause

```
DROP {TEMPFILE} {Temporary File Name}
```

This is used to reduce the amount of data storage space in a disk temporary tablespace. While the data storage space can be freely extended by adding more data files, a data file can be deleted only when it is not in use, that is, when no extents have been allocated to the data file.

6.6 Managing Tablespaces

ALTER Temporary File Size Clause

```
ALTER {TEMPFILE} {Temporary File Name}
    [{AUTOEXTEND [AUTOEXTEND Clause]}
    {SIZE [SIZE Clause]}]
```

Various attributes of each temporary file in a disk temporary tablespace, including its current size, maximum size, extension increment size, and whether it is automatically increased in size, can be changed. The specified current size and maximum size must be greater than the amount that is currently being used.

RENAME Temporary File Clause

```
RENAME {TEMPFILE} {The path and name of the existing temporary file}
    TO {The path and name of a new temporary file}
```

This command can be used to change the location of a data file and thereby change the file system in which the data in a tablespace are stored. This clause can be used in any startup phase, regardless of whether the applicable tablespace is online or offline. However, it can only be used with offline tablespaces in the service phase.

6.6.3.3 ALTER Memory Tablespace Clause

This can be used with system or user-defined tablespaces in memory, and has the following options. Checkpoint paths can be added, deleted or changed during any startup phase. However, during the service phase, only tablespaces that are offline can be modified.

```
ALTER TABLESPACE {Tablespace Name}
    {ADD Checkpoint Path Clause
    DROP Checkpoint Path Clause
    RENAME Checkpoint Path Clause
    ALTER Tablespace Size Clause}
```

ADD Checkpoint Path Clause

```
ADD CHECKPOINT PATH {Directory Path}
```

This is used to set an additional checkpoint image path.

DROP Checkpoint Path Clause

```
DROP CHECKPOINT PATH {Directory Path}
```

This is used to delete an existing checkpoint image path.

RENAME Checkpoint Path Clause

```
RENAME CHECKPOINT PATH {The existing directory path}
    TO {A new directory path}
```

This is used to change an existing checkpoint image path to a new path.

ALTER Tablespace Size Clause

```
ALTER
  { {AUTOEXTEND [AUTOEXTEND Clause] }
    {SIZE [SIZE Clause] } }
```

This is used to change the attributes of a memory tablespace, such as its maximum size, extension increment size, and whether it is automatically increased in size.

6.6.3.4 ALTER Volatile Tablespace Clause

This is used with volatile user-defined tablespaces, and has the following option:

```
ALTER TABLESPACE { Tablespace Name }
  {ALTER Tablespace Size Clause}
```

ALTER Tablespace Size Clause

```
ALTER
  { {AUTOEXTEND [AUTOEXTEND Clause] }
    {SIZE [SIZE Clause] } }
```

This is used to change the attributes of a volatile tablespace, such as its maximum size, extension increment size, and whether it is increased in size automatically.

6.6.3.5 ALTER Tablespace State Clause

The state of a tablespace can be either online or offline, which can be set using the following clause:

```
ALTER TABLESPACE { Tablespace Name }
  {ONLINE/OFFLINE/DISCARD}
```

ONLINE is the normal state of a tablespace. In this state, its objects can be accessed by users. In contrast, when a tablespace is offline, only tablespace-related DLL statements can be executed on it; the objects it contains cannot be accessed by users in other ways. This offline state can be used to overcome limitations, to perform a RENAME operation during the service phase, etc. However, system tablespaces must always remain online; that is, they cannot be taken offline. This clause cannot be used with volatile tablespaces.

The DISCARD option is used when ALTIBASE HDB can't be started due to a data error in one of the tablespaces currently in use.¹ DISCARDing the tablespace allows the user to start up ALTIBASE HDB with the remaining tablespaces. Because the only operation that can be performed on a discarded tablespace is the DROP operation, care should be taken when using this option. Additionally, tablespaces can be discarded only during the control phase. This option can be performed on both disk and memory data tablespaces.

-
1. For example, assume that the DBA has mistakenly deleted a checkpoint image file for a particular memory tablespace. In this case, since the memory tablespace cannot be loaded when the server is started, the DBA might first consider re-creating the deleted checkpoint image by performing media recovery. However, if archive logging has not been conducted, media recovery will be impossible, and thus this method will be unusable. In such cases, as long as the tablespace can be deleted without causing a problem, the DBA can discard the tablespace, restart the database without the tablespace, and then remove the tablespace.

6.6.4 Tablespace Backup and Recovery

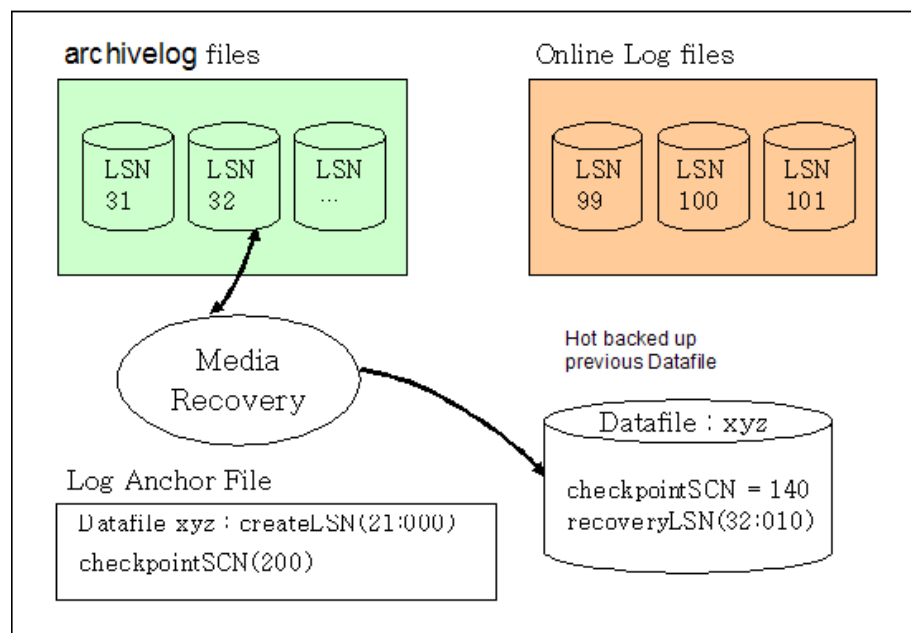
This section provides a simple overview of the concept of online and offline tablespace backup. For a more detailed explanation of backup and recovery in ALTIBASE HDB, please refer to the Backup and Recovery chapter of this manual and the *Getting Started Guide*.

6.6.4.1 Tablespace Online Backup (HOT Backup)

The term "tablespace online backup" refers to backup that is conducted while the tablespace is actively providing service. Because online backup does not influence the execution of transactions, it can be performed during the service phase. Online backup has the following characteristics:

- Online backup is only possible when the database is operating in archive log mode.
- In archive log mode, because all log files are backed up in a separate storage space, a sufficiently large storage space must be set aside, even if checkpointing and log flushing have just been conducted.
- Use the ALTER DATABASE BACKUP statement to perform online backup while the database is running.
- Even if a fault causes data files to be damaged or lost, media recovery can be performed to restore data files to the current point in time.

Figure 6-14 The Concepts of Media Recovery



- If data file xyz, which exists in a disk tablespace, is damaged, it can be restored using a data file that was previously created during a hot backup. A memory tablespace can be recovered using a checkpoint image file that was previously created during a hot backup.
- On the basis of the final checkpoint SCN (140) and recovery LSN (32:010), which are written in the header of the data file that was created during the backup, the file can be restored to the

current final checkpoint SCN (200).

- When the system is restarted, the most recent image of a data file or memory tablespace can be recovered by repeating all recent transactions using online logs and rolling back all uncommitted transactions using UNDO logs.

6.6.4.2 Offline Backup of Tablespaces (Cold Backup)

When a tablespace is backed up offline, the tablespace service is suspended while the backup is performed. Offline backup is faster than online backup, and thus enables recovery to be performed more quickly. Offline backup has the following characteristics:

- Offline backup is possible when the database is operating in noarchivelog mode.
- Offline backup is performed by copying data files, log files and log anchor files after the database is shut down normally.
- When a data file is damaged or lost due to a fault, it can be restored only up to the time point at which offline backup was most recently performed.

6.6.4.3 Offline Recovery

Recovery is a process in which the consistency of a database is restored using a backup image. Recovery cannot be performed while the database is online; it must be performed offline.

Recovery is performed by replacing the existing database with offline backup files while database service is stopped and then restarting the database.

6.7 Examples of Tablespace Use

This section provides examples of the use of memory tablespaces and volatile tablespaces.

6.7.1 Memory Tablespaces

6.7.1.1 Creating a Memory Tablespace - Basics

The simplest and easiest way to create a memory tablespace is to use the CREATE MEMORY TABLESPACE, specifying the initial size with the SIZE clause.

```
iSQL> CREATE MEMORY TABLESPACE USER_MEM_TBS SIZE 256M;
Create success.
```

Here, because the automatic extension mode was not set, it will default to OFF. When all 256 MB of the tablespace have been used, any attempt to allocate more space to the tablespace¹ will result in an error message saying that there is not enough room in the tablespace.

Additionally, one or more checkpoint paths, as specified in the MEM_DB_DIR property, will be used as the checkpoint paths for the new tablespace.

Supposing that two checkpoint paths are specified in the altibase.properties. Two paths have been saved for the MEM_DB_DIR property: dbs1 and dbs2, both of which are located in the ALTIBASE HDB home directory.

```
# altibase.properties
MEM_DB_DIR = ?/dbs1
MEM_DB_DIR = ?/dbs2
```

The following query can be executed to verify that dbs1 and dbs2, which were specified using the MEM_DB_DIR property, are the checkpoint paths for the USER_MEM_TBS tablespace created above.

```
iSQL> SELECT CHECKPOINT_PATH FROM V$MEM_TABLESPACE_CHECKPOINT_PATHS
WHERE SPACE_ID =
  (SELECT SPACE_ID
   FROM V$MEM_TABLESPACES
   WHERE SPACE_NAME='USER_MEM_TBS');
CHECKPOINT_PATH
-----
/alibase_home/dbs1
/alibase_home/dbs2
2 rows selected.
```

First, let's take a look at the files in the checkpoint folders. The 6 files shown below can be found in the dbs1 directory:

```
SYS_TBS_MEM_DATA-0-0
SYS_TBS_MEM_DATA-1-0
SYS_TBS_MEM_DIC-0-0
SYS_TBS_MEM_DIC-1-0
USER_MEM_TBS-0-0
USER_MEM_TBS-1-0
```

-
1. If a table is created in a tablespace, if data are entered into an existing table, or if the data in an existing table are changed, additional space is allocated from the tablespace.

All of these files are checkpoint image files for the memory tablespace. Their filename format is 'Tablespace Name-{Ping Pong Number}-{File Number}'. 'Ping Pong No.' is either 0 or 1, each of which indicates one of the two checkpoint images used for ping-pong checkpointing¹. In addition, because each of the checkpoint images can be stored as multiple files, 'File Number', at the end of the filename, indicates the number of each checkpoint image file, which begins at 0 and increments by 1. The size of the checkpoint image files is specified by using the SPLIT EACH clause with the CREATE TABLESPACE statement. Since the SPLIT EACH clause was not used in the CREATE MEMORY TABLESPACE statement above, the checkpoint image will be split into files 1 GB in size, which is the default value specified using the DEFAULT_MEM_DB_FILE_SIZE property. Because the space used by the above three tablespaces has not reached 1 GB yet, the only file number that can be seen is 0.

In the above example, SYS_TBS_MEM_DIC is a system dictionary tablespace containing metadata. This tablespace is automatically created when a database is created.

SYS_TBS_MEM_DATA is the default system data tablespace. When a user creates a table without specifying a tablespace, the data in the table are stored in this tablespace.

Finally, USER_MEM_TBS is the user-defined data tablespace that was created above.

For reference, the initial size, which is specified in the SIZE clause in the CREATE MEMORY TABLESPACE statement, must be a multiple of the extension increment size. For example, if the EXPAND_CHUNK_PAGE_COUNT property, which indicates the number of pages by which a memory tablespace will be incremented when it is expanded, is set to 128, because the size of one memory page is 32 KB, the default extension increment size of a memory tablespace will be 4 MB and the initial size can be set to a multiple of 4M.

If the size specified in the SIZE clause cannot be divided by the extension increment size, the following error will occur:

```
iSQL> CREATE MEMORY TABLESPACE USER_MEM_TBS SIZE 1M;
[ERR-110EE : The initial size of the tablespace must be a multiple of the
expand chunk size ( EXPAND_CHUNK_PAGE_COUNT * PAGE_SIZE(32K) = 4096K )]
```

6.7.1.2 Creating Memory Tablespaces - Details

This section examines various ways to create a memory tablespace.

In the following example, the initial size of the tablespace is set to 256 MB, the automatic extension mode is set to ON, and the tablespace is configured to extend by 128 MB every time it is extended, to a maximum of 1 GB.

```
iSQL> CREATE MEMORY TABLESPACE USER_MEM_TBS SIZE 256M
AUTOEXTEND ON NEXT 128M MAXSIZE 1G;
Create success.
```

Like its initial size, the automatic extension increment size of a tablespace must be set to a multiple of the EXPAND_CHUNK_PAGE_COUNT property * the size of one page, which is the default number of pages by which a tablespace is extended. For more information, please refer to 'Creating Memory Tablespace – Basics.'

-
1. To ensure the durability of tablespace data in memory, data are saved in disk files. The files in which tablespace data are stored are called images. In ping-pong checkpointing, which is used in ALTIBASE HDB, a pair of checkpoint images is maintained, and tablespace data are stored alternately in each of them.

6.7 Examples of Tablespace Use

A tablespace can be created with no MAXSIZE, as shown below. If the MAXSIZE clause is not specified, the system operates as if it were set to UNLIMITED.

```
iSQL> CREATE MEMORY TABLESPACE USER_MEM_TBS SIZE 256M
AUTOEXTEND ON NEXT 128M MAXSIZE UNLIMITED;
Create success.
```

In this case, USER_MEM_TBS is extended, but not past the point where the total space allocated to all memory tablespaces in the system exceeds MEM_MAX_DB_SIZE.

Checkpoint paths can also be specified when creating a memory tablespace, as seen below:

```
iSQL> CREATE MEMORY TABLESPACE USER_MEM_TBS SIZE 256M
CHECKPOINT PATH 'dbs1', '/new_disk/dbs2';
Create success.
```

In the above example, the relative path "dbs1" was specified for the checkpoint path, which has the same effect as if "\$ALTIBASE_HOME/dbs1" were specified. Additionally, the DBA must first manually create the checkpoint paths specified in the CREATE TABLESPACE statement in the actual file system and then grant write and file execution privileges for them before creating a tablespace.

The size of the files into which a checkpoint image is divided can also be specified, as seen below:

```
iSQL> CREATE MEMORY TABLESPACE USER_MEM_TBS SIZE 256M SPLIT EACH 512M;
Create success.
```

Like the initial size and the expansion increment size, the size of the files into which a checkpoint image is divided must be set to a multiple of the number of pages specified using the EXPAND_CHUNK_PAGE_COUNT property * the size of one page. For more information, please refer to 'Creating Memory Tablespace – Basics.'

A tablespace can be created offline and then taken online before it is used. Since a memory tablespace takes up the amount of system memory that was specified when it was created, in cases where a tablespace will not be used immediately after it is created, this practice can help optimize the use of system resources.

```
iSQL> CREATE MEMORY TABLESPACE USER_MEM_TBS SIZE 256M OFFLINE;
Create success.
iSQL> ALTER TABLESPACE USER_MEM_TBS ONLINE;
Alter success.
```

Here is an example that combines the memory tablespace creation options seen above:

```
iSQL> CREATE MEMORY TABLESPACE USER_MEM_TBS SIZE 256M
AUTOEXTEND ON NEXT 128M MAXSIZE 1G
CHECKPOINT PATH 'dbs1', '/new_disk/dbs2'
SPLIT EACH 512M OFFLINE;
Create success.
```

6.7.1.3 Adding a Checkpoint Path to a Memory Tablespace

This section describes how to add a checkpoint path to a memory tablespace.

The checkpoint paths for a memory tablespace can only be set during the control phase. After shutting down the ALTIBASE HDB server, restart it in the control phase.

```
$ isql -u sys -p manager -sysdba
iSQL(sysdba)> startup process
iSQL(sysdba)> startup control
```

In the control phase, the V\$TABLESPACES performance view, which pertains to all tablespaces, can be queried.

The V\$MEM_TABLESPACES performance view, which displays the attributes that are unique to memory tablespaces, can only be viewed in or after the meta phase. In the control phase, it is thus necessary to view memory tablespaces using V\$TABLESPACES.

The V\$MEM_TABLESPACE_CHECKPOINT_PATHS performance view can be used to view the checkpoint paths belonging to the USER_MEM_TBS tablespace, which was created earlier.

If the data in the tablespace change frequently, resulting in increased disk I/O during checkpointing, this can be alleviated by adding a new checkpoint path to a disk that is physically different from the disk used by the existing checkpoint path, as follows:

Let's add the "/new_disk/dbs3" path to USER_MEM_TBS.

In order to do this, the checkpoint path and directory to be added must first be created, and the ALTIBASE HDB process must be granted write and execute privileges for that directory. Supposing that ALTIBASE HDB is started using the "altibase" operating system user account, this would be conducted as follows:

```
$ su - root
$ mkdir /new_disk/dbs3
$ chown altibase /new_disk/dbs3
```

As shown below, a checkpoint path can now be added using the ADD CHECKPOINT PATH statement:

```
iSQL(sysdba)> ALTER TABLESPACE USER_MEM_TBS ADD CHECKPOINT PATH '/new_disk/
dbs3';
Alter success.
```

It is the DBA's responsibility to move or copy the checkpoint image files from the existing checkpoint path to the newly added checkpoint path. After a checkpoint path is added, if a new checkpoint image file is needed, the file is created in the new checkpoint path by ALTIBASE HDB.¹

6.7.1.4 Changing the Checkpoint Path for a Memory Tablespace

This section describes how to change a checkpoint path for a memory tablespace.

Checkpoint paths for memory tablespaces can only be set during the control phase. As noted in the [Adding a Checkpoint Path to a Memory Tablespace](#) section above, after shutting down the ALTIBASE HDB server, restart it in the control phase.

This example illustrates how to move an existing checkpoint path, which is "dbs1" in the ALTIBASE HDB home directory, to the newly installed disk "/new_disk".

To change a checkpoint path, the absolute path of the existing checkpoint path must be correctly entered. For information on viewing a checkpoint path for a tablespace during the control phase, please refer to the [Adding a Checkpoint Path to a Memory Tablespace](#) section above.

-
1. When checkpoint image files are created for a tablespace during checkpointing, they are alternately created in each of the checkpoint paths for that tablespace.

6.7 Examples of Tablespace Use

Just as when adding a checkpoint path, when changing a checkpoint path, the DBA must first manually create the directory and grant write and execute privileges for the directory to the OS user account under which ALTIBASE HDB is started. Again, it is assumed that the username under which the ALTIBASE HDB process is started is 'altibase.'

```
$ su - root
$ mkdir /new_disk/dbs1
$ chown altibase /new_disk/dbs1
```

Now the checkpoint path can be changed from the "dbs1" checkpoint directory, which is located in the ALTIBASE HDB home directory, to the "/new_disk/dbs1" path on the newly added disk using the RENAME CHECKPOINT PATH statement.

```
iSQL(sysdba)> ALTER TABLESPACE USER_MEM_TBS RENAME CHECKPOINT PATH
'/opt/altibase_home/dbs1' TO '/new_disk/dbs1';
Alter success.
```

Finally, all checkpoint images for the USER_MEM_TBS tablespace, which are located in the existing \$ALTIBASE_HOME/dbs1 directory, are moved to the /new_disk/dbs1 directory.

```
$ mv $ALTIBASE_HOME/dbs1/USER_MEM_TBS* /new_disk/dbs1
```

6.7.1.5 Removing a Checkpoint Path from a Memory Tablespace

This section describes how to remove a checkpoint path from a memory tablespace.

As noted above, checkpoint paths for memory tablespaces can only be set during the control phase, and thus it is first necessary to shut down the ALTIBASE HDB server and restart it in the control phase.

This example illustrates how to remove an existing checkpoint path, namely the "dbs2" directory located in the ALTIBASE HDB home directory.

To change a checkpoint path, the absolute existing checkpoint path must be entered correctly. For information on how to view the checkpoint paths for a tablespace during the control phase, please refer to the [Adding a Checkpoint Path to a Memory Tablespace](#) section above.

The \$ALTIBASE_HOME/dbs2 checkpoint path can now be removed using the DROP CHECKPOINT PATH statement as follows:

```
iSQL(sysdba)> ALTER TABLESPACE USER_MEM_TBS
DROP CHECKPOINT PATH '/opt/altibase_home/dbs2'
Alter success.
```

Finally, all of the checkpoint images for the USER_MEM_TBS tablespace that are located in the existing \$ALTIBASE_HOME/dbs2 directory must be moved to one of the other checkpoint paths defined for the USER_MEM_TBS tablespace.

```
$ mv $ALTIBASE_HOME/dbs2/USER_MEM_TBS* /new_disk/dbs1
```

6.7.1.6 Changing the Auto Extension Setting for a Memory Tablespace

This section describes how to change the auto extension settings for a memory tablespace.

If the AUTOEXTEND clause is not specified when a memory tablespace is created, by default, the tablespace is not automatically extended.

```
iSQL> CREATE MEMORY TABLESPACE USER_MEM_TBS SIZE 256M;
Create success.
```

In this case, the the simplest way to change a tablespace so that it extends automatically is as follows:

```
iSQL> ALTER TABLESPACE USER_MEM_TBS
ALTER AUTOEXTEND ON;
Alter success.
```

In the above example, the tablespace will be extended in increments equal to the number of pages specified in the EXPAND_CHUNK_PAGE_COUNT property, which is the default unit of extension for tablespaces.

In addition, the tablespace will be able to increase in size the same as if the maximum size were set to UNLIMITED, up to the limit at which the total size of all memory tablespaces in the system would exceed the MEM_MAX_DB_SIZE property.

Unlike disk tablespaces, checkpoint image files for memory tablespaces do not need to be managed by the DBA. This is because checkpoint image files are automatically created by ALTIBASE HDB when it is necessary to automatically increase the size of a database.

To specify the unit of extension for a memory tablespace, use a statement like the following:

```
iSQL> ALTER TABLESPACE USER_MEM_TBS
ALTER AUTOEXTEND ON NEXT 128M;
Alter success.
```

To specify the maximum size of a memory tablespace, use a statement like the following:

```
iSQL> ALTER TABLESPACE USER_MEM_TBS
ALTER AUTOEXTEND ON MAXSIZE 1G;
Alter success.
```

To specify both the unit of extension and the maximum size of a memory tablespace, use a statement like the following:

```
iSQL> ALTER TABLESPACE USER_MEM_TBS
ALTER AUTOEXTEND ON NEXT 128M MAXSIZE 1G;
Alter success.
```

To turn off the automatic extension setting for a memory tablespace, use a statement like the following:

```
iSQL> ALTER TABLESPACE USER_MEM_TBS
ALTER AUTOEXTEND OFF;
Alter success.
```

6.7.1.7 Bringing a Memory Tablespace Online or Taking It Offline

This example describes how to change a memory tablespace from an online state to an offline state and vice-versa.

All of the data in an ALTIBASE HDB memory tablespace are loaded into memory. To accomplish this, an amount of system memory equal to the amount of memory currently being used by a memory tablespace is allocated to the tablespace. ALTIBASE HDB provides functions for allocating memory to memory tablespaces and freeing memory so that DBAs can manage memory usage more easily.

Of course, if the memory that is being used by a memory tablespace is freed, all of the objects that

6.7 Examples of Tablespace Use

exist in that tablespace become temporarily inaccessible. To free the memory space being used by a memory tablespace, take the tablespace offline.

```
iSQL> ALTER TABLESPACE USER_MEM_TBS OFFLINE;  
Alter success.
```

When it is subsequently desired to use a table that exists in the memory tablespace that was taken offline, bring the tablespace online as follows:

```
iSQL> ALTER TABLESPACE USER_MEM_TBS ONLINE;  
Alter success.
```

6.7.2 Volatile Tablespaces

6.7.2.1 Creating a Volatile Tablespace

The statements for creating, changing and deleting volatile tablespaces are essentially the same as those for memory tablespaces. The main difference between them is that the statements related to checkpoint image files are of no use with volatile tablespaces.

A volatile tablespace 256 MB in size can be created using the following statement:

```
iSQL> CREATE VOLATILE DATA TABLESPACE USER_VOL TBS  
SIZE 256M;  
Create success.
```

In the above example, the size of the tablespace is fixed at 256 MB, that is, it is not automatically extended. A tablespace that automatically increases in size can be created using the following statement:

```
iSQL> CREATE VOLATILE DATA TABLESPACE USER_VOL_TBS  
SIZE 256M AUTOEXTEND ON;  
Create success.
```

In the above example, the initial size of the tablespace is 256 MB, but it can be automatically extended up to the size specified using the VOL_MAX_DB_SIZE property. The increment by which it automatically increases in size is 4 MB. To create a volatile tablespace for which the unit of automatic extension is 8 MB and the maximum size is 512 MB, execute a statement like the following:

```
iSQL> CREATE VOLATILE DATA TABLESPACE USER_VOL  
TBS SIZE 256M AUTOEXTEND ON NEXT 8M MAXSIZE 512M;  
Create success.
```

6.7.2.2 Modifying a Volatile Tablespace

The automatic extension mode, automatic extension increment size and maximum size settings for volatile tablespaces can be changed.

The following statement enables automatic extension for a volatile tablespace for which automatic extension was previously disabled:

```
iSQL> ALTER TABLESPACE USER_VOL_TBS ALTER AUTOEXTEND ON;  
Alter success.
```

If the above statement is executed on a tablespace for which automatic extension mode is already enabled, an error will occur.

The following statement enables automatic extension mode, sets the automatic extension increment size to 8 MB, and sets the maximum size of the tablespace at 512 MB.

```
iSQL> ALTER TABLESPACE USER_VOL_TBS
ALTER AUTOEXTEND ON NEXT 8M MAXSIZE 512M;
Alter success.
```

The following statement turns off the automatic extension mode. Before using this statement, the automatic extension mode must previously have been set to ON.

```
iSQL> ALTER TABLESPACE USER_VOL_TBS ALTER AUTOEXTEND OFF;
Alter success.
```

In order to change the automatic extension increment size for a tablespace from 4 MB to 8 MB, it is necessary to execute two statements, as seen below:

```
iSQL> ALTER TABLESPACE USER_VOL_TBS ALTER AUTOEXTEND OFF;
Alter success.
iSQL> ALTER TABLESPACE USER_VOL_TBS ALTER AUTOEXTEND ON NEXT 8M;
Alter success.
```

6.7.3 DROP Tablespace - for Disk, Memory and Volatile Tablespace

6.7.3.1 Discarding Tablespace – Removing Tablespace with Corrupt Data

This section describes how to discard a tablespace.

If the DBA accidentally deletes a data file of a disk tablespace or a checkpoint file of a memory tablespace, or if the contents of such a file are lost due to a media fault, it will become impossible to start ALTIBASE HDB.

In such cases, the first thing to try is to restore the lost or damaged file through media recovery. However, media recovery can only be performed when archive logging has been performed such that copies of all existing log files remain available in a separate archive.

If this is not the case, and media recovery is therefore impossible, the tablespace associated with the lost data file or checkpoint image file can be discarded using the ALTER TABLESPACE DISCARD statement, and ALTIBASE HDB can then be started with only the remaining tablespaces.

Once a tablespace has been discarded using the ALTER TABLESPACE DISCARD statement, the objects in the tablespace become inaccessible, and the only action that can subsequently be performed on the tablespace is to DROP it. Therefore, this statement should be used with caution.

In the following example, the memory tablespace USER_MEM_TBS is created and then, assuming that the checkpoint images for this tablespace have been deleted, ALTIBASE HDB can be started up with the remaining tablespaces after the tablespace is discarded.

First, create a memory tablespace as shown below:

```
iSQL> CREATE MEMORY TABLESPACE USER_MEM_TBS SIZE 256M;
Create success.
```

Then shut down ALTIBASE HDB and delete the checkpoint files for the tablespace. When an attempt is made to start up ALTIBASE HDB, the following error will occur:

```
[SM-WARNING] CANNOT IDENTIFY DATAFILE
```

6.7 Examples of Tablespace Use

```
[TBS:USER_MEM_TBS, PPID-0-FID-0] Datafile Not Found

[SM-WARNING] CANNOT IDENTIFY DATAFILE
[TBS:USER_MEM_TBS, PPID-1-FID-0] Datafile Not Found

[FAILURE] The data file does not exist.
Startup Failed....
[ERR-91015 : Communication failure.]
```

This is the error that ALTIBASE HDB will raise if one of the data files or checkpoint image files for a tablespace does not exist.

Now it is time to discard USER_MEM_TBS.

The Discard statement can be executed only during the control phase. So, start up ALTIBASE HDB in the control phase.

```
$ isql -u sys -p manager -sysdba
iSQL(sysdba)> startup control
```

Now USER_MEM_TBS, which is missing a checkpoint file, can be discarded.

```
iSQL(sysdba)> ALTER TABLESPACE USER_MEM_TBS DISCARD;
Alter success.
```

Then execute the STARTUP SERVICE command to start up ALTIBASE HDB in the service phase.

```
iSQL(sysdba)> startup service
Command execute success.
```

Issuing the ALTER TABLESPACE DISCARD command is merely the first step in discarding the tablespace. The tablespace and its objects must be manually removed using the DROP TABLESPACE INCLUDING CONTENTS statement.

```
iSQL> DROP TABLESPACE USER_MEM_TBS INCLUDING CONTENTS AND DATAFILES;
Drop success.
```

If a data file in a disk tablespace is lost, or if some of the contents of a data file become corrupt due to a media fault, ALTIBASE HDB can be started up after discarding the corresponding tablespace in the same way.

6.7.3.2 Removing Tablespace

This example demonstrates how to remove a tablespace.

If a tablespace contains no objects, it can be easily removed, as seen below. However, this method does not remove the data files of a disk tablespace or the checkpoint image files for a memory tablespace.

```
iSQL> DROP TABLESPACE MY_TBS;
Drop success.
```

If a tablespace contains objects, all objects in the tablespace can be dropped by using the INCLUDING CONTENTS clause together with the DROP statement, as shown below. However, even when using this method, the data files or checkpoint image files are not removed from the file system.

```
iSQL> DROP TABLESPACE MY_TBS
INCLUDING CONTENTS;
Drop success.
```


To remove referential constraints from tables that exist in other tablespace and refer to tables in the tablespace being dropped, use the CASCADE CONSTRAINTS clause together with the INCLUDING CONTENTS clause. However, even when using this method, the data files or checkpoint image files are still not removed from the file system.

```
iSQL> DROP TABLESPACE MY_TBS  
INCLUDING CONTENTS CASCADE CONSTRAINTS;  
Drop success.
```

To delete the data files in a disk tablespace or the checkpoint image files for a memory tablespace, use the AND DATAFILES clause right after the INCLUDING CONTENTS clause.

```
iSQL> DROP TABLESPACE MY_TBS  
INCLUDING CONTENTS AND DATAFILES;  
Drop success.
```

```
iSQL> DROP TABLESPACE MY_TBS  
INCLUDING CONTENTS AND DATAFILES  
CASCADE CONSTRAINTS;  
Drop success.
```

6.8 Managing Space in Tablespaces

This section describes how to manage space in tablespaces.

- [Estimating the Size Required for the Undo Tablespace](#)
- [Estimating the Size of Memory Tables](#)
- [Estimating the Size of Disk Tables](#)
- [Calculating Table Storage Space](#)

6.8.1 Estimating the Size Required for the Undo Tablespace

The undo tablespace is used for storing undo segments. Because insufficient undo tablespace can negatively affect the performance of transactions, the size of the undo tablespace should be managed so that it is always appropriate. If update transactions, especially transactions involving statements that take a long time to execute, occur frequently in the system, undo segments will be repeatedly extended. This can lead to a shortage of undo tablespace.

The user can set the undo tablespace to automatic extension mode, or alternatively can estimate the approximate amount of space required and set the undo tablespace to fixed mode, setting the maximum size to the estimate of the required space.

6.8.1.1 Auto Extension Mode of Undo Tablespace

It is not easy for the user to predict how much undo tablespace will be required when first running an application. In such cases, it is advisable to set the undo tablespace to automatic extension mode so that it increases automatically to the required size.

In ALTIBASE HDB, automatic extension mode is provided for the undo tablespace so that the size of the undo tablespace can be easily estimated in an application development environment. The undo tablespace is set to automatic extension mode by default. This can be changed using the ALTER TABLESPACE statement.

6.8.1.2 Fixed Size Mode of Undo Tablespace

If it is desired to fix the size of the undo tablespace, the required size must be estimated. To achieve this, the user must observe and analyze patterns of space usage in TSS and undo segments while client applications are running.

The required undo tablespace size can usually be satisfactorily estimated in the following way:

Size of Undo Tablespace = Long-Term Transaction Time (sec) x
(the number of undo pages allocated per second +
the number of TSS pages allocated per second) x page size (8kB)

In this example, supposing that it takes 600 seconds (10 minutes) for a typical long-term transaction to execute, and that 1000 undo pages and 24 TTS pages are allocated every second, approximately 4.7G of undo tablespace would be required, as calculated thus: $10 \times 60 \times (1000 + 24) \times 8\text{kB} = 4800\text{MB}$.

However, if it is difficult to estimate the size of undo tablespace in this way, it is also acceptable to

simply allocate large amounts of space, as long as disk space permits.

6.8.1.3 Undo Tablespace Extension

If update transactions (especially long-term transactions, that is, those that take a long time to be committed) occur frequently in the system, it is possible to run out of undo tablespace. In this situation, it is necessary to increase the size of the tablespace, either by adding one or more suitably sized data file(s) or increasing the size of the file(s) in undo tablespace using the ALTER TABLESPACE statement.

6.8.2 Estimating the Size of Memory Tables

6.8.2.1 Calculating the Size of Data

In ALTIBASE HDB, the size of the data in a memory table can be estimated based on the type of data in each column, the padding for alignment of the column, etc. Expressed as a mathematical formula, this would appear as follows:

data size = [(the total estimated size plus padding for each column) * the number of data records]

The estimated size of each data type is shown in the following table.

(P = Precision, V = Value length)

| Data Type | Estimated Column Size |
|-----------|-----------------------|
| INTEGER | 4 |
| SMALLINT | 2 |
| BIGINT | 8 |
| DATE | 8 |
| DOUBLE | 8 |
| CHAR | 2 + P |
| VARCHAR | 10 + V |
| NCHAR | 2 + P |
| NVARCHAR | 10 + V |
| BIT | 4 + (P / 8) |
| VARBIT | 12 + (V / 8) |
| FLOAT | 4 + (P + 2) / 2 |
| NUMERIC | 12 + (P + 2) / 2 |

In the above table, P (Precision) indicates the size of the column, which is defined when the table is

6.8 Managing Space in Tablespaces

created. Data longer than P cannot be inserted into a column of the corresponding data type. V (Value) is the actual length of the inserted data, so it follows that V cannot be greater than P.

In fixed-length columns, such as those of type CHAR, NCHAR, BIT etc., space equal to P is always occupied, and therefore the length of the column is fixed regardless of the actual length of the data. However, for variable-length columns, such as those of type VARCHAR, NVARCHAR, VARBIT, etc., the amount of space occupied varies according to the length of the data.

Unlike disk tables, memory tables contain padded space to increase the speed of data access. The size of this space varies according to the data type and the position of the column.

6.8.2.2 Estimating the Size of an Index

Memory indexes are not saved in the tablespace in which table data are saved; rather, they are saved in an independent memory space. Because a pointer that points to the data storage location is saved in each bucket of a node of a memory index, the index size can be estimated, regardless of the data type, based on the pointer size and the number of records currently saved in the table.

index size = (number of data records) * p
(p = Pointer Size)

In the above formula, p is the pointer size, that is, the size required to save one pointer. On a 32-bit system this size is 4 bytes, whereas on a 64-bit system it is 8 bytes. In this formula, the size of the index is taken as being equal to the total size of all leaf nodes (i.e. the lowest nodes on a B-Tree) of the index. In addition to leaf nodes, a B-Tree also comprises internal nodes (i.e. nodes higher than leaf nodes), but their total size is 1/128 that of the leaf nodes, which is so small that they can be safely ignored. Additionally the size of additional information used to manage the index is about 1/16 the size of the leaf nodes, which is also negligibly small. Therefore it is acceptable to calculate the total size of the index on the basis of the total size of all of the leaf nodes.

However, the value estimated using this formula can differ from the actual size of the index because it considers only the case where all buckets of all leaf nodes have key values saved therein. That is to say, if there are many empty buckets within nodes, the actual size of the index can be much greater than the estimated size. In this case, the index can be reduced in size by rebuilding it.

6.8.2.3 Example 1

Let's try to estimate the size of the data when a table is created as shown below:

```
CREATE TABLE T1 ( C1 Integer, C2 char(1024), C3 varchar(1024) ) tablespace
user_data01;
```

In this table, column C1 and column C2 are fixed-length columns, whereas column C3 is a variable-length column. Therefore, the size of a record will vary depending on the size of column C3. If the size of one record is calculated in consideration of this, as seen below, the size of the data in table T1 equals (the length of one record * the number of records).

```
[record header] = 24 bytes
[column C1] = 4 bytes
[column C2] = 2+P bytes = 2+ 1024 bytes
[column C3] = 10+ V bytes
```

- If the length of the data in column C3 is 200 bytes:

[record size] = 24 + (4) + (2+1024) + (10+200) + padding = (1264 + padding) bytes

- If the length of the data in column C3 is 500 bytes:

[record size] = 24 + (4) + (2+1024) + (10+500) + padding = (1564 + padding) bytes

6.8.2.4 Example 2

Let's calculate the size of the index for table T1, whose creation statements are shown below, assuming that table T1 currently contains 500,000 records and that the system is a 64-bit system.

```
CREATE TABLE T1 ( C1 Integer, C2 char(300), C3 varchar(500)) tablespace
user_data01;
CREATE INDEX T1_IDX1 ON T1( C1, C2, C3 );
```

[index size] = 500,000 records * 8 = 3.814 Megabytes

6.8.2.5 Example 3

Now let's calculate the size of the data and the size of the index for the table created as shown below, which currently contains 1,000,000 records, on a 64-bit system.

```
CREATE TABLE TEST001 (C1 char(8) primary key, C2 char(128), N1 integer,
IN_DATE date) tablespace user_data01;
```

- the size of one record and the total data size

[The size of a record] = 24[header size] + (2+8) + (2+128) + (4) + (8) = 176 bytes

[The total size of all records] = [176] * 1,000,000 records = 167.84 Mbytes

- the index size

[total index size] = 8 * 1,000,000 records = 7.629 Megabytes

Note that this value is calculated only based on the size of data and leaf nodes, and thus in reality, additional space will be used for the page header, index nodes, and memory for managing free pages.

6.8.3 Estimating the Size of Disk Tables

In ALTIBASE HDB, the size of a disk table can be calculated on the basis of the data types and data contents, that is, it is equal to [total length of a row in the table * number of rows]. The following table shows the length of each data type.

6.8 Managing Space in Tablespaces

(P = Precision, V = Value length)

| Data Type | Estimated Column Size | | |
|-----------|-----------------------|----------------------------------|------------------------|
| | Null | 250 bytes and below 250 bytes | Greater than 250 bytes |
| Integer | 1 | 5 | X |
| SmallInt | 1 | 3 | X |
| BigInt | 1 | 9 | X |
| Date | 1 | 9 | X |
| Double | 1 | 9 | X |
| Char | 1 | 1+P | 3+P |
| Varchar | 1 | 1+V | 3+V |
| NChar | 1 | 1+P | 3+P |
| NVarchar | 1 | 1+V | 3+V |
| Bit | 1 | 5+(P/8) | 7+(P/8) |
| Varbit | 1 | 5+(V/8) | 7+(V/8) |
| Float | 1 | 4+(V+2) / 2 | 6 +(V+2) / 2 |
| Numeric | 1 | 4+(V+2) / 2 | 6 +(V+2) / 2 |

In the above table, P (Precision) indicates the maximum size of the column, which is set when the table is created. Data longer than P cannot be inserted into a column of that type. Additionally, for fixed-length columns, such as those of type CHAR, NCHAR, BIT, etc., space equal to P is always occupied, and therefore the length of the column is fixed regardless of the actual length of the data.

V (Value) denotes the actual length of the inserted data, which of course cannot be greater than P. Additionally, the amount of space occupied by variable-length columns, such as those of type VARCHAR, NVARCHAR, VARBIT, etc., varies according to the length of the data. Therefore, the column size can vary depending on the size of the data.

6.8.3.1 Estimating Row Size

This section describes how to calculate the row size for a table having the schema shown below:

```
CREATE TABLE T1 ( C1 char(32), C2 char(1024), C3 varchar(512) )  
tablespace user_data02;
```

In this schema, column C1 and column C2 are fixed-length columns, whereas column C3 is a variable-length column. Therefore, the size of a row will vary depending on the size of column C3. The size of a row will also vary depending on whether any columns contain NULL values. If the size of one row is calculated in consideration of this, as seen below, the size of the data in table T1 equals (the total length of one row * the number of rows).

[Row Header] 34 Bytes

[column C1] 1+P Bytes = 1+32 Bytes
 [column C2] 3+P Bytes = 3+1024 Bytes
 [column C3] 3+V Bytes

- If the size of the data in column C3 is 200 bytes:

[Total Length of One Record] = 34 + (1+32) + (3+1024) + (3+200) = 1297 Bytes

- If the size of the data in column C3 is 500 bytes:

[Total Length of One Record] = 34 + (1+32) + (3+1024) + (3+500) = 1597 Bytes

- If column C2 is NULL and the size of column C3 is 300 bytes:

[Total Length of One Record] = 34 + (1+32) + (1) + (3+300) = 371 Bytes

- If column C3 is NULL:

[Total Length of One Record] = 34 + (1+32) + (3+1024) + (0) = 1094 Bytes

6.8.3.2 Estimating the Size of an Index

In ALTIBASE HDB, the size of a disk index can be calculated on the basis of the actual data types and data contents. The following table shows the length of each data type to use when calculating the size of an index:

(P = Precision, V = Value length)

| Data Type | Size of Index Key | | |
|-----------|-------------------|----------------------------------|------------------------|
| | Null | 250 bytes and below 250 bytes | Greater than 250 bytes |
| Integer | 4 | 4 | X |
| SmallInt | 2 | 2 | X |
| BigInt | 8 | 8 | X |
| Date | 8 | 8 | X |
| Double | 8 | 8 | X |
| Char | 1 | 1+P | 3+P |
| Varchar | 1 | 1+V | 3+V |
| NChar | 1 | 1+P | 3+P |
| NVarchar | 1 | 1+V | 3+V |
| Bit | 1 | 5+(P/8) | 7+(P/8) |
| Varbit | 1 | 5+(V/8) | 7+(V/8) |
| Float | 1 | 4+(V+2) / 2 | 6+(V+2) / 2 |

6.8 Managing Space in Tablespaces

(P = Precision, V = Value length)

| Data Type | Size of Index Key | | |
|-----------|-------------------|----------------------------------|------------------------|
| | Null | 250 bytes and below 250 bytes | Greater than 250 bytes |
| Numeric | 1 | $4+(V+2) / 2$ | $6+(V+2) / 2$ |

In the above table, P (Precision) and V (Value) respectively indicate the maximum size of the column, which is set when the table is created, and the size of the data that are actually inserted into the table.

The size of one index is calculated as follows:

$[10 \text{ (header length)} + (\text{total length of key columns})] * \text{number of records}$

The above formula is used to calculate the approximate size of leaf nodes (the lowest nodes on a B-Tree). In addition to leaf nodes, a B-Tree also comprises internal nodes (nodes higher than leaf nodes), but they can be safely ignored when the key column size is small.

However, if the key column size is greater than 2kB, the depth of the B*Tree increases, and thus the size of internal nodes must be included in the calculation because their size can approach 50% of the total size of leaf nodes.

The following shows how to estimate the size of index T1 for table T1, the creation statements for both of which are shown below.

```
CREATE TABLE T1 ( C1 Integer, C2 varchar(500)) tablespace user_data02;  
CREATE INDEX T1_IDX1 ON T1( C1, C2 );
```

Column C1 is always 4 bytes in size because it an integer type column. The length of column C2, which is a variable-length column, varies depending on the size of the data.

[Key Header] 10 bytes
[column C1] 4 bytes
[column C2] 1+V bytes

- If the size of the data in column C2 is 50 bytes:

[Total Length] = $10 + 4 + (1+50) = 65$ bytes

- If the size of the data in column C2 is 500 bytes:

[Total Length] = $10 + 4 + (3+500) = 517$ bytes

- If column C2 is NULL:

[Total Length] = $10 + 4 + 1 = 15$ bytes

6.8.3.3 Table Size Calculation Example

The following shows how to estimate the size of the table created as shown below, assuming that it contains 1,000,000 records. The table size comprises the total size of all the records plus the size of the index.

```
CREATE TABLE TEST001 (C1 char(8) primary key, N1 double unique, C2 char(128),
```



```
N2 integer, IN_DATE date) tablespace user_data02;
```

- Row Size and Total Data Size

```
Row Size: 34[Header] + (1+8) + (1+130) + (1+4) + (1+8) = 188 bytes
Total Size of Data: [ 188 ] * 1,000,000 data = 179.29 Megabytes
```

- Index Size

```
Index Size for one Row: 10[Header] + (1+8)[C1] = 19 bytes
Total Index Size: 19 * 1,000,000 data = 18.12 Megabytes
```

- Total Amount of Disk Space Occupied by TEST001

```
179.29 (Data Size) + 18.12 (Index Size) = 197.41 M bytes
```

The above calculation takes into consideration only the size of the data. In reality, additional space is also occupied by the page header, internal nodes, space for managing segments, etc. When the space used for these purposes is also considered, the total amount of space occupied by the table is determined to be about 240 Megabytes.

6.8.4 Calculating Table Storage Space

Below, table TEST001, which was used above for the estimation of table size, will be used to show how to determine the table size that is suitable for storing all of the records and indexes in the table. The following must be kept in mind when determining the suitable table size.

6.8.4.1 Consider the Relative Frequency of Transaction Types

If a lot of update transactions are executed on the table, PCTFREE should be set to a high value for better transaction performance, and PCTUSED should be set to a low value to ensure sufficient free space for update transactions.

In contrast, if a lot of insert transactions are performed on the table but the number of update transactions is low, PCTFREE should be set to a low value and PCTUSED should be set to a high value in order to minimize the amount of unnecessary free space.

- PCTFREE

The default value is 10. It can be set anywhere from 0 to 99 when a disk table is created. This is the percentage of free space in each page that is set aside in advance for updating existing records when saving data in tables. Therefore, supposing that PCTFREE has been set to 10 and only insert transactions occur, if the total size of the table is 100MB, the amount of space that can be used for the records and the index of the table is 90M.

- PCTUSED

The default value is 40. It can be set anywhere from 0 to 99 when a disk table is created. After the amount of free space in a particular page drops below the percentage specified in PCTFREE, no more data will be inserted into the page until the amount of used space subsequently drops below 40% (e.g. 39%) as a result of update or delete transactions. Therefore, greater amounts of free space must be allocated to tables on which updates transactions occur frequently.

Table 6-3 Table Size Estimation based on Relative Frequency of Transactions by Type

| Circumstances | Table Size Estimation |
|--|---|
| Only SELECT transactions occur, or record size doesn't increase during UPDATE transactions | <p>In the case where PCTFREE is set to 5 and PCTUSED is set to 90:</p> <p>① Estimated minimum table size: TEST001(Total size=215.53MB) The minimum size in which to save the table is calculated as follows: total table size / [1-(PCTFREE / 100)] = 215.53/0.95 \doteq 227MB</p> <p>② Weighted estimation: A weighting factor is taken into account in the determination of the minimum size. The weighting differs depending on the circumstances. The following is just one example of how to include weighting in the size determination. Minimum Size * [1- (PCTUSED / 100)] * 2 = 227 * 0.1 * 2 \doteq 45M</p> <p>③ Therefore, a table 272M in size should be created.</p> |
| UPDATE transactions occur frequently and tend to increase the size of records | <p>In the case where PCTFREE is set to 20 and PCTUSED is set to 40:</p> <p>① Estimated minimum table size: TEST001(total size=213.63MB) The minimum size in which to save the table is calculated as follows: total table size / [1-(PCTFREE / 100)] = 213.63/0.8 \doteq 267MB</p> <p>② Weighted estimation: A weighting factor is taken into account in the determination of the minimum size. The following is one example of how to include weighting in the size determination. Minimum Size * [1- (PCTUSED / 100)] * 2 = 267 * 0.6 * 2 \doteq 320M</p> <p>③ Therefore, a table approximately 587M in size should be created.</p> |
| INSERT and UPDATE transactions occur frequently but UPDATE transactions do not increase the size of rows | PCTFREE is set to 10 and PCTUSED is set to 60. |

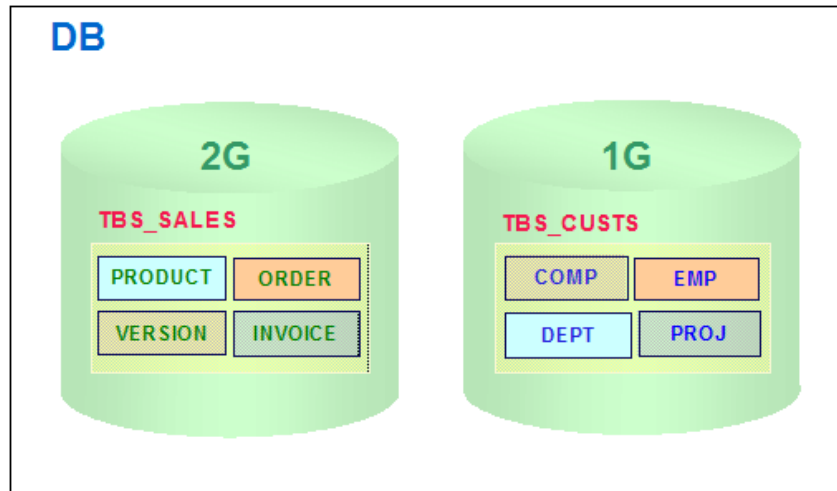
Note: The table size estimation method shown above should not be considered a rigid standard. It is also necessary to take into account the possibility that the amount of data will suddenly increase in the event of abnormal system operation..

6.8.4.2 Consider Suitable Backup Space

It is rare for a tablespace to have only one table saved in it. It is more efficient to group tables according to business purposes or backup strategies and store them collectively in a single tablespace.

In such cases, the size of a tablespace should be set in consideration of the time required to back up the tablespace.

The illustration below shows how tablespaces are organized in consideration of business purposes and backup strategies.

Figure 6-15 Determining Tablespace Size in Consideration of Backup Strategy

6.8.5 Tablespace Information

To help manage tablespaces, ALTIBASE HDB provides performance views and meta tables for monitoring the status of tablespaces.

`SYSTEM_.SYS_TBS_USERS`

Additionally, the following performance views can be used to obtain information about the size of the database, user access statistics, and the like:

`V$TABLESPACES`, `V$DATAFILES`, `V$MEM_TABLESPACES`

7 Partitioned Objects

This chapter contains the following sections:

- [What is Partitioning?](#)
- [Partitioned Objects](#)
- [Partition Conditions](#)
- [Partitioning Methods](#)

7.1 What is Partitioning?

Partitioning is the division of a large database object into several small pieces for easier management.

A large database object that has been partitioned is called a “partitioned object”, and each piece of a partitioned object is called a “partition”.

7.1.1 Partitioned Objects and Non-Partitioned Objects

When an end user accesses a partitioned object, the user cannot perceive any difference from a non-partitioned object. That is, from the user's perspective, both partitioned and non-partitioned objects are recognized as database objects, and it is not apparent whether a particular object has been partitioned. This allows the user to execute queries or DML statements (i.e. insert, delete and update records) in the same way regardless of whether an object is partitioned or not.

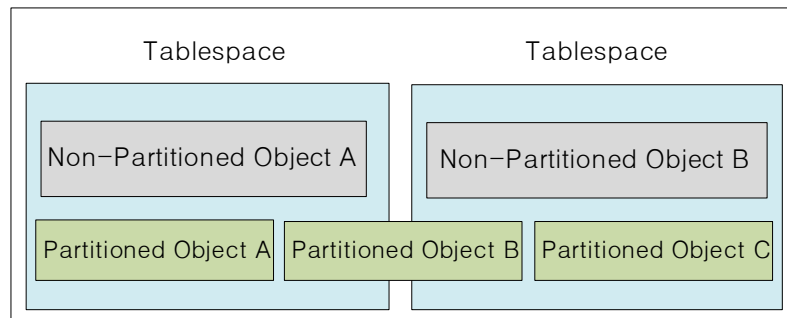
The differences between partitioned objects and non-partitioned objects pertain to database structure, as seen below:

7.1.1.1 Internal Structure

A non-partitioned object is dependent on, and stored in, only a single tablespace.

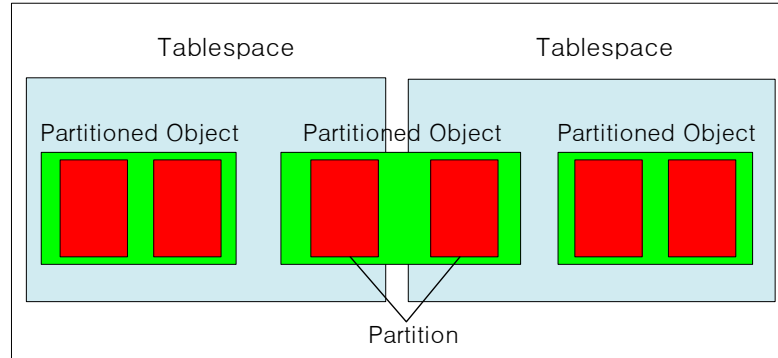
A partitioned object can be stored across multiple tablespaces. This is illustrated in [Figure 6-1].

Figure 7-1 The Relationships between Tablespaces, Partitioned Objects and Non-Partitioned Objects



Internally, a partitioned object comprises multiple partitions. Each partition has constraints on it, just like non-partitioned objects. Additionally, each partition is dependent only on a single tablespace.

A partitioned object causes partitions distributed across multiple tablespaces to appear as a single object. [Figure 6-2] shows the internal structure of a partitioned object.

Figure 7-2 The Internal Structure of a Partitioned Object

7.1.1.2 Advantages of Partitioned Objects

Thanks to the characteristics of its structure, a partitioned object realizes the following benefits:

- faster data loading and index rebuilding
- faster partial deletes
- faster table and index scans
- more flexible response to disk failures

7.1.2 Partition Key

A partition key is the basis on which tables are partitioned. A partition key consists of one or more of the columns of the table to be partitioned. These are called partition key columns.

A partition key column must have a data type that supports size comparisons (i.e. <, >, =). Only such columns can be used as partition key columns.

For example, when a record is inserted, it must be clear which partition will hold the inserted record. To satisfy this requirement, it must be possible to unambiguously compare partition key columns according to the relevant conditions. Therefore, data types on which size comparisons cannot be performed, such as the BINARY, GEOMETRY, BLOB and CLOB data types, cannot be used with partition key columns.

7.2 Partitioned Objects

Tables and indexes are the database objects that can be partitioned.

When a table is partitioned, it is called a “partitioned table”, and when an index is partitioned, it is called a “partitioned index”. Partitioned tables and partitioned indexes are collectively known as “partitioned objects”.

A partitioned object must satisfy the following rule:

$$\text{non-partitioned_object} \equiv \sum \text{partition} \dots \text{rule1}$$

The above rule means that a non-partitioned object must be equivalent to the sum of its partitions if it were partitioned. In other words, it is not possible to partition only part of a non-partitioned object.

7.2.1 Partitioned Tables

The term “partitioned table” refers to a large table that has been partitioned into multiple partitions based on partitioning conditions (range, list, or hash). This is illustrated in [Figure 6-3].

In [Figure 6-3], a non-partitioned table is partitioned based on color, resulting in a partitioned table that consists of three partitions.

From a structural point of view, the database object that conceptually bears the closest resemblance to a partitioned table is a union view. A union view allows multiple tables to be treated as a single object, but does not itself take up any physical space. Likewise, the partitions in a partitioned table occupy physical space, but the partitioned table itself does not.

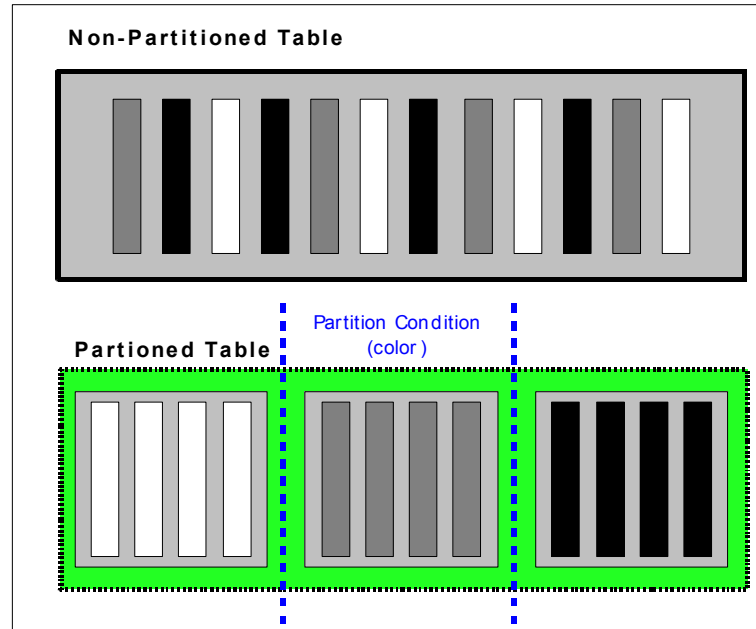
However, there are some characteristics of partitioned tables that differentiate them from union views.

- Updatability

Partitioned tables can be updated. In contrast, for union views, records can be updated by accessing individual tables, but not by accessing the union view.

- Index Range

Indexes can be built for partitioned tables. In contrast, indexes can be built for the individual tables that make up union views, but not for the union views themselves.

Figure 7-3 Partitioned Tables and Non-partitioned Tables

A partitioned table can thus be thought of as a union view that can be accessed to update records, and for which an index can be built.

Partitioned tables can be classified as follows based on the kind of storage medium on which the partitions are saved. ALTIBASE HDB only supports partitioned disk tables.

- **Partitioned Memory Table:** A partitioned table for which all partitions are stored in a memory tablespace.
- **Partitioned Disk Table:** A partitioned table for which all partitions are stored in a disk tablespace.

7.2.2 Partitioned Indexes

Indexes for partitioned tables can be classified according to the following:

- whether they are partitioned:
partitioned indexes vs. non-partitioned indexes
- the relationship between table and index:
global indexes vs. local indexes

7.2.2.1 Partitioned Indexes vs. Non-Partitioned Indexes

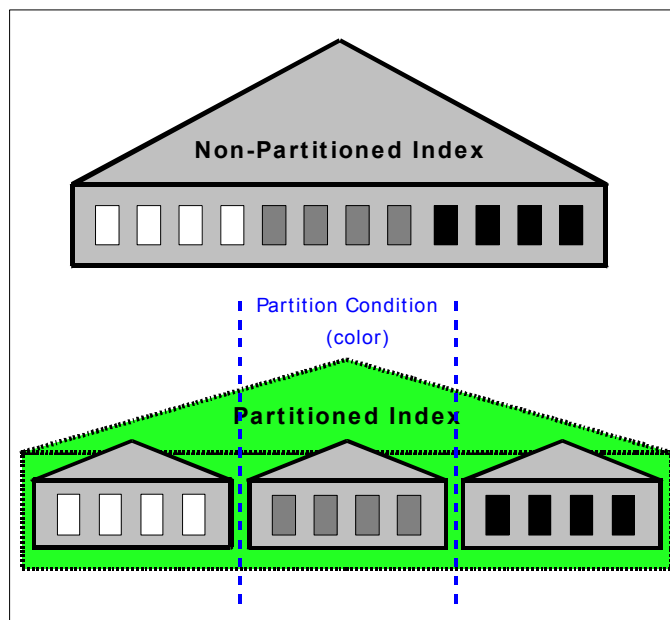
An index is classified either as a partitioned index or as a non-partitioned index based on whether it is partitioned.

The term “non-partitioned index” refers to an index that is not partitioned, whereas the term “parti-

7.2 Partitioned Objects

tioned index” refers to a large index that, just like a partitioned table, is divided into multiple partitions according to some partitioning criteria. This is illustrated in [Figure 6-4].

Figure 7-4 Partitioned Indexes vs. Non-Partitioned Indexes



[Figure 6-4] shows a non-partitioned index that has been partitioned based on color, resulting in a partitioned index that consists of three partitions.

Partitioned indexes, which are partitioned according to some partition conditions, are classified as either prefixed indexes or non-prefixed indexes based on the relationship between an index partition key and an index key.

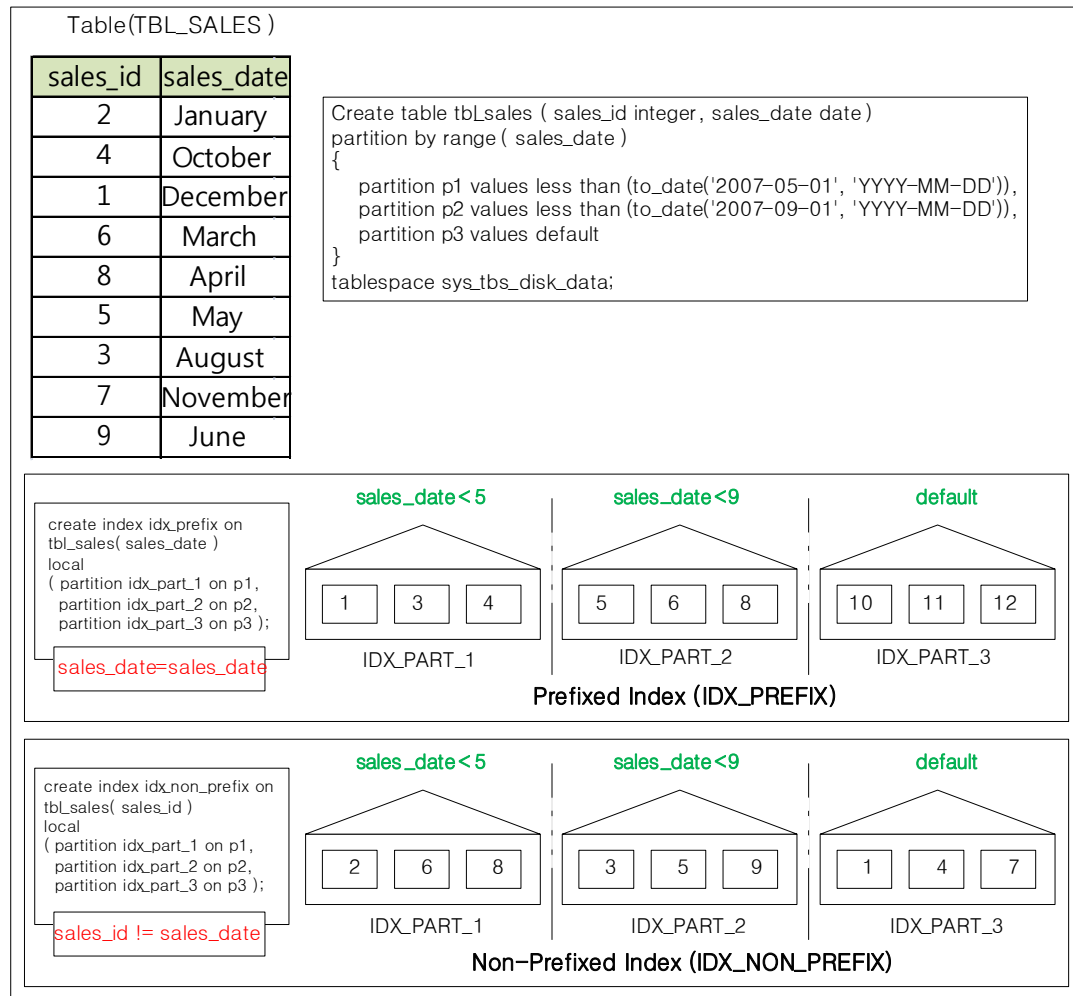
- **Prefixed Index**

In a prefixed index, the first column of an index key and the first column of an index partition key are the same column.

- **Non-prefixed Index**

In a non-prefixed index, the first column of an index key and the first column of an index partition key are not the same column.

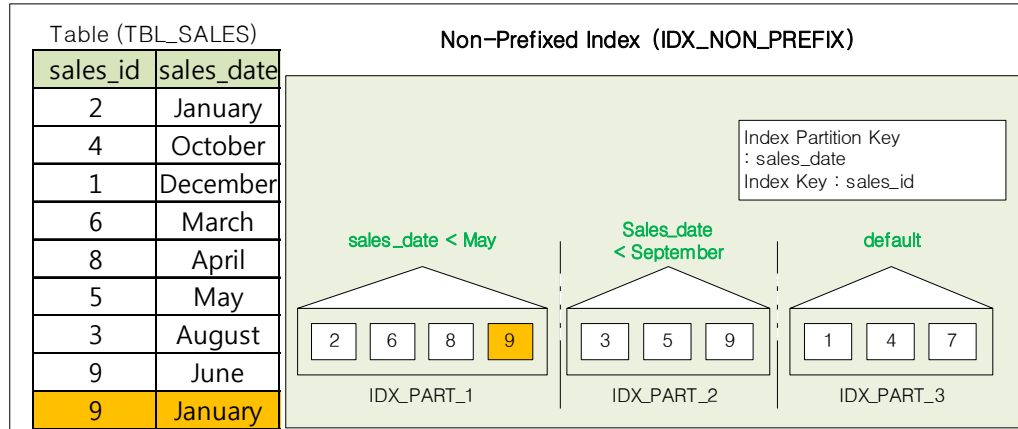
[Figure 6-5] shows the difference between a prefixed index and a non-prefixed index, using a table that consists of the “sales_id” and “sale_date” columns as an example.

Figure 7-5 Examples of Prefixed and Non-prefixed Indexes

In the above figure, the indexes are partitioned on the basis of the “sales_date” column. Each index is considered either a prefixed index or a non-prefixed index based on the key on which it was built.

In the figure, the prefixed index has the key “sales_date”. In other words, the index partition key and the index key are based on the same column. This type of index is called a “prefixed index”. In contrast, the non-prefixed index in the above figure is sorted by “sales_id”. Because this type of index is sorted according to a key other than the index partition key, it is called a “non-prefixed index”.

The reason for distinguishing between prefixed and non-prefixed indexes is related to the UNIQUE attribute. Since the index key of a prefixed index is the same as its index partition key, a unique check can be performed without having to check all of the partitions that belong to a partitioned index. However, with a non-prefixed index, all of the partitions in a partitioned index have to be checked. [Figure 6-6] displays an example of this:

Figure 7-6 Example of Unique Check with Non-Prefixed Index (Impossible)

In the example shown in [Figure 6-6], suppose that the index partition key of the table is the sales_date column and that duplicate values are not allowed for the sales_id column. If a non-prefixed index ("IDX_NON_PREFIX") is built using the sales_id column as the index key, consider whether a unique check can be performed using IDX_NON_PREFIX when records are inserted using the following SQL statement:

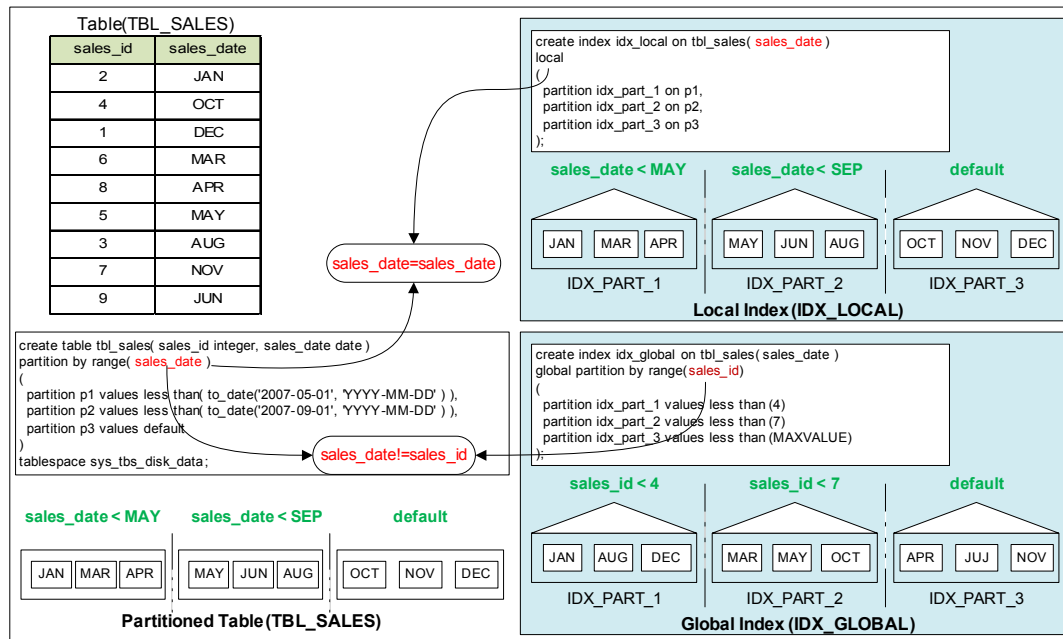
```
INSERT INTO TBL_SALES VALUES (9, January);
```

Because the value in the sales_date column of the record to be inserted is "January", a key will be inserted into IDX_PART_1. Even though a key value of 9 is present in the sales_id column in the IDX_PART_2 partition, the same value will be inserted without error into the IDX_PART_1 partition. Therefore, it is necessary to check all of the indexes when performing a unique check using a non-prefixed index.

7.2.2.2 Global Index and Local Index

An index is classified as either a global index or a local index based on the relationship between a table partition key and an index partition key. The term "global index" refers to an index in which the index partition key and the table partition key are different (index_partition_key != table_partition_key), whereas the term "local index" refers to an index in which the index partition key and the table partition key are the same (index_partition_key == table_partition_key).

Figure 7-7 Examples of Local and Global Indexes



[Figure 6-7] shows the difference between a global index and a local index with reference by way of example to a table that comprises the sales_id and sales_date columns. In the figure, the indexes are sorted by sales_date, and each index is categorized as either a local or global index based on which index partition key is used to partition it.

In the figure, a local index is partitioned into three partitions on the basis of the sales_date column. A partitioned index in which the index partition key (sales_date) and the table partition key (sales_date) are the same is called a local index.

In contrast, at the bottom of the figure, a global index is partitioned on the basis of the sales_id column. That is, the index partition key (sales_id) and the table partition key (sales_date) are not the same. This type of partitioned index is called a global index.

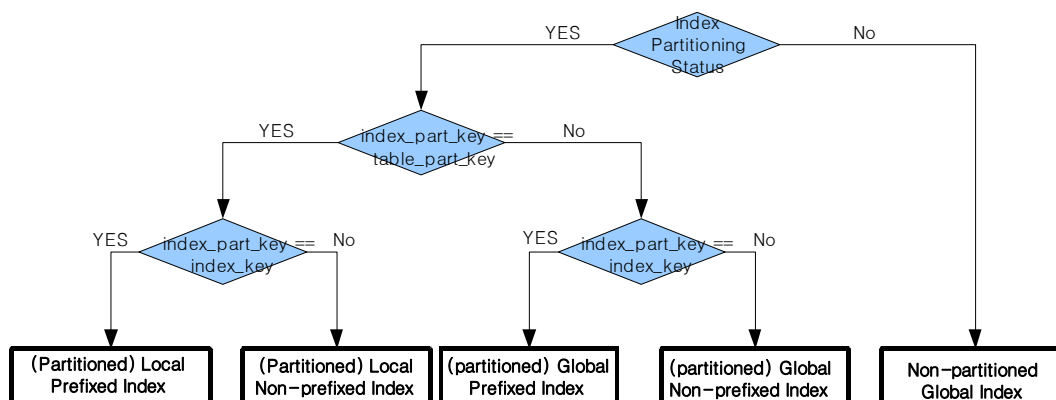
The reason that indexes are classified as global or local indexes is that they have different characteristics based on whether or not the table partition key and the index partition key are the same.

Building a partitioned index on the basis of a column other than the table partition key means that the keys in an index partition of a global index will point to several different table partitions. The result of this is that the use of a table modification statement (e.g. ALTER TABLEMERGE ...) on a partitioned table can cause the global index to be rebuilt.

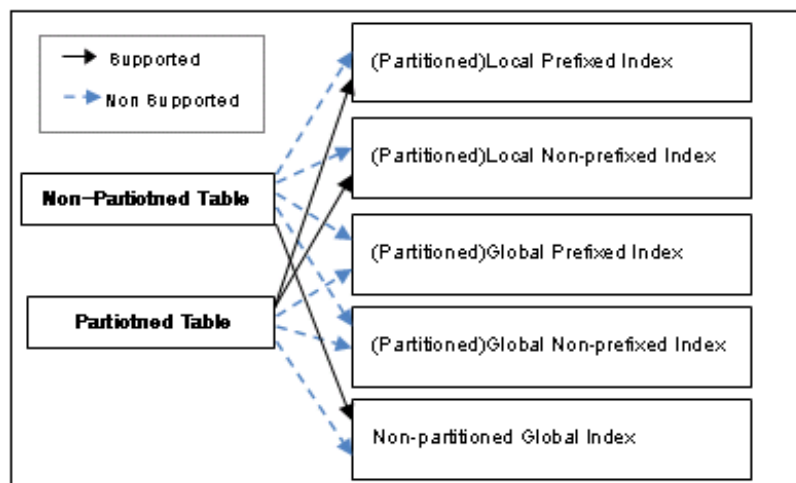
In contrast, for a local index, because a partitioned table modification statement only changes the local index of the partitions, overall consistency does not suffer.

7.2.2.3 Type of Index

[Figure 6-8] shows the types of indexes that have been examined so far:

Figure 7-8 Types of Indexes

At present, ALTIBASE HDB supports local indexes only. Global indexes are not supported. This is summarized in the following table:



Only non-partitioned indexes can be built for non-partitioned tables, and only local prefixed and non-prefixed indexes can be built for partitioned tables.

7.3 Partition Conditions

This section examines partition conditions and default partitions.

7.3.1 Partition Pre-Conditions

A partition condition is the basis on which partitioning is performed. Partition conditions must satisfy the following rule:

$$\text{partition_condition}^i \cap \text{partition_condition}^{i+1} = \emptyset \text{rule2}$$

The above rule means that there must not be any intersection between partition conditions for a partitioned table. If partition conditions intersect, it may not be clear in which partition a record is to be inserted. Therefore, if this rule is not satisfied when a partitioned object is created, the attempt to create the partitioned object will fail.

Additionally, partition conditions must output the same value at any point in time, regardless of the circumstances. If, for example, partitioning conditions determine at time t that a record is to be inserted into Partition A, if the same record were to be inserted at time $t+1$ instead, the partitioning conditions would have to make the same determination, namely that the record is to be inserted into Partition A. In order to satisfy this condition, the value output by a partition condition must be either a constant or a deterministic built-in function. A deterministic built-in function is an internal system function, rather than a user-defined function, that returns the same value at any given time.

7.3.2 The Default Partition

In ALTIBASE HDB, partition conditions must always satisfy the following rule:

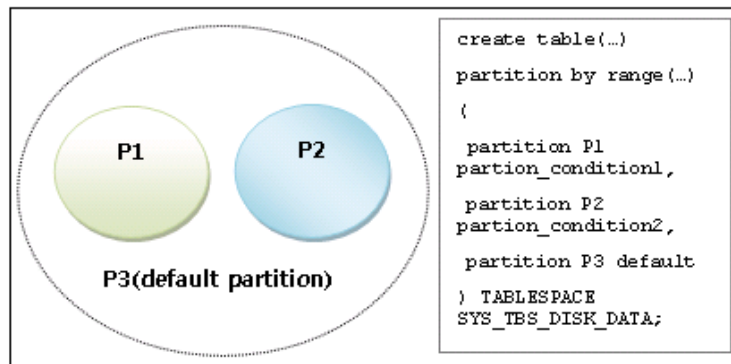
$$\text{column_domain} \equiv \bigcup \text{partition_condition rule3}$$

(The above rule states that the entire domain of the partition key columns must equal the union of domains that satisfy each partition condition. This means that all partition conditions must be specified when a partitioned table is created.

However, because in reality it is impossible for the author of query statements to specify conditions for all partitions, in ALTIBASE HDB the concept of a default partition has been introduced.

[Figure 6-9] illustrates the concept of a default partition with reference by way of example to a partitioned object having three partitions. In the statement below, the user specifies partition conditions (partition_condition1 and partition_condition2) for partitions P1 and P2, and declares P3 as a default partition. Therefore, if a record to be inserted into this object satisfies neither partition_condition1 nor partition_condition2, it will be inserted into partition P3. In other words, the default partition is equivalent to the portion of the domain of a partition key column that remains after the domains satisfying the user-specified partition conditions have been subtracted from the entire domain.

```
create table(...)
partition by range(...)
(
  partition P1 partition_condition1,
  partition P2 partition_condition2,
  partition P3 default) TABLESPACE SYS_TBS_DISK_DATA;
```

Figure 7-9 Default Partition Example

The default partition must be specified when a partitioned object is created. If no default partition is specified, the attempt to create the partitioned object will fail.

7.4 Partitioning Methods

Objects can be partitioned in three ways: range partitioning, list partitioning and hash partitioning.

In range partitioning, an object is partitioned based on ranges of partition key values. Range partitioning is suitable for data that are distributed across a linear range. In list partitioning, an object is partitioned based on sets of partition key values. List partitioning is useful with data that fall into discrete categories. In hash partitioning, an object is partitioned based on hash values that correspond to partition key values.

The following operations are supported on partitions created by each partitioning method:

Table 7-1 Operations Supported for Partitions

| Operation | Partitions created by Range Partitioning | Partitions created by List Partitioning | Partitions created by Hash Partitioning |
|-----------|--|---|---|
| Add | X | X | O |
| Coalesce | X | X | O |
| Drop | O | O | X |
| Merge | O | O | X |
| Rename | O | O | O |
| Split | O | O | X |
| Truncate | O | O | O |

7.4.1 Range Partitioning

Range partitioning is commonly used with date data types in situations where it is necessary to manipulate historical data.

The only partition condition that is supported when defining a partition is 'LESS THAN'. Default partitions are supported via the 'DEFAULT' clause.

The following is an example of range partitioning:

```
CREATE TABLE part_table
(
  sales_date DATE,
  sales_id NUMBER,
  sales_city VARCHAR(20),
  ....
)
PARTITION BY RANGE(sales_date)
(
  PARTITION part_1 VALUES LESS THAN ( TO_DATE('01-FEB-2006') ),
  PARTITION part_2 VALUES LESS THAN ( TO_DATE('01-MAR-2006') ),
  PARTITION part_3 VALUES LESS THAN ( TO_DATE('01-APR-2006') ),
  PARTITION part_def VALUES DEFAULT
) TABLESPACE SYS_TBS_DISK_DATA;
```

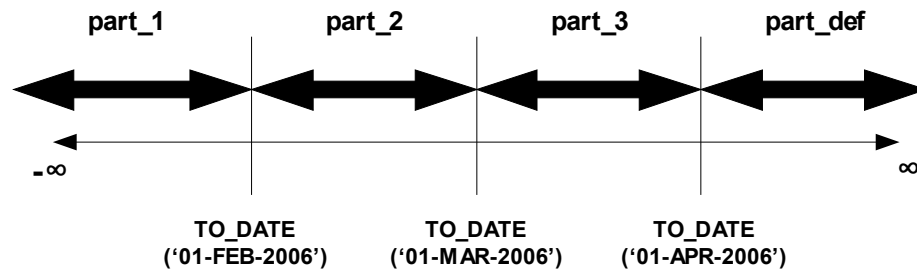
In the above example, the part_table table is created and range-partitioned into 4 partitions. The

7.4 Partitioning Methods

first three partitions handle data values prior to the beginning of February, March, and April, respectively. The default partition `part_def` handles data that do not satisfy any of the conditions for the other partitions.

[Figure 6-10] shows the above example in graphical form:

Figure 7-10 Partition Areas of a Range-Partitioned Table

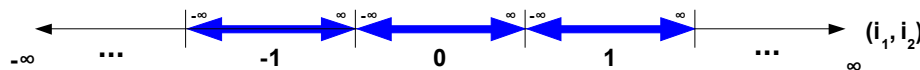


7.4.1.1 Multi-Column Partitioning

In multi-column partitioning, an object is partitioned using a partition key that is constructed on the basis of multiple columns. The concept of multi-column partitioning is similar to the concept of an index based on multiple keys.

The following figure depicts a partition key constructed on the basis of two columns (i_1, i_2) in one dimension.

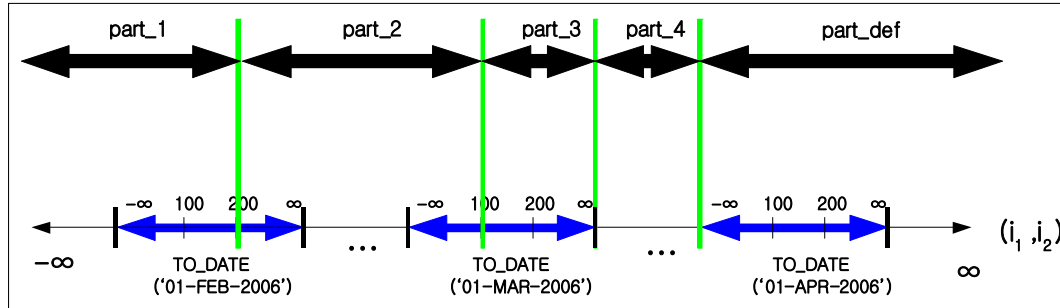
Figure 7-11 Partition Areas in Multi-Column Partitioning



The following describes multi-column partitioning with reference to an SQL statement as an example:

```
CREATE TABLE part_table
(
  sales_date DATE,
  sales_id NUMBER,
  sales_city VARCHAR(20),
  ....
)
PARTITION BY RANGE(sales_date, sales_id)
(
  PARTITION part_1 VALUES LESS THAN ( TO_DATE('01-FEB-2006'), 200),
  PARTITION part_2 VALUES LESS THAN ( TO_DATE('01-MAR-2006'), 100),
  PARTITION part_3 VALUES LESS THAN ( TO_DATE('02-MAR-2006') ),
  PARTITION part_4 VALUES LESS THAN ( TO_DATE('01-APR-2006') ),
  PARTITION part_def VALUES DEFAULT
) TABLESPACE SYS_TBS_DISK_DATA;
```

The above CREATE TABLE statement is illustrated below:

Figure 7-12 Partition Areas in Example SQL Statement

The following table shows the partition into which records will be inserted and the insertion conditions depending on the values of the records to be inserted.

| Value of the Record to be Inserted (sales_date, sales_id) | Partition into which Record Will be Inserted |
|---|--|
| TO_DATE('15-JAN-2006'), 100 | part_1 |
| TO_DATE('01-FEB-2006'), 100 | part_1 |
| TO_DATE('01-FEB-2006'), 200 | part_2 |
| TO_DATE('15-FEB-2006'), NULL | part_2 |
| TO_DATE('01-MAR-2006'), 50 | part_2 |
| TO_DATE('01-MAR-2006'), NULL | part_3 |
| TO_DATE('15-MAR-2006'), 200 | part_4 |
| NULL, 100 | part_def |
| NULL, NULL | part_def |

7.4.1.2 Operations on Range-Partitioned Objects

There are 5 operations that can be performed on range-partitioned objects. These are: SPLIT PARTITION, DROP PARTITION, MERGE PARTITION, RENAME PARTITION and TRUNCATE PARTITION. At present, no operation for changing partition conditions is supported.

The process of adding a partition to a range-partitioned object is the same as the process of splitting partition conditions. Therefore, to add a partition, use 'Split Partition'.

Similarly, deleting a partition is the same as deleting partition conditions. Therefore, to delete a partition, use DROP PARTITION. When a partition is deleted, the deleted partition conditions become included in the partition conditions of a neighboring partition. Additionally, DROP PARTITION is distinguished from MERGE PARTITION based on whether or not records are deleted.

A partition can be renamed using RENAME PARTITION. To delete records from a partition, use TRUNCATE PARTITION; all of the records stored in a partition can be deleted in this way.

7.4 Partitioning Methods

SPLIT PARTITION

Split Partition is an operation in which a partition of a partitioned object is divided into two partitions. SPLIT PARTITION can be performed in one of two ways:

- In-Place Split

With In-Place Split, some of the records are deleted from an existing partition and moved to a new partition; that is, the contents of the existing partition are changed.

In-Place Split is used when the name of one of the new partitions is the same as the name of the existing partition and the tablespace in which the new partitions are to be created is not specified. (Please refer to [Figure 6-13])

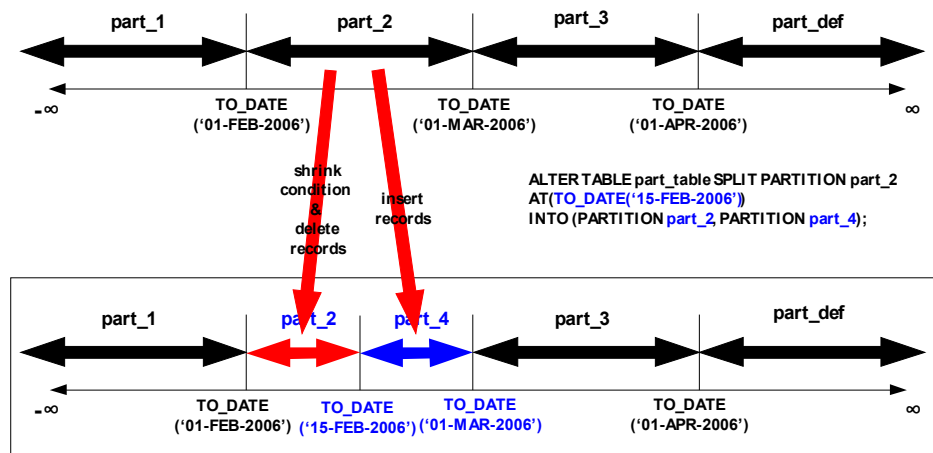
- Out-Place Split

With Out-Place Split, the contents of the existing partition are not changed. Instead, two new partitions are created, and then the records in the existing partition are divided and copied into them.

This method is used when the names of both of the new partitions are set differently from the name of the existing partition. Even if the name of one of the new partitions is the same as the name of the existing partition, this method is used when the tablespace in which the new partition having the same name of the existing partition is to be created is specified. (Refer to [Figure 6-14].)

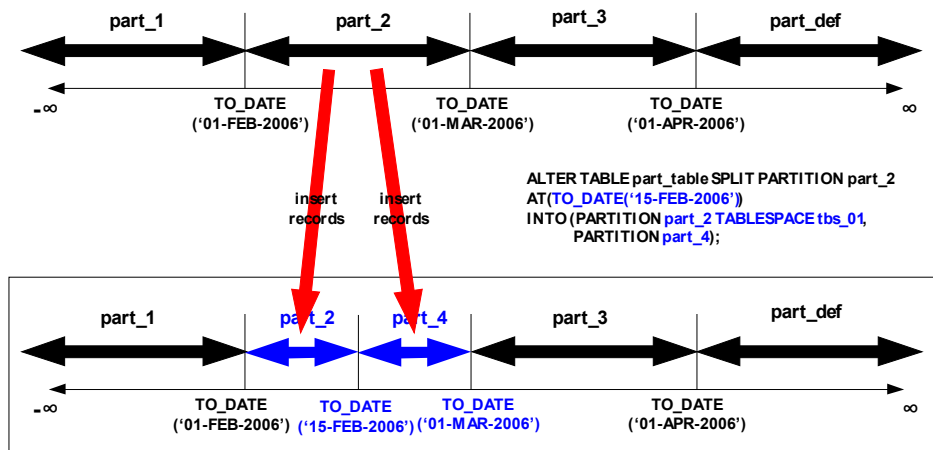
The Out-Place and In-Place partition splitting methods described above differ from each other in various aspects related to performance and efficiency. When using In-Place Split, because the existing partition is the same as one of the new partitions, only one new partition is created. Therefore, In-Place Split is advantageous from the aspect of efficiency of use of space.

Meanwhile, Out-Place Split comprises a process of creating two new partitions and inserting records into each of them. With In-Place Split, the operation that is performed on a record is a move operation, which comprises both an INSERT operation and a DELETE operation. In an MVCC environment, a DELETE operation has a more negative impact on system performance than an INSERT operation. Therefore, In-Place Split is more efficient when there is not enough storage space, whereas Out-Place Split realizes better performance in an MVCC environment when there is no shortage of storage space.

Figure 7-13 In-place Split of a Range-Partitioned Object

In the example shown in the above figure, part_2, which belongs to a partitioned object that originally comprises 4 partitions, is divided into part_2 and part_4.

① The new partition, part_4, is created, after which ② records are moved from part_2 to part_4 (move: INSERT & DELETE). Finally, ③ the conditions for part_2 are narrowed to the specified conditions.

Figure 7-14 Out-place Split of a Range-Partitioned Object

In the example shown in the above figure, part_2, which belongs to a partitioned object that originally comprises 4 partitions, is divided into part_2 and part_4. ① The new partitions part_2 and part_4 are created, and ② records are inserted from the old part_2 into the new part_2 and part_4. Finally, ③ the old part_2 is physically deleted.

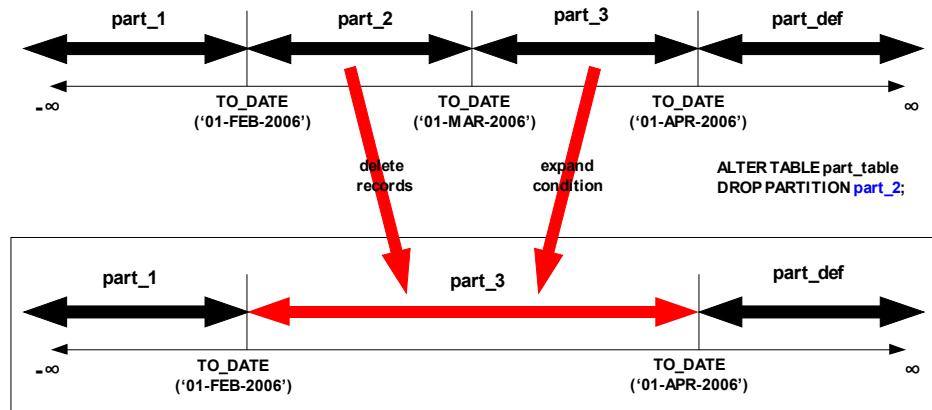
When the default partition is divided, the second partition in the INTO clause is automatically set as the default partition. This is because no command for specifying the default partition in the INTO subclause is supported.

7.4 Partitioning Methods

DROP PARTITION

Drop Partition is an operation in which a specific partition in a partitioned object is deleted. When a partition is deleted, all records and meta data in that partition are physically deleted. Furthermore, the conditions for that partition are incorporated in a neighboring partition.

Figure 7-15 DROPPing a Partition from a Range-Partitioned Object



The above figure illustrates an example in which a partition called `part_2` is dropped from a partitioned object that comprises 4 partitions.

① The physical space (records and meta data) of `part_2` is deleted, and ② its partition conditions are incorporated in the neighboring partition `part_3`, whose conditions are expanded to include the conditions of `part_2`.

MERGE PARTITION

MERGE PARTITION is an operation in which two specified partitions in a partitioned object are merged to form a single partition. The partitions to be merged must be neighboring partitions. The MERGE PARTITION operation can be conducted in one of two ways: In-Place Merge and Out-Place Merge.

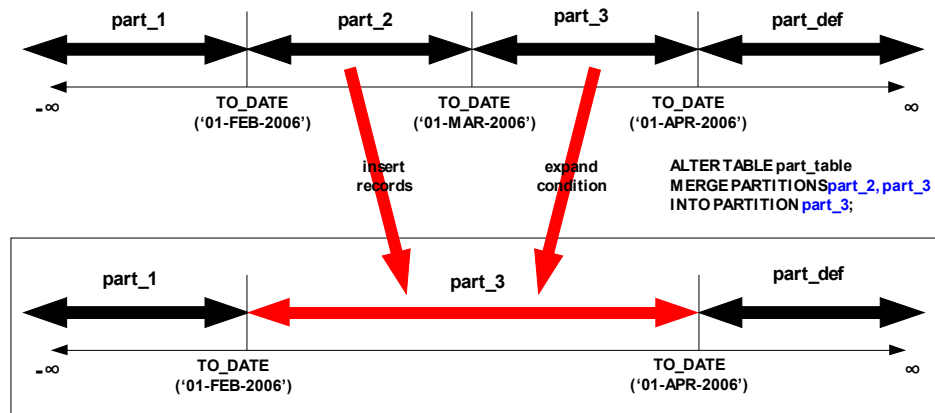
- In-Place Merge

Two existing partitions are merged into one of the partitions, and the records from the other partition are inserted into it. This method is used when the names of the new partition and one of the existing partitions are the same and the tablespace in which the new partition is to be created is not specified. Please refer to [Figure 6-16].

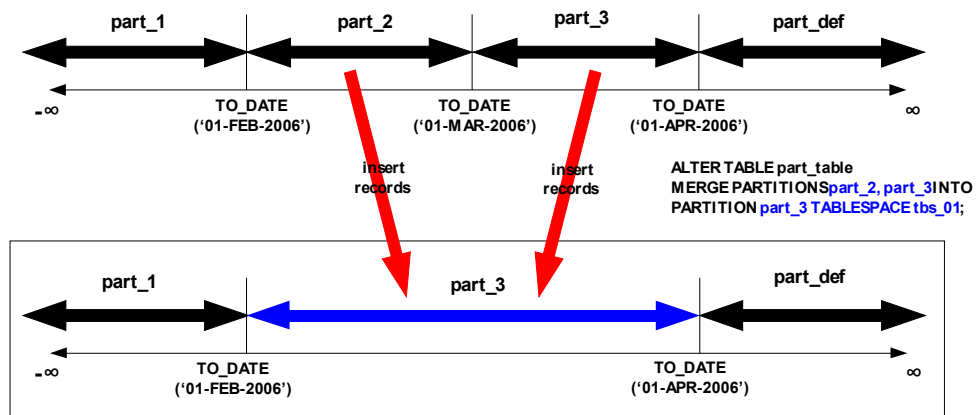
- Out-Place Merge

A new partition is created and the records in the existing partitions are copied into the new partition. This method is used when the name of the new partition is different from the names of the existing partitions. Additionally, even if the name of the new partition is the same as that of one of the existing partitions, this method is used if the tablespace in which the new partition is to be created is specified. Please refer to [Figure 6-17].

In-Place Merge and Out-Place Merge may differ from each other with respect to performance and efficiency. Because In-Place Merge does not create a new partition, but merely conducts an operation of INSERTing records, it is preferable to Out-place Merge from the aspect of performance.

Figure 7-16 In-Place Merge of a Range-Partitioned Object

In the example shown in the above figure, in a partitioned object comprising 4 partitions, the part_2 partition and the part_3 partition are merged into the original part_3 partition. ① The conditions for the existing part_3 are extended, and ② records from part_2 are inserted into part_3. Finally, ③ part_2 is physically deleted.

Figure 7-17 Out-Place Merge of a Range-Partitioned Object

In [Figure 6-17], in a partitioned object comprising 4 partitions, the part_2 partition and the part_3 partition are merged into a newly created part_3 partition.

① The new part_3 partition is created, and ② records from part_2 and the old part_3 are inserted into the new part_3. Finally, ③ part_2 and the old part_3 are physically deleted.

RENAME PARTITION

RENAME PARTITION changes the name of a partition without changing the partition conditions.

TRUNCATE PARTITION

TRUNCATE PARTITION does not change the partition conditions, but deletes all of the records stored in a partition.

7.4 Partitioning Methods

7.4.2 List Partitioning

In list partitioning, an object is partitioned based on a set of partition key values. List partitioning is commonly used when the range of partition key column values is not broad (e.g. January – December). List partitioning does not support multiple keys for partition key columns.

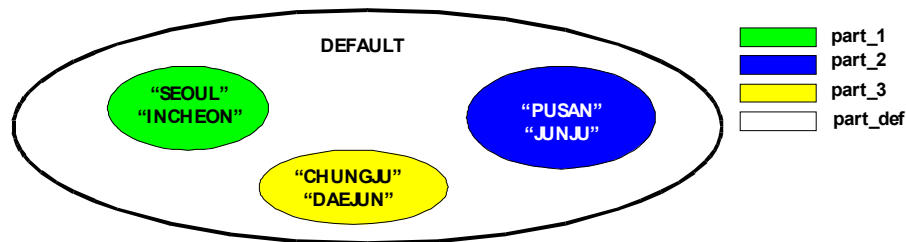
As in range partitioning, the use of the 'DEFAULT' statement is supported for the creation of a default partition.

The following is an example of list partitioning:

```
CREATE TABLE part_table(sales_date DATE,sales_id NUMBER,sales_city VARCHAR(20),...)
PARTITION BY LIST(sales_city)
(PARTITION part_1 VALUES ( 'SEOUL' , 'INCHEON' ),
 PARTITION part_2 VALUES ( 'PUSAN' , 'JUNJU' ),
 PARTITION part_3 VALUES ( 'CHUNGJU' , 'DAEJUN' ),
 PARTITION part_def VALUES DEFAULT)
TABLESPACE SYS_TBS_DISK_DATA;
```

In the above example, the part_table table is created and list-partitioned so that it has 4 partitions. The first three partitions manage data according to particular cities, while the default partition, part_def, handles data that do not meet any of the conditions specified for the other partitions. The example is illustrated below:

Figure 7-18 Partition Area of List-Partitioned Table



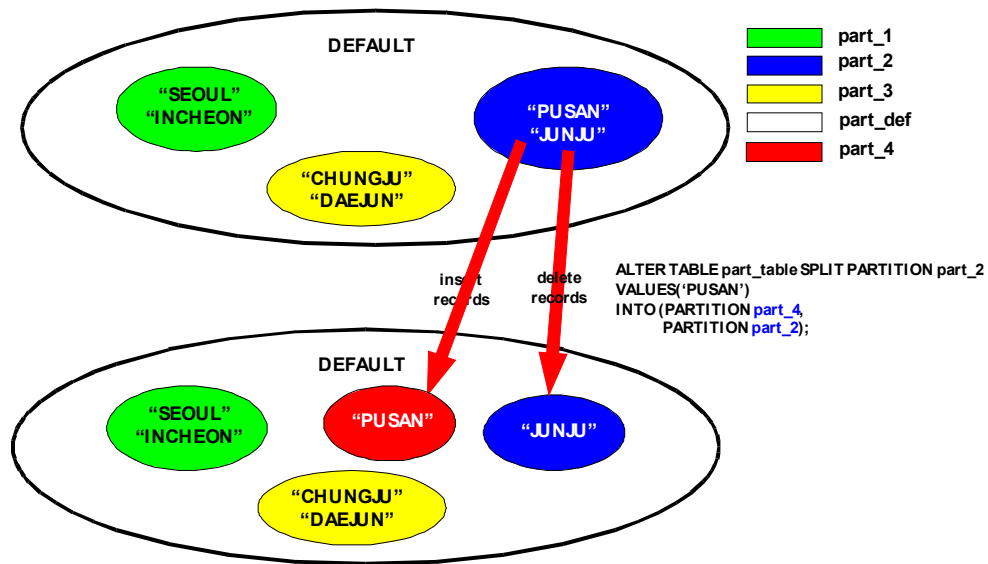
7.4.2.1 Operations on List-Partitioned Objects

5 types of operations can be performed on list-partitioned objects. These are: SPLIT PARTITION, DROP PARTITION, MERGE PARTITION, RENAME PARTITION and TRUNCATE PARTITION. The SQL statements that are used are the same as for range-partitioned objects. At present, no operation for changing partition conditions is supported.

SPLIT PARTITION

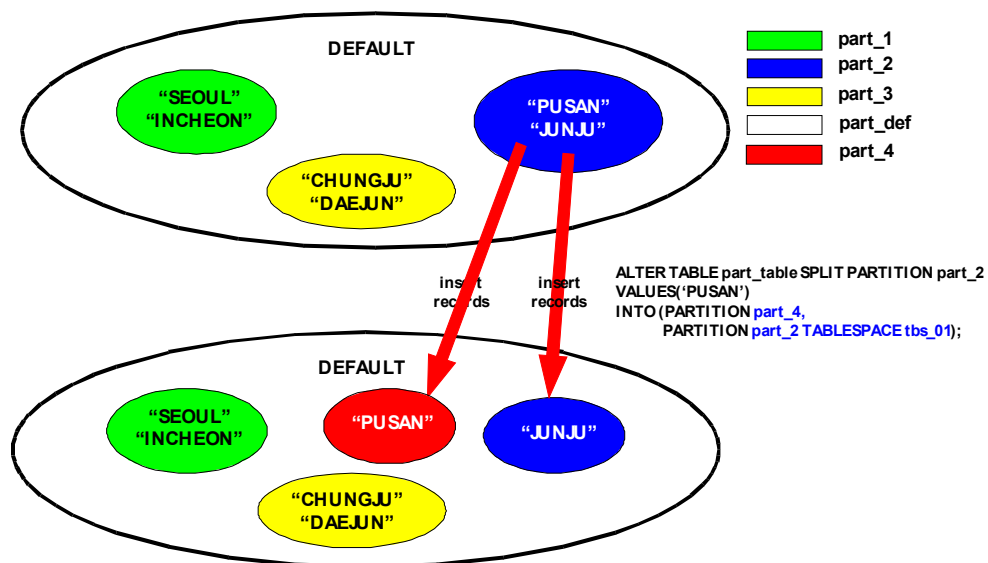
As in range partitioning, in list partitioning, the SPLIT PARTITION operation can be conducted as either In-Place Split or Out-Place Split. When splitting a partition, if the name of one of the new partitions is the same as the name of the old partition, whether In-Place Split or Out-Place Split is used depends on whether a tablespace is specified.

Figure 7-19 In-Place Split of a List-Partitioned Object



In the example shown in the above figure, in a partitioned object with 4 partitions, the part_2 partition is divided into the part_2 partition and the part_4 partition. ① A new partition, part_4, is created, and ② records are moved from part_2 to part_4 (move: INSERT & DELETE). Finally, ③ the conditions for the part_2 partition are narrowed to the newly specified conditions ({'PUSAN', 'JUNJU'} -> {'JUNJU'}).

Figure 7-20 Out-Place Split of a List-Partitioned Object

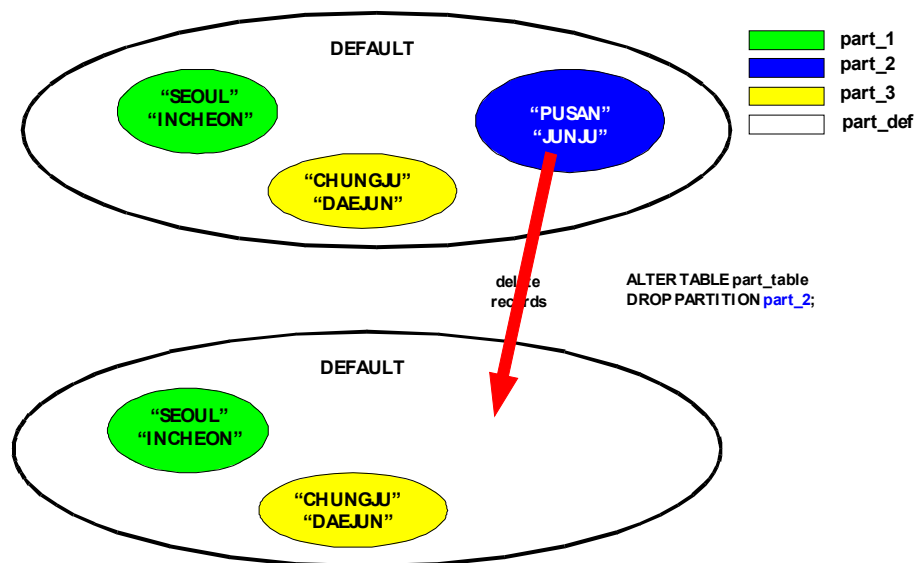


In the example shown in the above figure, the part_2 partition, which belongs to a partitioned object with 4 partitions, is divided into the part_2 partition and the part_4 partition. ① The new part_2 and part_4 partitions are created, and ② records from the old part_2 partition are inserted into the new part_2 partition and the part_4 partition. Finally, ③ the old part_2 partition is physically deleted.

DROP PARTITION

Dropping a partition from a list-partitioned object is similar to dropping one from a range-partitioned object, with the exception that the partition conditions of the partition to be deleted are incorporated in the conditions of the default partition rather than a neighboring partition.

Figure 7-21 DROPPing a Partition from a List-Partitioned Object

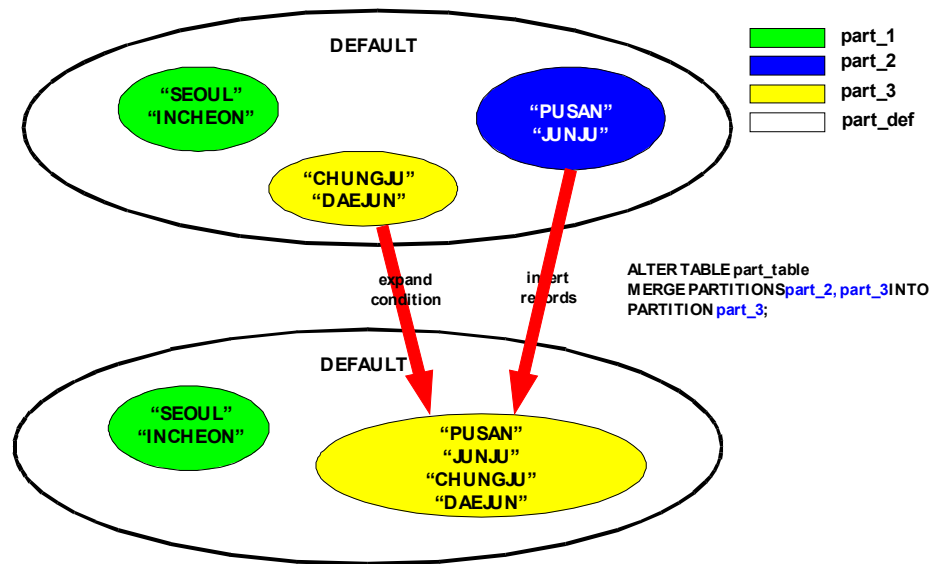


In the example shown in the above figure, the `part_2` partition is deleted from a partitioned object that has 4 partitions. ① The physical space (records and meta data) of the `part_2` partition is deleted, and ② the conditions for `part_2` are incorporated into those for the default partition `part_def`.

MERGE PARTITION

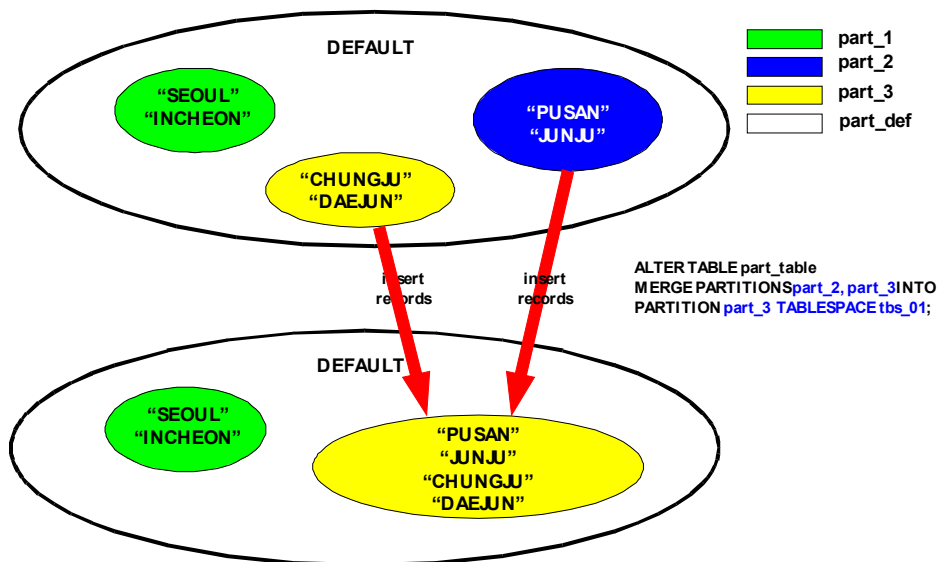
As with range-partitioned objects, there are two ways to merge the partitions of a list-partitioned object: In-Place Merge and Out-Place Merge. If the name of the new partition is the same as that of one of the partitions being merged, whether In-Place Merge or Out-Place Merge is used depends on whether a tablespace is specified.

Figure 7-22 In-Place Merge in a List-Partitioned Object



In the example shown in the above figure, the part_2 partition and the part_3 partition, which belong to a partitioned object with 4 partitions, are merged into the original part_3 partition. ① The conditions for the existing part_3 are extended, and ② records from the part_2 partition are inserted into the part_3 partition. Finally, ③ the part_2 partition is physically deleted.

Figure 7-23 Out-Place Merge in a List-Partitioned Object



In the example shown in the above figure, the part_2 and part_3 partitions, which belong to a partitioned object that has 4 partitions, are merged into a new part_3 partition. ① The new partition part_3 is created, and ② records from the part_2 partition and the original part_3 partition are inserted into the new part_3 partition. Finally, ③ the part_2 partition and the original part_3 partition are physically deleted.

7.4 Partitioning Methods

RENAME PARTITION

RENAME PARTITION changes the name of a partition without changing the partition conditions.

TRUNCATE PARTITION

This deletes all of the records from a partition without changing the partition conditions.

7.4.3 Hash Partitioning

In hash partitioning, an object is partitioned based on a hash value corresponding to a partition key. The partition key can consist of multiple columns. Hash partitioning is typically used for uniform load distribution rather than for manageability.

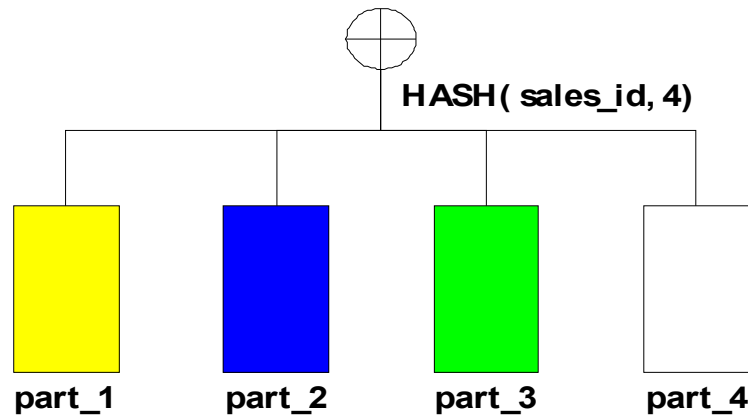
Due to the characteristics of hash functions, certain limitations are imposed on operations conducted on hash partitions. Unlike range partitions and list partitions, SPLIT PARTITION, DROP PARTITION, and MERGE PARTITION cannot be performed on hash partitions; however, operations such as ADD PARTITION and COALESCE PARTITION are supported.

Unlike range partitioning and list partitioning, in hash partitioning there is no default partition. This is because the hash function can accept all possible partition key values. The location where records having NULL partition key values are inserted depends on the hash value for the NULL data. Although the hash value corresponding to NULL data is constant, different values can be output depending on the data type. The location in which records having NULL partition key values are stored can be different depending on the column type.

The following is an example of hash partitioning:

```
CREATE TABLE part_table
(
  sales_date DATE,
  sales_id NUMBER,
  sales_city VARCHAR(20),
  ....
)
PARTITION BY HASH(sales_id)
(
  PARTITION part_1,
  PARTITION part_2,
  PARTITION part_3,
  PARTITION part_4
) TABLESPACE SYS_TBS_DISK_DATA;
```

In the above example, the hash-partitioned part_table is created with 4 partitions. Each partition manages data that are divided according to a hash function: HASH(sales_id, 4). An example is illustrated in [Figure 6-24]:

Figure 7-24 Partition Areas of a Hash-Partitioned Table

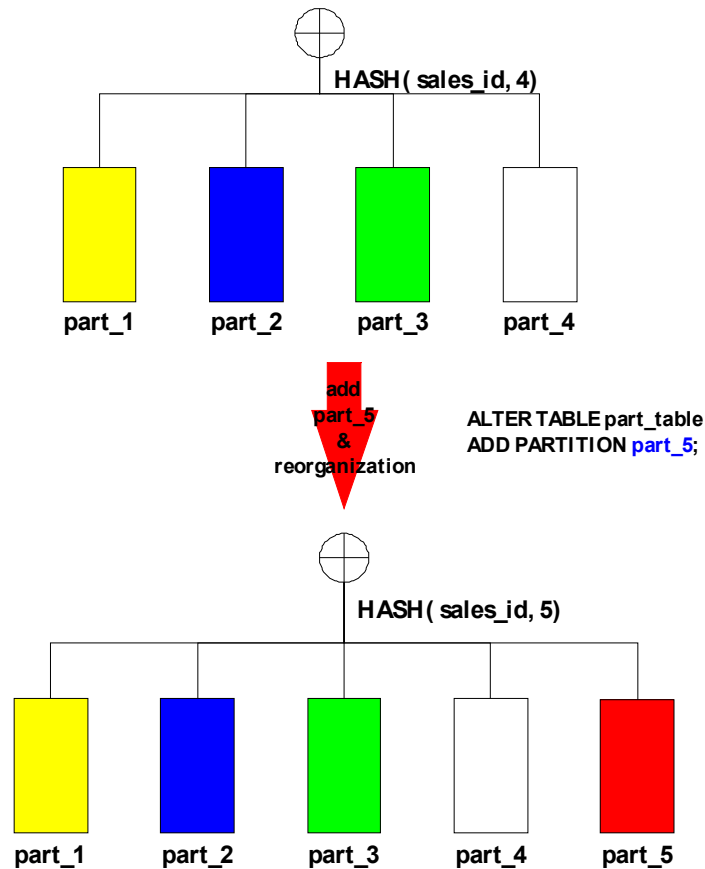
7.4.3.1 Operations on Hash-Partitioned Objects

4 types of operations can be performed on hash-partitioned objects. These are ADD PARTITION, COALESCE PARTITION, RENAME PARTITION and TRUNCATE PARTITION.

To add a partition to a hash-partitioned object, use ADD PARTITION. To get rid of a partition but keep its data, use COALESCE PARTITION. This operation deletes the last partition and reorganizes the entire partitioned object, combining the records from the deleted partition with the existing records in all other partitions. To rename a partition, use RENAME PARTITION. To truncate a partition, that is, delete all of the records in the partition, use TRUNCATE PARTITION.

ADD PARTITION

Adding a partition to a hash-partitioned object means increasing the number of hash keys. Adding a partition affects all existing partitions. If a hash key is changed, all of the records in a table are reorganized into the new partitions. The following figure illustrates the process of adding a partition.

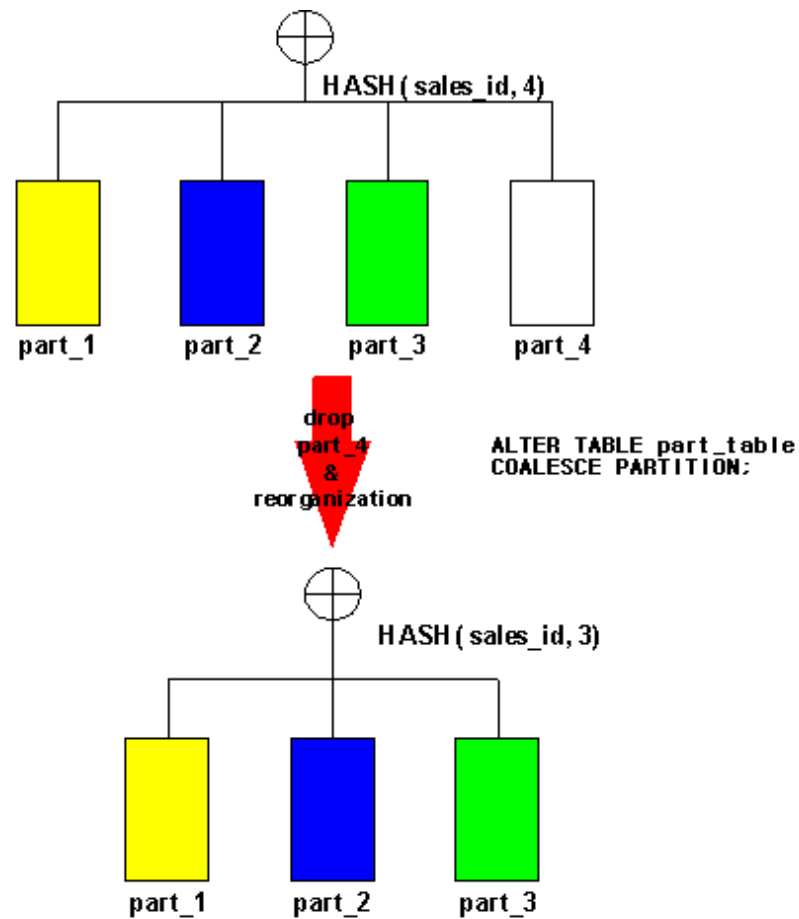
Figure 7-25 Adding a Partition to a Hash-Partitioned Object

In the example shown in the above figure, the `part_5` partition is added to a partitioned object that originally consists of 4 partitions. The new `part_5` partition is created, and ② the records in the four existing partitions are redistributed among the four existing partitions and the newly created partition.

COALESCE PARTITION

'Coalescing' the partitions of a hash-partitioned object means decreasing the number of hash keys. Coalescing a partition affects all existing partitions. To coalesce the partitions of a hash-partitioned object, use `COALESCE PARTITION`. The last partition is deleted, and the records from the deleted partition are redistributed throughout the partitioned object together with the existing records. When coalescing partitions, the name of the partition to delete cannot be specified. The partitions are deleted one by one, starting with the last one and ending with the first one.

For example, if a hash-partitioned object comprising 4 partitions (`part_1`, `part_2`, `part_3` and `part_4`) is coalesced, it is reduced to a partitioned object comprising 3 partitions (`part_1`, `part_2` and `part_3`). The following figure illustrates the process of coalescing partitions.

Figure 7-26 Coalescing the Partitions of a Hash-Partitioned Object

In the example shown in the above figure, the partitions of a hash-partitioned object consisting of 4 partitions are coalesced. ① The records in the four existing partitions are redistributed among the `part_1`, `part_2` and `part_3` partitions, and ② the last partition, `part_4`, is deleted.

RENAME PARTITION

`RENAME PARTITION` changes the name of a partition without changing the partition conditions.

TRUNCATE PARTITION

`TRUNCATE PARTITION` deletes all of the records from a partition without changing the partition conditions.

8 Managing Transactions

Concurrency control and data consistency are two of the most fundamental concepts in database management. This chapter explains how to manage transactions in an Altibase database.

This chapter contains the following sections:

- [Transactions](#)
- [Locking](#)
- [Multi-Version Concurrency Control \(MVCC\)](#)
- [Transaction Durability](#)

8.1 Transactions

A transaction is a logical unit of work that comprises one or more SQL statements. A transaction begins with the first execution of an SQL statement by a user, and ends when it is committed or rolled back, either explicitly with a COMMIT or ROLLBACK statement or implicitly when a DDL statement is issued.

8.1.1 Definition of Transaction

Transactions let users guarantee the consistency of changes to data, as long as the SQL statements within a transaction are grouped logically. A transaction should consist of all of the necessary parts for one logical unit of work—no more and no less. Data in all referenced tables that are in a consistent state before the transaction begins should also be in a consistent state after it ends. Transactions should consist of only the SQL statements that make one consistent change to the data.

Transferring money from one bank account to another is a representative example of a transaction. In order to transfer \$100 from account A to account B, the following tasks must be completed:

- Decrease the balance of account A by \$100.
- Increase the balance of account B by \$100.
- Make a record of the fact that money was transferred from account A to account B.

When a transaction is normally executed on a database that was in a consistent state before the transaction, the database will still be consistent after the transaction. If even one of the three tasks constituting the above transaction is not performed correctly, the integrity of the database will be compromised, and either the holder of account A, the holder of account B or the bank will suffer damages.

To maintain database integrity, a properly executed transaction must exhibit the four ACID properties: Atomicity, Consistency, Isolation, and Durability.

- Atomicity - Either all of the statements that constitute a transaction are completely executed, or none of them are. That is, the transaction cannot be partially successful.
- Consistency - A properly executed transaction does not break the consistency of the database.
- Isolation - When multiple transactions are underway at the same time, none of the transactions have access to the results of the other transactions.
- Durability - Once a transaction has been committed, the resultant changes are not lost regardless of the circumstances, such as system failure.

8.1.2 Transaction Termination

A transaction will be terminated if any of the following occurs.

- The transaction is terminated when the user executes the ROLLBACK statement without a SAVEPOINT clause, or executes the COMMIT statement.
- When the user executes a DDL statement, the transaction is committed.

- When the user disconnects from ALTIBASE HDB, the transaction is committed.
- If the user session terminates abnormally, the current transaction is rolled back.

8.1.3 Statements

The term “statement” refers to an SQL statement within a transaction. SQL statements fall into the three following categories:

- DCL (Data Control Language): This type of statement is used to change the status of the database, its properties, or its physical configuration.
- DDL (Data Definition Language): This type of statement is used to create, change, or delete database logical structural elements, such as tables, sequences, indexes, and the like. Examples are the CREATE TABLE, CREATE INDEX, ALTER, and DROP statements.
- DML (Data Manipulation Language): Data Manipulation Language commands are used to insert, delete, modify, or view the actual data saved in a database. Example DML statements are the UPDATE, INSERT, and DELETE statements.

A statement is usually just a single SQL statement, but one or more underlying statements are executed when a stored procedure or function is executed.

If an error occurs while a statement is being executed, all of the data affected by the statement are restored to their original state. To make this possible, a so-called “Implicit Save Point” is set before each statement is executed, and the database is restored to this point if an error occurs.

8.1.4 Commit

Committing a transaction means permanently saving the results of all SQL statements executed within the transaction up to that point in time and ending the transaction. When a transaction is committed, the database is moved from a previous state, in which it had integrity, to a new state that also has integrity.

When a transaction is committed in ALTIBASE HDB, the following tasks are performed:

- A transaction commit log is written to a log file.
- Resources that are no longer needed by the transaction and thus can be released are handed over to the Garbage Collector.
- The status of the transaction is changed to “committed”.
- Resources allocated during execution of the transaction, such as locks and temporary memory, are released.

8.1.5 Rollback

When a fatal error occurs while a transaction is underway such that execution of the transaction cannot continue, all SQL statements executed by the transaction must be undone, and the database must be returned to the state that existed before execution of the transaction. This is referred to as “rolling back” a transaction.

8.1 Transactions

A transaction is rolled back by executing operations that effectively undo, or compensate for, each of the operations that was logged during the transaction.

When a transaction is rolled back in ALTIBASE HDB, the following tasks are performed:

- The log records are read in the opposite order in which they were written, and compensatory operations are executed.
- A transaction rollback log is recorded.
- Resources that were allocated for insert operations, etc. are returned to the garbage collector.
- The status of the transaction is changed to “rolled back.”
- Resources allocated during execution of the transaction, such as locks and temporary memory, are released.

8.1.6 Explicit Savepoint

To manage a long transaction by dividing it into several portions, explicit save points can be declared at the start point of each portion.

Because each explicit save point can have a name, multiple save points can be declared within a single transaction. If an error occurs after an explicit save point is declared, the transaction can be rolled back to the save point to restore the database.

When a transaction is rolled back to an explicit save point, all of the resources such as table and row locks acquired since that savepoint are released, and all savepoints declared since that point are cancelled.

8.2 Locking

The purpose of a “lock” is to set access rights to a particular object in a database.

ALTIBASE HDB uses locks to control concurrent access to data. When data are updated, those data are locked until the update is committed. Until that happens, no one else can access the locked data. This helps ensure the integrity of the data in the system.

8.2.1 Locking Modes

Locks are acquired at row level or table level depending on their purpose.

Common uses for locks include the following:

- to ensure that only one user can modify a record at a time
- to ensure that a table cannot be dropped if it is being queried
- to ensure that one user cannot delete a record while another is updating it

8.2.1.1 Table Level Lock Modes

Table 8-1 Lock Modes

| Lock Mode | Description | Property |
|-----------|-----------------------------------|--|
| S | Shared Lock | The holder of the lock can read all of the records in a table, and does not lock individual records. Other transactions that only read the table can be executed at the same time. |
| X | Exclusive Lock | The holder of the lock can read and modify all of the records in a table, and does not lock individual records. No other transactions can read or modify the table. |
| IS | Intent Shared Lock | This mode is the same as S mode except that the lock holder locks individual records before reading them. |
| IX | Intent Exclusive Lock | The lock holder can read and modify records after obtaining a lock on the records. Multiple transactions that write to different records in the same table can exist at the same time. |
| SIX | Shared with Intent Exclusive Lock | The lock holder can read and modify records after obtaining a lock on the records. Only one transaction can update records in the table. |

8.2.1.2 Intention Mode Lock - IS, IX, SIX

There are many kinds of objects that can be locked. These objects can have various sizes within the

8.2 Locking

database. Examples of objects that can be locked include the entire database, schemas, tables, records and columns. Arranged in descending order of size, these are:

database > schema > table > record > column

Lock granularity refers to the size of the object to be locked. If locking were supported only for large objects, concurrency control would suffer. Described in greater detail, suppose that multiple transactions attempt to perform operations on individual records in a table. If locking were supported only for objects larger than records, (e.g. for the entire table,) when a transaction is executed, even if it performed an operation on only a single record, all other transactions wishing to perform operations on other records in the table would have to wait until the operation that started first was successfully completed.

Therefore, it is most efficient to make the record the smallest lockable unit. To acquire a lock on the smallest unit, a lock on larger objects must also be acquired. This is referred to as the “lock granularity protocol”.

When acquiring a lock on larger objects, it is advisable to choose a suitable lock mode from among the various lock modes that are provided, so that multiple transactions can perform operations on the same table as long as they are not accessing the same records. So-called “intention mode locking” is used for this purpose.

8.2.1.3 Lock Compatibility

The term “lock compatibility” refers to compatibility between lock modes, that is, to whether a request to place a particular kind of lock on an object will be accepted when the object in question has already been locked by another transaction.

The compatibility between the various lock modes is set forth in the following table:

Table 8-2 Lock Mode Compatibility

| Requested Lock Mode | Mode of Lock Already Granted to Other Transaction | | | | | |
|---------------------|---|----|----|-----|---|---|
| | NONE | IS | IX | SIX | S | X |
| IS | O | O | O | O | O | - |
| IX | O | O | O | - | - | - |
| SIX | O | O | - | - | - | - |
| S | O | O | - | - | O | - |
| X | - | - | - | - | - | - |

8.2.1.4 Record-Level Lock Modes

Table 8-3 Record-Level Lock Modes

| Lock Mode | Description | Property |
|-----------|-------------|---------------------------|
| S | Shared Lock | Records can only be read. |

| Lock Mode | Description | Property |
|-----------|----------------|--------------------------|
| X | Exclusive Lock | Records can be modified. |

The INSERT, UPDATE, and DELETE DML statements obtain X locks on individual records, whereas read operations obtain S locks on individual records.

Normally, S locks conflict with X locks, and the two types of locks are not considered intercompatible. However, thanks to the ALTIBASE HDB MVCC (Multi-Version Concurrency Control) implementation, these kinds of locks do not conflict with each other. Thus, read operations can be performed on records that are being updated, and update operations can be performed on records that are being read.

8.3 Multi-Version Concurrency Control (MVCC)

In ALTIBASE HDB, MVCC (Multi-Version Concurrency Control) is used to ensure the consistency of records. MVCC is a concurrency control method in which, when a DML statement is executed on a record, the record is maintained in its original state and the DML statement is executed on a copy of the record to create a new version of the record. In this way, a transaction that is performing an operation on a record will not affect another transaction that is reading the same record.

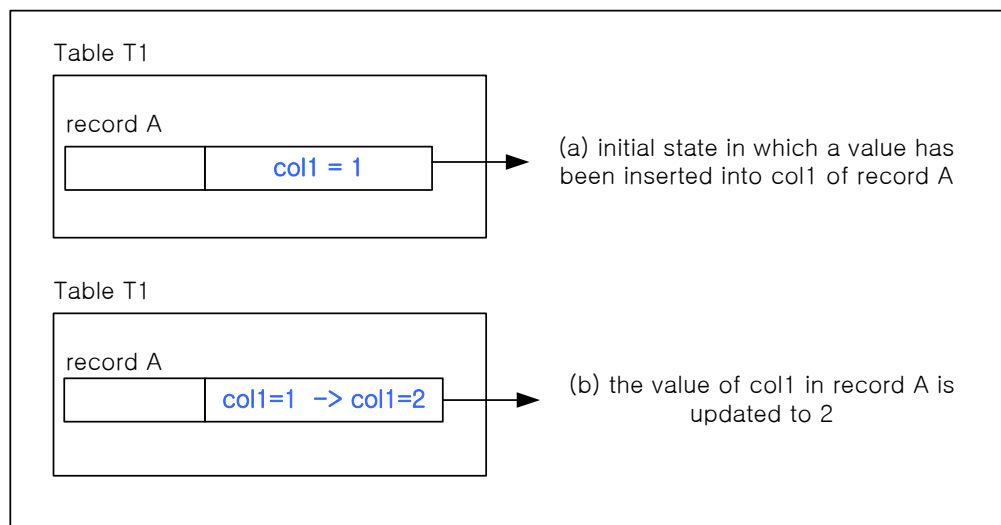
The MVCC concurrency control method cannot be implemented in the same way for memory tablespaces and disk tablespaces due to the differences in their characteristics. ALTIBASE HDB uses so-called “Out-Place MVCC” for memory tablespaces and “In-Place MVCC” for disk tablespaces. Because these two techniques superficially appear to work in the same manner, there is no need for users to distinguish between the two.

This section briefly describes the internal processes that are conducted to support MVCC when each kind of DML statement is executed. First, the cases in which MVCC is not used are described first. Out-place MVCC for memory tablespaces is then described, followed by in-place MVCC for disk tablespaces. Finally, some cautionary notes to keep in mind when using MVCC are described.

8.3.1 Updating without Using MVCC

For the sake of comparison with MVCC, this section describes how an update statement is internally handled in a non-MVCC environment. The following figure shows how the records in a table are changed in response to an update operation when MVCC is not being used.

Figure 8-1 Non-MVCC Transaction



In the above [Figure 7-1], (a) illustrates the state in which record A has been initially inserted into table T1. If col1 of record A is updated to the value of 2, as shown in (b) above, record A is modified in its original location without changing the amount of space allocated to T1. Similarly, DELETE operations are also performed in the original location of the record.

When MVCC is not being used, an UPDATE or DELETE operation does not change the amount of space allocated to a table. The space allocated to a table can be increased only by the execution of an INSERT statement.

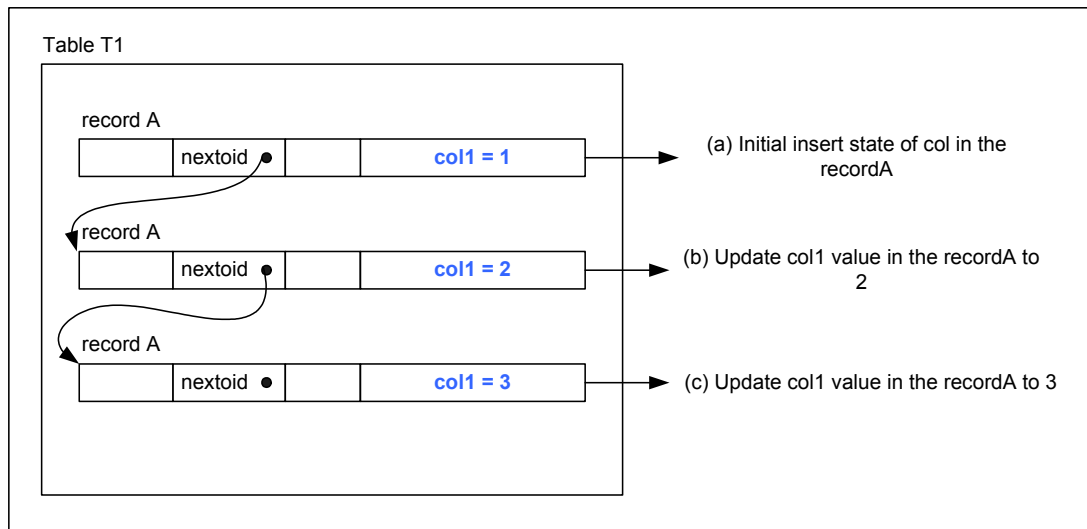
8.3.2 Out-Place MVCC and Memory Tablespaces

In out-place MVCC, which is used with memory tablespaces in ALTIBASE HDB, a new version of a record is created and associated with previous versions of the record every time an UPDATE operation occurs.

8.3.2.1 UPDATE Operation

The following [Figure 7-2] shows the effect of executing an UPDATE statement when using out-place MVCC.

Figure 8-2 Transaction using Out-Place MVCC



In the initial state, in which record A has been inserted into table T1, as shown at (a), if the value in col1 of record A in table T1 is updated to 2, an identical record is created and the value in this record is changed to 2, as shown at (b). Therefore, table T1 occupies one more slot than it did before the transaction took place.

When the new version of record A is created, a pointer in the header of the original version of record A is used to indicate the newly added record. In this way, different versions of the same record can therefore be managed simultaneously.

If another UPDATE operation is performed on record A, which is still in the state indicated by (b) in the above figure, yet another record will be created, and the UPDATE operation will be executed on the new record. Ultimately, the number of versions of the same record will equal the number of UPDATE operations performed on the record.

So, does the tablespace increase in size without limit as more UPDATE operations are performed?

When each transaction that performs an UPDATE operation on a particular record is committed, only the most recent of the multiple versions of the record is kept. Preceding versions do not need to be saved in the database. Unnecessary versions are deleted by the garbage collector, and the emptied spaces previously occupied by these versions are reused by subsequent INSERT/UPDATE statements. Therefore, even though a new version of a record is created every time an UPDATE operation occurs, the database does not occupy infinitely increasing amounts of space.

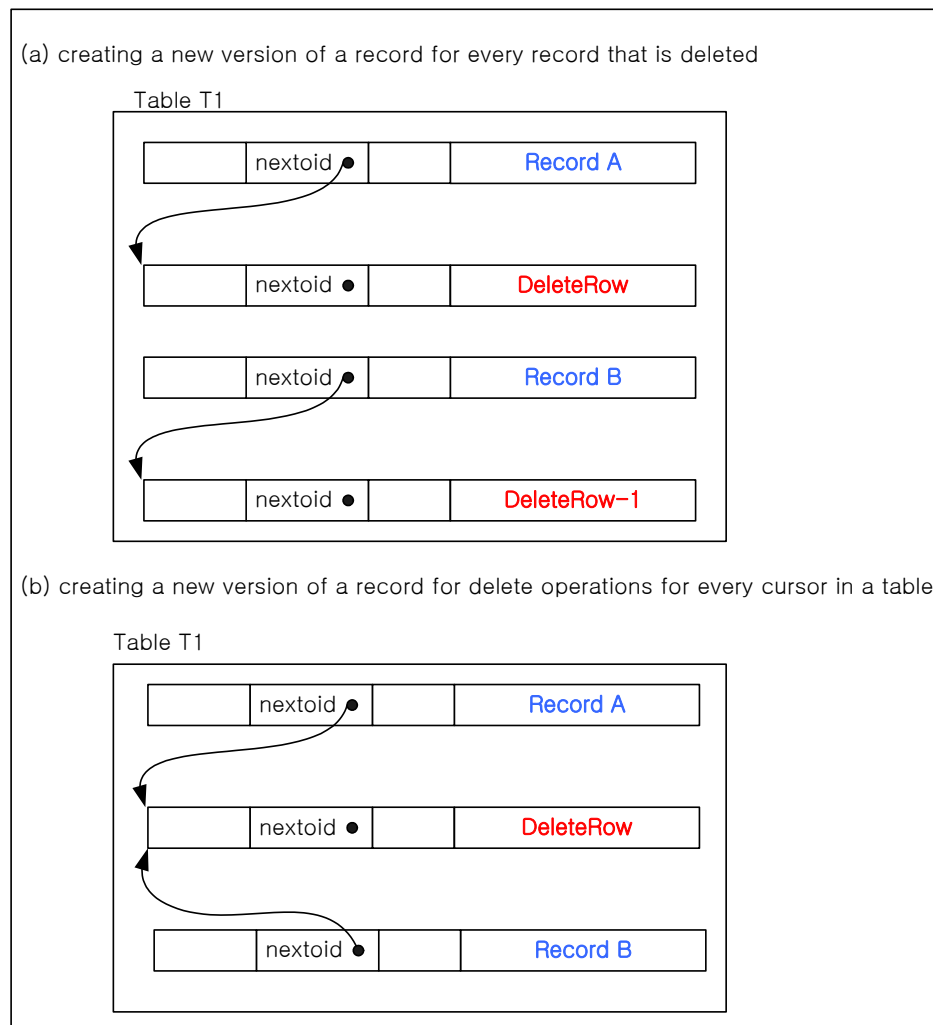
8.3 Multi-Version Concurrency Control (MVCC)

8.3.2.2 DELETE Operation

Just like an UPDATE operation, whenever a DELETE operation is executed on a record, a new version of the record is created. Unlike an UPDATE operation, however, the new version of the record to be deleted does not actually contain any data. Therefore, it is not necessary to create a new version of every record when a DELETE operation is performed. It is sufficient to create a single version representing all deleted records.

The following figure shows how much space is used depending on whether or not new versions of each record are created when a DELETE operation is performed:

Figure 8-3 DELETE Transaction using MVCC



The case indicated by (a) in the above figure represents the case in which a new version is created for every record that is deleted. If a transaction deletes records A and B using a single DELETE statement, new versions will be created for each record, and thus table T1 will have two additional records.

In the case indicated by (b), only one additional record is created, even though the DELETE statement deletes multiple records. As shown in the above figure, the case indicated by (b) generates fewer unnecessary versions of records, thereby increasing the efficiency of space utilization. ALTI-BASE HDB uses the case indicated by (b) when DELETE statements are executed on tables in mem-

ory tablespace.

8.3.3 In-Place MVCC and Disk Tablespaces

According to in-place MVCC, which ALTIBASE HDB uses for disk tablespaces, when an UPDATE operation occurs, the contents of columns that belong to the original records and are being changed are written as so-called “undo log records” to an undo page, which exists in undo tablespace, and the new data are written to the location of the original record.

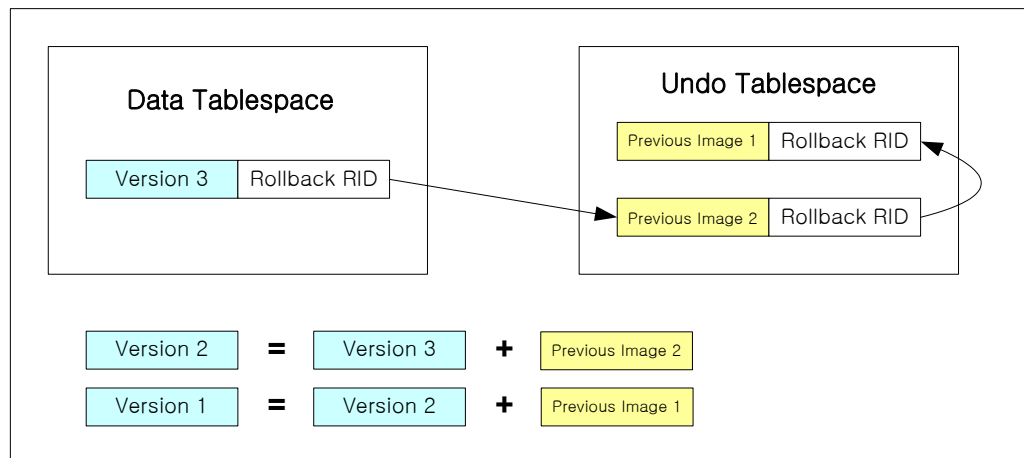
8.3.3.1 Insert Operation

When a record is first inserted, the system allocates space for the record in a data tablespace and creates the record. The system also allocates an area for an undo log record in an undo tablespace and creates the undo log record. Finally, the system links the location of the undo log record with the rollback RID of the actual record in the data tablespace.

8.3.3.2 Update Operation

Assuming that version 1 is the record that was originally inserted, the following figure shows how version 1 is updated to version 2 and then to version 3.

Figure 8-4 Using MVCC with Disk Tablespaces



As shown in the above figure, the most recent image of a record always exists in the data tablespace. If the execution of some statement starts before version 3 is committed, it executes on the basis of the previous version, which is version 2, because it cannot read version 3. In such cases, the statement copies the image of version 3 to a buffer that it manages privately. It then reads the previous version 2 from the location indicated by the rollback RID of the record and stores this in its private buffer, where it copied version 3. If the statement cannot read version 2 either, it repeats the process and creates its own copy of version 1.

If the statement cannot read version 1, this means that execution of the statement started before the record was initially inserted, so the statement will ignore the record.

8.3 Multi-Version Concurrency Control (MVCC)

8.3.3.3 Clearing the Undo Log Record Area

In disk tablespace, the size of the data tablespace does not increase much if a great number of update operations occur in a short period of time. However, because the effect of in-place MVCC is to increase the number of undo log records, the amounts of space being used in the undo tablespace would increase in this situation. Because the size of the undo tablespace is set when the CREATE DATABASE statement is executed, and cannot be changed, it is necessary to reuse undo log records. Undo log records are registered in the undo tablespace header when transactions are committed, and are managed in a linked list. When it is no longer necessary to refer to undo log records pertaining to particular transactions, the records are cleared from the system. In contrast, when a transaction is rolled back, the undo log records are cleared immediately instead of being registered in the undo tablespace.

The undo log records that are created by insert operations are managed separately from those created by update and delete operations. This is so that the undo log records created by insert operations can be cleared immediately when transactions are committed.

8.3.3.4 Delete Operation

Delete operations are executed in the same way as update operations. Because the information that is altered by a delete operation is a delete flag that is set in the header of the record, only information about the record header is recorded in undo log record images.

The space occupied by a deleted record is not reused immediately. First, all index keys pertaining to the record are deleted, then the actual record is deleted, and then the garbage collector removes the delete undo log record pertaining to the deleted record. Then, the space occupied by the record can be reused.

8.3.4 In-Place MVCC vs. Out-Place MVCC

The in-place method, which is used with disk tablespaces, checks record versions differently than the out-place method, which is used with memory tablespaces. In the out-place method, a Commit SCN is saved for every transaction that creates a new version of a record, and this Commit SCN is used to check record versions. That is, a read transaction reads versions that have a Commit SCN lower than the SCN of the read transaction. The commit SCN of a transaction is set when the transaction is committed, and is written to all versions of records created by that transaction.

In contrast, setting the Commit SCN for a transaction in a disk tablespace requires all record versions created by the transaction to be accessed, which is unfeasible in practice. This is because transaction performance is greatly reduced, attributable to disk I/O expense.

A TSS (Transaction Status Slot) is a kind of record that indicates the current state of a transaction. Each TSS has a commit SCN written in it. TSSs are permanently written in undo tablespace, and when they are no longer needed, they are deleted by the Ager. Deleted TSSs can be reused for other transactions.

A transaction that is being committed does not set a Commit SCN in all of the record versions that it has created; instead, it sets the Commit SCN in the TSS with which it is associated. Additionally, when a record is updated, a TSS identifier is written in the record, and the written TSS identifier is used by the transaction for checking record versions. That is, the transaction compares its SCN with the Commit SCN of the TSS of each record, and only reads records having Commit SCNs lower than its SCN.

8.3.5 Considerations when Using MVCC

ALTIBASE HDB uses MVCC for concurrency control with both memory and disk tablespaces. Because MVCC is different than Single Version Concurrency Control (SVCC), its precursor, there are a few things to keep in mind when using MVCC.

- Transactions that take a long time increase the size of the database.

If a particular transaction takes a long period of time to execute and is not committed, because there is the chance that the transaction may need to read images of previous versions of data, the garbage collector will not be able to delete previous images created by other transactions (previous record versions in the case of memory tables, and undo log records in the case of disk tables) and the index keys for these records. This increases the size of memory tables and the amount of space used in disk undo tablespace. In addition, in order to roll back the transaction later, log files cannot be deleted, which can consume all of the space on the file system containing the log files.

- A large number of simultaneous transactions increases the size of the database.

Previous images created by MVCC are cleared by the garbage collector. If the number of simultaneous transactions is much greater than the number of CPUs, the garbage collector may not have enough time to delete previous images, and the size of the database will continue to increase.

- Large amounts of UPDATE operations increase the size of the database.

If operations that generate large amounts of previous versions of data are executed frequently, memory tables increase in size, and the undo tablespace, which is used with disk tables, increases in size as well.

- Large numbers of previous images decrease performance.

If excessive numbers of previous images remain in the database for the reasons mentioned above, the cost of searching for a specific record might increase, which can degrade overall performance.

- Repeatable Reads Vs. Consistent Reads

In typical DBMSs in which SVCC is implemented, when records are read, an S lock is set, which conflicts with X locks, with the result that records that are being read cannot be altered. Accordingly, the isolation level of the database is set to Repeatable Read. In contrast, in ALTIBASE HDB, a record can be updated while it is being read, and thus Consistent Read is the default isolation level. As a result, a transaction that repeatedly reads a table without being committed might obtain different results every time. To avoid this, set the isolation level to Repeatable Read or use the SELECT FOR UPDATE command.

8.4 Transaction Durability

Generally, the term “transaction” refers to an independent unit that serially reads and updates a stored object (typically a page or record; implemented differently depending on the DBMS).

To improve performance, DBMSs interleave multiple transactions so that they can be executed simultaneously. Concurrency control ensures that transactions are performed concurrently without violating data integrity, and guarantees that the result of multiple transactions executed simultaneously is the same as if the transactions were executed sequentially without overlapping in time.

Therefore, DBMSs are designed to guarantee that transactions have the four following properties, so that all data can be accurately managed in the event of unexpected system failures.

- Atomicity
- Consistency
- Isolation
- Durability

8.4.1 Concept

Durability, which is one of the four properties of a transaction, means that after a transaction has been committed, the committed transaction must be guaranteed, even if a database failure occurs before the changed data are physically written to disk.

To ensure the durability of transactions, DBMSs manage transaction logs, that is, the redo log record, which contains changes to the data page. If a system failure occurs before the data changed by a committed transaction are written to disk, the DBMS reads the logs when the system is restarted, and the data are restored according to the contents of the logs.

Transaction durability is an important factor in determining transaction processing performance. In memory-based DBMSs, which exhibit performance tens of times better than disk-based DBMSs, guaranteeing transaction durability has a much bigger impact on performance than in disk-based DBMS.

For example, logs for all database updates must be written to a log file on disk in order for a DBMS to provide complete transaction durability. Disk I/O occurs when all of the logs in the memory log buffer are written to log files, and this disk I/O acts as a bottleneck in processing transactions, which degrades processing performance. That is to say, there is a tradeoff between complete transaction durability and transaction processing performance.

ALTIBASE HDB guarantees complete transaction durability and provides a transaction durability management method that allows the balance between transaction processing performance and transaction durability to be controlled in order to realize high transaction-processing performance in multiple-system implementations.

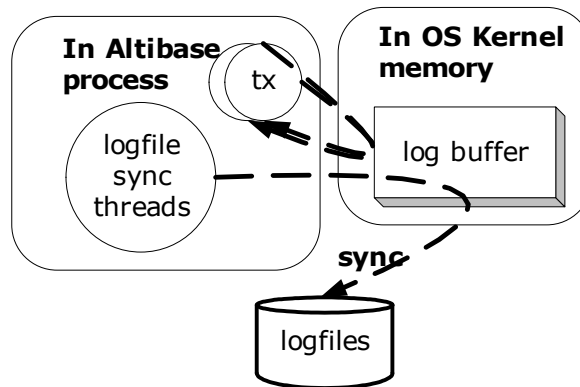
8.4.2 How to Manage Durability

In ALTIBASE HDB, durability is managed using the `COMMIT_WRITE_WAIT_MODE` and `LOG_BUFFER_TYPE` properties in the `altibase.properties` file. `COMMIT_WRITE_WAIT_MODE` specifies

whether a transaction waits until an update log has been written to a log file on disk. This property can be specified for the entire system or for individual sessions. LOG_BUFFER_TYPE specifies the type of log buffer that is used when update logs are written to a log file. This property can't be changed while the system is running. For a more detailed explanation of these properties, please refer to the *General Reference*.

- The case where a transaction does not wait until logs have been written to disk and a kernel log buffer is used:

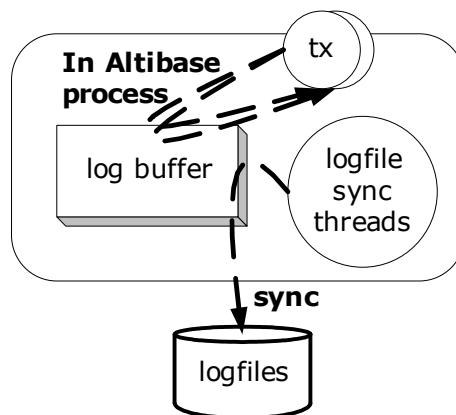
Figure 8-5 Durability in The Case where a Transaction Does Not Wait until Logs Have Been Written to Disk and a Kernel Log Buffer Is Used:



Set COMMIT_WRITE_WAIT_MODE and LOG_BUFFER_TYPE to 0 and 0, respectively. With the default ALTIBASE HDB durability property settings, update logs are stored in the log buffer of the OS kernel area, and transactions do not wait until their update logs have been written to the log file.

- The case where a transaction does not wait until logs have been written to disk and a memory log buffer is used:

Figure 8-6 Durability in The Case where a Transaction Does Not Wait until Logs Have Been Written to Disk and a Memory Log Buffer Is Used:

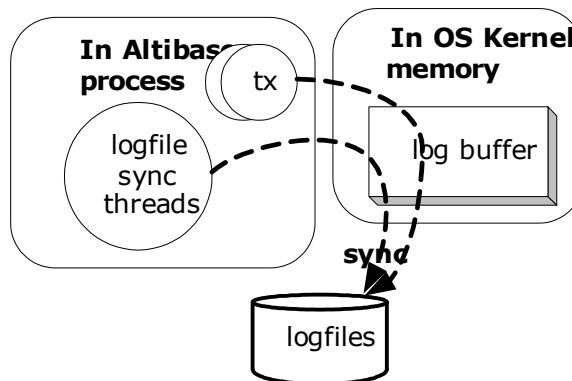


Set COMMIT_WRITE_WAIT_MODE and LOG_BUFFER_TYPE to 0 and 1, respectively. With this method, transactions store their update logs in a memory log buffer, and the sync thread itself flushes the logs in the log buffer to the log file.

8.4 Transaction Durability

- The case where a transaction waits until logs have been written to disk and a kernel log buffer is used:

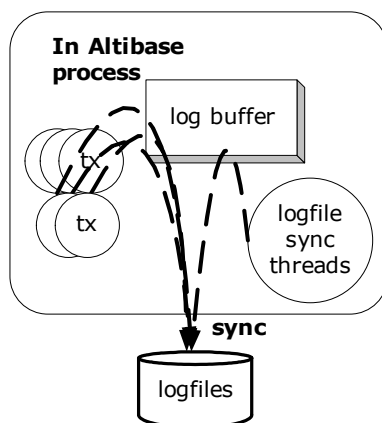
Figure 8-7 Durability in The Case where a Transaction Waits until Logs Have Been Written to Disk and a Kernel Log Buffer Is Used:



Set COMMIT_WRITE_WAIT_MODE and LOG_BUFFER_TYPE to 1 and 0, respectively. With this method, transaction update logs are stored in the log buffer of the OS kernel area, and logs for committed transactions are written directly to a log file.

- The case where a transaction waits until logs have been written to disk and a memory log buffer is used:

Figure 8-8 Durability in The Case where a Transaction Waits until Logs Have Been Written to Disk and a Memory Log Buffer Is Used:



Set COMMIT_WRITE_WAIT_MODE and LOG_BUFFER_TYPE to 1 and 1, respectively. With this method, transactions store their update logs in a memory log buffer, and logs for committed transactions are written directly to a log file, as mentioned above.

9 Database Buffer Manager

In ALTIBASE HDB, data objects in disk tablespace must be loaded from disk into memory in order for them to be accessed or updated. Memory that is used temporarily in this way is referred to as a “buffer”, and in ALTIBASE HDB this memory is collectively called the “buffer pool”.

If all of the data on disk were loaded into the buffer pool, it would be possible to access any data quickly without incurring any disk I/O expense. However, because the amount of memory is limited, it is only possible to load some of the data that exist on disk into the buffer pool. When data that have been loaded into the buffer pool are removed to make way for other data, this is called data replacement. Because this has such a strong impact on system performance, an efficient algorithm must be used to determine which data are to be saved in the buffer pool for a long time.

In ALTIBASE HDB, the entity that manages the buffer pool is known as the buffer manager. The main role of the buffer manager is to save more frequently accessed data longer in the buffer, that is, to manage the buffer efficiently.

This chapter describes the structure and function of the buffer manager, how to manage the buffer pool, related properties, and so on.

- [Structure of the Buffer Manager](#)
- [Managing Database Buffers](#)
- [Related Database Properties](#)
- [Statistics for Buffer Management](#)

9.1 Structure of the Buffer Manager

9.1.1 Components

The components of the buffer manager include the buffer area, the buffer pool, buffer frames, and Buffer Control Blocks (BCBs). The buffers in the buffer pool are organized as follows: an LRU list, a prepare list, a flush list, a checkpoint list and a hash table. This section describes the components of the buffer manager.

9.1.1.1 Buffer Area

The buffer area is the prepared memory space which is assigned to the buffer pool. The size of the buffer managed by the buffer manager depends on the size of the buffer area.

9.1.1.2 Buffer Pool

The buffer pool is the core element of the buffer manager, and is the implementation of the buffer replacement policy. It loads the requested data pages into the buffer and returns the memory address of the area into which the pages were loaded. Internally, the buffer pool manages BCBs using the hash table, LRU list, prepare list and flush lists.

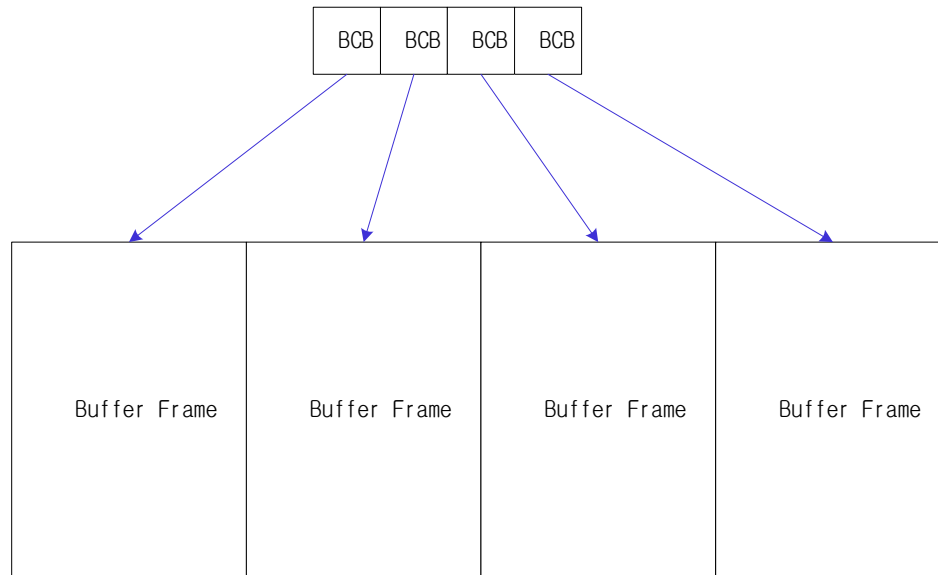
9.1.1.3 Buffer Frame

A buffer frame is space that has been set aside so that a single page can be loaded into memory, and is thus the same size as a page. A set of buffer frames constitutes a buffer pool.

9.1.1.4 Buffer Control Block (BCB)

Buffer Control Blocks (BCBs) contain information about buffer frames. One BCB corresponds to a single buffer frame. The buffer manager uses BCBs to manage information about all of the pages loaded into buffers, whereas buffer frames are merely the space into which pages can be loaded. Each BCB maintains the address of the buffer frame to which it corresponds.

The following figure and table describe the structure of buffer control blocks.

Figure 9-1 BCB Structure

The following information is maintained in BCBs:

Table 9-1 BCB Information

| Property | Description |
|----------------------|---|
| Buffer Frame Status | This is the current status of the buffer frame. Possible values: FREE/ CLEAN/DIRTY FREE: no page is loaded in the buffer frame CLEAN: page is loaded in buffer frame but not updated DIRTY: page is loaded in buffer frame and updated but not written to disk |
| Buffer Frame Address | The address of the buffer frame corresponding to the BCB |
| Space ID | The identifier of the tablespace containing the page |
| Page ID | The unique identifier of the page in the tablespace |
| Page Owner Lock | In order to access a page, it is first necessary to acquire a lock. Read, Write, and Fix mode locks can be acquired. A page in the buffer can be accessed after a lock on a BCB corresponding to the page is acquired in a particular mode. |
| Modified LSN | The next time the contents of the disk buffer are flushed to disk, the portion of the disk buffer that will be flushed to disk is the portion corresponding to all changes up to and including the change corre- sponding to this LSN. |
| Fix Count | This is the number of transactions that are simultaneously accessing a page. If this value is 1 or higher, the page can't be replaced, whereas if it is 0, the page can be replaced. |

9.1 Structure of the Buffer Manager

| Property | Description |
|-------------|--|
| Touch Count | This is the number of transactions that have accessed a page since it was loaded into a buffer. This value is used to determine whether the page is hot or cold. |

9.1.1.5 Hash Table

When the ALTIBASE HDB server receives a request for a page, it searches for the BCB of the page in the hash table in order to check whether the page has already been loaded into the buffer. The BCBs of all pages that are loaded into the buffer are registered in the hash table.

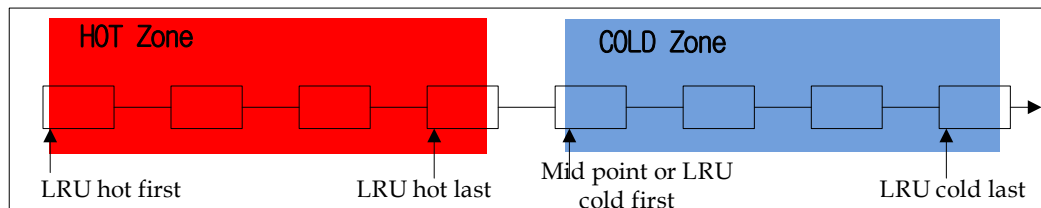
9.1.1.6 LRU (Least Recently Used) List

This is used to determine which buffers have not been accessed for a long time so that they can be replaced first.

In ALTIBASE HDB, the LRU list is separated into hot and cold zones, and can thus be called a “hot-cold LRU list”. Buffers that are accessed frequently are placed in the hot zone, whereas those that are not accessed frequently are placed in the cold zone. When a buffer needs to be replaced, only the cold zone is searched, meaning that hot buffers are not considered as replacement candidates.

When a page is first loaded into a buffer, it is inserted at the mid-point (LRU cold first) of the LRU list. When allocating a buffer to the new data page, if there are no free buffers in the prepare list, the end (LRU cold last) of this list is searched first, and a cold buffer is then replaced. The buffer that is replaced is called a “victim”.

Figure 9-2 hot-cold LRU list



Buffers that are read frequently are moved to the “LRU hot first” position in the hot zone. Meanwhile, dirty buffers, that is, buffers containing pages that have been updated but haven't been flushed to disk, are moved to the flush list. Additionally, clean buffers, that is, buffers containing pages that haven't been updated, are designated as replacement buffers as long as they are not in the hot zone.

The relative size of the hot zone can be set using the HOT_LIST_PCT property. The default is 50, which means that half of the LRU list is used as the hot zone.

9.1.1.7 Flush List

If dirty buffers are found while an LRU list is searched for buffers to replace, they are moved to the flush list. The flush list is a collection of buffers containing pages that have been updated but that haven't been written to disk yet. However, not all dirty buffers are on the flush list. This is because they are moved to the flush list not at the time point at which they are updated, but when the LRU list is searched for buffers to replace.

When replacement flushing occurs, updated pages on this list are written to disk, and clean buffers are thus obtained.

9.1.1.8 Prepare List

All buffers on the flush list that have been written to disk are moved to the prepare list. That is to say, the prepare list consists of clean buffers, for which flushing is complete.

When the buffer manager searches for buffers to replace, it first looks in the prepare list. If it can't find suitable buffers in the prepare list, it then searches the LRU list. However, even if a buffer is on the prepare list, it isn't necessarily clean. This is because the contents of the buffer can be updated.

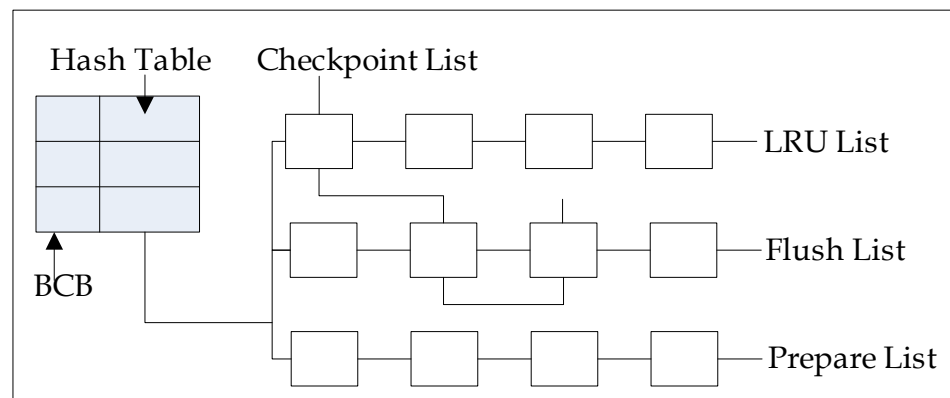
9.1.1.9 Checkpoint List

The LRU list, flush list and prepare list are mutually exclusive, so one buffer can't exist on two or more lists. However, because the checkpoint list is managed independently of the other lists, buffers on any of the other 3 lists may also be found on the checkpoint list.

Dirty buffers, that is, updated buffers, are present on the checkpoint list, and all buffers on the checkpoint list are dirty. Buffers on this list are assigned LSNs corresponding to the time points at which they were first updated, and are sorted and managed on that basis. When the checkpoint list is flushed, the buffers on the checkpoint list having the lowest LSNs are flushed first.

[Figure 9-3] shows that all of the buffers on the LRU list, flush list and prepare list can be accessed using a hash table.

Figure 9-3 Buffer Pool



9.1.1.10 List Multiplexing

The LRU list, flush list, prepare list and checkpoint list can each be multiplexed. List multiplexing prevents list lock contention when multiple database clients make simultaneous requests.

The number of each kind of list can be specified using the `BUFFER_LRU_LIST_CNT`, `BUFFER_PREPARE_LIST_CNT`, `BUFFER_FLUSH_LIST_CNT`, and `BUFFER_CHECKPOINT_LIST_CNT` properties, and can be checked by querying the `LRU_LIST_COUNT`, `PREPARE_LIST_COUNT`, `FLUSH_LIST_COUNT`, and `CHECKPOINT_LIST_COUNT` columns of the `V$BUFFPOOL_STAT` performance view. However, these values cannot be changed while the server is running.

9.1 Structure of the Buffer Manager

9.1.2 BCB State Transition

The status of each BCB is always one of FREE, CLEAN or DIRTY.

9.1.2.1 FREE

This status indicates that no pages are loaded in the buffer. The status of most buffers will be free when the system is first started up. Additionally, when a tablespace is dropped or taken offline, the status of all buffers corresponding to pages in that tablespace is free.

9.1.2.2 CLEAN

This status indicates that pages are loaded into the buffer but haven't been updated. In this status, the contents of buffer pages is the same as the contents of pages on disk. The buffer can be accessed using a hash table. The buffer can be found on one of the LRU list, flush list and prepare list, but cannot be present on the checkpoint list.

9.1.2.3 DIRTY

This status indicates that the pages in the buffer have been updated, but have not yet been written to disk. Dirty buffers are present on the checkpoint list, and also on one of the LRU list, flush list and prepare list. Dirty pages become clean pages once flushed.

9.1.3 Flush Thread

The flusher performs flushing, which means writing to disk the contents of all buffers that are to be replaced. In ALTIBASE HDB, two kinds of flushing are performed: replacement flushing and checkpoint flushing.

- Replacement Flushing

In replacement flushing, updated buffers that have not been accessed for a long time are flushed so that they can be replaced.

- Checkpoint Flushing

In checkpoint flushing, buffers that were first updated a long time previously are flushed to decrease the amount of time taken to perform checkpointing.

Replacement flushing occurs periodically but is also executed forcibly when replacement buffers can't be found. Similarly, checkpoint flushing also occurs periodically, but can also be executed by users using the ALTER SYSTEM CHECKPOINT command.

The flusher performs periodic checkpoint flushing only in the case where it has first performed replacement flushing and then has no pending tasks. That is, if the flusher finds buffers to be replaced after waiting for some specified time, it writes them to disk. The flusher waits for an amount of time that is greater than that specified using DEFAULT_FLUSHER_WAIT_SEC and less than that specified using MAX_FLUSHER_WAIT_SEC. After performing replacement flushing, the flusher again waits for the specified amount of time, and if there are no buffers to be replaced at that time, it decides whether or not to perform checkpoint flushing or to continue waiting.

The conditions that must be satisfied in order for checkpoint flushing to occur are as follows:

- The amount of time specified in CHECKPOINT_FLUSH_MAX_WAIT_SEC has passed since the flusher most recently performed flushing.
- The number of logs pertaining to pages that must be recovered exceeds some specified number when the system is restarted. That is, the lowest LSN of the LSNs of pages that have been updated and haven't been flushed to disk equals the value of CHECKPOINT_FLUSH_MAX_GAP.

However, even if the flusher is instructed to wait for a long time, if the number of buffers to be replaced becomes high as the result of frequent transactions, forced flushing may occur.

The number of buffer frames that can be flushed in one flusher cycle can be specified for checkpoint flushing using the CHECKPOINT_FLUSH_COUNT property.

Additionally, multiple flushers can be specified using the BUFFER_FLUSHER_CNT property. However, this property cannot be changed while the server is running. Each flusher can be started or paused using the ALTER SYSTEM START/STOP FLUSHER statement.

Please refer to the *SQL Reference* for more detailed information about SQL.

9.2 Managing Database Buffers

9.2.1 Access Modes

A lock must be acquired in order to access a page. Access modes are categorized, according to the kind of authority that is granted, into read, write and fix modes, which are outline below:

Table 9-2 Buffer Access Mode

| Access mode | Description |
|-------------|--|
| Read | This access mode is only for reading pages that have been loaded into the buffer. Multiple transactions can access the buffer at the same time. |
| Write | This access mode is for writing to the pages that have been loaded into the buffer. In this mode, only one transaction can access a page at a time. |
| Fix | In this access mode, after the page has been uploaded to the buffer, it is guaranteed that it will not be replaced by the buffer manager. If a transaction accesses and reads a page in fix mode, the accuracy of the data cannot be guaranteed. In order to be sure that the data being read are correct, the page must be accessed in read mode. |

The relationship between permissions and access modes are shown in [Table 8-3] below:

Table 9-3 Permissions and Access Modes

| Permissions | Transaction Mode | | |
|-------------|------------------|-------|-----|
| | Read | Write | Fix |
| Read | O | X | O |
| Write | X | X | O |
| Fix | O | O | O |

As shown in [Table 8-3], when a write mode access request is made, if some other transaction has already acquired a read or write mode lock, the request will either be queued or fail. Meanwhile, if some other transaction has already acquired a read mode lock, a request for access in read mode will succeed, but a request for access in write mode will fail.

9.2.2 Standby Modes

The standby mode determines whether to wait or to return an error immediately when access cannot be granted in the requested mode because the requested page is being used by another transaction.

| Access mode | Description |
|-------------|--|
| Wait | If another transaction has already acquired a lock and access in the requested mode cannot be granted, wait until the lock is released after the other transaction finishes its task. |
| No-Wait | If another transaction has already acquired a lock and the access in the requested mode cannot be granted, return an error without waiting for the other transaction to finish its task. |

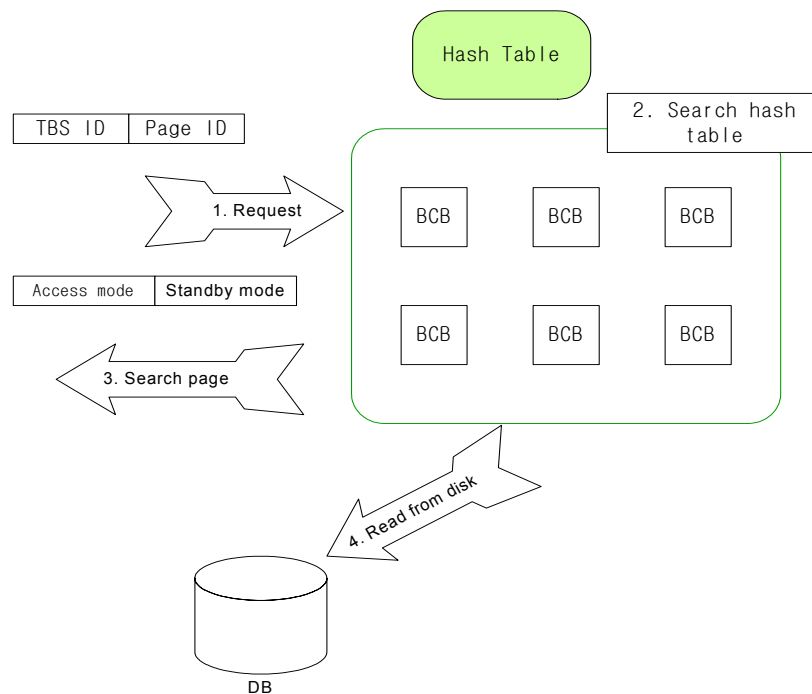
9.2.3 Page Request Procedure

1. Search the Hash Table

The buffer manager receives a page request, which includes information about the page ID, access mode, and wait mode.

When the buffer manager receives a request, it first checks the hash table for the BCB. If the requested page has already been loaded into a buffer, the BCB for that page will be found in the hash table. If the BCB cannot be found in the hash table, this means that the page has not been loaded into a buffer, and the page must be read from disk and loaded into a buffer.

Figure 9-4 Searching the Hash Table



2. Acquire a Lock

9.2 Managing Database Buffers

In ALTIBASE HDB, in order to guarantee that data are accurately read, pages that are being read from disk may not be accessed by other transactions until they have been completely read.

This is accomplished by requiring that write privileges be acquired when reading from disk. This means that if a transaction is able to acquire read or writes privileges for a page, the data were not being read from disk at that point in time.

As shown in [Table 8-5], whether a lock is granted differs depending on the access mode, on whether pages are read from disk, and on the standby mode.

| Access Mode | Read from Disk | Standby Mode | Result |
|-------------|----------------|----------------|--|
| Fix | O | Not applicable | Allow after waiting for reading to finish |
| Fix | X | Not applicable | Allow |
| Read | O | Wait | Allow when the access mode is permitted; stand by if access fails |
| Read | O | No-Wait | Allow when the access mode is permitted; stand by if access fails |
| Read | X | Wait | Allow when the access mode is permitted; stand by if access fails |
| Read | X | No-Wait | Allow when the access mode is permitted; return an error if access fails |
| Write | O | Wait | Allow when the access mode is permitted; stand by if access fails |
| Write | O | No-Wait | Allow when the access mode is permitted; stand by if access fails |
| Write | X | Wait | Allow when the access mode is permitted; stand by if access fails |
| Write | X | No-Wait | Allow when the access mode is permitted; return an error if access fails |

As seen in the above table, if access is requested in fix mode, it will be granted immediately regardless of the privileges with which other locks are set for the current page. However, if the page is being read from disk, the request must stand by until the page has been completely read from disk.

Moreover, when a request for access with read or write privileges is made in No-Wait mode, if the page is being read from disk, the request must stand by until the page has been completely read from disk.

9.2.4 Reading Pages from Disk

If a page requested by the buffer manager has not been loaded into a buffer, the page is read from disk according to the following procedure.

1. An Available BCB is Acquired

When a page has not been loaded into a buffer, the ALTIBASE HDB server must first acquire a BCB in order to load the page into a buffer. To find a buffer into which to load the page, the prepare list is searched first. Finding a free buffer on the prepare list is the easiest way to load a page.

However, if no free buffer can be obtained in this way, the next step is to find a buffer to replace. When the system is initially started up, there are a lot of free buffers, but after that the likelihood of finding a free buffer in the buffer pool is low unless a tablespace is dropped or taken offline. Additionally, no buffers are newly freed after flushing, because buffers are used to store the contents of pages even after they have been flushed.

2. A Buffer is Replaced

The prepare list is checked first when searching for buffers to replace. This is because the prepare list contains clean buffers, which were flushed and moved from the flush list.

If buffers suitable for replacement can't be found on the prepare list, the LRU list is searched. However, if no clean buffers can be found even when the LRU list is searched, another prepare list is checked. This process is repeated until a clean buffer is found or until all prepare lists have been searched.

However, if no clean buffers can be found even after all prepare lists have been searched, the flushers are instructed to operate, and the search for a buffer pauses to wait for a buffer to be added to a prepare list. This waiting time is specified in the `BUFFER_VICTIM_SEARCH_INTERVAL` property. If no buffers can be found during this waiting time, the search proceeds to the next prepare list. In ALTIBASE HDB, this phenomenon is called "victim search warp".

If the value of `VICTIM_SEARCH_WARP`, which can be observed in the `V$BUFFPOOL_STAT` performance view, is high, this indicates long waits for buffer replacement. If this problem persists, increase either the size of buffers or the number of flush threads, and restart the system.

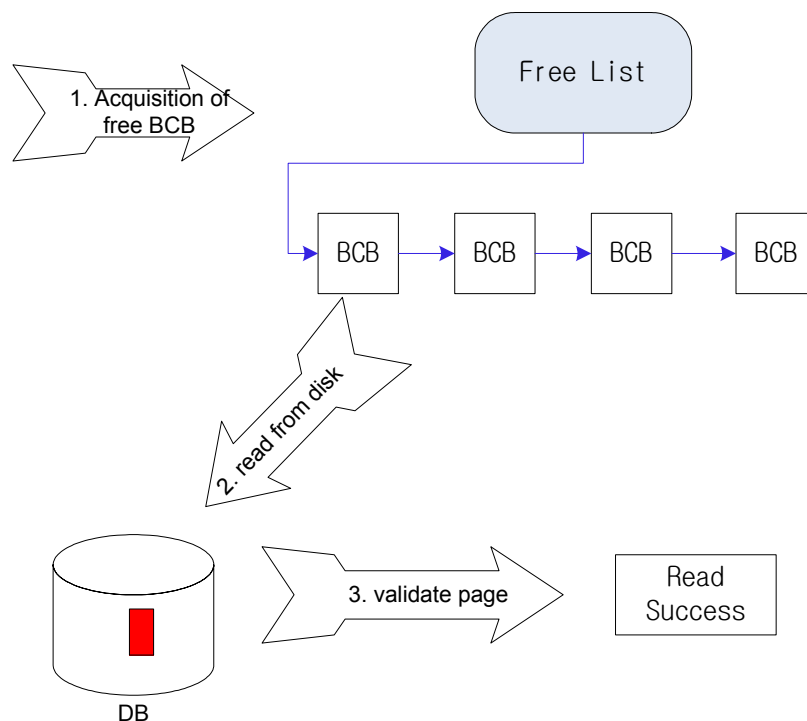
In order for buffers on the LRU list to be replaced, they must meet the following conditions:

- They have never been fixed (buffers whose fix count = 0).
- They have pages loaded into them, but those pages have not been modified (clean buffers).
- They are not hot (buffers whose touch count < `HOT_TOUCH_CNT`).

If buffers suitable for replacement are found, they are removed from the hash table, and the next task is performed.

3. The Page is Verified after Being Read

After a BCB has been acquired, a page is loaded from the disk to the buffer. However, a portion of the page on the disk may have been lost or corrupted due to some unforeseeable circumstance such as a hard disk error or a power failure. If such a problem goes unnoticed, users might be presented with invalid data, so it is important that the ALTIBASE HDB server be aware of such problems. Therefore the ALTIBASE HDB server checks the integrity of each page immediately after the page has been loaded from disk to the buffer.

Figure 9-5 Verify the Page

9.2.5 Flushing

1. Selecting Buffers to Flush

Pages that have been modified since they were loaded into buffers are flushed either when they are selected as victims or during checkpointing. Pages that have been loaded into buffers must meet the following conditions in order to be flushed:

- They have been updated at least once.
- They have never been fixed (fix count = 0).

Pages can be read by other transactions while being flushed. Therefore, in order to perform flushing, read mode privileges are required.

2. Copying Pages to the I/O Buffer

Once pages to be flushed have been chosen, they are first copied into I/O buffer memory before being written to disk. They are written to disk after being copied there because I/O tasks are expensive and time-consuming compared to memory operations.

When buffer pages have been copied to the I/O buffer, this buffer can be read and updated by other transactions. However, if the I/O buffer were not used, it would be impossible to read or update these pages while they are being written to disk (that is, while the I/O task is being conducted).

3. Log Flushing

Before a modified page is written to disk, a log of the changes must first be written to disk. This is called the WAL (Write-Ahead Logging) protocol.

Additionally, a checksum value is calculated and written to check whether the page has been corrupted when a page is loaded from disk into a buffer. Every time the page is read from disk, the checksum value is calculated and compared with the checksum for the page to verify the integrity of the page.

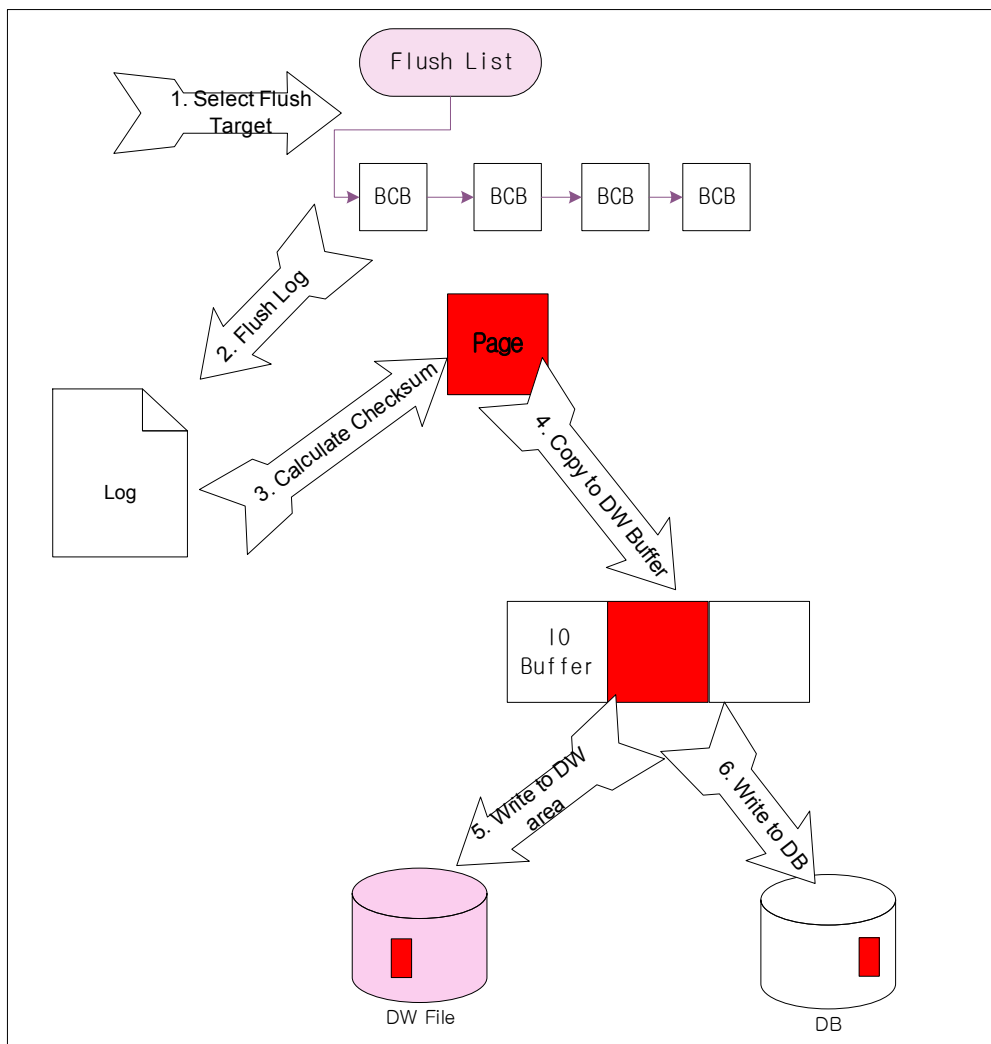
4. Writing Pages in the I/O Buffer to Disk

When the pages in the I/O buffer are written to disk, the entire contents of the buffer are first written all at once to a so-called “double-write file”, and then each page is written to a data file.

The reason for writing pages to disk twice in this way is that it is impossible to recover partially written pages, such as those that are generated if a system failure occurs while pages are being written to disk. Such inconsistent pages cannot be recovered even using redo logs.

In ALTIBASE HDB, the directory in which the double write file is saved is specified using the `DOUBLE_WRITE_DIRECTORY` property. This file is used to verify the consistency of data files when the system is started and to perform system recovery. If this file doesn't exist, these tasks are not conducted.

Figure 9-6 Flushing Pages to Disk



9.3 Related Database Properties

To use the buffer manager, the properties in the `altibase.properties` file must be suitably set. The properties related to buffers are listed below. For detailed information about each of these properties, please refer to the *General Reference*.

- `BUFFER_AREA_CHUNK_SIZE`
- `BUFFER_AREA_SIZE`
- `BUFFER_CHECKPOINT_LIST_CNT`
- `BUFFER_FLUSH_LIST_CNT`
- `BUFFER_FLUSHER_CNT`
- `BUFFER_HASH_BUCKET_DENSITY`
- `BUFFER_HASH_CHAIN_LATCH_DENSITY`
- `BUFFER_LRU_LIST_CNT`
- `BUFFER_PINNING_COUNT`
- `BUFFER_PINNING_HISTORY_COUNT`
- `BUFFER_PREPARE_LIST_CNT`
- `BUFFER_VICTIM_SEARCH_INTERVAL`
- `BUFFER_VICTIM_SEARCH_PCT`
- `CHECKPOINT_FLUSH_COUNT`
- `CHECKPOINT_FLUSH_MAX_GAP`
- `CHECKPOINT_FLUSH_MAX_WAIT_SEC`
- `DEFAULT_FLUSHER_WAIT_SEC`
- `HIGH_FLUSH_PCT`
- `HOT_LIST_PCT`
- `HOT_TOUCH_CNT`
- `LOW_FLUSH_PCT`
- `LOW_PREPARE_PCT`
- `MAX_FLUSHER_WAIT_SEC`
- `TOUCH_TIME_INTERVAL`

9.4 Statistics for Buffer Management

Statistical information about the buffer manager can be checked using the performance views provided with ALTIBASE HDB. Please refer to the *General Reference* for detailed descriptions of the available performance views.

Information related to the buffer pool can be viewed using V\$BUFFERPOOL_STAT, and information related to the flusher can be checked using V\$FLUSHINFO and V\$FLUSHER. Statistical information related to the buffer frames managed by the buffer manager can be viewed using V\$BUFFPAGEINFO, and statistical information related to the buffer pool of undo tablespace can be viewed using V\$UNDO_BUFF_STAT.

Because the statistical information accumulates from the time the server is started, to obtain statistics for a particular period, calculation must be performed for all columns in this way: (present value - value at the start of the period of interest).

9.4.1 Calculating Hit Ratio

The cumulative hit ratio of the buffer pool can be checked in the HIT_RATIO column of the V\$BUFFPOOL_STAT performance view.

The Hit Ratio can be calculated using the following formula:

$$\text{Hit Ratio} = (\text{GET_PAGES} + \text{FIX_PAGES} - \text{READ_PAGES}) / (\text{GET_PAGES} + \text{FIX_PAGES})$$

Ex)

```
iSQL> select hit_ratio from X$BUFFER_POOL_STAT;
```


10 Backup and Recovery

This chapter explains the ALTIBASE HDB backup and recovery features, which are provided to help prevent the loss of data in the event of unforeseen circumstances, such as the loss or damage of a disk or data file, and explains how to manage database backup and recovery tasks.

This chapter contains the following sections:

- [Database Backup](#)
- [Database Recovery](#)
- [Backup and Recovery Examples](#)

10.1 Database Backup

This section describes the backup methods and backup policies available for use with memory and disk tablespaces, which depend on whether the database is running in Archivelog mode or Noarchivelog mode.

10.1.1 The ALTIBASE HDB Backup Policy

ALTIBASE HDB supports the following kinds of backup operations:

- Logical backup
 - Utility backup
- Physical backup
 - Offline backup
 - Online backup

When logical backup is performed using the aexport or iLoader utility, script files for creating tables, indexes and files in which table records are written are created.

Physical backup means that the data files and log anchor files constituting the database are copied.

Physical backup is categorized as either online backup or offline backup, depending on whether the service is interrupted while a snapshot of the data file or files is taken.

Before performing offline backup, the database server must be shut down normally, and then all tablespace files, log anchor files and log files must be copied.

In online backup, the database's data files, log anchor files, etc. are copied without interrupting service. During this process, some uncommitted data may also be backed up. Therefore, in the event of recovery, log files will be required in order to undo these uncommitted transactions. Additionally, online backup is only possible in archivelog mode, in which archive log files are created.

When performing online backup, the entire database can be backed up, or a specific tablespace or log anchor file can be backed up as desired.

The statements that would be used in each case are as follows:

- To back up the entire database

```
alter database backup database to '/backup_dir';
```
- To back up an individual tablespace:

```
alter database backup tablespace SYS_TBS_DISK_DATA to '/backup_dir';
alter tablespace SYS_TBS_DISK_DATA begin backup;
cp SYS_TBS_DISK_DATA-DATA-FILES /backup_dir
...
alter tablespace SYS_TBS_DISK_DATA end backup;
```

SYS_TBS_MEM_DIC, which is the memory tablespace that contains system catalog data, is backed up not only by backing up the tablespace, but also when the entire database is backed up.

The following table describes the various backup modes in ALTIBASE HDB:

Table 10-1 Backup Methods

| Backup Type | Backup Method | Backup Object | Restoration Method | Possible while Online? |
|--------------------------------------|---|--|---|------------------------|
| Backup using the iLoader utility | Use the iLoader "out" command | User-defined tables | Use the iLoader "in" command. | O |
| Online backup of entire database | Use the SQL statement: ALTER DATABASE BACKUP DATABASE TO backup_dir; | All data files and log anchor files in all tablespaces in the system | 1> Use the UNIX "cp" command 2> ALTER DATABASE RECOVER DATABASE; | O |
| Online backup of specific tablespace | Use the SQL statement: ALTER DATABASE BACKUP TABLESPACE tablespace name TO backup_dir; or 1. ALTER TABLESPACE tablespace name BEGIN BACKUP; 2. Use the UNIX "cp" command cp <original_files> <backup_dir> 3. ALTER TABLESPACE tablespace name END BACKUP; | All data files in the tablespace | 1> Use the UNIX "cp" command 2> ALTER DATABASE RECOVER DATABASE; | O |
| Offline backup | 1> Shut down the database 2> Use the UNIX "cp" command | The entire database | Use the UNIX "cp" command | X |
| Backup Time Comparison | iLoader < online backup < offline backup | | | |

10.1.1.1 Classifying Backup According to Scope

- Database-Level Backup

All data files in the database are backed up.

Ensures that all backup-related log files are archived.

- Tablespace-Level Backup

All data files in a specific memory or disk tablespace are backed up.

10.1 Database Backup

Because the archiving of backup-related log files is not guaranteed, they must be archived separately using a DCL statement.

10.1.1.2 Classifying Backup according to Method

- Database-Driven Backup

Files are copied by the ALTIBASE HDB server.

When a single DCL statement is executed, the entire DB (or specified tablespace) is backed up according to a predetermined order.

Can be used to perform both database-level backup and tablespace-level backup.

- DBA-Driven Backup

Files are copied by the DBA.

Because multiple tablespaces can be backed up in parallel, integration with 3rd-party backup solutions is supported.

Can be used to perform only tablespace-level backup.

10.1.2 Database Mode

The database runs in either archivelog mode or noarchivelog mode, depending on the way that online log files, in which all changes made to data are recorded, are managed.

In archivelog mode, when a log file fills up, logging continues in a new log file, and the previous log file is copied to an archive directory. The archive log directory is set using the ARCHIVE_DIR property in the \$ALTIBASE_HOME/conf/altibase.properties file.

In noarchivelog mode, these log files are deleted automatically by the system after checkpointing.

[Table 9-2] outlines the pros and cons of each database mode.

Table 10-2 Pros and Cons of Each Database Mode

| Database Mode | Pros | Cons |
|-----------------|--|--|
| archivelog mode | <ul style="list-style-type: none">- Media recovery is supported. The database can be recovered up to the current point even if data files are lost or corrupted. | <ul style="list-style-type: none">- Disk space is required to store archive log files.- Because the DBA must provide a separate storage device in which to store archive logs, organize files, etc., the burden of management tasks is increased.- If there is not enough disk space for archive logs, a fault occurs. |

| Database Mode | Pros | Cons |
|-------------------|--|---|
| noarchivelog mode | - The DBA does not have to manage archive log files. | - The DBA can only perform recovery using offline backups, even if data files are corrupted. Changes to the data that were made between the time of the backup and the time point at which the data file was corrupted cannot be recovered. |

The database mode is determined when the database is created using the `CREATE DATABASE` statement, and can be changed during the control phase when the database is started.

An example of creating a database in archivelog mode is shown below:

```
create database mydatabase INITSIZE=100M archivelog character set US7ASCII
national character set UTF16;
```

The following example shows how to change the database mode to archivelog mode during the control phase when starting up the database:

```
% isql -silent -u sys -p manager -sysdba
iSQL(sysdba)> startup control;
iSQL(sysdba)> alter database archivelog;
```

10.1.3 Online Backup and Media Recovery in Different Database Modes

Table 10-3 Backup and Recovery in Different Database Modes

| Mode | Backup Method | Media Recovery Type |
|-------------------|---|--|
| noarchivelog mode | Offline backup | Full database recovery |
| archivelog mode | Online backup (offline backup is possible) | Complete recovery - Full database recovery Incomplete recovery - Cancel-based recovery - Time-based recovery |
| Either mode | Backup using the iLoader utility | Recovery using the iLoader utility |

10.1.4 Online Backup and Media Recovery in Different Tablespace States

Table 10-4 Online Backup and Media Recovery in Different Tablespace States

| Tablespace State | Online Backup | Media Recovery |
|------------------|---------------|----------------|
| Online | Yes | Yes |

10.1 Database Backup

| Tablespace State | Online Backup | Media Recovery |
|------------------|---------------|----------------|
| Offline | Yes | Yes |
| Discarded | No | No |
| Dropped | No | No |
| Backup | No | n/a |

10.1.5 Cautions when Performing Online Backup

Online backup and checkpointing cannot be conducted at the same time.

Because the contents of the database that are maintained in memory are written to disk during checkpointing, and online backup only guarantees that the data are backed up to the point in time at which the backup was made, checkpointing and online backup are considered mutually exclusive, and cannot be run at the same time.

If a request to perform online backup is received during checkpointing, the online backup will start after checkpointing is complete.

Similarly, if a request to perform checkpointing is received while online backup is underway, checkpointing will start after the online backup is complete. Because ALTIBASE HDB is a hybrid database, one of its characteristics is that memory tablespaces are backed up first, followed by disk tablespaces, when the database is backed up.

Checkpointing cannot be performed on memory tablespaces while they are being backed up. When the backup of memory tablespaces is complete and the backup of disk tablespaces is underway, memory tablespaces can be checkpointed, but disk tablespaces cannot.

Replication data are also backed up when replication is active.

When an Altibase database on which replication is active is backed up, the replication-related data are backed up in their current state. Therefore, if a database backup is used to perform recovery on a different host, problems will occur when replication is subsequently executed, attributable to the difference in the hosts' network addresses.

Therefore, if replication is active, it is necessary to either: (1) stop replication and delete all replication-related data before performing backup, or (2) in two systems on which replication is active, if performing recovery for one system using the backup files from the other system, ensure that replication is stopped on all systems and prevent replication from starting during database recovery.

10.2 Database Recovery

In every database system, the possibility of a system or hardware failure always exists. If a failure that affects the database occurs, then recovery must be performed to restore the database. The goals after such failure are to ensure that the effects of all committed transactions are preserved in the recovered database and to return to normal operation as quickly as possible while insulating users from problems caused by the failure.

10.2.1 The ALTIBASE HDB Recovery Policy

ALTIBASE HDB supports the following types of recovery:

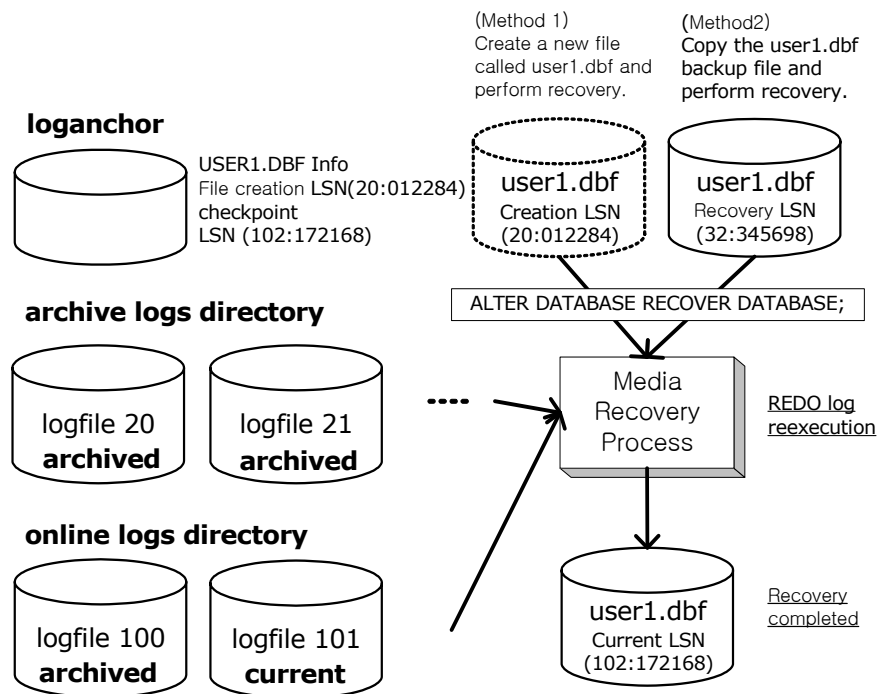
- Restart Recovery
- Media Recovery

Restart recovery is automatically executed during the startup phase if the ALTIBASE HDB process was terminated abnormally due to a system crash or software error.

If a data file is lost or corrupted, media recovery uses a snapshot of a previous backup of the data file and re-executes all operations from the recovery start LSN to the current LSN to restore all data from the backup of the data file to the current point in time.

Whether it is necessary to perform media recovery is determined by checking whether the version of the data file which is written in the log anchor file is the same as the version of the current data file. This is illustrated below:

Figure 10-1 The ALTIBASE HDB Recovery Process



10.2 Database Recovery

Media recovery can be performed only during the control phase when starting up the database. That is, Altibase only supports offline media recovery.

10.2.1.1 Example

The following is an example of media recovery. The data file 'user1.dbf,' which belonged to the tablespace TEST, has been lost. The lost data file is restored to its current state using the file 'user1.dbf,' an online backup that was made two days ago.

```
% cp /bck/user1.dbf $ALTIBASE_HOME/dbs
% isql -silent -u sys -p manager -sysdba
[ERR-00000 : Connected to idle instance]
iSQL(sysdba)> startup control;
Trying Connect to Altibase.. Connected with Altibase.
TRANSITION TO PHASE : PROCESS
TRANSITION TO PHASE : CONTROL
Command execute success.
iSQL(sysdba)> alter database recover database;
Alter success.
```

The data file 'user1.dbf' in the tablespace TEST has been restored.

```
iSQL(sysdba)> startup service;
```

Media recovery has been completed, and thus the system progresses to the service phase.

10.2.2 Complete vs. Incomplete Recovery

The ALTIBASE HDB media recovery policies support both complete recovery and incomplete recovery.

"Complete recovery" means restoring data files up to the current time point in the case where online logs and archive logs have not been lost.

"Incomplete recovery" refers to the case in which archive log files or online log files have been lost, and thus the database is recovered to the point in time immediately before the log files were lost, or the case where the entire database is rewound to a certain time point in the past in order to restore the database at that point in time.

The following is an example of complete recovery:

```
ALTER DATABASE RECOVER DATABASE;
```

Incomplete recovery can be classified into two cases:

- To rewind the entire database to a certain point in the past:

Copy all database backup files, which were created on September 10, 2007, and use them to perform recovery.

```
ALTER DATABASE RECOVER DATABASE UNTIL TIME '2007-09-10:17:55:00';
```

The entire DB is rewound to September 10, 2007, 5:55 pm.

- When the database cannot be restored to the present point in time due to the corruption of a particular online log file, it is restored to the time point immediately before the corruption of

the online log file using the following command:

```
ALTER DATABASE RECOVER DATABASE UNTIL CANCEL;
```

If incomplete recovery was performed in the control startup phase, the following statement must be subsequently executed in order to proceed to the meta startup phase.

```
ALTER DATABASE db_name META RESETLOGS;
```

Because the database has been restored to a specific past time point, it will be necessary to prevent automatic recovery ("restart recover") from executing when the database is restarted. To accomplish this, the online logs are initialized using the resetlogs option.

When the database is started up in the meta startup phase, if the resetlogs option is used to reset the logs, the entire database must be backed up, either offline or online.

The reason for this is as follows: if the logs are reset when proceeding to the meta startup phase and then another media error occurs two days later, it will only be possible to recover the data to the time point before the logs were reset. That is, changes made to the data in the two days since the logs were reset will be lost.

10.2.3 Cautions when Performing Media Recovery

Due to the media recovery algorithm, the current log anchor files must be used for media recovery whenever possible.

When recovery is performed, only the data files should be restored from backup copies. Except for special cases, log anchor files should not be restored from backup copies.

One such special case is the case where a user accidentally deletes a tablespace using the `DROP TABLESPACE` command. Because there are no data pertaining to the dropped tablespace in the current log anchor files at that time, it is acceptable to restore log anchor files from backup copies.

When recovering a data file of a memory tablespace, the stable memory data file must be used to restore the other memory data file.

Because ping-pong checkpointing is used for memory tablespaces in ALTIBASE HDB, two data files pertaining to each memory tablespace are maintained on disk. A pair of data files, in which the same image is saved, is stored in a location set using the `MEM_DB_DIR` property. Both data files must exist in order for ALTIBASE HDB to operate normally. At any particular point in time, the memory tablespace is only using one of these data files.

In order to reduce the time required to perform backups, only the most recently checkpointed data file for the memory tablespace is backed up.

Therefore, when performing recovery using the backup of the memory tablespace data file, the identical image must be written to the other data file.

The data files containing the backup of memory tablespace are as follows:

```
SYS_TBS_MEM_DIC-1-0,  
SYS_TBS_MEM_DIC-1-1,  
SYS_TBS_MEM_DIC-1-2
```

The data files for a memory tablespace must be copied in the following manner:

10.2 Database Recovery

```
% cp SYS_TBS_MEM_DIC-1-0 $ALTIBASE_HOME/dbs;  
% cp SYS_TBS_MEM_DIC-1-0 $ALTIBASE_HOME/dbs/SYS_TBS_MEM_DIC-0-0;  
% cp SYS_TBS_MEM_DIC-1-1 $ALTIBASE_HOME/dbs;  
% cp SYS_TBS_MEM_DIC-1-1 $ALTIBASE_HOME/dbs/SYS_TBS_MEM_DIC-0-1;  
% cp SYS_TBS_MEM_DIC-1-2 $ALTIBASE_HOME/dbs;  
% cp SYS_TBS_MEM_DIC-1-2 $ALTIBASE_HOME/dbs/SYS_TBS_MEM_DIC-0-2;
```

If a tablespace is added or deleted or the name of a tablespace is changed, the dictionary tablespace (SYS_TBS_MEM_DIC) and the changed tablespace will need to be backed up. Otherwise, the entire database will need to be backed up.

```
iSQL(sysdba) > ALTER DATABASE BACKUP TABLESPACE SYS_TBS_MEM_DIC TO '/  
backup_dir';
```

Because log anchor files include information about the tablespaces in a database, they must be backed up along with the dictionary tablespace whenever the structure of a tablespace is changed.

```
iSQL(sysdba) > ALTER DATABASE BACKUP LOGANCHOR TO 'anchor_path';
```

10.3 Backup and Recovery Examples

10.3.1 Backing Up and Restoring Tables Using the iLoader Utility

Backup can be used in order to be prepared for unforeseen problems with individual tables, or to back up a particular table for some specific reason.

10.3.1.1 Before Backup

Before a backup can be performed, it is necessary to create a schema file, called a FORM file, for the table to be backed up. The FORM file contains basic information about the table, such as the name and data type of each column.

Ex.) To create a form file for table t1 (a file called t1.fmt will be created):

```
iloaders> formout -T t1 -f t1.fmt
```

10.3.1.2 Backup

Use the iLoader utility with the `out` option. A backup of the table will be created having the name specified by the user.

Ex.) Back up table t1 (the form file to use is t1.fmt, and the backup file that will be created is t1.dat)

```
iloaders> out -d t1.dat -f t1.fmt
```

10.3.1.3 Restore

Use the iLoader utility with the `in` option.

If the table being restored contains records, the user can keep the existing records or overwrite them. If the user does not explicitly specify what to do with existing records, they will be kept.

Ex.) Restore table t1

```
iloaders> in -d t1.dat -f t1.fmt
```

10.3.2 Offline Backup and Recovery

Offline backup and recovery is generally used when the database is running in `noarchivelog` mode.

10.3.2.1 Cautions When Performing Offline Backup

Before performing offline backup, stop all services related to ALTIBASE HDB.

If offline backup is performed while the database is running, the contents of log files will change while they are being backed up, and thus the backup will not be performed correctly. Therefore, be sure that ALTIBASE HDB is stopped before performing offline backup.

10.3 Backup and Recovery Examples

10.3.2.2 How to Perform Backup

Use the UNIX cp (copy) command, or a similar command depending on the OS, to back up all of the data files, log files and log anchor files in all tablespaces. In ALTIBASE HDB, not only the memory data files but also the data files and log anchor files pertaining to disk tablespaces must be backed up.

The location where the data files for memory tablespace are stored is set using the MEM_DB_DIR property in the ALTIBASE HDB property file \$ALTIBASE_HOME/conf/altibase.properties. To back up the data files for memory tablespace, all of the directories specified using MEM_DB_DIR must be copied.

The location of the log anchor files is set using the LOGANCHOR_DIR property in the \$ALTIBASE_HOME/conf/altibase.properties file. The files in the directory specified using LOGANCHOR_DIR must be copied in order to back up the log anchor files. Furthermore, the data files for disk tablespaces should be copied after consulting the data dictionary.

Ex)

```
$ALTIBASE_HOME/conf/altibase.properties
MEM_DB_DIR=$ALTIBASE_HOME/dbs0
MEM_DB_DIR = $ALTIBASE_HOME/dbs1
LOGANCHOR_DIR = $ALTIBASE_HOME/logs
```

Disk tablespaces have been only system tablespace, undo tablespace, and temp tablespace.

Location where backup file stores: /home/backup

```
$cp -r $ALTIBASE_HOME/dbs0 /home/backup
$cp -r $ALTIBASE_HOME/dbs1 /home/backup
$cp -r $ALTIBASE_HOME/logs /home/backup
$cp -r $ALTIBASE_HOME/dbs/system*.dbf /home/backup
$cp -r $ALTIBASE_HOME/dbs/undo.dbf /home/backup
$cp -r $ALTIBASE_HOME/dbs/temp.dbf /home/backup
```

10.3.2.3 How to Perform Recovery

The ALTIBASE HDB properties file that was used when the database was backed up must be used when performing recovery. Any databases remaining in shared memory are deleted. The backup files created during backup can be restored using the “cp” copy command or equivalent. It is necessary to have sufficient privileges to access these files.

In the following example, the database that was backed up above is restored.

Ex) Restore using database backup

```
$cp -r /home/backup/dbs0 $ALTIBASE_HOME/dbs0
$cp -r /home/backup/dbs1 $ALTIBASE_HOME/dbs1
$cp -r /home/backup/logs $ALTIBASE_HOME/logs
$cp -r /home/backup/system*.dbf $ALTIBASE_HOME/dbs
$cp -r /home/backup/undo.dbf $ALTIBASE_HOME/dbs
$cp -r /home/backup/temp.dbf $ALTIBASE_HOME/dbs
```

10.3.3 Database-Driven Online Backup

10.3.3.1 Database-Level Online Backup

The entire database is backed up online to the /backup_dir directory.

```
iSQL(sysdba)> ALTER DATABASE BACKUP DATABASE TO '/backup_dir';
% ls /backup_dir
SYS_TBS_MEM_DIC-0-0
SYS_TBS_MEM_DATA-0-0
system001.dbf
system002.dbf
undo001.dbf
loganchor0
loganchor2
loganchor1
```

10.3.3.2 Tablespace-Level Online Backup

The stable versions of the data files in the SYS_TBS_MEM_DIC tablespace are backed up online to the /backup_dir directory.

```
iSQL(sysdba)> alter database backup tablespace SYS_TBS_MEM_DIC to '/
backup_dir';
% ls /backup_dir
SYS_TBS_MEM_DIC-0-0
```

10.3.3.3 Loganchor Online Backup

All loganchor files are backed up online to the /backup_dir directory.

```
iSQL(sysdba)> alter database backup loganchor to '/backup_dir';
% ls /backup_dir
loganchor0 loganchor1 loganchor2
```

10.3.4 DBA-Driven Online Backup

10.3.4.1 Tablespace-Level Online Backup

Back up the data files in the USER_MEMORY_TBS and USER_DISK_TBS tablespaces online to the /backup_dir directory.

The data files in the memory tablespace are backed up online after checking that they are stable copies.

```
iSQL(sysdba)> ALTER TABLESPACE user_memory_tbs BEGIN BACKUP;
iSQL(sysdba)> SELECT * FROM V$STABLE_MEM_DATAFILES;
V$STABLE_MEM_DATAFILES.MEM_DATA_FILE
-----
/altibase_home/dbs/USER_MEM_TBS-0-0

% cp $ALTIBASE_HOME/dbs/USER_MEMORY_TBS-0-0 /backup_dir/

iSQL(sysdba)> ALTER TABLESPACE user_memory_tbs END BACKUP;
iSQL(sysdba)> ALTER TABLESPACE user_disk_tbs BEGIN BACKUP;

% cp $ALTIBASE_HOME/dbs/USER_DISK_TBS.dbf /backup_dir/
iSQL(sysdba)> ALTER TABLESPACE user_disk_tbs END BACKUP;

% ls /backup_dir
USER_MEMORY_TBS-0-0 USER_DISK_TBS.dbf
```

10.3 Backup and Recovery Examples

10.3.5 Completing Online Backup

When manually backing up the database online, the final step is to issue a command to forcibly archive the backup-related log files. This command closes the current log file, even if it is not full. Logging continues in the next log file.

```
iSQL(sysdba) > ALTER SYSTEM SWITCH LOGFILE;
```

A message indicating that online backup is complete is written to altibase_sm.log. In this example of manual backup, the completion of backup is indicated by a message saying that the log file named logfile15341 is being archived.

```
[2007/09/18 14:42:38] [Thread-6] [Level-9]
Waiting logfile15341 to archive
```

```
[2007/09/18 14:42:43] [Thread-6] [Level-9]
Database-Level Backup Completed [SUCCESS]
```

10.3.6 Media Recovery Example 1

Suppose that the database is operating in archivelog mode and that the data file \$ALTIBASE_HOME/dbs/abc.dbf, which was not backed up, has been lost.

Note: Data files in a memory tablespace cannot be recovered in this way.

10.3.6.1 Recovery Procedure

Check which archive log files are required for complete recovery.

```
iSQL(sysdba) > SELECT NAME, CREATE_LSN_LFGID,
CREATE_LSN_FILENO FROM V$DATAFILES;
-----
..
/altibase_home/dbs/abc.dbf
0 18320
```

To check the log file that was most recently deleted, view the contents of the log anchor file using the “dumpla” utility.

```
% dumpla loganchor0
```

```
[LOGANCHOR HEADER]
Binary DB Version          [ 5.4.1 ]
Archivelog Mode            [ Archivelog ]
Begin Checkpoint LSN       [ 0, 20345, 469859 ]
End Checkpoint LSN         [ 0, 20345, 470300 ]
Disk Redo LSN              [ 0, 20345, 469859 ]
Global SN                  [ 9476323 ]
Server Status              [ SERVER SHUTDOWN ]
Log File Group Count       [ 1 ]
Log File Group             [ 0 ]
End LSN                    [ 0, 20345,470341 ]
ResetLog LSN               [ 4294967295, 4294967295, 4294967295 ]
Last Created Logfile Num   [ 20350 ]
Delete Logfile(s) Range   [ 20344 ~ 20333 ]
Update And Flush Count     [ 316 ]
New Tablespace ID          [ 8 ]
```

Check whether the log files ranging from logfile18320 to logfile20344 exist in the directory specified using the ARCHIVE_DIR property. If not, copy the archive log files from a backup storage device to the directory specified using the ARCHIVE_DIR property.

All log files after logfile20345 are online log files that exist in the directory specified using the LOG_DIR property. That is, the log files ranging from logfile18320 to logfile20345 are required in order to completely recover the lost abc.dbf file.

To avoid the waste of disk space that would occur if duplicate log files existed in the directories specified using both the ARCHIVE_DIR and LOG_DIR properties, ALTIBASE HDB directly reads the log files from the directory specified using the ARCHIVE_DIR property.

Create the lost abc.dbf file by executing the following command during the control startup phase.

```
iSQL(sysdba) > ALTER DATABASE CREATE DATAFILE 'abc.dbf';
```

Perform complete media recovery by executing the following command during the control phase.

```
iSQL(sysdba) > ALTER DATABASE RECOVER DATABASE;
```

10.3.7 Media Recovery Example 2

Suppose that the database is operating in archivelog mode and that the data files in the USER_DISK_TBS tablespace were backed up three days ago.

All of the data files in the USER_DISK_TBS tablespaces were lost this morning.

10.3.7.1 Backup Procedure

The backup from three days ago was performed as follows:

```
iSQL(sysdba) > ALTER DATABASE BACKUP TABLESPACE user_disk_tbs TO '/backup1';
iSQL(sysdba) > ALTER SYSTEM SWITCH LOGFILE;
% ls /backup1
USER_DISK_TBS01.dbf USER_DISK_TBS02.dbf USER_DISK_TBS03.dbf
```

10.3.7.2 Recovery Procedure

After determining which archive log files are required for complete recovery, these files are copied into the archive directory. The method of determining which archive log files are required is to check the headers of the data files that are to be recovered. The header information can be viewed using the dumpddf utility, as seen below:

```
% dumpddf -m -f USER_DISK_TBS01.dbf
[BEGIN DATABASE FILE HEADER]

Binary DB Version      [ 5.4.1 ]
Redo LSN                [ 0, 4, 2257550 ]
Create LSN              [ 0, 0, 657403 ]

[END DATABASE FILE HEADER]
```

The above results indicate that the logfile4 archive log file and subsequent files are required in order to restore the database using the backup data files.

Copy the data file backups in the backup_dir directory to the original location of the data files for the

10.3 Backup and Recovery Examples

USER_DISK_TBS tablespace, which is the \$ALTIBASE_HOME/dbs/ directory.

```
% cp /backup_dir/*.dbf $ALTIBASE_HOME/dbs;
```

Perform complete media recovery during the control phase.

```
iSQL(sysdba) > ALTER DATABASE RECOVER DATABASE;
```

10.3.8 Media Recovery Example 3

Suppose that the database is operating in archivelog mode and that the data files in the USER_DISK_TBS tablespace were backed up seven days ago.

This afternoon, the /disk1 file system, which contained the data files for the USER_DISK_TBS tablespace, was corrupted, but the /disk2 file system remains intact.

In this case, because the /disk1 partition is corrupt, the backup data files are moved to the healthy partition, which is /disk2, in order to perform media recovery.

10.3.8.1 Backup Procedure

The backup from seven days ago was performed as follows:

```
iSQL(sysdba) > ALTER DATABASE BACKUP TABLESPACE user_disk_tbs TO '/
backup_dir';
iSQL(sysdba) > ALTER SYSTEM SWITCH LOGFILE;

% ls /backup_dir
USER_DISK_TBS01.dbf USER_DISK_TBS02.dbf
```

10.3.8.2 Recovery Procedure

After checking which archive log files are required to perform complete recovery, these files are copied to the archive directory.

Copy the backups of the files for the USER_DISK_TBS tablespace, which are in the backup_dir directory, to the /disk2 filesystem.

```
% cp /backup_dir/*.dbf /disk2/dbs;
```

Change the file path for the data files for the USER_DISK_TBS tablespace during the control phase.

```
iSQL(sysdba) > ALTER DATABASE RENAME DATAFILE
'/disk1/dbs/USER_DISK_TBS01.dbf' TO
'/disk2/dbs/USER_DISK_TBS01.dbf';
iSQL(sysdba) > ALTER DATABASE RENAME DATAFILE
'/disk1/dbs/USER_DISK_TBS02.dbf' TO
'/disk2/dbs/USER_DISK_TBS02.dbf';
```

Note: The alter tablespace command can also be used to perform this task.

```
iSQL(sysdba) > ALTER TABLESPACE user_disk_tbs RENAME DATAFILE
'/disk1/dbs/USER_DISK_TBS02.dbf' TO
'/disk2/dbs/USER_DISK_TBS02.dbf';
```

Check the V\$DATAFILE performance view to verify that the data file path was correctly changed.


```
iSQL(sysdba) > SELECT * FROM V$DATAFILES;
```

Complete media recovery can be performed by executing the following command during the control phase.

```
iSQL(sysdba) > ALTER DATABASE RECOVER DATABASE;
```

10.3.9 Media Recovery Example 4

Suppose that the database is operating in archivelog mode and that a user accidentally dropped the summary table.

- A. The time of completion of the most recent full online backup: September 18, 2007, 12:00
- B. The time when the table was dropped: September 18, 2007, 15:00
- C. The current time: September 18, 2007, 19:00

In order to recover the summary table, it will be necessary to perform incomplete media recovery to restore the database to its state at 14:30 on September 18, which is 3.5 hours before the current time.

10.3.9.1 Backup Procedure

When the last backup was performed, the entire database was backed up, as shown below:

```
iSQL(sysdba) > ALTER DATABASE BACKUP DATABASE TO '/backup_dir';
iSQL(sysdba) > ALTER SYSTEM SWITCH LOGFILE;
```

10.3.9.2 Recovery Procedure

1. Copy the backups of the data files to their original location.

```
% cp /backup_dir/*.dbf $ALTIBASE_HOME/dbs;
```

2. Because the ping-pong checkpointing technique is used with memory tablespaces, when the database is backed up, only the stable data files for a memory tablespace are copied.

Ex.) The backups of the data files for a memory tablespace are as follows:

```
SYS_TBS_MEM_DIC-1-0,
SYS_TBS_MEM_DIC-1-1,
SYS_TBS_MEM_DIC-1-2
SYS_TBS_MEM_DATA-0-0,
SYS_TBS_MEM_DATA-0-1,
SYS_TBS_MEM_DATA-0-2
```

3. Since the backup of the memory tablespace contains valid data files, they can be copied without checking their validity.

```
% cp SYS_TBS_MEM_DIC-1-0 $ALTIBASE_HOME/dbs
% cp SYS_TBS_MEM_DIC-1-1 $ALTIBASE_HOME/dbs
% cp SYS_TBS_MEM_DIC-1-2 $ALTIBASE_HOME/dbs

% cp SYS_TBS_MEM_DATA-0-0 $ALTIBASE_HOME/dbs
% cp SYS_TBS_MEM_DATA-0-1 $ALTIBASE_HOME/dbs
% cp SYS_TBS_MEM_DATA-0-2 $ALTIBASE_HOME/dbs
```

10.3 Backup and Recovery Examples

4. Incomplete recovery uses the backups of the log anchor files. Copy the log anchor files from the backup storage device.

```
% cp /backup_dir/loganchor* $ALTIBASE_HOME/logs
```

5. Check which archive log files are required for incomplete recovery, as shown below:

```
iSQL(sysdba)> SELECT LAST_DELETED_LOGFILE FROM V$LFG;  
LAST_DELETED_LOGFILE  
-----  
15021
```

6. Check the altibase_sm.log file, which is in the \$ALTIBASE_HOME/trc directory, to verify that the backup-related log files were forcibly archived at the end of the backup.

```
[2007/09/18 13:59:59] [Thread-6] [Level-9]  
Waiting logfile15341 to archive
```

7. According to the results above, copy all files ranging from logfile15021 to logfile15341, which is the last archive log file in which logging was performed at the end of backup, from the directory specified using the ARCHIVE_DIR property (or a backup device) to the directory specified using the LOG_DIR property.

Unlike complete recovery, when performing incomplete recovery, the duplication of log files is inevitable.

8. Because SYS_TBS_DISK_TEMP was not backed up, create the corresponding file.

```
iSQL(sysdba)> ALTER DATABASE CREATE DATAFILE 'temp001.dbf';
```

9. Perform incomplete media recovery as shown below.

```
iSQL(sysdba)> ALTER DATABASE RECOVER DATABASE UNTIL TIME '2007-09-  
18:14:30:00';
```

10. Use stable versions of all data files for the complete backup of the memory tablespace to create the copies of these files.

```
% cp SYS_TBS_MEM_DIC-1-0 $ALTIBASE_HOME/dbs/SYS_TBS_MEM_DIC-0-0;  
% cp SYS_TBS_MEM_DIC-1-1 $ALTIBASE_HOME/dbs/SYS_TBS_MEM_DIC-0-1;  
% cp SYS_TBS_MEM_DIC-1-2 $ALTIBASE_HOME/dbs/SYS_TBS_MEM_DIC-0-2;  
  
% cp SYS_TBS_MEM_DATA-0-0 $ALTIBASE_HOME/dbs/SYS_TBS_MEM_DATA-1-0;  
% cp SYS_TBS_MEM_DATA-0-1 $ALTIBASE_HOME/dbs/SYS_TBS_MEM_DATA-1-1;  
% cp SYS_TBS_MEM_DATA-0-2 $ALTIBASE_HOME/dbs/SYS_TBS_MEM_DATA-1-2;
```

11. Because incomplete media recovery has been performed, the resetlogs option must be used when proceeding to the meta startup phase.

```
iSQL(sysdba)> ALTER DATABASE MYDB META RESETLOGS;
```

12. Once resetlogs has been performed, perform a complete database backup.

```
iSQL(sysdba)> ALTER DATABASE BACKUP DATABASE TO 'backup_dir';
```

10.3.10 Media Recovery Example 5

Suppose that the database is running in archivelog mode, that log files numbered between #499 and 600 exist, and that log file #570 has been lost.

The database will be recovered up to log file #569. The log files that are related to all backup versions of data files are those between #499 and #550.

10.3.10.1 Recovery Procedure

Copy the data files and log anchor files required to perform incomplete recovery.

Check which archive log files are required to perform incomplete recovery.

Apply the redo records of the log files from #499 to #569 to the database, but not the redo records of log file #570 and subsequent files.

```
iSQL(sysdba) > ALTER DATABASE RECOVER DATABASE UNTIL CANCEL;
```

Since incomplete media recovery has been performed, the resetlogs option must be used when proceeding to the meta startup phase.

```
iSQL(sysdba) > ALTER DATABASE MYDB META RESETLOGS;
```

Once resetlogs is complete, perform a complete database backup.

```
iSQL(sysdba) > ALTER DATABASE BACKUP DATABASE TO '/backup_dir';
```

10.3.11 Media Recovery Example 6

In some cases, media recovery can be performed even if the database is running in noarchivelog mode. One such case is the case where data files in temporary tablespace are lost. This is because there are no changes that must be reapplied to the database during recovery of the temporary tablespace.

10.3.11.1 Recovery Procedure

In the control startup phase, create a new file called temp001.dbf to replace the lost file for the SYS_TBS_DISK_TEMP tablespace.

```
iSQL(sysdba) > ALTER DATABASE CREATE DATAFILE 'temp001.dbf';
```

Start the server.

```
iSQL(sysdba) > ALTER DATABASE dbname SERVICE;
```

10.3.12 Media Recovery Example 7

Suppose that the database is running in archivelog mode and that the data files that belong to the SYS_TBS_MEM_DIC dictionary tablespace have been lost.

10.3.12.1 Backup Procedure

Perform tablespace-level backup of the SYS_TBS_MEM_DIC tablespace.

```
iSQL(sysdba) > ALTER DATABASE BACKUP TABLESPACE SYS_TBS_MEM_DIC TO '/backup_dir';
```

10.3 Backup and Recovery Examples

10.3.12.2 Recovery Procedure

After checking which archive log files are required in order to perform complete recovery, copy these files to the archive directory. The method of determining which archive log files are required is to check the headers of the checkpoint image files that are to be recovered. The header information can be viewed using the dumpci utility, which will yield results similar to those shown below.

```
% dumpci -m -f SYS_TBS_MEM_DIC-0-0
[BEGIN CHECKPOINT IMAGE HEADER]

Binary DB Version          [ 5.4.1 ]
LogFileGroup Count        [ 1 ]
LogFileGroup-0 Redo LSN    [ 0, 4, 2257550 ]
LogFileGroup-0 Create LSN  [ 0, 0, 657403 ]

[END CHECKPOINT IMAGE HEADER]
```

The above results indicate that the logfile4 archive log file and subsequent files are required in order to restore the database using the backup data files. Only the stable data files in a memory tablespace are backed up. Supposing that the backup file is SYS_TBS_MEM_DIC-0-0, copy the backup data file as shown below:

```
% cp /backup_dir/SYS_TBS_MEM_DIC-0-0 $ALTIBASE_HOME/dbs;
```

In the results of \$ALTIBASE_HOME/bin/dumpla loganchor0, check the “Stable Checkpoint Image Num” tablespace attribute for the tablespace named SYS_TBS_MEM_DIC.

```
% dumpla loganchor0
[ TABLESPACE ATTRIBUTE ]
Tablespace ID              [ 0 ]
Tablespace Name            [ SYS_TBS_MEM_DIC ]
New Database File ID       [ 0 ]
Tablespace Status          [ ONLINE ]
TableSpace Type            [ 0 ]
Checkpoint Path Count      [ 0 ]
Autoextend Mode            [ Autoextend ]
Shared Memory Key          [ 0 ]
Stable Checkpoint Image Num. [ 1 ]
Init Size                  [ 4 MBytes ( 129 Pages ) ]
Next Size                  [ 4 MBytes ( 128 Pages ) ]
Maximum Size               [ 134217727 MBytes ( 4294967295 Pages ) ]
Split File Size            [ 1024 MBytes ( 32768 Pages ) ]

[ MEMORY CHECKPOINT PATH ATTRIBUTE ]
Tablespace ID              [ 0 ]
Checkpoint Path            [ /home/altibase_home/dbs ]
[ MEMORY CHECKPOINT IMAGE ATTRIBUTE ]
Tablespace ID              [ 0 ]
File Number                [ 0 ]
Create LSN                 [ 0, 0, 2028 ]
Create On Disk (PingPong 0) [ Created ]
Create On Disk (PingPong 1) [ Created ]
```

Because the backup datafile number is [0] and the current stable datafile number is [1], copy the tablespace as shown below.

```
% cd $ALTIBASE_HOME/dbs
% cp SYS_TBS_MEM_DIC-0-0 SYS_TBS_MEM_DIC-1-0
```

Perform media recovery during the control phase.

```
iSQL(sysdba) > ALTER DATABASE RECOVER DATABASE;
```

Create a copy of the data file generated by the above process in the restored memory tablespace.

```
% cp SYS_TBS_MEM_DIC-1-0 $ALTIBASE_HOME/dbs/SYS_TBS_MEM_DIC-0-0;
```

Now that media recovery is complete, restart recovery will occur automatically.

```
iSQL(sysdba) > ALTER DATABASE dbname SERVICE;
```

10.3.13 Media Recovery Example 8

Suppose that the database is running in archivelog mode and that a user deleted the USER_DISK_TBS tablespace by mistake. The tablespace was deleted at 22:30 on April 6, 2007. The database will be restored to its state as of 10 minutes before that time, when the tablespace existed.

10.3.13.1 Backup Procedure

When the last backup was performed, the entire database was backed up, as shown below:

```
iSQL(sysdba) > ALTER DATABASE BACKUP DATABASE TO '/backup_dir';
```

10.3.13.2 Recovery Procedure

Copy the backups of the data files and log anchor files that are required for incomplete recovery.

Copy the data files for all disk tablespaces in the backup of the database to the original location of the data files.

```
% cp /backup_dir/*.dbf $ALTIBASE_HOME/dbs
% cp /backup_dir/SYS_TBS_* $ALTIBASE_HOME/dbs
```

Check which archive log files are required to perform incomplete recovery.

Incomplete recovery uses the backups of the log anchor files. Copy the log anchor files from the backup storage device.

```
% cp /backup_dir/loganchor* $ALTIBASE_HOME/logs;
```

Because the SYS_TBS_DISK_TEMP tablespace is not backed up, create a new data file for it.

```
iSQL(sysdba) > ALTER DATABASE CREATE DATAFILE 'temp001.dbf'
```

Perform incomplete media recovery.

```
iSQL(sysdba) > ALTER DATABASE RECOVER DATABASE UNTIL TIME '2007-04-
06:22:20:00';
```

Because incomplete media recovery has been performed, logs must be reset when proceeding to the meta startup phase.

```
iSQL(sysdba) > ALTER DATABASE mydb META RESETLOGS;
```

Start the server.

```
iSQL(sysdba) > alter database mydb service;
```

10.3 Backup and Recovery Examples

Because the logs have been reset, it is recommended that the entire database be backed up.

```
iSQL(sysdba) > ALTER DATABASE BACKUP DATABASE TO '/backup_dir';
```

11 SQL Tuning

This chapter describes how to perform SQL tuning to improve query performance.

This chapter comprises the following sections:

- [Introduction to SQL Tuning](#)
- [Query Optimization Procedures](#)
- [Analyzing Execution Plans](#)
- [Using Optimizer Hints](#)

11.1 Introduction to SQL Tuning

The term “SQL tuning” refers to a variety of tasks that are undertaken to maximize the performance of SQL statements, including the following:

- rewriting SQL statements,
- redesigning database schemas,
- adjusting property values, and
- adjusting operating system kernel parameters.

Two of these SQL tuning methods, namely rewriting SQL statements and redesigning database schemas, require a deep understanding of how queries are processed in ALTIBASE HDB, because query processing and the associated limitations have such a strong effect on query performance. Additionally, because our true hybrid database uses both memory and disk tables, it is also important to understand how the characteristics of each of these two types of storage media affect query processing.

This section will briefly outline the issues related to query performance and the basic concepts that are required in order to undertake SQL tuning tasks.

- [Performance Issues](#)
- [SQL Tuning Basics](#)
- [Memory Tables and Disk Tables](#)
- [Query Processing Overview](#)

11.1.1 Performance Issues

11.1.1.1 The Execution Plan Tree

When a client requests that an SQL statement be executed, the Query Processor checks the syntax and validity of the statement and then optimizes the statement to create an execution plan tree. If host variables exist, ALTIBASE HDB will access the database and retrieve records as directed by the execution plan after these variables are bound. Because most of the time spent handling SQL statement execution requests is spent actually executing the statements themselves, the structure of the execution plan tree will have a significant impact on the efficiency with which complex SQL statements are executed.

11.1.1.2 SQL Plan Cache

In ALTIBASE HDB, SQL statement execution plans can be shared between sessions. This reduces the expense associated with creating an execution plan every time an SQL statement is executed, which means that a performance improvement can be expected. To use this functionality, the value of the SQL_PLAN_CACHE_SIZE property, which is related to the SQL Plan Cache, must be set appropriately.

11.1.1.3 Database Schema and Data Volume

When deciding whether to deploy memory tables or disk tables, it is advisable to consider the characteristics of client applications and the efficiency with which resources are utilized. Generally, frequently used data should be stored in memory tables, whereas less frequently used data should be stored in disk tables.

In consideration of the types of SQL statements to be used by applications, the structure of the table schema and index should be designed carefully so that the expense incurred by executing each SQL statement, which is dependent in part on the number of times a table is accessed, the number of records that are accessed, and the number of disk pages that are accessed, can be minimized.

In addition, it can be expensive to compare column values when tables contain large amounts of data. Therefore, it is important to select suitable data types in order to minimize the expense associated with data conversion and comparison.

It is thus necessary to implement SQL statements so as to ensure that as few records as possible are accessed, reduce the number of direct comparisons, and choose suitable data types for columns and values that will be compared so as to avoid data conversion when performing comparison operations.

11.1.1.4 Application Logic (Data Access Order)

If several clients have established sessions with an Altibase database, the order in which each client accesses tables can affect system performance. In the case where a single transaction uses several DML statements to access several tables, if an application was poorly designed, for example if it accesses tables in an inappropriate order, then the performance of all applications can suffer due to the time spent waiting to acquire locks. Thus, it is important to pay attention to the logic of client applications.

11.1.1.5 System Resource Usage

System resources that affect the performance with which a single SQL statement is processed include the amount of available memory, the size of memory buffers, disk performance and CPU performance.

The performance of search queries is, of course, not affected by disk performance when only memory tables are used. However, when querying disk tables, query performance is affected both by disk performance and by the amount of available space in a memory buffer. Accordingly, a consistent response time can be expected when querying memory tables, but when querying disk tables, performance can vary depending on the prevailing circumstances, such as the frequency of disk swapping, at the time point at which the statement was executed.

When processing queries that contain ORDER BY or GROUP BY clauses or the like, the query processor uses sorting or hashing methods, and thus memory temporary tables or disk temporary tables are used to store the intermediate results of such processing.

When intermediate results are stored in memory temporary tables, they are stored in an area of memory that is not available to the memory database. Therefore, in order to avoid degradation in the performance of the database, be sure that no memory swapping occurs, which can happen when memory tables increase in size and thus require additional space.

When intermediate results are stored in disk temporary tables, they might consume disk space that is occupied by the disk temporary tablespace, and might consume space in the memory buffer.

11.1 Introduction to SQL Tuning

Because performance is greatly affected by the amount of available space in the memory buffer, storing intermediate results can have a great impact on performance.

Additionally, when processing queries, many operators must be processed, which leads to increased CPU usage. Therefore, query processing performance is also greatly affected by CPU performance and by the proportion of the CPU that is allocated to processing queries.

11.1.2 SQL Tuning Basics

This section briefly illustrates the general tuning methodology for ALTIBASE HDB users.

11.1.2.1 How to Measure SQL Performance

When developing an application using one of the ALTIBASE HDB APIs, such as the APRE, ODBC, or JDBC API, the amount of time that has actually elapsed while processing the query can be determined using a function written for that purpose within the application. The elapsed query time can also be determined using the following iSQL command:

```
SET TIMING ON;
```

When the same query is repeatedly executed on a memory table using iSQL, performance will be quite consistent; however, in the case of a disk table, performance can be expected to improve with repeated executions, because fewer buffer replacements occur as the query is repeated. This means that if many queries are simultaneously executed on a disk table, consistent performance cannot be guaranteed due to variation associated with buffer replacement.

However, using iSQL to tune an SQL statement usually results in a comparable performance improvement when the SQL statement is implemented within an application.

11.1.2.2 Analyzing Execution Plan Trees

The execution plan trees for SQL DML statements, such as SELECT, INSERT, UPDATE or DELETE statements, can only be checked using iSQL. Because DELETE, UPDATE and MOVE statements are optimized in the same way as SELECT statements, execution plan trees are internally created for them in a similar manner.

In the case of INSERT statements, the execution plan tree for the SELECT portion of an INSERT INTO SELECT statement can be checked.

An execution plan tree typically needs to be checked against the following criteria. For more detailed information, please refer to the next section.

- Is the access method efficient?
- Are the joins appropriate, and suitably ordered?
- Are the data types and operations suitable?
- Is any unnecessary hashing or sorting performed?

11.1.2.3 Query Tuning

Once the factors that degrade performance have been identified using the execution plan tree, per-

formance can be improved by taking appropriate measures. Query tuning methods will be discussed in detail when the query optimization process and the concept of the query executor are explained.

11.1.3 Memory Tables and Disk Tables

ALTIBASE HDB is the first true HDB (Hybrid Database), and as such supports both memory tables and disk tables. Therefore, understanding the differences in how queries are processed for memory tables and disk tables will be of great help when tuning queries.

The fundamental differences between query processors for memory tables and disk tables are as follows:

| Query Processor | Item | Memory Tables | Disk Tables |
|-----------------|--|-------------------------|-----------------------|
| Executor | Object identifier | Pointer | OID (RID) |
| | Buffer management | N/A | Limited buffer |
| | Join methods | One-pass algorithms | Multi-pass algorithms |
| Optimizer | Main cost | CPU | Disk |
| | Index selection | Minimize record access | Minimize disk I/O |
| | Cost factor | $T(R)$, $V(R.a)$, etc | $+ B(R)$, M |
| Reference | $T(R)$: Number of records in Table R , $V(R.a)$: Cardinality of values in column $R.a$ $B(R)$: Number of disk pages for Table R , M : Number of available memory buffers | | |

Query processing can be broadly categorized as being handled either by the optimizer or the executor. The optimizer creates an execution plan tree by calculating the cost of execution of a statement, whereas the executor actually executes the statement according to each of the nodes of the execution plan tree.

The ALTIBASE HDB query executor and optimizer were designed not only to account for the differences in the various kinds of storage media, but also to allow execution plans to be created without distinguishing between the types of storage media while still taking those characteristics into account when processing queries. In other words, even when processing a statement that queries both memory tables and disk tables, ALTIBASE HDB can take the characteristics of the respective kinds of storage media into account and still handle them using the same process.

11.1.3.1 The Executor

As seen in the above table, the operation of the executor varies conceptually depending on the storage medium in which the table is saved.

The so-called “object identifier”, which identifies records, is implemented as a pointer for memory tables, whereas in the case of disk tables it is an identifier that can be converted to a particular disk address, similar to a resource identifier (RID). This difference means that unlike memory tables, in which the records can be accessed directly, disk tables require addresses to be converted in order to

11.1 Introduction to SQL Tuning

access records.

Buffers are not used when processing queries on memory tables, and thus no special consideration of buffers is required. In contrast, queries on disk tables are processed within memory buffers, which are limited, and thus if a record that is being sought cannot be found in a buffer (known as “buffer miss”), buffer replacement occurs, which causes disk I/O.

Join methods are categorized into the following:

- nested loop join
- hash-based join
- sort-based join
- merge join

For each join method, intermediate results are stored if necessary. These results are processed using either a one-pass or multi-pass algorithm, depending on whether the results can all be loaded into the limited amount of memory at one time. A one-pass algorithm is used when all of the intermediate results can be loaded into the available memory, whereas a multi-pass algorithm is used in order to minimize buffer replacement in the case where not all of the intermediate results can be loaded into the available memory. A one-pass algorithm is always used with memory tables, as no memory buffer is used and thus there is no buffer-related limit, whereas either a one-pass or two-pass algorithm is used for disk tables in consideration of the limited amount of buffer memory.

As outlined above, the method in which the query executor processes queries differs depending on the kind of storage medium, and great differences in performance are exhibited as a result.

11.1.3.2 The Optimizer

As was also seen in the preceding table, the query optimizer also differs conceptually and in how it operates depending on the storage medium in which the table is saved.

The optimizer calculates the expense associated with queries, and thus when querying memory tables, it creates an execution plan that minimizes CPU usage. In contrast, when querying disk tables, the optimizer creates an execution plan that minimizes disk I/O. That is, the optimizer creates an execution plan that minimizes the use of the resource that has the greatest effect on query performance.

When selecting the access method, for memory tables, the optimizer selects the access method that minimizes the number of records that are read, whereas for disk tables, the optimizer selects the access method that can minimize disk I/O. This difference is due to the fact that in most cases using an index to query a memory table guarantees better performance than scanning the entire table, while in the case of disk tables, depending on how the data are distributed, scanning the entire table can in some cases result in less disk I/O than when using an index, thus yielding better performance.

The optimizer uses various statistical data as factors when calculating the expense of executing a query. When calculating the expense for a query of a memory table, statistical data such as the number of records in the table $[T(R)]$, the number of different values in a column $[V(R.a)]$, and the maximum and minimum values in a column are used. For a disk table, in addition to these, other statistical data, such as the number of disk pages being used by the table $[B(R)]$ and the number of available memory buffer pages $[M]$, are also included in the calculation.

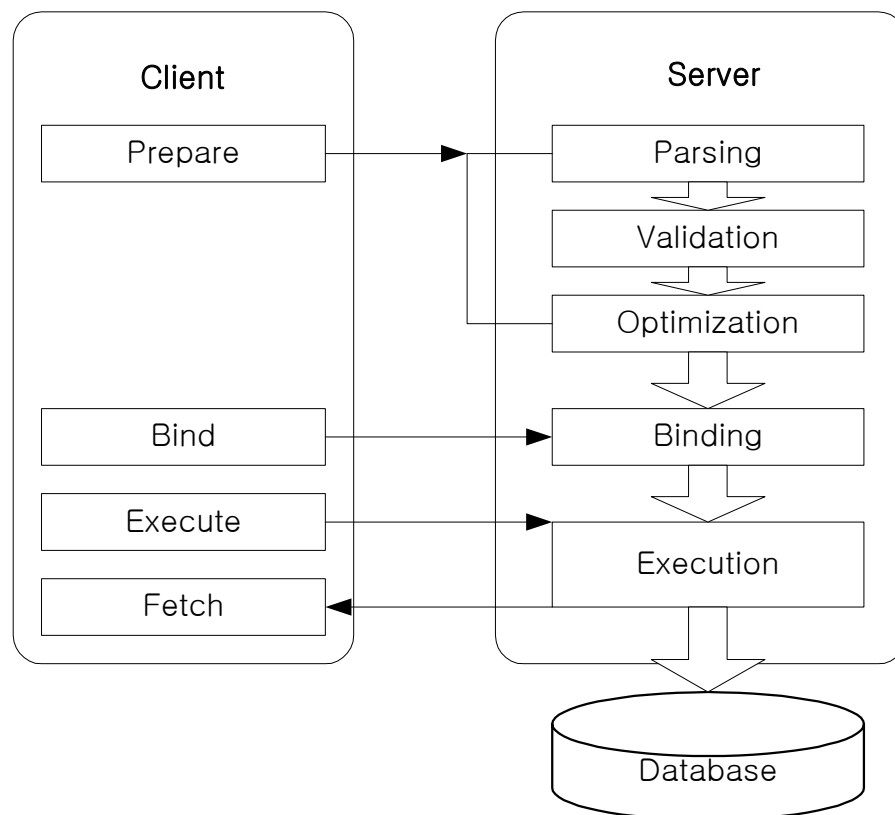
11.1.4 Query Processing Overview

It is impossible to suggest cardinal SQL tuning rules that can be applied to consistently exhibit optimum performance in all applications. SQL tuning methods are many and various, and differ according to the design of the database and the business logic of the client application. Therefore, this section will describe how an SQL statement is processed and what factors affect system performance, and suggest a general query tuning methodology.

The Query Processor is the DBMS module that is responsible for processing SQL statements. When a user requests the execution of an SQL statement, the Query Processor checks the validity of the statement, decides how and in what order to access the database for optimum performance, searches for records that satisfy the given conditions, performs operations on the records as required, and finally returns the results of processing to the client.

The following figure roughly illustrates the tasks involved in processing queries.

Figure 11-1 Query Processing Order



The tasks performed at each step are as follows:

1. **Parsing:** The syntax of the SQL statement is checked, and a parse tree containing analytic information about the statement is created.
2. **Validation Check:** A semantic check is performed, and the parse tree is expanded using data retrieved from meta tables.
3. **Optimization:** An optimal execution plan is created based on various statistics, and access

11.1 Introduction to SQL Tuning

costs are estimated on the basis of the parse tree.

4. Binding: Host variables are bound to the created execution plan.
5. Execution: The SQL statement is executed according to the execution plan tree.

Understanding the query optimization and query execution process in ALTIBASE HDB is very important in order to tune queries effectively. How to tune a query will be described in detail in the following section.

11.2 Query Optimization Procedures

The query optimization process is a process of creating execution plans that can be used for a given SQL statement and selecting the most efficient execution plan by evaluating the expense that would be incurred by executing each plan. This process has the greatest influence on query performance. The more complicated a query is, the more its performance depends on the precision of the optimization process.

This section contains the following topics:

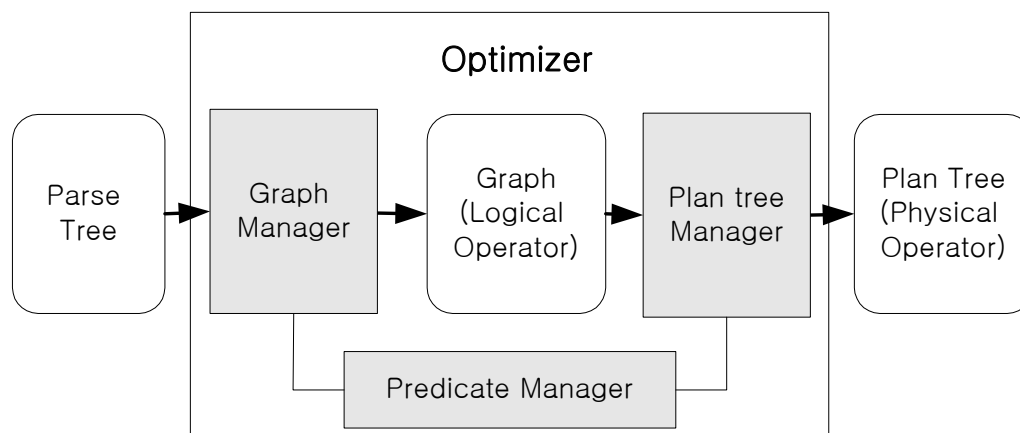
- [Query Optimization](#)
- [Query Normalization](#)
- [Condition Clauses](#)
- [Determining the Access Method](#)
- [Determining the Join Order](#)
- [Determining the Join Method](#)
- [Optimization of Group Operations](#)
- [Optimization of DISTINCT Clause](#)
- [Optimization of Set Operations](#)
- [Optimizing ORDER BY Clauses](#)
- [Optimizing and Partitioned Tables](#)
- [Optimization of the Projection](#)
- [Subquery Optimization](#)

11.2.1 Query Optimization

11.2.1.1 Optimizer Structure

Before a given SQL statement is optimized, it is first parsed and validated, and then a parsing tree for the statement is created. The optimizer undertakes various cost evaluation processes to finally suggest an efficient execution plan tree for the statement.

The structure of the optimizer is as shown below.

Figure 11-2 The Procedure of Query Processing

The optimizer broadly comprises the graph manager, the execution plan manager, and the predicate manager. The role of each manager is described below:

- The Graph Manager: this uses a given parsing tree to create a graph comprising logical operators, and determines the optimization method by calculating the cost of executing the statement according to the graph.
- The Execution Plan Manager: this uses the created graph to create an execution plan tree for the given optimization method.
- The Predicate Manager: this provides information required by the graph manager and the execution plan manager in order to determine and create an execution plan tree.

11.2.1.2 Type of Graph Elements (Logical Operators)

There are 12 types of graph elements. The functions of the main graph elements are described below:

Table 11-1 Types of Graph

| Classification | Graph Element | Function | Related SQL Clauses |
|----------------|---------------|---------------------------------------|---------------------|
| Critical path | Selection | A method for accessing a single table | FROM, WHERE |
| | Join | Joining method and join order | FROM, WHERE |

| Classification | Graph Element | Function | Related SQL Clauses |
|-------------------|---------------|--|---|
| Non critical path | Grouping | Methods of performing grouping and aggregation | GROUP BY, HAVING, aggregation operation |
| | Distinction | A method of throwing out duplicates of records | DISTINCT |
| | Set | Methods of performing set operations | UNION, UNION ALL, INTERSECT, MINUS |
| | Sorting | A method of performing a sort operation | ORDER BY |
| | Projection | A method of performing a projection operation | SELECT |

From the point of view of the optimizer, the FROM and WHERE clauses, which are known as the “critical path”, are most critical in determining query performance. Other clauses, such as the GROUP BY and ORDER BY clauses, are called the “non-critical path”.

The optimizer uses the selection and join graph elements to determine how to execute the critical path, and uses the grouping, distinction, set, sorting, and projection graph elements to determine how to execute the non-critical path.

11.2.1.3 Real-time Statistics

The optimizer uses real-time statistics to calculate the cost of the various execution plans. In addition to basic statistics, the optimizer comes up with new statistical information during the course of optimization and provides various estimates on the basis of cost calculations.

The ALTIBASE HDB optimizer uses the following basic real-time statistics:

Table 11-2 Real-time Statistics

| Symbols | Description |
|----------|--|
| T(R) | Number of Records in Table R |
| V(I) | Cardinality of values in Index I, Number of different index key values |
| V(R.a) | Cardinality of values in Column R.a, Number of different values |
| MIN(R.a) | Minimum value in Column R.a |
| MAX(R.a) | Maximum value in Column R.a |
| B(R) | Number of disk pages in Table R |
| M | Number of pages in the memory buffer |

The above real-time statistical information is updated without any additional load whenever data are modified, and thus relatively accurate statistics are maintained by the database server without any user intervention. Because it uses these real-time statistical data, the optimizer can evaluate the

11.2 Query Optimization Procedures

cost of various execution plans and create an efficient execution plan.

However, these statistical data alone are not sufficient to optimize all queries. There are environments in which the optimizer cannot always create an efficient execution plan. Therefore, users are required to perform some tasks in order to improve the performance of particular queries, and thus it is very important that users conceptually understand query optimization in ALTIBASE HDB.

11.2.1.4 The Query Optimization Process

In ALTIBASE HDB, the optimizer performs so-called “bottom-up” optimization, meaning that it first determines the optimization method for the critical path, which has the greatest influence on the performance of the optimizer, and then processes the non-critical path.

The optimization process for determining the execution method for the critical path is as follows:

1. Normalization- A user-defined WHERE clause is converted into two normalized forms: CNF (Conjunctive Normal Form) and DNF (Disjunctive Normal Form). The condition clauses are used to calculate the cost of the critical path for each normalized form. The costs are compared to determine the execution plan for the critical path.
2. Each normalized form (CNF and DNF) is processed according to the following procedure:
 - a. Classification of Condition Clauses - Each normalized condition clause is categorized according to whether it pertains to a single table or to multiple tables.
 - b. Determining the Access Method- the access method is set by preferentially choosing the least costly index for all of the condition clauses that pertain to each table. The access method selected in this way might be changed during the join process.
 - c. Determining the Join Order - The join order is the factor that has the greatest influence on query performance when processing complex queries, and is determined using a greedy algorithm.
 - d. Determining the Join Method: After the join order is determined, the optimal join method is chosen for each join order.

After the critical path is optimized, the optimization process for determining the execution method for the non-critical path is conducted according to the following procedure:

1. Determining the Method of Executing Group/Aggregate Operations - If the statement contains a GROUP BY or HAVING clause or an aggregate operation, the execution method for processing it is determined.
2. Determining the Method of Executing a DISTINCT Clause - If the statement contains a DISTINCT clause, the execution method for processing it is determined.
3. Determining the Method of Executing a SET Clause - If the statement contains a UNION, UNION ALL, INTERSECT, or MINUS clause, the execution method for processing it is determined.
4. Determining the Method of Executing an ORDER BY Clause - If the statement contains an ORDER BY clause, the execution method for processing it is determined.
5. Determining the Method of Executing Projection - The method for performing projection for the columns etc. referred to in a SELECT statement is determined. When a subquery contains a SELECT statement, various optimization methods can be used.

11.2.2 Query Normalization

11.2.2.1 Concepts of Normalization

User-defined WHERE condition clauses can be written in a wide variety of ways. In order to process a wide variety of conditions effectively and consistently, the optimizer rewrites user-defined WHERE clauses in a standardized form. This process is called normalization.

Normalization is the process of extending and transforming condition clauses using logical expressions such as AND, OR, and NOT. When AND operators are handled as the highest-level expression, this is so-called “Conjunctive Normal Form (CNF)”; whereas in so-called “Disjunctive Normal Form (DNF)”, OR operators are handled as the highest-level expression.

In CNF normalization, the structure of a given condition clause is rearranged so that any AND operators are taken as the highest-level operators and any OR operators are taken as lower level operators. The following example shows the structure that results when a condition clause is altered using CNF normalization:

- WHERE (i1 = 1 AND i2 = 1) OR i3 = 1
CNF: (i1 = 1 OR i3 = 1) AND (i2 = 1 OR i3 = 1)

In DNF normalization, the structure of a given condition clause is rearranged so that any OR operators are taken as higher-level operators and AND operators are taken as lower level operators. The following example shows the structure that results when a condition clause is altered using DNF normalization:

- WHERE (i1 = 1 OR i2 = 1) AND i3 = 1
DNF: (i1 = 1 AND i3 = 1) OR (i2 = 1 AND i3 = 1)

That is, the optimizer selects the type of normalization to use for a given condition clause by comparing the cost of executing the clause if it is CNF normalized with the cost of executing the clause if it is DNF normalized.

The optimizer usually chooses CNF-based execution plans, and only chooses DNF-based execution plans in certain cases. Consider for example the following query:

- SELECT * FROM T1 WHERE i1 = 1 OR i2 = 1;
CNF Normalization: AND (i1 = 1 OR i2 = 1)
DNF Normalization: (i1 = 1 AND) OR (i2 = 1 AND)

For the condition clause shown above, if table T1 has no index, or if it has an index but the index is defined on the basis of only one column, the condition clause will be processed in CNF. This is because it is most efficient to process the query by scanning all of table T1.

However, if an index is defined for each of the i1 and i2 columns, it will be most efficient to process the condition clause in DNF so as to obtain the results of (i1 = 1) and (i2 = 1) separately and then combine them.

As described above, the type of normalization that is chosen can differ, even for the same condition clause, depending on whether an index is present. The optimizer determines the type of normalization by comparing the cost of execution in each case.

11.2 Query Optimization Procedures

Of course, if there is no logical operator, or if there is no OR operator, the optimizer will use CNF normalization, whereas if there are one or more OR operators, the optimizer creates an execution plan by constructing both CNF and DNF forms and comparing the cost of each.

11.2.2.2 Normalization Tuning

As described above, the normalization process inevitably results in the extension of condition clauses. Therefore, writing a condition clause so that it is already constructed in a normalized form will prevent extension of the condition clause and eliminate unnecessary comparison operations.

For example, the condition clause shown below can be processed in either CNF or DNF form, but it can be modified so that it is only possible to execute it in CNF by modifying the query as shown.

- `WHERE i1 = 1 OR i1 = 2 OR i1 = 3`
CNF Normalization: `i1 IN (1, 2, 3)`

In a similar example, by changing each of the condition clauses shown below into CNF and DNF form, respectively, unnecessary normalization tasks are eliminated and redundant comparisons are prevented, thus improving performance.

- `WHERE (i1 = 1 AND i2 = 1) OR (i1 = 2 AND i2 = 2)`
CNF Normalization: `(i1, i2) IN ((1,1), (2,2))`
- `WHERE (i1 = 1 AND i2 = 1) OR (i3 = 1 AND i4 = 1)`
DNF Normalization: `(i1, i2) = (1, 1) OR (i3, i4) = (1, 1)`

Of course, when writing a condition clause in normalized form, certain information about the table, including whether indexes are defined for it, must be considered. Furthermore, even when the user writes a condition clause in normalized form, the optimizer may choose a type of normalization other than that which is desired. This can be controlled by using hints, which will be explained in detail later in the section describing hints (please refer to [Using Optimizer Hints](#)).

11.2.3 Condition Clauses

11.2.3.1 Classification of Condition Clauses

The conditional expression in a WHERE clause is changed to have a standard form through normalization, and the optimizer classifies each of the conditions so that the critical path that uses this condition clause can be processed efficiently.

Condition clauses can be broadly classified into the following three types:

- Conditions that pertain only to a single table, e.g. `T1.i1 = 1`
- Conditions that pertain to multiple tables, e.g. `T1.i1 = T2.i1` or `T1.i1 + T2.i1 > 0`
- Conditions that are independent of tables, e.g. `1 = 1` or `1 = ?`

These condition clause classifications are used to choose the access method and to determine the join order and join methods. Writing suitable condition clauses is the single most important element of SQL tuning.

11.2.3.2 Using Constant Filters

Conditions that consistently evaluate to either TRUE or FALSE regardless of the values in tables, such as, for example, $1 = 1$ or $1 <> 1$, are called “constant filters”. Because constant filters always have the same logical value, they need to be evaluated only once, and no additional processing expense associated with comparison operations is incurred upon subsequent executions of the query. That is, if the logical value of the constant filter resolves to, for example, FALSE, it need not be checked again, nor is there any need to access a table.

Although constant filters may seem meaningless, they have a wide variety of uses, as will be seen below.

In one example, constant filters can be used when it is desired only to configure the schema of a table without populating it with data. An example of this is shown below:

```
CREATE TABLE T3 AS SELECT * FROM T1, T2 WHERE 1 <> 1;
```

As seen in the above example, a constant filter can be used to create table T3, which is a table that contains columns corresponding to all of the columns in tables T1 and T2, but is not populated with any of the data that tables T1 and T2 contain.

Moreover, as seen in the example below, constant filters can also be used to limit search privileges:

```
SELECT * FROM T1, T2 WHERE T1.i1 = T2.a1 AND ? > 20;
```

As shown in the above example, a host variable corresponding to age can be set in order to prevent users who do not meet certain criteria from executing queries or to prevent unnecessary loads caused by the execution of queries that would not return any results if executed.

In addition, the optimizer can process a condition clause containing a subquery as though it were a constant filter in order to prevent repeated execution of the subquery:

```
SELECT * FROM T1
WHERE EXISTS ( SELECT * FROM T2 WHERE T2.date = SYSDATE );
```

In the above example, the EXISTS condition is a constant filter that has no relationship with the data stored in table T1.

11.2.3.3 Push Selection Method

Condition clauses that pertain to single tables, which are arrived at through the process of normalizing and classifying condition clauses, are the most important element in determining access methods. The optimizer can apply any of a variety of optimization methods depending on the form of condition clauses.

The above-described form of push selection is the most typical and general case. It consists of classifying condition clauses so that the conditions that pertain to the tables in question can be evaluated first.

Consider for example the following query:

```
SELECT * FROM T1, T2 WHERE T1.i1 = T2.a1 AND T1.i2 = 'abc';
```

Two execution plans could be created for the condition clause in the above query. Specifically, the optimizer can either first combine tables T1 and T2 and then compare the clauses, or it can first evaluate the condition clause pertaining only to table T1 in order to obtain a small result set, and then

11.2 Query Optimization Procedures

join this result set with table T2. This is the most common form of push selection: processing the conditions in a WHERE clause before executing joins.

The optimizer considers various forms of push selection on the basis of a variety of factors, including the cost of execution and mathematical consistency. The types of push selection used by the optimizer are as follows:

- Push selection on views
- Push selection on outer joins
- Push selection on joins

11.2.3.4 Push Selection on Views

With the so-called “push selection on views” method, the conditions in the WHERE clause are evaluated within the view when a user-defined view is queried.

For example, given the view and the query defined below:

```
CREATE VIEW V1(a1, a2) AS SELECT i1, i2 FROM T1 WHERE i2 > 20;  
SELECT * FROM V1 WHERE a1 = 1;
```

If the optimizer, in the course of optimizing the above query, concludes that using the push selection on views method will result in the most efficient execution plan, it determines that the condition in the WHERE clause is to be processed internally within the view in the form shown below. Expressed conceptually, it would appear as follows:

```
SELECT * FROM ( SELECT i1, i2 FROM T1 WHERE i2 > 20 AND i1 =1 ) V1;
```

In other words, if the query is changed so that it has the structure shown above, it is optimized because it uses the index for the T1.i1 column. However, the optimizer does not always decide to use this kind of push selection.

If users understand the internal structure of views, users can write suitable condition clauses so that the optimizer will choose push selection on views. In addition, users can use hints to explicitly instruct the optimizer to choose push selection on views.

However, using push selection on views does not necessarily guarantee performance equal to or better than that of the original query.

One optimization method that contrasts with the push selection on views method is the view materialization method. With this method, when a view is used repeatedly by a query, the contents of the view are temporarily stored while the query is being processed.

For example, given the view defined as seen below and the associated query:

```
CREATE VIEW V1(a1, a2) AS SELECT i1, SUM(i2) FROM T1 GROUP BY i1;  
SELECT * FROM V1 WHERE V1.a2 > ( SELECT AVG(a2) FROM V1 WHERE a1 > 10 );
```

When the view materialization method is used to process the above query, the top-level query and the subquery use the contents of view V1, which are temporarily stored. That is, the query defined for the view does not need to be executed repeatedly in order to obtain the contents of V1.

Because hinting the optimizer to use the push selection on views method to process this kind of query can actually worsen performance, discretion is advised.

11.2.3.5 Push Selection on Outer Joins

Various push selection methods are available in the case where an outer join is used in a FROM clause. What these methods have in common is that the conditions described in the WHERE clause are evaluated before the outer join is processed.

Consider for example the following query:

```
SELECT * FROM T1 LEFT OUTER JOIN T2 ON T1.i1 = T2.a1 WHERE T1.i1 = 1;
```

If the above query is processed using push selection, it is conceptually handled as though it had the following form:

```
SELECT * FROM (SELECT * FROM T1 WHERE T1.i1 = 1) T1 LEFT OUTER JOIN T2 ON
T1.i1 = T2.a1;
```

The $T1.i1 = 1$ condition is processed before the join condition pertaining to the left outer join is processed in order to reduce the size of the T1 result set.

This method is chosen only after confirming the mathematical consistency and evaluating the cost. For example, the following three queries, when executed, yield different results depending on the circumstances; that is, they are not identical queries:

```
SELECT * FROM T1 LEFT OUTER JOIN T2 ON T1.i1 = T2.a1 WHERE T2.a1 = 1;
SELECT * FROM T1 LEFT OUTER JOIN (SELECT * FROM T2 WHERE T2.a1 = 1) T2 ON
T1.i1 = T2.a1;
SELECT * FROM T1 LEFT OUTER JOIN T2 ON T1.i1 = T2.a1 AND T2.a1 = 1;
```

In order to ensure mathematical consistency and thus obtain the same results, the optimizer uses push selection to evaluate the above queries, as shown below:

```
SELECT * FROM T1 LEFT OUTER JOIN
(SELECT * FROM T2 WHERE T2.a1 = 1) T2
  ON T1.i1 = T2.a1
 WHERE T2.a1 = 1;
```

In the case of a left outer join, if it is desired to move the WHERE condition to the ON clause for processing, it is necessary to leave the WHERE clause as shown below in order to ensure the consistency of results.

```
SELECT * FROM T1 LEFT OUTER JOIN T2
  ON T1.i1 = T2.a1 AND T2.a1 = 1
 WHERE T2.a1 = 1;
```

Even if the optimizer does not use push selection to process the outer join, users can expect a similar effect merely by adding a condition to the ON clause. Of course, it is important to ensure that adding such conditions does not affect the consistency of results and actually results in an improvement in performance.

11.2.3.6 Push Selection on Joins

In the case where a join condition and a single table condition exist, the so-called “push selection on joins” method improves performance by adding another table condition that is similar to the existing table condition.

Consider for example the following query:

```
SELECT * FROM T1, T2 WHERE T1.i1 = T2.a1 AND T1.i1 = 1;
```

11.2 Query Optimization Procedures

In cases where there is no index that can be used to process a query, adding a table condition similar to that shown below is a common way of improving performance:

```
SELECT * FROM T1, T2 WHERE T1.i1 = T2.a1 AND T1.i1 = 1 AND T2.a1 = 1;
```

In the above case, performance can be improved by reducing the size of the T2 result set. Using the push selection on joins method in this way is particularly effective in processing views created through set operations.

Suppose that a view and query have been defined as follows:

```
CREATE VIEW V1(a1, a2)AS ( SELECT m1, m2 FROM T2
                           UNION ALL SELECT x1, y1 FROM T3 );
SELECT * FROM T1, V1 WHERE T1.i1 = V1.a1 AND T1.i1 = 1;
```

Even if indexes were defined for both the T2.m1 and T3.x1 columns, neither of those indexes would not be of any use when the query is executed. In this case, modifying the query using the push selection on joins method would have the following effect:

```
SELECT * FROM T1, V1 WHERE T1.i1 = V1.a1 AND T1.i1 = 1 AND V1.a1 = 1;
```

When the push selection on views method and the so-called “push selection on sets” method are applied to queries defined like that shown above, it becomes possible to use the indexes for both the T2.m1 and T3.x1 columns.

```
SELECT * FROM T1, ( SELECT m1 A1, m2 A2 FROM T2 WHERE T2.m1 = 1
                    UNION ALL
                    SELECT x1, y1 FROM T3 WHERE T3.x1 = 1 ) V1
  WHERE T1.i1 = V1.a1 AND T1.i1 = 1;
```

Proper description of condition clauses as shown above helps the optimizer create a more efficient execution plan. Additionally, users can improve performance by explicitly modifying condition clauses.

Adding a condition clause does not necessarily guarantee better performance. Rather, users should bear in mind that in certain cases adding an unsuitable conditional clause will increase the expense of comparison operations.

11.2.4 Determining the Access Method

The first thing to consider in SQL tuning is determining whether to use indexes to access tables. Most query processing performance issues are resolved by determining whether to use indexes, making this the most important question in SQL tuning.

11.2.4.1 Processing Type of Condition Clause

The optimizer normalizes and classifies condition clauses, and then determines how to access the tables and how to process the conditions based on the classified condition clauses.

Condition clauses are classified as follows according to the order in which they are processed and how they are processed:

Table 11-3 Processing Condition Clauses

| Condition Processing Method | Position in Processing Sequence | Description | Example |
|-----------------------------|--|---|---------------|
| Key Range | 2 | A condition for which an index can be used to perform a range scan | T1.i1 > 1 |
| Key Filter | 3 | A condition for which range scanning is not possible, but that can be checked by comparing index keys | T1.i2 = 1 |
| Constant Filter | 1 | A condition that can be evaluated by checking it once, regardless of the data in the table | ? = 1 |
| Filter | 4 | A condition that requires that the data be directly compared | T1.i4 = 'abc' |
| Subquery filter | 5 | A condition containing a subquery | EXISTS () |
| Example | Index on T1(i1, i2, i3) Query: SELECT * FROM T1 WHERE ? = 1 AND i1 > 1 AND i2 = 2 AND i4 = 'abc' AND EXISTS (SELECT * FROM T2 WHERE T2.a1 = T1.i3); | | |

As shown in the above table, all conditions can be processed according to one of the five processing methods.

With the key range processing method, an index is used to find a range of data that meet the condition in question. Only the locations of the minimum value and the maximum value that meet the condition need to be determined. The data that fall within that range are accepted without checking every item individually against the condition in question. This method thus guarantees improved performance because it obviates additional comparison operations.

The key filter processing method is not a method of using an index to compare a range of values with a condition. Rather, the keys stored in the index are used to perform comparison operations. In greater detail, when this processing method is used, only the pages in which an index is stored are accessed in order to perform the comparison, which reduces the number of times that pages are accessed, thus improving performance. This method can only be used with disk table indexes. Because separate key values are not stored for memory table indexes, using this method with a memory table yields no significant improvement in performance over the filter method.

With the constant processing method, a condition is processed only once before any other conditions are evaluated, and if the logical result is FALSE, the data need not be examined any further.

The filter processing method is used for conditions for which an index cannot be used. With this method, the data are directly read and compared against the condition. In cases where there are multiple filters, the optimizer estimates the expense associated with processing each filter, compares them, and sets the filter processing order so that the filters that will incur the lowest processing expense are processed before other filters.

The subquery filter processing method is typically the most expensive comparison operation. It is thus processed after all other comparison operations have been completed. The optimizer uses various optimization methods to improve the performance of the subquery filter. These methods will be

11.2 Query Optimization Procedures

described in detail later in the section on subquery processing (please refer to [Subquery Optimization](#)).

11.2.4.2 Considerations when Creating Indexes

In order to find the key range that allows a given condition clause to be processed most efficiently, an index is required. However, the following must be taken into consideration when creating an index:

Creating indexes can improve the performance of search operations, but storage space is required in order to manage them, which means that they consume additional system resources. Moreover, creating too many indexes can degrade performance due to the cost of maintaining the indexes when data are inserted, deleted or updated.

When processing queries on memory tables, using an index for a search operation (known as “index scan”) almost always yields better performance than scanning an entire table (“full table scan”).

However, this is not necessarily the case with disk tables. The reason for this is that, while the pattern by which pages are accessed is consistent when performing a full table scan, the pattern by which index scans are performed is irregular and may thus incur excessive disk I/O in some cases.

Generally, for disk tables, the creation of an index is recommended only if a search that uses an index scan is expected to return less than 10% of the records in a table, or a very small number of records.

Therefore, when deciding whether to create an index, how often it is expected that queries using that index will be executed, the expected improvement in performance resulting from the use of the index, the total system resources, and the concomitant degradation in performance of INSERT, DELETE, and UPDATE queries must all be taken into consideration.

11.2.4.3 Index Selection by the Optimizer

The optimizer chooses the most efficient access method based on the given conditions and the indexes associated with the table.

To accomplish this, the optimizer uses various statistical data to evaluate the cost of using each index, and choose the most efficient access method. Once the access method has been determined, the optimizer decides on the method (e.g. the key range method, the filter method, etc.) to use in order to process each condition clause.

What follows is an approximation of the formula that the optimizer uses to calculate the cost of each access method:

Access Cost + Disk I/O Cost

| Conceptual Access Cost Calculation | |
|------------------------------------|---|
| Full scan | : $T(R)$ |
| Index Scan | : $\log(T(R)) + T(R) * \text{MAX}(1/V(\text{Index}), \text{selectivity})$ |

| Conceptual Disk I/O Cost Calculation | |
|--------------------------------------|--|
| Full scan | : $B(R)$ |
| Index Scan : | |
| Buffer Miss | : $[T(R) / V(R.a)] * (1 - M/B(R))$ |
| No Buffer Miss | : $B(R) * (1 - (\log V(R.a) / \log T(R)))$ |

The optimizer estimates the cost as a function of both the access cost and disk I/O cost. Because memory tables do not contain disk pages, the disk I/O cost is automatically excluded from calculations made for memory tables.

When the optimizer calculates the cost for each index, it does so by calculating the cost for the conditions that can be processed using that index. The factor that most strongly affects this cost calculation is the efficiency or “selectivity” of each of these conditions.

The selectivity of a condition is the proportion of all records that are selected using the condition. That is, the lower the selectivity, the better the performance, which is attributable to the smaller number of records in the result set.

Consider the following condition:

```
WHERE i1 = 1 AND i2 = 1
```

In the above condition, if an index has been defined for each of the i1 and i2 columns, the most important factor to consider when deciding which index to use is the selectivity of the condition. For instance, if there are 100 different values in the i1 column [$V(i1) = 100$] and 1000 in the i2 column [$V(i2) = 1000$], the selectivities of the conditions are calculated as follows:

- Selectivity of $(i1 = 1) = 1/V(i1) = 1/100$
- Selectivity of $(i2 = 1) = 1/V(i2) = 1/1000$

That is, for this query, the more efficient access method is to use the index that was defined for column i2.

The above access determination method can generally be used to create a suitable execution plan. However, because this method is based on cost calculation, an unsuitable access method is sometimes chosen under certain circumstances.

Consider the following query:

```
SELECT * FROM soldier
WHERE gender = 'female' AND rank = 'lieutenant';
```

Supposing that indexes have been defined for both the gender and rank columns, the optimizer determines, on the basis of a cost estimation, that it would be most appropriate to use an index to evaluate the condition for the rank column in the above query.

However, if the user is aware that the number of result sets that meet the condition ($gender = 'female'$) is very small, it would be appropriate to use a hint or the like to instruct the optimizer to use another index.

11.2 Query Optimization Procedures

11.2.4.4 Using Composite Indexes

When a composite index is used, the optimizer checks the condition clauses and chooses the one that allows key range processing to be performed on the maximum number of conditions. The more conditions that can be processed using key range scanning, the better the performance that can be expected.

Performance can vary greatly depending on how condition clauses that reference user-defined indexes are written. Therefore, an understanding of the principles behind composite indexes is very helpful when undertaking SQL tuning tasks.

The following example shows how various conditions are processed:

Given a composite index based on the T1(i1, i2, i3, i4) columns:

```
WHERE i1 = 1 AND i2 = 1 AND i3 = 1 AND i4 = 1
```

All of the conditions in the above WHERE clause can be processed using the composite index and key range processing.

```
WHERE i1 = 1 AND i2 > 0 AND i3 = 1 AND i4 = 1
Key Range      : i1 = 1, i2 > 0
Filter(or Key filter): i3 = 1, i4 = 1
```

In the above example, it is determined that key range processing can be performed on two of the conditions, and that the others will be processed using a filter. This is because key range processing can only be performed within a range for which maximum and minimum values can be set, and thus cannot be performed on the conditions that follow the inequality operation.

```
WHERE i1 = 1 AND i3 = 1 AND i4 = 1
Key Range      : i1 = 1
Filter          : i3 = 1, i4 = 1
```

In the above example, even though all of the operations are equality operations, key range processing can only be performed on the first condition. This is because key range processing can only be performed when the conditions match the order of the columns on which the composite index is defined, and when no columns are missing. That is, key range processing cannot be used to evaluate the conditions that reference the i3 and i4 columns, because no condition corresponding to the i2 column precedes them.

As described above, in the case of a composite index, only conditions that appear in the same order as the order of the key columns, with no columns missing, can be evaluated using key range processing, and only if those conditions use equality operations.

The example shown below illustrates that when it is desired to add an index to support the execution of a frequently used query, it is advisable to create the index so that it is of maximum usefulness.

```
WHERE i1 > 0 AND i2 = 1
```

- Proper index: Index on T1(i2, i1)

Key range processing can be used to evaluate the conditions that reference the i2 and i1 columns without a filter.

- Improper index: Index on T1(i1, i2)

Key range processing can be used to evaluate the condition that references the i1 column, but filter processing is used to evaluate the condition that references the i2 column. Therefore, this

index is inefficient.

11.2.4.5 Index and comparison operator

Just because an index and a condition have been defined with reference to the same column does not necessarily mean that the index can be used.

In greater detail, whether the index can be used is determined by how the condition clause is authored and the types of comparison operators it contains, so it is advisable to keep this in mind.

The types of comparison operators and whether indexes can be used with them are shown below:

Table 11-4 Use of Comparison Operators with Indexes

| Type | Comparison Operator | Can be used with index? | Remarks |
|--------------------|---------------------|-------------------------|---|
| Simple comparison | = | O | |
| | != | O | |
| | < | O | |
| | <= | O | |
| | > | O | |
| | >= | O | |
| Area comparison | BETWEEN | O | |
| | NOT BETWEEN | O | |
| Member comparison | IN | O | |
| | NOT IN | O | |
| Pattern comparison | LIKE | O | Yes: T1.i1 LIKE 'abc%' No: T1.i1 LIKE '%abc' |
| | NOT LIKE | X | |
| NULL comparison | IS NULL | O | |
| | IS NOT NULL | O | |
| Exists comparison | EXISTS | X | |
| | NOT EXISTS | X | |
| | UNIQUE | X | |
| | NOT UNIQUE | X | |

11.2 Query Optimization Procedures

| Type | Comparison Operator | Can be used with index? | Remarks |
|------------------|---------------------|-------------------------|---------|
| Quantitative ANY | =ANY | O | |
| | !=ANY | O | |
| | <ANY | O | |
| | <=ANY | O | |
| | >ANY | O | |
| | >=ANY | O | |
| Quantitative ALL | =ALL | O | |
| | != ALL | O | |
| | < ALL | O | |
| | <= ALL | O | |
| | > ALL | O | |
| | >= ALL | O | |

The comparison operators related to geometry data types are shown below. When an index has been defined for a column containing a geometry data type, it can only be used if it is an R-Tree index.

Table 11-5 Geometry-Related Comparisons and Indexes

| Type | Comparison Operator | Status | Remarks |
|---------------------|---------------------|--------|-------------------|
| Geometry comparison | CONTAINS | O | R-Tree index only |
| | CROSSES | O | |
| | DISJOINT | X | |
| | DISTANCE | X | |
| | EQUALS | O | |
| | INTERSECTS | O | |
| | ISEMPTY | X | |
| | ISMBRWITHIN | O | |
| | ISMBRINTERSECTS | O | |
| | ISMBRCONTAINS | O | |
| | ISSIMPLE | X | |
| | NOT CONTAINS | X | |
| | NOT CROSSES | X | |
| | NOT EQUALS | X | |
| | NOT OVERLAPS | X | |
| | NOT RELATE | X | |
| | NOT TOUCHES | X | |
| | NOT WITHIN | X | |
| | OVERLAPS | X | |
| | RELATE | X | |
| | TOUCHES | O | |
| | WITHIN | O | |

Similar to the case noted above, just because an index has been defined for a column and the index can be used with a comparison operator does not necessarily mean that the index can always be used.

Whether the index can be used is determined by the characteristics of the condition clause. The characteristics of condition clauses that determine how indexes can be used are as follows. (Note: the examples of condition clauses are based on the assumption that the columns are INTEGER type columns.)

Table 11-6 Characteristics of Condition Clauses and the Use of Indexes

| Characteristics of Condition Clause | Example of Valid Use | Example of Invalid Use |
|---|--------------------------------------|--|
| The comparison operator must be capable of using an index. | T1.i1 = 1 | T1.i1 NOT LIKE 'a' |
| The comparison must reference a column. | T1.i1 = 1 | 1 = 3 |
| No operations must be performed on the column. | T1.i1 = 3 - 1 | T1.i1 + 1 = 3 |
| Columns can be referenced only on only one side of the operator. | (T1.i1, T1.i2) = (1, 1) T1.i1 = 1 | (T1.i1, 1) = (1, T1.i2) T1.i1 = T1.i2 |
| No conversion operation must be performed on the value in the column. | T1.i1 = SMALLINT'1' | T1.i1 = 1.0 |

As described above, care must be taken when writing condition clauses in order for it to be possible to use indexes. In particular, operations on column values and type conversions must be avoided.

The data types and data conversions that can be used with indexes will be described in detail later (please refer to [Indexes and Data Types](#)).

11.2.4.6 Indexes and Data Types

Just because a column referenced in a WHERE clause has an index associated with it does not necessarily mean that the user will always be able to perform so-called “index scanning”, that is, use the index to help perform the search. Index scanning may or may not be possible, depending on the column’s data type and whether the data would need to be converted.

```
SELECT * FROM T1 WHERE T1.i1 = ?
```

For example, assuming that the i1 column is a VARCHAR type column and is also the PRIMARY KEY column for the table T1, checking the execution plan of the above query using the iSQL utility will indicate that index scanning is possible. However, if the column is bound to a numeric data type, such as the INTEGER type, the values in the column must be implicitly converted to the numeric type in order to perform the comparison. This makes index scanning impossible.

Therefore, whether it is possible to use index scanning to process a query should be checked using the iSQL utility. Even if index scanning is used in the execution plan, the value that is actually bound in the application and the data type of the column must be checked. If these are not checked, the application may perform full scanning, that is, it may scan the entire table, which will result in performance that is very different from that measured within the iSQL utility.

The data types and whether they can be used with indexes are described below. The darkened cells indicate that the values in key columns will be type-converted when they are compared.

Table 11-7 Data Types and the Use of Indexes

| VALUE KEY | CHAR | VARCHAR | SMALLINT | INTEGER | BIGINT | NUMERIC | FLOAT | REAL | DOUBLE | DATE | BLOB | NIBBLE | BYTE | GEOMETRY |
|--------------|------|---------|----------|---------|--------|---------|-------|------|--------|------|------|--------|------|----------|
| CHAR | O | O | X | X | X | X | X | X | X | X | - | - | - | - |
| VARCHAR | O | O | X | X | X | X | X | X | X | X | - | - | - | - |
| SMALLINT | X | X | O | O | O | O | O | O | O | - | - | - | - | - |
| INTEGER | X | X | O | O | O | O | O | O | O | - | - | - | - | - |
| BIGINT | X | X | O | O | O | O | O | O | O | - | - | - | - | - |
| NUMERIC | O | O | O | O | O | O | O | O | O | - | - | - | - | - |
| FLOAT | O | O | O | O | O | O | O | O | O | - | - | - | - | - |
| REAL | X | X | O | O | O | O | O | O | O | - | - | - | - | - |
| DOUBLE | O | O | O | O | O | O | O | O | O | - | - | - | - | - |
| DATE | O | O | - | - | - | - | - | - | - | O | - | - | - | - |
| BLOB | - | - | - | - | - | - | - | - | - | - | O | - | - | - |
| NIBBLE | - | - | - | - | - | - | - | - | - | - | - | O | - | - |
| BYTE | - | - | - | - | - | - | - | - | - | - | - | - | O | - |
| GEOMETRY | - | - | - | - | - | - | - | - | - | - | - | - | - | O |

Data types can be broadly categorized into the following character type group and numeric type groups. Indexes can be used to perform comparisons between data types that fall within the same group.

| | | | |
|-----------------|-----------------------|------------------------|--|
| Character types | | | CHAR, VARCHAR |
| Numeric types | Native | Integer Type | BIGINT, INTEGER, SMALLINT |
| | | Real Number Type | DOUBLE, REAL |
| | Non-Native (Exponent) | Fixed decimal data | NUMERIC, DECIMAL, NUMBER(p), NUMBER(p,s) |
| | | Fixed decimal datatype | FLOAT, NUMBER |

Indexes can be used to perform comparisons between the CHAR and VARCHAR types in the character type group, or to perform comparisons between the data types in the integer type group, the real number type group, or the non-native type group.

- Character types

11.2 Query Optimization Procedures

`char_column = VARCHAR 'abc'`

`varchar_column = CHAR 'abc'`

- Numeric types

`integer_column = DOUBLE '1'`

`number_column = DOUBLE '1'`

`integer_column = NUMBER '1'`

Different numerical data types are converted and compared according to the criteria shown in the following conversion matrix:

| | Integer Type | Real Number Type | Exponent |
|------------------|------------------|------------------|------------------|
| Integer Type | Integer Type | Real Number Type | Exponent |
| Real Number Type | Real Number Type | Real Number Type | Real Number Type |
| Exponent | Exponent | Real Number Type | Exponent |

Such conversion can also be performed on the values in index columns, in which case the comparison operation is performed on the value resulting from conversion of the value in the index column.

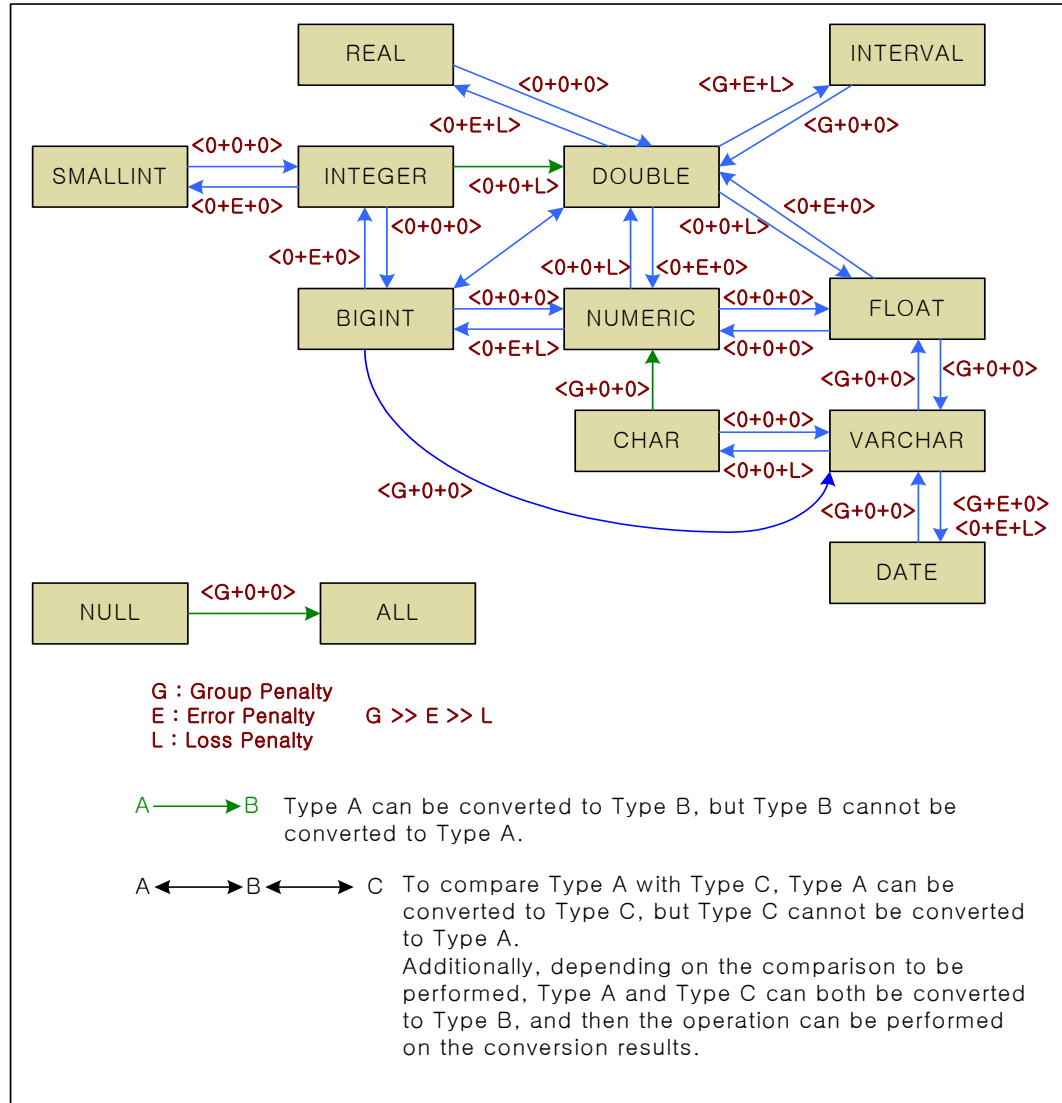
Therefore, the speed of execution is slower for an index scan for which conversion needs to be performed, such as the comparison `bigint_col = NUMERIC '1'`, than for an index scan for which conversion need not be performed, such as the comparison `bigint_col = BIGINT '1'`.

Even in cases not involving index scans, the performance of comparison operations that compare two values is strongly dependent on the data types of the values being compared. When values having the same data type are compared, no data conversion expense is incurred, and thus performance is optimal. However, when values having different data types are compared, efforts must be made to minimize data conversion.

For example, when comparing a numeric type with a character type, the shortest conversion path will be that between the `FLOAT` and `VARCHAR` types, respectively. However, when choosing data types it is of course necessary to consider factors other than data conversion costs, for example differences in the data storage space requirements and format of the different data types.

The following figure schematically shows the data type conversion paths.

Figure 11-3 Data type Conversion Paths



In order to be able to use index columns in search operations, it is very important that conditional clauses be authored as described above. Because the type of the conditional clause can affect performance, care must be taken when writing WHERE clauses and when authoring client applications.

After deciding on the access method for each table, the optimizer processes joins. The decision on the access method for each table is not final, and may be revised while processing joins.

11.2.5 Determining the Join Order

The performance of complicated queries is most affected by the order of joins and how they are processed. Thus, determining whether the join order and join processing methods are suitable, and subsequently revising them when appropriate, is of great help in improving query performance.

11.2 Query Optimization Procedures

The optimizer progresses through the following sequence when processing joins:

1. Groups joins according to the join conditions
2. Determines the join relationships for each group
3. Decides on the join order for each group
4. Decides on the join method for each group
5. Decides on the order and method for joining groups to each other

Understanding how the optimizer performs each stage of the process is helpful during SQL tuning.

This section describes how the optimizer determines the join order. The join order is determined based on a greedy algorithm that is based on join selectivity.

11.2.5.1 Classifying Join Groups

Considering all tables, including those between which joins are not defined, would serve only to increase the workload, and would be of little help in determining a suitable join order. In other words, it is more efficient to place the tables that are joined together in respective groups and then determine the join order within each group.

Consider for example the following query:

```
SELECT * FROM T1, T2, T3, T4, T5
WHERE T1.i1 = T2.a1 AND T2.a2 = T3.m1
      AND T4.x1 = T5.z1;
```

Join Group Classification: (T1, T2, T3), (T4, T5)

As shown in the above example, no joins are defined between the tables in the two different groups. Considering all of the tables at one time when determining the join order would only add unnecessarily to the processing expense.

This method of classification is efficient in most cases, but may not be suitable depending on the nature of the application or the way that the data are organized. In such cases, the join order can be controlled using hints.

After classifying the tables into join groups in this way, the join order within each group is determined as follows:

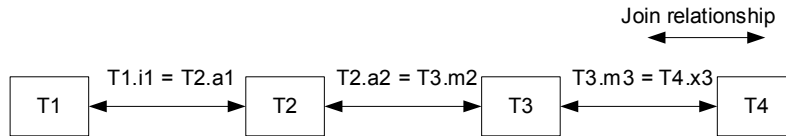
11.2.5.2 Configuring Joining Relationships

The purpose of determining the join relationships within each join group is to set a more efficient join order than could be set by simply minimizing the cost of comparisons between tables that are not directly joined.

Consider the following query:

```
SELECT * FROM T1, T2, T3, T4
WHERE T1.i1 = T2.a1 AND T2.a2 = T3.m2
      AND T3.m3 = T4.x3;
```

The join relationships in the above query can be expressed as follows:



When managing join relationships and setting the join order, considering and assessing the cost only of direct join relationships prevents tables that are not directly joined, such as tables T1 and T4 above, from being chosen to be joined first.

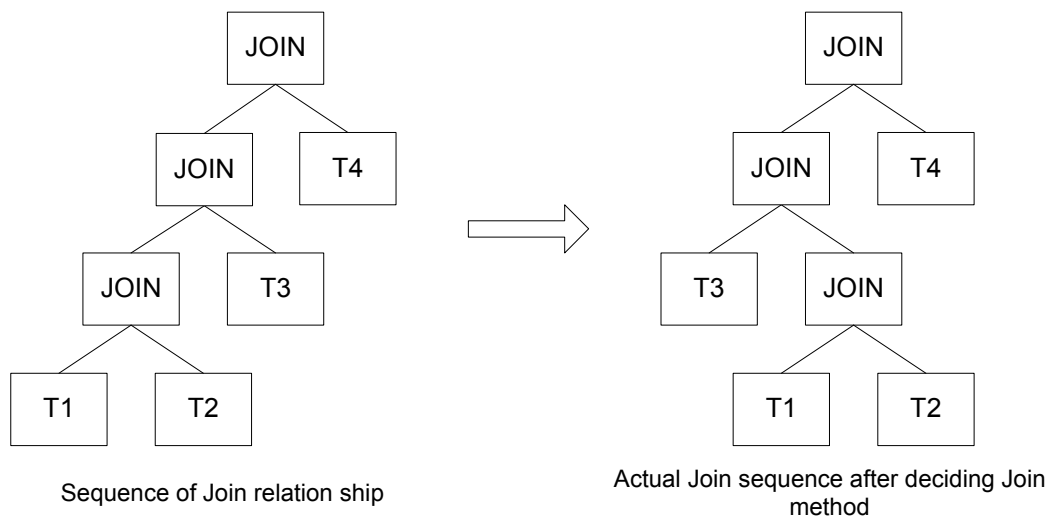
Considering the join relationships usually helps the optimizer determine an efficient join order, but does not always guarantee that the optimizer will arrive at the most efficient join order. Thus, it is possible to control the join order using hints when necessary.

11.2.5.3 Deciding the Sequence of Joining Relationship

The most efficient join order is determined using the join relationships created above.

The way that the join order is determined is to choose the joins with the highest selectivity to be processed first. Then, which joins are the most efficient is determined again, starting with these joins. However, this join order is not the actual join order. The actual join order is finally set when the join method is determined.

Figure 11-4 Join Ordering



As shown in the figure above, the join order that is set at this stage only represents the depth of the join relationships. The actual join order is determined by selecting the joining method.

The most important factor in determining the join order is join selectivity, which means the ratio of the size of the result set created by combining two tables to the size of the original tables. In other words, when determining the join order, it is not the actual size of the result set that is important; what is important is how much the size of the result set is reduced.

The optimizer calculates join selectivity as shown below. A detailed description of this formula is beyond the scope of this document.

Conceptual Join Selectivity Calculation

$$[T(R) * T(S) / \text{MAX}[V(R.a), V(S.a)] / [T(R) + T(S)]]$$

By using join selectivity to determine the join order in this way, the join relationships are ordered such that the result set is reduced in size by higher ratios, which means that an appropriate join order is likely to be determined later when the joining method is determined.

For example, assume that the number of records in each of tables R, S, T and U referred to in a query and the number of results in the result sets created by joining the tables are as follows:

$T(R) = 10, T(S) = 10, T(R \text{ JOIN } S) = 10$
 $T(T) = 1000, T(U) = 1000, T(T \text{ JOIN } U) = 100$

In the above example, although the number of records in the result set obtained by joining tables R and S is smaller than the result set obtained by joining tables T and U, the latter join relationship is more important because it reduces the number of results in the result set by a greater proportion.

The join order set by the optimizer using the join relationships cannot be guaranteed to be the optimum join order in all cases. Thus, the join order can be controlled using join order hints.

After the join order has been provisionally set, the method of joining each pair of tables is determined, at which time the join order is finalized.

11.2.6 Determining the Join Method

Once the join order has been determined, the method of joining each pair of tables is decided. The costs of each of a variety of join methods are compared and used in the determination of the join order, join direction and joining methods.

The joining methods supported in ALTIBASE HDB are broadly classified into the following four types:

- [Nested Loop Join Methods](#)
- [Sort-based Join methods](#)
- [Hash-based Join Methods](#)
- [Merge Join Methods](#)

The joining methods and the conditions under which they can be used are shown below:

Table 11-8 Join Method & Join Order

| Joining Method | Joining Method Name | Join Direction | |
|---|------------------------|----------------|-------------|
| | | Left=>Right | Right=>Left |
| Nested Loop | Full nested loop | C, I, L | C, I, R |
| | Full store nested loop | C, I, L, F | C, I, R, F |
| | Index nested loop | I, L | I, R |
| | Anti outer nested loop | F | F |
| Sort-Based | One pass sort join | I, L, F | I, R, F |
| | Multi pass sort join | I, L, FI | I, R, F |
| Hash-Based | One pass hash join | I, L, F | I, R, F |
| | Multi pass hash join | I, L, F | I, R, F |
| Merge-Based | Index merge join | I | I |
| | Sort merge join | I | I |
| Possible Join Types: <ul style="list-style-type: none"> • C (Cartesian Product): The combination of two tables that are not joined • I (Inner Join): A typical join between two joined tables • L (Left Outer Join): A join between two tables that have a left outer join relationship between them • R (Right Outer Join): A join between two tables that have a right outer join relationship between them • F (Full Outer Join): A join between two tables that have a full outer join relationship between them | | | |

The optimizer determines which of the above joining methods are possible, uses cost estimation to select the most efficient joining method from among them, and determines the join direction. Once the joining method has been determined, the outer table is displayed on the left, and the inner table is displayed on the right.

Which joining method has been chosen can be checked by analyzing the execution plan tree, and can be controlled using hints. This section describes the joining methods supported in ALTIBASE HDB.

11.2.6.1 Nested Loop Join Methods

The nested loop joining methods are as follows:

- Full nested loop join
- Full store nested loop join
- Index nested loop join
- Anti outer nested loop join

11.2 Query Optimization Procedures

When the full nested loop joining method is used, every row of one table is joined to every row of the other table. There is no relationship established between the two tables.

```
SELECT * FROM T1, T2;
```

When the full store nested loop joining method is used, full nested loop joining is performed after the result set from the inner table is stored. This method is likely to be used in cases where the size of the result set can be greatly reduced by processing conditions other than join conditions, and usually uses the Cartesian product of join groups.

```
SELECT * FROM T1, T2 WHERE T1.i1 = 1 AND T2.i1 = 1;
```

When the index nested loop joining method is used, one or more indexes are used to process join conditions. This method is likely to be used if the number of records in the outer table is small and if an index has been defined for the inner table.

```
Index on T2(i1)
SELECT * FROM T1, T2 WHERE T1.i1 = T2.i1 AND T1.i1 = 1;
```

The anti outer nested loop joining method is only used with full outer join, and can be used only if indexes have been defined for the columns corresponding to the join condition in both the outer table and the inner table. In such cases, this method is likely to be selected over other methods.

```
Index on T1(i1), Index on T2(i1)
SELECT * FROM T1 FULL OUTER JOIN T2 ON T1.i1 = T2.i1;
```

The cost of execution of each of the joining methods can be roughly calculated thus: (access cost + disk I/O cost). The access cost and disk I/O cost for each joining method are calculated as follows:

Table 11-9 Estimated Cost 1

| Joining Method | Access Cost | Disk I/O Cost | |
|--|----------------------------------|--|----------------------------------|
| | | No Buffer Miss | Buffer Miss |
| Full Nested | $T(O) * T(I)$ | $B(O) + B(I)$ | $B(O) + T(O) * B(I)$ |
| Full Store Nested | $T(O) * T(I) + S(I)$ | $B(O) + S(I) + B(I)$ | $B(O) + S(I) + T(O) * B(I)$ |
| Index Nested | $T(O) * [\log(T(I)) + s * T(I)]$ | $B(O) + B(I) * [1 + (\log s / \log T(I))]$ | $B(O) + T(O) * s * [1 - M/B(I)]$ |
| Anti Outer | $T(O) * [\log(T(I)) + s * T(I)]$ | $B(O) + B(I) * [1 + (\log s / \log T(I))]$ | $B(O) + T(O) * s * [1 - M/B(I)]$ |
| Key: O: Outer Table; I: Inner Table T(): Record Count; B(): Disk Page Count; S(): Storage Cost; M: Buffer Page; s: Join Condition Selectivity | | | |

11.2.6.2 Sort-based Join methods

The following sort-based joining methods are used:

- One-pass sort-based join

- Multi-pass sort-based join

When sort-based joining methods are used, the inner table is sorted and then stored, and range scanning is performed based on the join conditions. This method is likely to be used in cases where there is no index and join conditions that use inequality operators are processed.

```
SELECT * FROM T1, T2 WHERE T1.i1 > T2.i1;
```

The one-pass sort-based joining method can be used when the amount of data in the inner tables is low enough that the data can be managed in the buffer. This method can always be used when a memory table is used as the inner table.

The multi-pass sort-based joining method is used when the amount of data in the inner table is too great to be managed in the buffer. This method is used to reduce the amount of disk I/O.

With this method, both the outer table and the inner table are sorted and then saved. Because the records are evaluated against the join conditions in the order in which the data in the outer table are sorted, this method increases the likelihood that the same disk page will be accessed to retrieve the corresponding record for the repeated table, thereby reducing disk I/O expense.

The cost of execution of each of the joining methods can be roughly calculated thus: (access cost + disk I/O cost). The access cost and disk I/O cost for each joining method are calculated as follows:

Table 11-10 Estimated Cost 2

| Joining Method | Access Cost | Disk I/O Cost | |
|--|--|-----------------------------|----------------------------------|
| | | No Buffer Miss | Buffer Miss |
| One-Pass Sort | $S(I) + T(O) * [\log(T(I)) + s * T(I)]$ | $B(O) + S(I) + B(I)$ | $B(O) + 3S(I) + T(O) * s * B(I)$ |
| Multi-Pass Sort | $S(O) + S(I) + T(O) * [\log(T(I)) + s * T(I)]$ | $S(O) + S(I) + B(O) + B(I)$ | $3S(O) + 3S(I) + B(O) + B(I)$ |
| Key: O: Outer Table; I: Inner Table; T(): Record Count; B(): Disk Page Count; S(): Array Storage Cost; M: Buffer Page; S: Join Condition Selectivity | | | |

11.2.6.3 Hash-based Join Methods

The hash-based joining methods are as follows:

- One-pass hash-based join
- Multi-pass hash-based join

When hash-based joining methods are used, the inner table is stored in a hash structure, and range scanning is then performed using the join conditions. This method can only be used with join conditions that use equality operators, and is likely to be chosen in cases where there are no indexes.

```
SELECT * FROM T1, T2 WHERE T1.i1 = T2.i1;
```

The one-pass hash-based joining method can be used when the amount of data in the inner table is low enough that the data can be managed in the buffer. This method can always be used when a memory table is used as the inner table.

11.2 Query Optimization Procedures

The multi-pass hash-based joining method is used when the amount of data in the inner table is too great to be managed in the buffer. This method is used to reduce the amount of disk I/O.

The outer table is divided and stored in temporary tables. The inner table is divided and stored in other temporary tables. The same hash function is used to divide both the outer table and the inner table. Therefore, this method increases the likelihood that the records in the inner table that satisfy the join condition will be on the same disk page as each other, thereby reducing disk I/O.

The cost of execution of each of the joining methods can be roughly calculated thus: (access cost + disk I/O cost). The access cost and disk I/O cost for each joining method are calculated as follows:

Table 11-11 Estimated Cost 3

| Joining Method | Access Cost | Disk I/O Cost | |
|---|---------------------------------|-----------------------------|--|
| | | No Buffer Miss | Buffer Miss |
| One-Pass Hash | $H(I) + T(O) * s * T(I)$ | $B(O) + H(I) + B(I)$ | $B(O) + 3H(I) + T(O) * s * B(I)$ |
| Multi-Pass Hash | $H(O) + H(I) + T(O) * s * T(I)$ | $H(O) + H(I) + B(O) + B(I)$ | $3H(O) + 3H(I) + B(O) + T(O) * s * B(I)$ |
| Key: O: Outer Table; I: Inner Table; T(): Record Count; B(): Disk Page Count; S(): Hashing Storage Cost; M: Buffer Page; S: Join Condition selectivity | | | |

11.2.6.4 Merge Join Methods

The merge joining method is very efficient when the data in both tables have been sorted. With this method, there is no concept of a so-called “driving table”, meaning a table that is read and used as a basis to search for corresponding records in the other table.

Both of the tables must have been sorted on the basis of the join key. However, if either (or both) of the tables uses an index for which the key column is the same as the join key column, then it is not necessary to sort that table (because it is sorted already).

```
Index on T1(i1), Index on T2(a1)
```

```
SELECT * FROM T1, T2 WHERE T1.i1 = T2.a1;
```

The cost of execution of each of the joining methods can be roughly calculated thus: (access cost + disk I/O cost). The access cost and disk I/O cost for each joining method are calculated as follows:

Table 11-12 Estimated Cost 4

| Table Type | Access Cost | Disk I/O Cost | |
|-----------------|---------------|----------------|----------------|
| | | No Buffer Miss | Buffer Miss |
| Indexed Table | $T(O)$ | $B(O)$ | $B(O)$ |
| Non-Index Table | $S(O) + T(O)$ | $S(O) + B(O)$ | $3S(O) + B(O)$ |

| Table Type | Access Cost | Disk I/O Cost | |
|---|-------------|----------------|-------------|
| | | No Buffer Miss | Buffer Miss |
| Key: In Merge Join, the cost of the two tables is calculated; O: Table; T(): Record Count; B(): Disk Page Count; S(): Array Storage Cost | | | |

So far, we have described the optimization process for the critical path, that is, the FROM and WHERE clauses, which has the greatest effect on performance.

What follows is a description of the optimization process for the non-critical path, organized according to statement.

11.2.7 Optimization of Group Operations

After optimizing the critical path, the optimizer optimizes the so-called “non-critical path”, which means GROUP BY and HAVING clauses and aggregate operations.

The method to use when optimizing aggregate operations is determined by calculating and comparing the costs of hashing and sorting. The following methods are applied, if possible:

- [GROUP BY Using Sort Order](#)
- [MIN/MAX Optimization Using Sort Order](#)
- [Avoiding Use of the DISTINCT Keyword from within a MIN or MAX Operation](#)
- [Dependence on Sort Order when using DISTINCT with Aggregate Functions](#)

11.2.7.1 GROUP BY Using Sort Order

Hashing or sorting is usually required in order to classify groups according to the values in the columns referenced in the GROUP BY clause. This type of operation requires large amounts of storage space and incurs high CPU usage.

The optimizer can use this method if there is an index that can be used to process the GROUP BY clause, or if the order of the columns is known due to actions taken when the critical path was optimized.

The optimizer avoids performing sorting or hashing in the following cases:

- When the columns in the GROUP BY clause are used in the index
- When the data have already been sorted on the basis of the columns in the GROUP BY clause in a previous operation

Taking the following query as an example, the optimizer creates an execution plan that eliminates the need to perform any additional hashing or sorting. Because this type of optimization is not possible for all queries, it is necessary to determine whether it is possible by analyzing the execution plan.

11.2 Query Optimization Procedures

```
Index on T1(i1)

SELECT SUM(i2) FROM T1 GROUP BY i1;
SELECT SUM(i2) FROM T1 WHERE i1 > 0 GROUP BY i1;
```

The following example shows how to perform optimization using an indexed column that is referenced in a GROUP BY clause:

```
Index IDX1 on T1(i1, i2), Index IDX2 on T1(i1, i3)

SELECT * FROM T1 WHERE i1 > 0 GROUP BY i1, i3;
```

When processing the above query, the optimizer can use *IDX1* or *IDX2* to process the critical path. If *IDX1* is used, hashing and sorting are both required in order to process the GROUP BY clause. In contrast, no additional storage is required if *IDX2* is used.

The user analyzes the execution plan tree, observes that *IDX1* is being used, and therefore uses hints to force the optimizer to use *IDX2*.

For the following query, assume that no index has been defined for table *T1*:

```
SELECT * FROM T1 WHERE i3 > 0 GROUP BY i1, i3;
```

In the above case, creating the composite index 'T1(i3, i1)' can improve query performance. The order in which the columns appear in the GROUP BY clause is not important, because the result sets for 'GROUP BY i1, i3' and 'GROUP BY i3, i1' are the same. If a user creates a composite index such as 'T1(i3, i1)', 'i3>0', range scanning cannot be used to evaluate the predicate.

11.2.7.2 MIN/MAX Optimization Using Sort Order

In the case of MIN and MAX aggregations when no GROUP BY clause is defined, it is easy to find the minimum and maximum values using an index. This method is known as "MIN/MAX Optimization using Sort Order".

That is, for the following type of query, the optimizer can use an index to process the MIN and MAX operations.

```
Index on T1(i1)

SELECT MIN(i1) FROM T1;
SELECT MAX(i1) FROM T1 WHERE i1 < 0;
```

In greater detail, to determine the value of MIN, the index is searched starting with the smallest value, whereas to determine the value of MAX, the index is searched starting with the largest value.

If a query whose purpose is to retrieve a MIN or MAX value is executed frequently, creating an index for the corresponding column will greatly improve performance because not all of the records will need to be searched.

However, because an index can only be searched in one direction when using this method, if both MIN and MAX values for the same column are to be retrieved, as shown in the following example, the optimizer will choose full scanning.

```
SELECT MIN(i1), MAX(i1) FROM T1;
```

Therefore, it is recommended that the user rewrite the query as shown below:

```
SELECT MIN(i1) FROM T1
```

```
UNION ALL
SELECT MAX(i1) FROM T1;
```

11.2.7.3 Avoiding Use of the DISTINCT Keyword from within a MIN or MAX Operation

The following two queries are computationally equivalent. That is, the DISTINCT keyword is irrelevant.

```
SELECT MIN(DISTINCT i2) FROM T1 GROUP BY i1;
SELECT MIN(i2) FROM T1 GROUP BY i1;
```

Of course, the optimizer will remove the DISTINCT keyword from within a MIN or MAX operation. Nevertheless, users should avoid writing such meaningless statements.

11.2.7.4 Dependence on Sort Order when using DISTINCT with Aggregate Functions

When an aggregate function is executed on the result of a DISTINCT operation on a column, such as SUM(DISTINCT i1), additional storage space and hash operations are required in order to remove redundancies.

However, the use of the DISTINCT keyword with an aggregate function does not require additional storage space and hash operations if an index is used, or if the result set is known to have been properly sorted due to a preceding optimization task. Note that this only applies to queries that do not contain a GROUP BY clause.

```
SELECT SUM(DISTINCT i1) FROM T1;
SELECT COUNT(DISTINCT i1) FROM T1 WHERE i1 > 0;
```

In cases where queries similar to those shown above are frequently used, creating an index for the column in question can greatly enhance performance, because it will obviate the need for additional storage space and hash operations.

11.2.8 Optimization of DISTINCT Clause

The DISTINCT keyword can usually be processed using a hash or sort operation. The optimizer chooses the execution method by estimating the cost of each.

If an index has been defined for a column that is referenced in a DISTINCT operation, or if the result set is known to have been properly sorted due to the optimization of a sub-statement, the optimizer optimizes the query so that it can be processed without requiring additional storage space. This is known as “DISTINCT Optimization using Sort Order”.

For example, queries similar to those that follow can be processed using “DISTINCT Optimization using Sort Order”, thus obviating the need for additional storage space:

```
Index on T1(i1, i2, i3)

SELECT DISTINCT i1, i2, i3 FROM T1;
SELECT DISTINCT i2, i1 FROM T1 WHERE i1 > 0;
```

The columns referenced in the DISTINCT clause must be present in the index. However, they do not have to be referenced in the same order as they appear in the key column of the index.

Thus, the index would not be useful when processing the following query:

11.2 Query Optimization Procedures

```
SELECT DISTINCT i1, i3 FROM T1;
```

With a good understanding of these characteristics, the user can rewrite the first query shown below so that it resembles the second query shown below, so that the “DISTINCT Optimization using Sort Order” optimization method can be used.

```
SELECT DISTINCT i1, i3 FROM T1 WHERE i2 = 0;  
SELECT DISTINCT i1, i2, i3 FROM T1 WHERE i2 = 0;
```

The above query was optimized by adding the *i2* column to the DISTINCT clause. Of course, what makes this possible is the WHERE clause, which ensures that only a single value for *i2* will be returned (*i2* = 0).

Therefore, when creating indexes, the user is advised to give consideration not only to WHERE clauses but also to frequently used DISTINCT clauses.

11.2.9 Optimization of Set Operations

Set operations such as UNION, INTERSECT, and MINUS (but not UNION ALL) require the management of intermediate results.

When multiple set operations are to be performed, the optimizer performs optimization such that the smallest result set can be processed first. However, the assessment of which result set is the smallest is performed on the basis of cost estimation, and thus may not be accurate.

Therefore, the user is advised to explicitly state the operation order if the user can anticipate the sizes of the result sets relative to each other.

```
SELECT * FROM large  
INTERSECT  
SELECT * FROM medium  
INTERSECT  
SELECT * FROM small;  
  
(SELECT * FROM small  
INTERSECT  
SELECT * FROM medium )  
INTERSECT  
SELECT * FROM large;
```

One error that users frequently make when optimizing set operations is failing to distinguish between UNION and UNION ALL.

The UNION operation removes redundancies from the result set, whereas the UNION ALL operation does not. Therefore, the performance of UNION is invariably inferior to that of UNION ALL when they are executed on the same table.

Therefore, it is preferable to use the UNION ALL operation in cases where it is certain that the result set contains no redundancies.

11.2.10 Optimizing ORDER BY Clauses

The ORDER BY clause requires results to be sorted. The optimizer checks whether each of the following optimization methods can be applied and then sets an execution plan.

- [ORDER BY Using Sort Order](#)

- [The LIMIT Clause and Sorting](#)

11.2.10.1 ORDER BY Using Sort Order

If an index can be used, or if, in the process of optimizing a subquery, a column referenced in an ORDER BY clause is known to have been sorted, the optimizer need not perform any additional sorting tasks in order to process an ORDER BY clause.

For example, the optimizer can process the ORDER BY clause in the following query without undertaking any sorting operations:

```
Index on T1(i1, i2, i3)

SELECT * FROM T1 ORDER BY i1, i2, i3;
SELECT * FROM T1 WHERE i1 = 0 ORDER BY i1, i2;
```

Note that this optimization method is possible only if the columns in the ORDER BY clause are in the same order that they appear in the index.

Therefore, by analyzing the execution plan and using hints, the sorting task for processing the ORDER BY clause can be obviated.

```
Index IDX1 on T1(i1, i2), Index IDX2 on T1(i3)

SELECT * FROM T1
  WHERE i3 IS NOT NULL
  ORDER BY i1, i2;
SELECT /*+ INDEX(T1, IDX1) */ * FROM T1
  WHERE i3 IS NOT NULL
  ORDER BY i1, i2;
```

In the above example, it is advisable to use a hint to ensure that the *IDX1* index, rather than the *IDX2* index, is used to process the WHERE clause, because this eliminates the sorting cost associated with the ORDER BY clause and is thus more efficient.

11.2.10.2 The LIMIT Clause and Sorting

When the LIMIT clause is used, only part of the result set is returned. The following query only returns five of the records in table *T1*.

```
SELECT * FROM T1 LIMIT 5;
```

When the ORDER BY clause and the LIMIT clause are used together, the optimizer performs LIMIT sorting optimization. When this method is used, only the records specified in the LIMIT clause, rather than all records, are sorted.

For example, to sort the results of the following query, only five records need to be sorted.

```
SELECT * FROM T1 ORDER BY i1 LIMIT 5;
```

Therefore, in cases where it is not necessary to sort and return the entire result set, using the LIMIT statement can improve performance and the efficiency with which resources are used.

11.2.11 Optimizing and Partitioned Tables

When a partitioned table is accessed, unneeded partitions are removed through the processes of

11.2 Query Optimization Procedures

partition pruning and filtering, and an execution plan tree that only references the partitions containing records that satisfy predicate conditions is created.

Partition pruning optimizes so-called “fixed predicates”, that is, those for which the conditions do not change during execution. However, with so-called “variable predicates”, because the partitions containing records that satisfy the predicate conditions continue to change, it is not possible to create a plan tree containing only the partitions that contain records that meet the predicate conditions simply by removing unneeded partitions from the plan tree.

Variable predicates can be optimized through partition filtering.

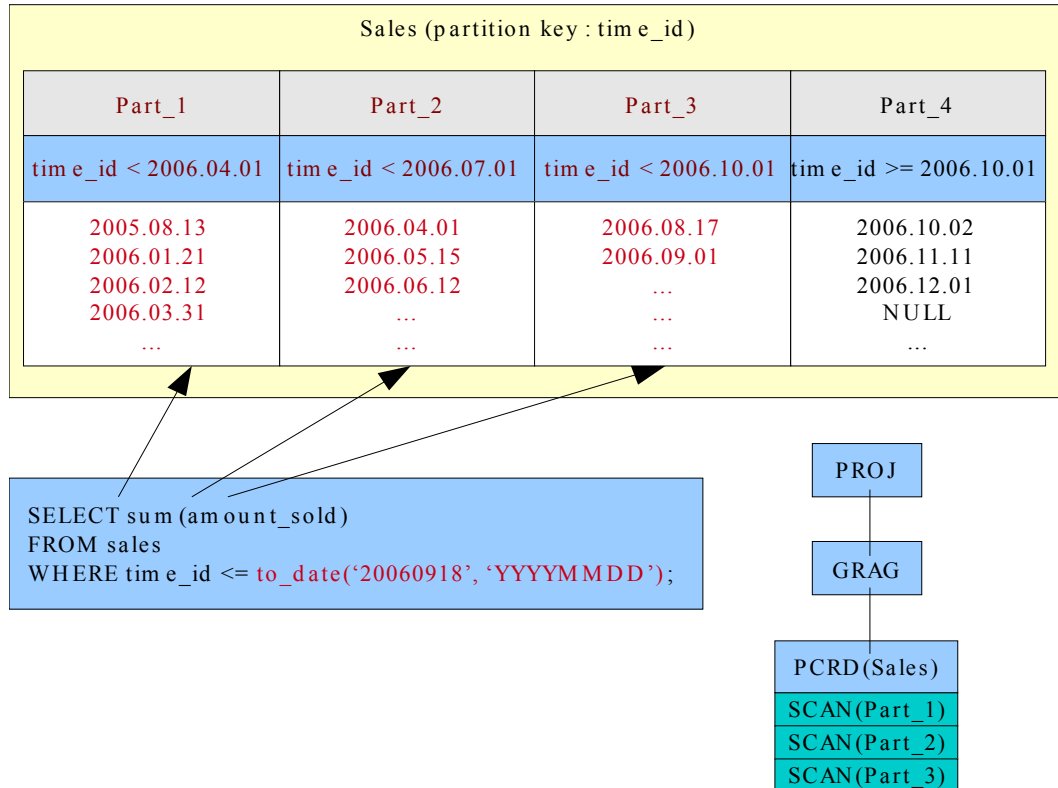
- [Range Partition Pruning](#)
- [List Partition Pruning](#)
- [Hash Partition Pruning](#)
- [Partition Filtering](#)

11.2.11.1 Range Partition Pruning

The predicates for which range partition pruning is possible have the same form as conditions that allow indexes to be used for searches. Range partition pruning is not possible for conditions that contain host variables or for join conditions.

Examples of such predicates include all predicates that allow the use of indexes and represent ranges, such as:

- `I1 = 1`
- `i1 between 1 and 10`
- `i1 > 5`

Figure 11-5 An Example of Range Partition Pruning

The top half of [Figure 10-5] depicts the “Sales” table, which is partitioned into 4 partitions based on time.

At the optimization stage, partition pruning is performed using the predicate that references the partition key time_id. Accordingly, the plan tree specifies that only the Part_1, Part_2 and Part_3 partitions, which contain records that meet the predicate conditions, are to be searched.

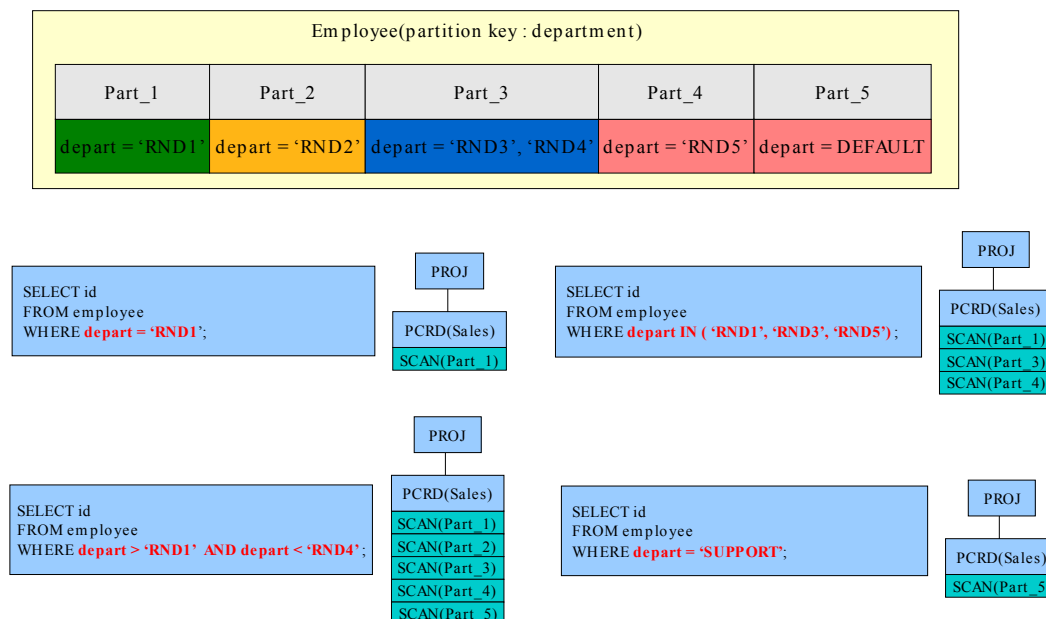
PCRD is a partition-coordinator execution node having several child nodes, each of which is a scan node for a corresponding partition of the partitioned table.

11.2.11.2 List Partition Pruning

The predicates with which list partition pruning is possible have the same form as conditions for which searching can be performed using indexes, and can only have the equality (=) operator. List partition pruning is not possible for join conditions or for conditions that contain host variables.

Examples of such predicates include:

- I1 = 1
- i1 in (4,5,6)
- i1 IS NULL

Figure 11-6 An Example of List Partition Pruning

The top half of [Figure 10-6] depicts the *Employee* table, which is partitioned into 5 partitions based on department name.

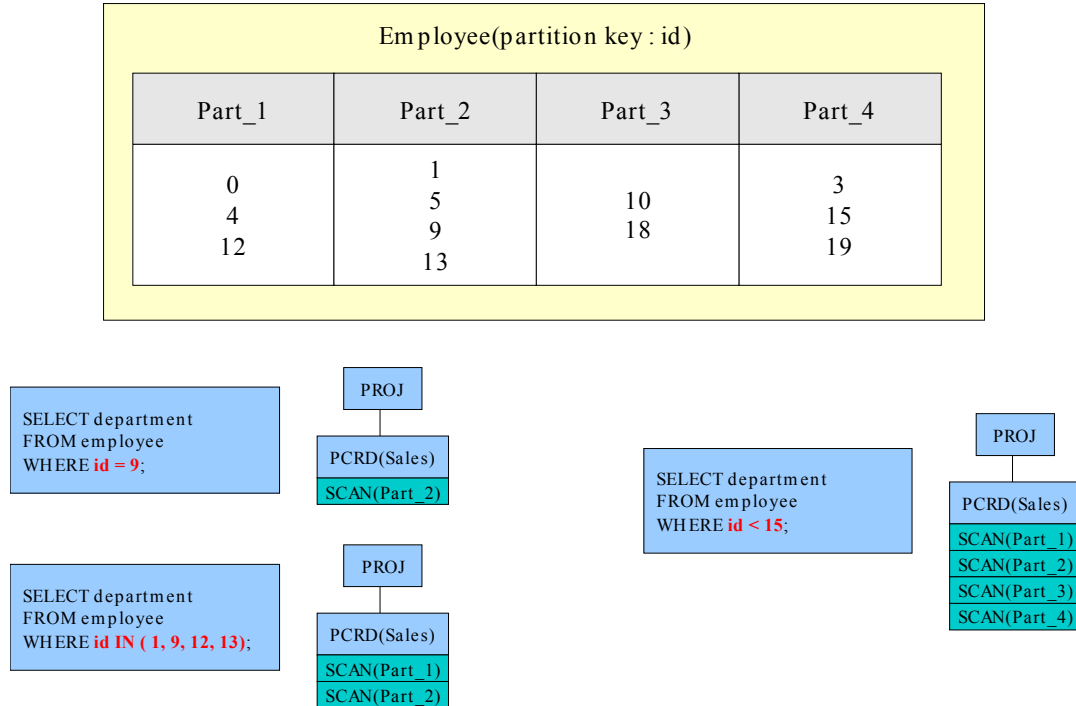
During the optimization step, the predicate that references the partition key *depart* is used in partition pruning, and thus the plan tree specifies that only the partitions that contain records that meet the predicate conditions are to be searched.

11.2.11.3 Hash Partition Pruning

The predicates with which hash partition pruning is possible have the same form as conditions for which searching can be performed using indexes, and can only have the equality (=) operator. Hash partition pruning is not possible for join conditions or for conditions that contain host variables.

Examples of such predicates include:

- `i1 = 1`
- `i1 in (4,5,6)`
- `i1 IS NULL`

Figure 11-7 An Example of Hash Partition Pruning

The top half of [Figure 10-7] depicts the *Employee* table, which is partitioned into 4 partitions based on *id*.

During the optimization step, the predicate that references the partition key *id* is used in partition pruning, and thus the plan tree specifies that only the partitions that contain records that meet the predicate conditions are to be searched.

11.2.11.4 Partition Filtering

The predicates with which partition filtering is possible have the same form as predicates with which partition pruning is possible, and include join conditions and conditions that contain host variables.

Examples of such predicates include:

- I1 = ?
- T1.i1 = T2.i1
- I1 < ?

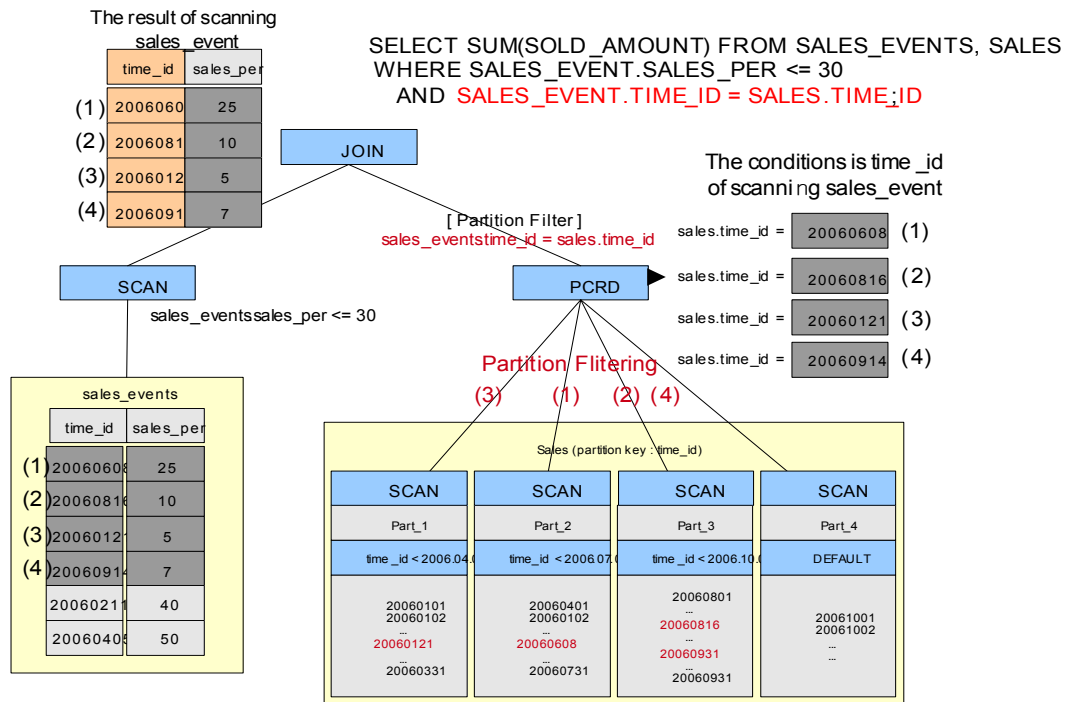
Figure 11-8 An Example of Partition Filtering

Figure [10-8] shows the partition filtering technique during execution. After records that meet the conditions are retrieved one by one from the sales_event table in the order (1), (2), (3), (4), the time_id column of these records becomes the sales_events.time_id variable predicate, which is used to determine which partitions contain records that satisfy the predicate condition.

Because the value (1) of the predicate falls within the Part_2 partition, which ranges between 2006.04.01 and 2006.06.29, partition filtering is performed, and thus only the Part_2 partition is searched for records that meet the conditions.

The same process is repeated for the predicate values (2), (3) and (4) in order to locate and search partitions that contain records that satisfy the predicate condition. The step of finding partitions that satisfy predicate conditions is known as “partition filtering”.

11.2.12 Optimization of the Projection

“Projection” is the process of specifying, in the SELECT statement, which of the available columns are to appear in the result set, along with any desired calculated results.

11.2.12.1 Considering the Order of Operations

An operation that always returns a fixed value, such as that shown below, is executed only one time, and the result is output repeatedly upon subsequent executions.

```
SELECT 1 + 1 FROM T1;
```

ALTIBASE HDB converts expressions that include the four basic arithmetic operations and the like into post-fix notation for processing, which can have an effect on performance, depending on the

order in which the operations are performed.

For example, the following queries yield the same result:

```
Query 1: SELECT i1 * 2 * 3 FROM T1;
Query 2: SELECT i1 * (2 * 3) FROM T1;
```

In query 1, the order of operations is $[i1 * 2] * 3$, and thus the $(* 2)$ and $(* 3)$ operations must both be executed again every time the value of $i1$ changes. However, in query 2, the result of $(2 * 3)$, which is 6, is calculated first, and thus only the $(i1 * 6)$ expression needs to be re-evaluated when the value of $i1$ changes.

That is, performance may vary depending on how the order of operations is set. This is true not only for the main SELECT clause, but for all portions of an SQL query, including the WHERE clause.

```
Query 1: WHERE i1 + 3 > 100;
Query 2: WHERE i1 > 100 - 3;
```

Of the two queries shown above, query 2 can be expected to realize better performance than query 1.

There are some fine points to be kept in mind when modifying formulas within queries in this way. Supposing that column $i1$ is an INTEGER type column, it would be more efficient to rewrite query 1 so that it looks like query 3 below:

```
Query 1: WHERE i1 * 3 > 100;
Query 2: WHERE i1 > 100 / 3;
Query 3: WHERE i1 > 33;
```

If query 1 were modified so that it resembled query 2, $100/3$ would evaluate to a real number, and thus it would be impossible to use an index with column $i1$. However, even though query 3 is not mathematically equivalent to query 1, the same results can be expected, because $i1$ is an INTEGER type column. Moreover, query 3 allows the use of an index.

As was seen above, carefully considering the order of operations and the data type can yield performance improvements without compromising the consistency of results.

Optimization of the projection consists mainly of optimization of any SELECT statement that is present within a subquery, which will be described in detail in the section that follows, which pertains to subquery optimization.

11.2.13 Subquery Optimization

Together with joins, subqueries have the greatest effect on performance.

The objective here is to minimize the use of subqueries generally, and further to eliminate those subqueries that are found to be unnecessary. Consider the following example:

```
SELECT * FROM T1
WHERE (T1. i1 = 1 AND
      T1.i2 IN ( SELECT a2 FROM T2
                WHERE T2.a1 = T1.i1 ) )
OR (T1. i1 = 2 AND
    T1.i2 IN ( SELECT a2 FROM T2
                WHERE T2.a1 = T1.i1 ) );

SELECT * FROM T1
WHERE T1. i1 IN (1, 2)
```

11.2 Query Optimization Procedures

```
AND T1.i2 IN ( SELECT a2 FROM T2
               WHERE T2.a1 = T1.i1 );
```

The above two queries are computationally equivalent. It is important to eliminate unnecessary subqueries so as to reduce the total number of subqueries.

The optimizer considers a wide variety of methods when optimizing a subquery, and chooses the one that is estimated to be the least costly while still ensuring mathematical consistency. A thorough understanding of the optimization methods implemented in the optimizer can help the user make modifications to queries that will greatly improve performance.

When the optimizer determines how to execute a subquery, it checks whether each of the following optimization methods can be used, and predicts the cost of execution.

- Removing columns from a SELECT Subquery within an EXISTS or UNIQUE Clause
- Eliminate Quantified Predicates
- NJ Transformation
- Invariant Portion of Subquery
- Store And Search
- Subquery Key Range

11.2.13.1 Subquery Type

Before explaining how the optimizer optimizes subqueries, the different subquery types will be described. Understanding each of the subquery types is very important, because different optimization methods are used for each type of subquery.

Subqueries are classified into the following four types depending on whether they reference external columns and on whether they contain so-called “group operations”. For the purposes of this document, the term “group operation” should be understood to include GROUP BY and HAVING clauses and aggregate operations.

| | No external columns are referenced. | One or more external columns are referenced. |
|---|-------------------------------------|--|
| No group operations are present. | Type N (No) | Type J (Join) |
| One or more group operations are present. | Type A (Aggregation) | Type JA (Join+Aggregation) |

- Type N: As shown in the following example, this type of subquery does not reference any external columns, and contains no group operations.

```
SELECT * FROM T1
WHERE T1.i1 IN ( SELECT T2.a1 FROM T2 );
```

- Type J: As shown below, this type of subquery references at least one external column but does not contain any group operations.

```
SELECT * FROM T1
WHERE EXISTS ( SELECT * FROM T2 WHERE T2.a1 = T1.i1 );
```

- Type A: As shown below, this type of subquery does not reference any external columns, but contains at least one group operation.

```
SELECT * FROM T1
WHERE i1 > ( SELECT SUM(a1) FROM T2 );
```

- Type JA: As shown below, this type of subquery references at least one external column and contains at least one group operation.

```
SELECT * FROM T1
WHERE i1 IN ( SELECT a1 FROM T2
              WHERE T2.a2 = T1.i2
              GROUP BY a1
              HAVING SUM(a2) > 0 );
```

Because type N and type A subqueries do not reference any external columns, they will always return consistent results. In contrast, type J and type JA subqueries must be executed again whenever external columns are modified.

That means that if a query can be written so that its subqueries do not reference external columns, better performance can be expected.

The optimizer determines which optimization method to use based on the above subquery types.

11.2.13.2 Removing SELECT Lists from EXISTS or UNIQUE Subqueries

It is not necessary to use a SELECT list in an EXISTS or UNIQUE subquery.

The optimizer removes such columns while optimizing the subquery, resulting in a query that has the meaning shown below:

```
SELECT * FROM T1
WHERE EXISTS ( SELECT a1, a2 * 1, a3 + a4 FROM T2 );
SELECT * FROM T1
WHERE EXISTS ( SELECT 1 FROM T2 );
```

However, when this optimization method is used, consistent results are guaranteed only for type N and type J subqueries, whereas the results may be inconsistent if it is used with type A or type JA subqueries.

Consider for example the following query statements:

```
SELECT * FROM T1
WHERE EXISTS ( SELECT COUNT(*) FROM T2 );
SELECT * FROM T1
WHERE EXISTS ( SELECT 1 FROM T2 );
```

At first glance, it seems that the two queries shown above would return the same result, but if there were no records in table T2, the queries would return different results.

It is permissible to modify queries in order to improve performance as long as the user keeps this point in mind and ensures that consistent results are still returned.

11.2 Query Optimization Procedures

11.2.13.3 Eliminating Quantified Predicates

If a quantified predicate, such as ANY or ALL, is used together with an inequality operator in a subquery, the quantified predicate can be removed, as shown below. This optimization method is called the “eliminate quantified predicate” method.

```
SELECT * FROM T1 WHERE i1 >=ANY ( SELECT a1 FROM T2 );  
SELECT * FROM T1 WHERE i1 >= ( SELECT MIN(a1) FROM T2 );
```

```
SELECT * FROM T1 WHERE i1 >ALL ( SELECT a1 FROM T2 );  
SELECT * FROM T1 WHERE i1 > ( SELECT MAX(a1) FROM T2 );
```

Because the optimized queries check only one value, rather than all records, against the MIN and MAX expressions, better performance can be expected. In addition, when a query is modified in this way, it is also possible to use an index to improve performance, in the same way as the “MIN/MAX Optimization using Sort Order” optimization method described earlier.

However, because the results can be inconsistent in the case where there are no records in table T2, the optimizer does not always modify subqueries in this way. This method can be used only with type A and type J subqueries. The optimizer decides whether to use this subquery optimization method after estimating the cost.

Therefore, if the user understands the above optimization method and is sure that the results will be consistent, the user can improve performance by explicitly modifying the query, rather than leaving the decision to the optimizer.

11.2.13.4 N to J Transformation

When the so-called “NJ Transformation” method is used, a type N subquery is changed into a type J subquery, or a type J subquery is modified slightly.

Consider the following example:

```
SELECT * FROM T1  
  WHERE i1 IN ( SELECT a1 FROM T2 );  
SELECT * FROM T1  
  WHERE i1 IN ( SELECT a1 FROM T2 WHERE T2.a1 = T1.i1 );
```

In the above example, the original query is a type N subquery that does not refer to an external column, but is modified using the N to J Transformation method so that it references an external column.

The following example shows how a type J subquery is changed. An additional predicate is added and pushed down.

```
SELECT * FROM T1  
  WHERE i1 IN ( SELECT a1 FROM T2  
                WHERE T2.a2 = 1 );  
SELECT * FROM T1  
  WHERE i1 IN ( SELECT a1 FROM T2  
                WHERE T2.a1 = 1  
                AND T2.a1 = T1.i1 );
```

With this optimization method, performance is improved by using the index that has been defined for the T2.a1 column. When the T2.a1 column and T1.i1 column are known not to contain any NULL values, this optimization method can be applied. In such cases, the optimizer estimates the cost of execution and determines whether to use this method.

This optimization method is usually efficient in cases where it is expected that the amount of data returned by the subquery will be greater than the amount of data returned by the main query. As long as the user is certain that the results will be consistent despite the query modification, a great performance improvement can be expected when the query is explicitly modified in this way.

11.2.13.5 Invariant Area of Subquery

Subqueries that reference external columns, that is, type J and type JA subqueries, do not always need to be repeatedly executed in their entirety.

Consider the following example:

```
SELECT * FROM T1
WHERE i2 > ( SELECT SUM(T2.a2 + T3.x2)
            FROM T2, T3
            WHERE T2.a1 = T3.x1
              AND T3.x1 = 1
              AND T2.a1 = T1.i1 );
```

The subquery in the above example is a type JA subquery, as it references the T1.i1 column, but the condition (T3.x1 = 1) does not need to be executed repeatedly. That is, the intermediate result output by this condition can always be reused, and is thus known as the “invariant area” of the subquery.

The optimizer identifies such invariant areas and determines through cost estimation whether it is worthwhile to store and reuse these invariant areas. The user can determine whether this optimization method is being used by analyzing the execution plan.

The related analysis method is described in the section entitled “[Analyzing Execution Plans](#)”.

11.2.13.6 Store and Search

All areas of type N and type A subqueries are invariant. The optimization method that takes advantage of this fact by storing all of the results of a subquery and using them to retrieve records is called the “Store and Search” optimization method.

Consider the following example:

```
SELECT * FROM T1
WHERE i1 IN ( SELECT SUM(a2)
            FROM T2
            GROUP BY a1 );
```

When this method is used, the results of the subquery are stored and searched to determine whether they include the value of i1.

This optimization method can only be used with type N and type A subqueries, and is an efficient method if there is no index for the i1 column.

Of course, the optimizer checks whether this optimization method is possible and estimates the cost when determining whether to use this method.

11.2.13.7 Subquery Key Range

Under circumstances in which the store and search optimization method can be used, if an index

11.2 Query Optimization Procedures

has been defined for the *i1* column, the stored results can be used in conjunction with the index. This is called the “Subquery Key Range” optimization method.

```
SELECT * FROM T1
  WHERE i1 IN ( SELECT SUM(a2)
                FROM T2
                GROUP BY a1 );
```

The subquery key range optimization method is not always efficient. When processing type N subqueries, the following factors should be kept in mind:

```
SELECT * FROM T1
  WHERE i1 IN ( SELECT a1 FROM T2 );
```

Depending on the data, the performance that can be realized when using the index for the *T1.i1* column may vary compared to the performance that can be realized when using the N to J transformation method, which was described above, together with the index for the *T2.a1* column.

Generally, the Subquery Key Range optimization method is efficient when the amount of data retrieved by the main query is high, whereas the N to J Transformation method is efficient when the amount of data retrieved by the subquery is high.

In other words, using an index to process query ranges containing large amounts of data is more likely to realize good performance. The optimizer chooses between the above two methods on the basis of cost estimation, but users can also explicitly modify the query to improve performance.

What follows is an example of how a query can be modified so that the optimization method desired by the user will be applied:

```
SELECT * FROM T1
  WHERE i1 IN ( SELECT a1 FROM T2 );
```

When it is desired to apply the Subquery Key Range optimizer method, the query would be altered thus:

```
SELECT * FROM T1
  WHERE i1 IN ( SELECT DISTINCT a1 FROM T2 );
```

When it is desired to apply the NJ Transformation optimizer method, the query would be altered thus:

```
SELECT * FROM T1
  WHERE i1 IN ( SELECT a1 FROM T2 WHERE T2.a1 = T1.i1 );
```

Changing queries in this way can encourage the optimizer to use a desired optimization method, but does not guarantee that it will. Additionally, when modifying queries, ensuring that the consistency of results is not compromised is of paramount importance.

11.2.13.8 Converting Subqueries to Joins

Under certain circumstances, query performance can be improved by changing subqueries, such as *IN* or *EXISTS*, into joins. However, because transforming this kind of subquery into a join in this way does not necessarily improve performance, and because the consistency of results can be compromised, caution is required.

In the example shown below, the *IN* subquery can be changed into a join only if the *UNIQUE* constraint has been set for the *T2.a1* column.

```

SELECT * FROM T1
  WHERE T1.i1 IN ( SELECT T2.a1 FROM T2 );
SELECT T1.* FROM T1, T2
  WHERE T1.i1 = T2.a1;

```

In the same way, the EXISTS subquery shown below can be changed into a join as shown only if the UNIQUE constraint has been set for the *T2.a1* column.

```

SELECT * FROM T1
  WHERE EXISTS ( SELECT * FROM T2
                  WHERE T2.a1 = T1.i1 );
SELECT T1.* FROM T1, T2
  WHERE T1.i1 = T2.a1;

```

So far, the optimization process and query tuning procedure have been described. How these query optimization methods have been implemented can be determined by analyzing the execution plan, which is described in detail in the next section.

11.3 Analyzing Execution Plans

When tuning SQL statements, the first task that is required is to analyze the execution plan tree. This section describes how to obtain and interpret information about the execution plan tree, as well as how to analyze information about each execution node that the execution plan tree comprises.

11.3.1 Analysis Methods

11.3.1.1 SQL Statements Used to Analyze the Execution Plan

Execution plan trees can be analyzed using iSQL. Execution plan trees only provide information pertaining to SELECT statements. In order to obtain information about an execution plan tree, the following command must be executed before the relevant SELECT statement is executed:

```
ALTER SESSION SET EXPLAIN PLAN = option;
```

The option can be set to ON, OFF or ONLY. The default setting is OFF.

- **ON:** After the SELECT statement is executed, the resultant records are displayed along with the execution plan.
- **ONLY:** The process of preparing the SELECT statement is conducted and the execution plan for the SQL statement is displayed, but the statement itself is not actually executed. This option is used when it is desired merely to check the execution plan for a SELECT statement to which one or more host variables are bound, or one that takes a long time to execute.
- **OFF:** After the SELECT statement is executed, the resultant records are displayed, but no execution plan information is provided.

To obtain more detailed information on how the conditions in a WHERE clause written by a user are processed, use the following command:

```
ALTER SYSTEM SET TRCLOG_DETAIL_PREDICATE = 1;
```

When the property is set to 1, as shown above, which signifies "ON", information about how the conditions in the WHERE clause are processed, such as whether fixed key range processing, variable key range processing, or filter processing is used, are displayed. This allows the user to check which predicates are executed using index scanning in the case of queries that have complicated WHERE clauses. Note, however, that this information might not be output in cases where the query has been changed in order to optimize it.

Consider for example the output of the query statement shown below:

```
iSQL> alter system set trclog_detail_predicate = 1;
Alter success.
iSQL> alter session set explain plan = on;
Alter success.
iSQL> select * from t1 where i1 = 1;
T1.I1
-----
1
1 row selected.
```

[When TRCLOG_DETAIL_PREDICATE = 1 and EXPLAIN PLAN = ON]

```

PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 4 )
  SCAN ( TABLE: T1, INDEX: IDX1, ACCESS: 1, SELF_ID: 2 )
    [ FIXED KEY ]
  OR
  I1 = 1

```

[When TRCLOG_DETAIL_PREDICATE = 0 and EXPLAIN PLAN = ON]

```

PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 4 )
  SCAN ( TABLE: T1, INDEX: IDX1, ACCESS: 1, SELF_ID: 2 )

```

[When TRCLOG_DETAIL_PREDICATE = 0 and EXPLAIN PLAN = ONLY]

```

PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 4 )
  SCAN ( TABLE: T1, INDEX: IDX1, ACCESS: ??, SELF_ID: 2 )

```

When EXPLAIN PLAN = ONLY, the execution plan is created, but the query is not actually executed. Therefore, the values of items that are normally determined after the query is executed, such as the number of records that were accessed ("ACCESS"), are displayed using question marks ('??').

11.3.1.2 Interpreting Execution Plan Trees

What follows is a brief explanation of how to interpret an execution tree, with reference to an example. The order of execution varies depending on the overall structure of the execution plan tree, which comprises individual execution nodes that are connected in tree form.

Example

```

SELECT * FROM T1, T2, T3
WHERE T1.I1 = T2.I1 AND T2.I1 = T3.I1
In case execution plan of query is as follows:

[EXECUTION PLAN]
6----- PROJECT ( COLUMN_COUNT: 9, TUPLE_SIZE: 36 ) ——— Top node
5----- JOIN ( REF_ID: 3 )
3----- JOIN ( REF_ID: 2 )
1----- SCAN ( TABLE: T1, FULL SCAN, ACCESS: 8, SELF_ID: 1 )
2----- SCAN ( TABLE: T2, INDEX: IDX2, ACCESS: 104, SELF_ID: 2 )
4----- SCAN ( TABLE: T3, INDEX: IDX3, ACCESS: 1144, SELF_ID: 3 )

```

Bottom node

Each individual node in the execution plan tree is displayed on its own row. The further the node is indented from the left, that is, the further it is located towards the right, the lower the node, and the earlier it is executed relative to other nodes.

In the above example, the PROJECT node is the root node, while the SCANS of tables T1 and T2 are the lowest nodes (also known as "leaf nodes"). In the case where multiple nodes are indented the same amount, such as the T1 and T2 scan nodes, the node that appears first will be handled as the higher node, that is, the left node.

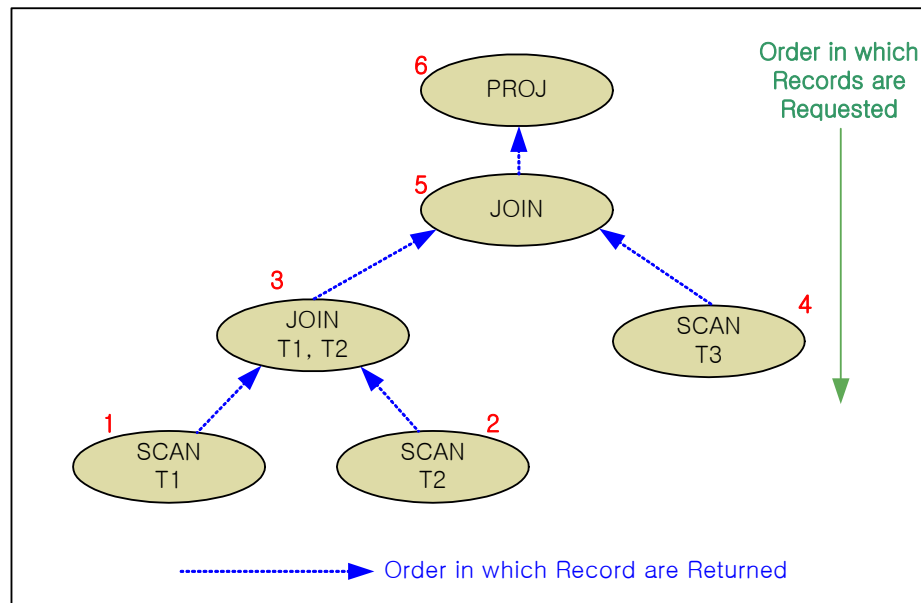
The request to fetch records is handled in a top-down manner, whereas the records are returned in a bottom-up manner.

In the above example, the node that accesses the database first is the node that scans table T1, followed in succession by the nodes that scan tables T2 and T3. The numbers shown in red to the left of each node indicate the order of execution.

11.3 Analyzing Execution Plans

The following figure shows the execution plan described above in tree diagram form.

Figure 11-9 Sequence in which Records are Requested and Fetched



11.3.2 Types of Execution Nodes

An execution plan tree comprises various kinds of nodes. These execution nodes are also known as "physical operators". Execution nodes are the elements that actually execute queries. As described above, the flow of execution can be checked by examining the execution plan tree.

Execution nodes are classified as shown below according to the number of child nodes they possess and by whether they store intermediate results:

- **Unary Non-Materialization Node:**
Has one child node, if any, and does not store intermediate results.
- **Unary Materialization Node**
Has one child node, if any, and stores intermediate results.
- **Binary Non-Materialization Node**
Has two child nodes and does not store intermediate results.
- **Binary Materialization Node**
Has two child nodes and stores intermediate results.
- **Multiple Non-Materialization Node**
Has one or more child nodes and does not store intermediate results.

According to this classification, there are 26 kinds of physical operators in ALTIBASE HDB, which are

shown below. Hereinafter, the abbreviation for each execution node type will be used in the descriptions of execution plan trees.

Table 11-13 Types of Execution Nodes

| Classification | Abbreviated Name | Display Name | Function |
|-------------------------|------------------|-------------------|---|
| Single Non-Storage Node | SCAN | SCAN | Retrieving data from tables using various access path methods |
| | FILT | FILTER | Filtering out data that can't be filtered out by the access path method |
| | PROJ | PROJECT | Processing projection |
| | GRBY | GROUPING | Processing grouping |
| | AGGR | AGGREGATION | Performing aggregate operations |
| | VIEW | VIEW | Organizing records into views |
| | VSCN | VIEW-SCAN | Retrieving data from view |
| | COUNT | COUNT | Processing specifically for COUNT(*) |
| | HIER | HIER | Processing hierarchical queries |
| Single Storage Node | SORT | SORT | Sorting records |
| | HASH | HASH | Hashing records |
| | GRAG | GROUP-AGGREGATION | Grouping using hashing and performing aggregate functions |
| | HSDS | DISTINCT | Discarding redundant records using hashing |
| | VMTR | MATERIALIZATION | Managing views that are stored in temporary tables |
| | STOR | STORE | Storing records |
| | LMST | LIMIT-SORT | Sorting for limit clauses |

11.3 Analyzing Execution Plans

| Classification | Abbreviated Name | Display Name | Function |
|-----------------------------------|------------------|-----------------------|--|
| Binary Non-Storage Node | JOIN | JOIN | Processing joins |
| | MGJN | MERGE-JOIN | Processing merge joins |
| | LOJN | LEFT-OUTER-JOIN | Processing LEFT OUTER joins |
| | FOJN | FULL-OUTER-JOIN | Processing FULL OUTER joins |
| | AOJN | ANTI-OUTER-JOIN | Processing ANTI OUTER joins |
| | CONC | CONCATENATION | Combining results returned by child nodes |
| | BUNI | BAG-UNION | Processing BAG UNIONS |
| Binary Materialization Node | SITS | SET-INTERSECT | Performing SET INTERSECT operations |
| | SDIF | SET-DIFFERENCE | Performing SET DIFFERENCE operations |
| Multiple Non-Materialization Node | PCRD | PARTITION-COORDINATOR | Managing scans of partitions of partitioned tables |

11.3.2.1 Features of Materialization Nodes

A materialization node is a node that stores intermediate results in order to perform its operations. Materialization nodes store intermediate results in temporary tables, which are classified as either memory temporary tables or disk temporary tables depending on the storage medium.

Memory temporary tables are stored in separately allocated memory, which is released immediately after the query has finished executing. Disk temporary tables are stored in user-defined temporary tablespaces, and the data therein are managed using a memory buffer. This memory buffer is also released when the query has finished executing.

The determination of whether to use a memory temporary table or a disk temporary table is made as follows: a memory temporary table is used if all of the nodes below a materialization node use only memory tables. If one or more disk tables are used, a disk temporary table is used. This decision rule can be overruled using a hint.

In the following examples, it can be seen in the execution plans that sorting is handled differently depending on whether memory or disk temporary tables are used.

The execution plan for the query below contains two nodes that store intermediate results in memory tables for the purposes of removing redundancies and sorting records. Of course, when memory temporary tables are used, the information about the number of disk pages is not provided.


```

iSQL> select distinct i3 from memory_table order by i3;
i3
-----
0
1
2
3
4
5 rows selected.

-----
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 4 )
  SORT ( ITEM_SIZE: 24, ITEM_COUNT: 5, ACCESS: 5, SELF_ID: 2, REF_ID: 0 )
    DISTINCT ( ITEM_SIZE: 24, ITEM_COUNT: 5, BUCKET_COUNT: 1024,
      ACCESS: 5, SELF_ID: 1, REF_ID: 0 )
      SCAN ( TABLE: MEMORY_TABLE, FULL SCAN, ACCESS: 16384, SELF_ID: 0 )
-----

```

In the following example, there are two materialization nodes that store intermediate results in disk tables in order to remove redundancies and sort records. When a temporary disk table is used, the information about the number of disk pages is present. That is, whether the temporary tables are stored in memory or on disk can be verified by observing whether the `DISK_PAGE_COUNT` information appears in the execution node.

```

iSQL> select distinct i3 from disk_table order by i3;
i3
-----
0
1
2
3
4
5 rows selected.

-----
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 4 )
  SORT ( ITEM_SIZE: 32, ITEM_COUNT: 5, DISK_PAGE_COUNT: 3,
    ACCESS: 5, SELF_ID: 2, REF_ID: 0 )
    DISTINCT ( ITEM_SIZE: 24, ITEM_COUNT: 5, DISK_PAGE_COUNT: 6,
      ACCESS: 5, SELF_ID: 1, REF_ID: 0 )
      SCAN ( TABLE: DISK_TABLE, FULL SCAN, ACCESS: 16384,
        DISK_PAGE_COUNT: 28, SELF_ID: 0 )
-----

```

All materialization nodes contain information that helps determine whether the data stored in temporary tables need to be refreshed. This information is indicated by `REF_ID`. If a record returned by the execution node corresponding to a particular `REF_ID` is modified, that temporary table is refreshed to reflect the current data. In the above example, it can be seen that it is not necessary to refresh a temporary table.

11.3 Analyzing Execution Plans

11.3.3 Unary Non-Materialization Nodes

A unary non-materialization node comprises only one child node, if any, manages only a single record, and does not require any additional storage area.

Below, the function of each kind of execution node having these characteristics will be explained, along with how to analyze them in an execution plan.

11.3.3.1 SCAN Node

A so-called "SCAN node" is a physical entity that executes SELECT operations according to the relational model. A SCAN node has no child nodes. It directly accesses a table and retrieves records from the table.

If there are one or more predicates related to the table, one of the following five methods is used:

- Fixed key range
- Variable key range
- Constant filter
- Filter
- Subquery filter

The following SCAN node information can be observed by setting EXPLAIN PLAN to ON (or ONLY):

Table 11-14 Scan Node Information

| Item | Description | Remarks |
|-----------------|---|--|
| SCAN | The name of the execution node | |
| TABLE | The name of the table being accessed | |
| ACCESS | The number of records that were actually accessed | |
| DISK_PAGE_COUNT | The number of disk pages in the table | No information is provided for memory tables |
| SELF_ID | The identifier of the execution node | |

Memory Tables vs. Disk Tables

Memory tables and disk tables differ slightly with regard to the information that is displayed. The number of disk pages occupied by the table is displayed only for disk tables.

Below, the information displayed for a SCAN node pertaining to a memory table is compared with that for a SCAN node pertaining to a disk table.

The following example shows the information displayed for a SCAN node pertaining to a memory table in the case where the default tablespace for the SYS user is set to SYS_TBS_MEMORY.

```

iSQL> create table m1 ( i1 integer );
Create success.
iSQL> select * from m1;
M1.I1
-----
No rows selected.
-----
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 4 )
SCAN ( TABLE: M1, FULL SCAN, ACCESS: 0, SELF_ID: 0 )
-----

```

The example below shows the information displayed for a SCAN node pertaining to a table that was explicitly created in the SYS_TBS_DATA tablespace, which is a disk tablespace. The number of disk pages occupied by the table is displayed.

```

iSQL> create table d1 ( i1 integer ) tablespace sys_tbs_data;
Create success.
iSQL> select * from d1;
D1.I1
-----
No rows selected.
-----
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 4 )
SCAN ( TABLE: D1, FULL SCAN, ACCESS: 0, DISK_PAGE_COUNT: 2, SELF_ID: 0 )
-----

```

Table Name

In the example below, the name of the table accessed by the SCAN node is displayed. Additionally, the alias that was defined in the query can also be seen.

```

iSQL> select * from t1 v1;
V1.I1
-----
No rows selected.
-----
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 4 )
SCAN ( TABLE: T1 V1, FULL SCAN, ACCESS: 0, SELF_ID: 0 )
-----

```

Access Method and Number of Records Accessed

The most important information in query tuning is whether a full scan is performed or an index is used, along with the estimate of how many records will be accessed. The greater the number of records that are accessed, the lower the performance, so the estimate of the number of records that will be accessed is critical.

The example below shows the case where a query is processed using a full scan. The example shows the number of records accessed in order to perform the comparison in the WHERE clause.

11.3 Analyzing Execution Plans

```
iSQL> select * from t1 where i1 = 1000;
T1.i1    T1.i2
-----
1000      0
1 row selected.

PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 8 )
SCAN ( TABLE: T1, FULL SCAN, ACCESS: 16384, SELF_ID: 2 )
```

The next example shows the execution plan information that is displayed when an index is used to access the same table. The example shows that when *IDX1* is used, only one record is accessed in order to evaluate the condition in the WHERE clause.

```
iSQL> create index idx1 on t1(i1);
Create success.
iSQL> select * from t1 where i1 = 1000;
T1.i1    T1.i2
-----
1000      0
1 row selected.

PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 8 )
SCAN ( TABLE: T1, INDEX: IDX1, ACCESS: 1, SELF_ID: 2 )
```

It can be seen that the same execution plan is created to evaluate this condition even when a condition pertaining to another column is added.

```
iSQL> select * from t1 where i1 = 1000 and i2 = 0;
T1.i1    T1.i2
-----
1000      0
1 row selected.

PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 8 )
SCAN ( TABLE: T1, INDEX: IDX1, ACCESS: 1, SELF_ID: 2 )
```

If an index is added to the *T1.i2* column and a query is then executed using that index, the execution plan shown below will result. It can be seen that using the index for the *T1.i2* column degrades efficiency, even though the query is the same.

```

iSQL> create index idx2 on t1(i2);
Create success.
iSQL> select /*+ INDEX(T1, idx2) */ * from t1 where i1 = 1000 and i2 = 0;
T1.I1      T1.I2
-----
1000        0
1 row selected.

PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 8 )
SCAN ( TABLE: T1, INDEX: IDX2, ACCESS: 163, SELF_ID: 2 )

```

When no hints are given, as in the example shown below, the optimizer chooses the best index to use on the basis of a cost estimation.

```

iSQL> select * from t1 where i1 = 1000 and i2 = 0;
T1.I1      T1.I2
-----
1000        0
1 row selected.

PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 8 )
SCAN ( TABLE: T1, INDEX: IDX1, ACCESS: 1, SELF_ID: 2 )

```

As illustrated above, it is necessary for the user to view the SCAN node information to check whether a suitable access method was chosen, and, if necessary, to create an index that is appropriate for the query when no such index exists.

TRCLOG_DETAIL_PREDICATE

The TRCLOG_DETAIL_PREDICATE property is set to 1, that is, it is turned on, in order to output information about how the conditions are processed by the SCAN node. This property is useful when checking whether a condition is using an index.

In the following example, because this property has been enabled, the method that is used to process each condition is shown. It can be seen that the ($i1 = 1000$) condition is processed as a fixed key range that uses an index, and that the ($i2 = 0$) condition is processed as a filter.

11.3 Analyzing Execution Plans

```
iSQL> alter system set trclog_detail_predicate = 1;
Alter success.
iSQL> alter session set explain plan = on;
Alter success.
iSQL> select * from t1 where i1 = 1000 and i2 = 0;
T1.I1    T1.I2
-----
1000      0
1 row selected.

PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 8 )
SCAN ( TABLE: T1, INDEX: IDX1, ACCESS: 1, SELF_ID: 2 )
[ FIXED KEY ]
AND
OR
  I1 = 1000
[ FILTER ]
AND
OR
  I2 = 0
```

A different index is used to process the same query in the example below. It can be seen that when the *IDX2* condition is used, the condition that is processed using the index and the condition that is processed without using an index are switched.

```
iSQL> select /*+ INDEX(T1, idx2) */ * from t1 where i1 = 1000 and i2 = 0;
T1.I1    T1.I2
-----
1000      0
1 row selected.

PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 8 )
SCAN ( TABLE: T1, INDEX: IDX2, ACCESS: 163, SELF_ID: 2 )
[ FIXED KEY ]
AND
OR
  I2 = 0
[ FILTER ]
AND
OR
  I1 = 1000
```

As seen above, determining whether a condition in a *WHERE* clause is processed using an index is of great help when tuning queries. However, this information might not be output if the query is changed by the optimizer during the course of optimization.

11.3.3.2 FILT (Filter) Node

A so-called “filt node” is a physical entity that filters out data. This kind of node has a single child node and checks the result set returned from the child node against the relevant predicates without directly accessing any tables.

The following filt node information can be observed by setting *EXPLAIN PLAN* to *ON* (or *ONLY*):

Table 11-15 FILT Node Information

| Item | Description | Remarks |
|--------|--------------------------------|---------|
| FILTER | The name of the execution node | |

A FILT node does not provide any additional data beyond the name of the node. If the TRCLOG_DETAIL_PREDICATE property is turned on (i.e. set to 1), information about the predicates processed by the filter is displayed.

Displaying FILT Node Information

As can be seen below, the only information that is displayed for a FILT node is the name of the node; no other information is shown. In this example, the FILT node is used to process the (HAVING i2 < 2) condition.

```
iSQL> select sum(i1) from t1 group by i2 having i2 < 2;
SUM(i1)
-----
1336600
1336764
2 rows selected.

-----
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 8 )
FILTER
AGGREGATION ( ITEM_SIZE: 16, GROUP_COUNT: 100 )
GROUPING
SCAN ( TABLE: T1, INDEX: IDX2, ACCESS: 16384, SELF_ID: 2 )
-----
```

The TRCLOG_DETAIL_PREDICATE property must be set to 1 in order to examine the FILT node more closely. In the example below it can be seen that the FILT node is used to process the (i2 < 2) condition.

```
iSQL> alter system set trclog_detail_predicate = 1;
Alter success.
iSQL> select sum(i1) from t1 group by i2 having i2 < 2;
SUM(i1)
-----
1336600
1336764
2 rows selected.

-----
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 8 )
FILTER
[ FILTER ]
AND
OR
i2 < 2
AGGREGATION ( ITEM_SIZE: 16, GROUP_COUNT: 100 )
GROUPING
SCAN ( TABLE: T1, INDEX: IDX2, ACCESS: 16384, SELF_ID: 2 )
-----
```

11.3 Analyzing Execution Plans

11.3.3.3 PROJ (Projection) Node

A so-called “PROJ node” is a physical entity that executes projection operations according to the relational model. This node comprises a single child node, and extracts required columns from the result set returned by the child node.

The following proj node information can be observed by setting EXPLAIN PLAN to ON (or ONLY):

Table 11-16 PROJ Node Information

| Item | Description | Remarks |
|--------------|---------------------------------|---------|
| PROJECT | The name of the execution node | |
| COLUMN_COUNT | Number of projected columns | |
| TUPLE_SIZE | Size of records to be projected | |

Displaying PROJ Node Information

A PROJ node is the node that formats the final result of a query. The information about a PROJ node that is displayed in the execution plan is shown below. It can be seen that the query result comprises two columns and that the size of each of the resultant record is 8 bytes.

```
iSQL> select * from t1 where i1 = 1000;
T1.I1    T1.I2
-----
1000     0
1 row selected.

PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 8 )
SCAN ( TABLE: T1, INDEX: IDX1, ACCESS: 1, SELF_ID: 2 )
```

11.3.3.4 GRBY (GROUP BY) Node

A so-called “GRBY node” is a physical entity that performs sort-based grouping according to the relational model. This kind of node comprises a single child node, and examines the records returned by the child node to determine whether the previous record and the current record belong in the same group.

This kind of node is used to when queries that perform the following operations are executed:

- Identifying whether two records belong in the same group using the sort order
- Removing redundancies using the sort order
- Processing DISTINCT aggregate operations using the sort order

The following grby node information can be observed by setting EXPLAIN PLAN to ON (or ONLY):

Table 11-17 GRBY Node Information

| Item | Description | Remarks |
|----------|--------------------------------|---------|
| GROUPING | The name of the execution node | |

No information other than the node name is displayed for GRBY nodes.

Grouping using the Sort Order

When a GRBY node is used to check whether two records belong in the same group using the sort order, the execution plan appears as shown below. The example below shows that the GRBY node is used to process the (GROUP BY I3) clause, which underlies the distinct i3 clause shown, without requiring any additional storage space. This was previously noted in the description of how the sort order can be used to process a GROUP BY clause during optimization. (Please refer to [GROUP BY Using Sort Order](#).)

```
iSQL> select distinct i3 from t1;
i3
-----
0
1
2
3
4
5 rows selected.

-----
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 4 )
GROUPING
SCAN ( TABLE: T1, INDEX: IDX3, ACCESS: 16384, SELF_ID: 0 )
-----
```

Removing Redundant Values Using the Sort Order

When a GRBY node is used along with the sort order to eliminate redundant values, the execution plan has the form shown below. The example below shows that the GRBY node was used to process the (DISTINCT i3) clause without using any additional storage space. This was already explained in the description of how to use the sort order to optimize DISTINCT clauses (Please refer to [Optimization of DISTINCT Clause](#).)

```
iSQL> select distinct i3 from t1;
i3
-----
0
1
2
3
4
5 rows selected.

-----
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 4 )
GROUPING
SCAN ( TABLE: T1, INDEX: IDX3, ACCESS: 16384, SELF_ID: 0 )
-----
```

Processing of Distinct Aggregation Using the Sort Order

When a GRBY node is used along with the sort order to process a DISTINCT clause within an aggregate function, the execution plan that results is as shown below. The example below shows that the GRBY node is used to eliminate redundancies, which is stipulated by the (DISTINCT i2) clause, when (COUNT(DISTINCT i2)) is processed. This was already explained in the description of the use of the sort order to optimize clauses in which aggregate functions are executed on clauses containing the DISTINCT keyword (please refer to [Optimization of DISTINCT Clause](#).)

```
iSQL> select count(distinct i2) from t1;
COUNT(distinct i2)
-----
100
1 row selected.
-----
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 8 )
GROUP-AGGREGATION ( ITEM_SIZE: 24, GROUP_COUNT: 1,
                    BUCKET_COUNT: 1, ACCESS: 1,
                    SELF_ID: 3, REF_ID: 1 )

GROUPING
SCAN ( TABLE: T1, INDEX: IDX2, ACCESS: 16384, SELF_ID: 1 )
-----
```

11.3.3.5 AGGR (Aggregate) Node

A so-called “AGGR node” is a physical entity that performs aggregate operations according to the relational model. This node comprises a single child node and uses no additional space to store intermediate results. This node performs aggregate operations on records in the same group.

An aggregate node is used in the execution of queries that have the following characteristics:

- When the sort order is used to perform aggregate operations
- When aggregate operations are performed on clauses that contain the DISTINCT keyword

The following AGGR node information can be observed by setting EXPLAIN PLAN to ON (or ONLY):

Table 11-18 AGGR Node Information

| Item | Description | Remarks |
|-------------|--|---------|
| AGGREGATION | The name of the execution node | |
| ITEM_SIZE | The size of a record for one group | |
| GROUP_COUNT | The number of groups created by the execution node | |

Sort Order Based Aggregation

When an AGGR node is used along with the sort order to perform aggregate operations, execution plan information similar to that shown below is output. From the example below, it can be seen that the AGGR node uses data classified by the GRBY node to perform the SUM(i2) operation. These records are organized into groups using the SUM(i2) and GROUP BY i3 expressions. The example

below shows that there are five groups, each containing one or more 16-byte records.

```

iSQL> select sum(i2) from t1 group by i3;
SUM(i2)
-----
155530
158807
162084
165361
168638
5 rows selected.
-----
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 8 )
AGGREGATION ( ITEM_SIZE: 16, GROUP_COUNT: 5 )
GROUPING
  SCAN ( TABLE: T1, INDEX: IDX3, ACCESS: 16384, SELF_ID: 1 )
-----

```

Aggregation Containing DISTINCT

The AGGR node requires additional space to remove redundancies only in the case where a DISTINCT clause is contained within an aggregate function. The example below shows the use of an AGGR node to process the SUM(DISTINCT i2) clause.

```

iSQL> select sum( distinct i2 ) from t1 group by i3;
SUM( distinct i2 )
-----
950
970
990
1010
1030
5 rows selected.
-----
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 8 )
AGGREGATION ( ITEM_SIZE: 16, GROUP_COUNT: 5 )
GROUPING
  SCAN ( TABLE: T1, INDEX: IDX3, ACCESS: 16384, SELF_ID: 1 )
-----

```

11.3.3.6 VIEW Node

VIEW nodes are used to display virtual tables according to the relational model. This kind of node is used to display a user-defined view or make a result set created by a set operation appear as a single table.

The following VIEW node information can be observed by setting EXPLAIN PLAN to ON (or ONLY):

Table 11-19 VIEW Node Information

| Item | Description | Remarks |
|-----------|--------------------------------|--------------------------|
| VIEW | The name of the execution node | |
| View name | View name | When a name is available |

11.3 Analyzing Execution Plans

| Item | Description | Remarks |
|---------|--|---------|
| ACCESS | The number of records that are accessed using the view | |
| SELF_ID | The identifier of the execution node | |

When a user-defined view is queried, the output for the VIEW node appears as shown below. The subnodes of the VIEW node illustrate the execution plan for the SELECT statement that forms the basis of the user-defined view.

```
iSQL> create view v1(a1, a2) as select i3, sum(i2) from t1 group by i3;
Create success.
iSQL> select * from v1;
V1.A1      V1.A2
-----
0          155530
1          158807
2          162084
3          165361
4          168638
5 rows selected.

PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 16 )
VIEW ( V1, ACCESS: 5, SELF_ID: 2 )
  PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 16 )
    AGGREGATION ( ITEM_SIZE: 16, GROUP_COUNT: 5 )
      GROUPING
        SCAN ( TABLE: T1, INDEX: IDX3, ACCESS: 16384, SELF_ID: 1 )
```

VIEW nodes are also used for queries that use set operations, an example of which is shown below. A VIEW node is created in order to manage the results of an INTERSECT operation as though they were a single table. In this case, the view does not have its own name.

```
iSQL> select i1, i3 from t1 intersect select i3, i1 from t1;
i1          i3
-----
1           1
2           2
3           3
4           4
4 rows selected.

PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 8 )
VIEW ( ACCESS: 4, SELF_ID: 2 )
  SET-INTERSECT ( ITEM_SIZE: 24, ITEM_COUNT: 16384, BUCKET_COUNT: 8192,
ACCESS: 4, SELF_ID: 4, L_REF_ID: 0, R_REF_ID: 1 )
    PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 8 )
      SCAN ( TABLE: T1, FULL SCAN, ACCESS: 16384, SELF_ID: 0 )
    PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 8 )
      SCAN ( TABLE: T1, FULL SCAN, ACCESS: 16384, SELF_ID: 1 )
```

11.3.3.7 VSCN (View-Scan) Node

The so-called “VSCN node” is the physical entity that performs SELECT operations on temporarily stored views according to the relational model. If it has a child node, it is always a VMTR node.

This node is created during query optimization when it has been determined that it would be more efficient to store the contents of a view for processing.

The following VSCN node information can be observed by setting EXPLAIN PLAN to ON (or ONLY):

Table 11-20 VSCN Node Information

| Item | Description | Remarks |
|---------|---|---------|
| VSCN | The name of the execution node | |
| VIEW | The name of the view | |
| ACCESS | The number of records in the view that are accessed | |
| SELF_ID | The identifier of the execution node | |

An example of the use of a VSCN node is shown below. In the example below, the same view is accessed by both the main query and the subquery. The optimizer concluded that it would be more efficient to temporarily store the contents of the view, and thus they were stored using the VMTR node. The execution plan shows that the VSCN node accesses the contents of the view.

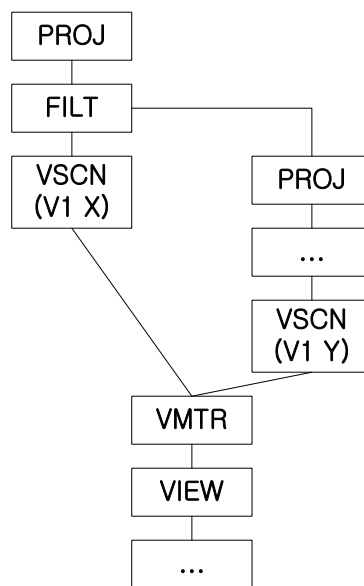
```

iSQL> select * from v1 x where x.a2 > ( select avg(y.a2) from v1 y );
X.A1      X.A2
-----
3          165361
4          168638
2 rows selected.
-----
PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 16 )
  FILTER
    ::SUB-QUERY BEGIN
      PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 22 )
        STORE ( ITEM_SIZE: 32, ITEM_COUNT: 1, ACCESS: 10, SELF_ID: 11, REF_ID: 4 )
          VIEW ( ACCESS: 1, SELF_ID: 10 )
            PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 22 )
              GROUP-AGGREGATION ( ITEM_SIZE: 64, GROUP_COUNT: 1,
                BUCKET_COUNT: 1, ACCESS: 1,
                SELF_ID: 9, REF_ID: 4 )
                VIEW-SCAN ( VIEW: V1 Y, ACCESS: 5, SELF_ID: 4 )
                  MATERIALIZATION ( ITEM_SIZE: 24, ITEM_COUNT: 5 )
                    VIEW ( ACCESS: 5, SELF_ID: 7 )
                      PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 16 )
                        AGGREGATION ( ITEM_SIZE: 16, GROUP_COUNT: 5 )
                          GROUPING
                            SCAN ( TABLE: T1, INDEX: IDX3, ACCESS: 16384, SELF_ID: 1 )
                        ::SUB-QUERY END
                      VIEW-SCAN ( VIEW: V1 X, ACCESS: 5, SELF_ID: 2 )
                    -----

```

In the execution plan shown above example, the VSCN node for the (V1 X) view does not seem to have any child nodes. However, this VSCN node has a VMTR node (displayed as “MATERIALIZATION”) as its child node. Part of the above execution plan is shown diagrammatically below:

11.3 Analyzing Execution Plans



The above diagram shows clearly that VSCN(V1 X) and VSCN(V1 Y) have the same child node.

11.3.3.8 COUNT Node

In the relational model, COUNT nodes are used specifically to process COUNT(*) operations in queries that do not contain GROUP BY clauses.

The COUNT node information that is shown by setting EXPLAIN PLAN to ON (or ONLY) is similar to that for a SCAN node, and is as shown below:

Table 11-21 COUNT Node Information

| Item | Description | Remarks |
|-----------------|---|--|
| COUNT | The name of the execution node | |
| TABLE | The name of the table being accessed | |
| ACCESS | The number of records that were actually accessed | |
| DISK_PAGE_COUNT | The number of disk pages in the table | No information is provided for memory tables |
| SELF_ID | The identifier of the execution node | |

An example of the use of a COUNT node is shown below. The example shows that the value of COUNT(*) is obtained without actually accessing the data, thanks to the use of an index.

```

iSQL> select count(*) from t1 where i1 > 1000;
COUNT
-----
15384
1 row selected.
-----
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 8 )
COUNT ( TABLE: T1, INDEX: IDX1, ACCESS: 1, SELF_ID: 2, REF_ID: 2 )
-----

```

11.3.3.9 HIER (Hierarchy) Node

HIER nodes execute hierarchical queries using special operations not found in the relational model. This node has a single child node, which is always a scan node.

The following HIER node information can be observed by setting EXPLAIN PLAN to ON (or ONLY). Most of the data for a HIER node is similar to that of a SCAN node.

Table 11-22 HIER Node Information

| Item | Description | Remarks |
|-----------------|---|---|
| HIER | The name of the execution node | |
| TABLE | The name of the table being accessed | |
| ACCESS | The number of records that were actually accessed | |
| DISK_PAGE_COUNT | The number of disk pages in the table | No information is provided for memory tables. |
| SELF_ID | The identifier of the execution node | |

An example of the use of a HIER node is shown below:

```

iSQL> select count(*) from t1 start with i3 = 0 connect by prior i3 = i1 ignore loop;
COUNT
-----
3276
1 row selected.
-----
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 8 )
GROUP-AGGREGATION ( ITEM_SIZE: 24, GROUP_COUNT: 1,
                     BUCKET_COUNT: 1, ACCESS: 1,
                     SELF_ID: 5, REF_ID: 2 )
HIER ( TABLE: T1, INDEX: IDX1, ACCESS: 3276, SELF_ID: 2 )
SCAN ( TABLE: T1, INDEX: IDX3, ACCESS: 3276, SELF_ID: 2 )
-----

```

11.3.4 Unary Materialization Nodes

Unary Materialization Nodes have one child node, if any, and require specially allocated storage space in order to execute their functions.

Below, the function of each kind of Unary Materialization Node will be explained, along with how to analyze them in an execution plan.

11.3.4.1 Node for SORT Execution

The so-called “SORT node” is the physical entity that performs sort operations according to the relational model. It has one child node, and uses a temporary table to save intermediate results.

The following SORT node information can be observed by setting EXPLAIN PLAN to ON (or ONLY):

Table 11-23 SORT Node Information

| Item | Description | Remarks |
|-----------------|--|--|
| SORT | The name of the execution node | |
| ITEM_SIZE | The size of the record to be sorted | |
| ITEM_COUNT | The number of records to be sorted | |
| DISK_PAGE_COUNT | The number of disk pages that the temporary table comprises | No information is provided for temporary memory tables |
| ACCESS | The number of records that are accessed | |
| SELF_ID | The identifier of the execution node | |
| REF_ID | A node identifier used to determine whether to refresh a temporary table | |

SORT nodes are used for various purposes. The following is a description of the execution plan tree for each purpose.

Used for ORDER BY Clause

A SORT node is used when an additional sorting operation is required in order to process an ORDER BY clause. The example below shows the use of a SORT node to process an ORDER BY clause.


```

iSQL> select i3, sum(i2) from t1 group by i3 order by 2;
i3      SUM(i2)
-----
0       155530
1       158807
2       162084
3       165361
4       168638
5 rows selected.
-----
PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 16 )
  SORT ( ITEM_SIZE: 24, ITEM_COUNT: 5, ACCESS: 5, SELF_ID: 5, REF_ID: 2 )
    AGGREGATION ( ITEM_SIZE: 16, GROUP_COUNT: 5 )
      GROUPING
        SCAN ( TABLE: T1, INDEX: IDX3, ACCESS: 16384, SELF_ID: 2 )
-----

```

Used for GROUP BY Clause

A SORT node can be created to perform sorting to classify records into groups when processing a GROUP BY clause. The example below shows a SORT node that was created to process the (GROUP BY i4) clause.

```

iSQL> select i4, sum(distinct i2) from t1 group by i4;
i4      SUM(distinct i2)
-----
0       950
1       970
2       990
3      1010
4      1030
5 rows selected.
-----
PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 16 )
  AGGREGATION ( ITEM_SIZE: 16, GROUP_COUNT: 5 )
    GROUPING
      SORT ( ITEM_SIZE: 16, ITEM_COUNT: 16384, ACCESS: 16384,
        SELF_ID: 2, REF_ID: 1 )
        SCAN ( TABLE: T1, FULL SCAN, ACCESS: 16384, SELF_ID: 1 )
-----

```

Used for DISTINCT Clause

A SORT node can be used for sort-based elimination of redundancies when processing a DISTINCT clause. The example below shows a SORT node that was created to process the (DISTINCT i4) clause.

11.3 Analyzing Execution Plans

```
iSQL> select DISTINCT i4 from t1;
i4
-----
0
1
2
3
4
5 rows selected.

-----
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 4 )
GROUPING
  SORT ( ITEM_SIZE: 16, ITEM_COUNT: 16384, ACCESS: 16384,
        SELF_ID: 1, REF_ID: 0 )
    SCAN ( TABLE: T1, FULL SCAN, ACCESS: 16384, SELF_ID: 0 )
-----
```

Used for Joins

A SORT node can be used to process joins.

The example below shows the use of a SORT node to process a join. A SORT execution node was created in order to use sort-based join processing to evaluate records against the (t1.i1 = t2.i1) join condition.

```
iSQL> select count(*) from t1, t2 where t1.i1 = t2.i1;
COUNT
-----
16384
1 row selected.

-----
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 8 )
GROUP-AGGREGATION ( ITEM_SIZE: 24, GROUP_COUNT: 1,
                    BUCKET_COUNT: 1, ACCESS: 1,
                    SELF_ID: 4, REF_ID: 2 )
JOIN
  SCAN ( TABLE: T1, FULL SCAN, ACCESS: 16384, SELF_ID: 1 )
  SORT ( ITEM_SIZE: 16, ITEM_COUNT: 16384, ACCESS: 16384,
        SELF_ID: 3, REF_ID: 2 )
    SCAN ( TABLE: T2, FULL SCAN, ACCESS: 16384, SELF_ID: 2 )
-----
```

11.3.4.2 HASH Node

A so-called “HASH node” performs hash operations according to the relational model. A HASH node has one child node and uses a temporary table to save intermediate results.

The following HASH node information can be observed by setting EXPLAIN PLAN to ON (or ONLY):

Table 11-24 HASH Node Information

| Item | Description | Remarks |
|------|--------------------------------|---------|
| HASH | The name of the execution node | |

| Item | Description | Remarks |
|-----------------|--|---|
| ITEM_SIZE | The size of each record to be hashed | |
| ITEM_COUNT | The number of records to be hashed | |
| BUCKET_COUNT | The number of hash buckets | |
| DISK_PAGE_COUNT | The number of disk pages that the temporary table comprises | No information is provided for temporary memory tables. |
| ACCESS | The number of records that are accessed | |
| SELF_ID | The identifier of the execution node | |
| REF_ID | A node identifier used to determine whether to refresh a temporary table | |

HASH nodes are used for various purposes. The following is a description of the execution plan tree for each purpose.

Used to Process Joins

A HASH node can be used to process a join.

The example below shows a HASH node that was created to process a join. The HASH node was created in order to use hash-based join processing to check records against the ($t1.i1 = t2.i1$) join condition.

```

ISQL> select count(*) from t1, t2 where t1.i1 = t2.i1;
COUNT
-----
16384
1 row selected.

PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 8 )
GROUP-AGGREGATION ( ITEM_SIZE: 24, GROUP_COUNT: 1,
                     BUCKET_COUNT: 1, ACCESS: 1,
                     SELF_ID: 4, REF_ID: 2 )

JOIN
SCAN ( TABLE: T1, FULL SCAN, ACCESS: 16384, SELF_ID: 1 )
HASH ( ITEM_SIZE: 24, ITEM_COUNT: 16384, BUCKET_COUNT: 1024,
      ACCESS: 16384, SELF_ID: 3, REF_ID: 2 )
SCAN ( TABLE: T2, FULL SCAN, ACCESS: 16384, SELF_ID: 2 )

```

Used to Process a Subquery Search

A HASH node can be used to perform a comparison operation within a subquery.

The example below shows the use of a HASH node to process the ($i4$ in (select $i4$...)) clause. The HASH node hashes and stores the values of $t2.i4$, and checks for values that correspond to every value of $t1.i4$.

11.3 Analyzing Execution Plans

```
iSQL> select count(*) from t1 where i4 in ( select i4 from t2 );
COUNT
-----
16384
1 row selected.

PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 8 )
GROUP-AGGREGATION ( ITEM_SIZE: 24, GROUP_COUNT: 1,
                    BUCKET_COUNT: 1, ACCESS: 1,
                    SELF_ID: 5, REF_ID: 1 )
SCAN ( TABLE: T1, FULL SCAN, ACCESS: 16384, SELF_ID: 1 )
::SUB-QUERY BEGIN
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 4 )
  HASH ( ITEM_SIZE: 24, ITEM_COUNT: 5, BUCKET_COUNT: 1024,
        ACCESS: 16384, SELF_ID: 4, REF_ID: 2 )
  VIEW ( ACCESS: 16384, SELF_ID: 3 )
    PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 4 )
      SCAN ( TABLE: T2, FULL SCAN, ACCESS: 16384, SELF_ID: 2 )
::SUB-QUERY END
```

11.3.4.3 GRAG (Group-Aggregation) Node

The so-called “GRAG node” performs hash-based grouping and aggregation operations according to the relational model. This kind of node has a single child node and uses a temporary table to save intermediate results.

The GRAG node information that can be seen by setting EXPLAIN PLAN to ON (or ONLY) is as follows:

Table 11-25 GRAG Node Information

| Item | Description | Remarks |
|--------------------|--|---|
| GROUP-AGGREGA-TION | The name of the execution node | |
| ITEM_SIZE | The size of each record to be hashed for the grouping operation | |
| GROUP_COUNT | The number of records to be grouped | |
| BUCKET_COUNT | The number of hash buckets | |
| DISK_PAGE_COUNT | The number of disk pages that the temporary table comprises | No information is provided for temporary memory tables. |
| ACCESS | The number of records that are accessed | |
| SELF_ID | The identifier of the execution node | |
| REF_ID | A node identifier used to determine whether to refresh a temporary table | |

The example below shows how a GRAG node is used to perform hash-based grouping and to process aggregate functions. The GRAG node was used to process (GROUP BY i4), (AVG(i1)), and (SUM(i2)).

```

iSQL> select avg(i1), sum(i2) from t1 group by i4:
AVG(I1)    SUM(I2)
-----
8191      158807
8192      162084
8193      165361
8194      168638
8192.5    155530
5 rows selected.
-----
PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 32 )
GROUP-AGGREGATION ( ITEM_SIZE: 80, GROUP_COUNT: 5,
                    BUCKET_COUNT: 1024, ACCESS: 5,
                    SELF_ID: 2, REF_ID: 1 )
SCAN ( TABLE: T1, FULL SCAN, ACCESS: 16384, SELF_ID: 1 )
-----

```

11.3.4.4 HSDS (DISTINCT) Node

The so-called “HSDS node” performs hash-based repetition elimination operations according to the relational model. This kind of node has one child node and uses a temporary table to save intermediate results.

The following HSDS node information can be observed by setting EXPLAIN PLAN to ON (or ONLY):

Table 11-26 HSDS Node Information

| Item | Description | Remarks |
|-----------------|--|---|
| DISTINCT | The name of the execution node | |
| ITEM_SIZE | The size of each record to be hashed for the DISTINCT operation | |
| ITEM_COUNT | The number of records that remain after redundancies have been removed | |
| BUCKET_COUNT | The number of hash buckets | |
| DISK_PAGE_COUNT | The number of disk pages that the temporary table comprises | No information is provided for temporary memory tables. |
| ACCESS | The number of records that are accessed | |
| SELF_ID | The identifier of the execution node | |
| REF_ID | A node identifier used to determine whether to refresh a temporary table | |

HSDS nodes are used for various purposes. The following is a description of the execution plan tree for each purpose.

Used for DISTINCT Clause

An HSDS node can be used to process a DISTINCT clause.

11.3 Analyzing Execution Plans

The example below shows the use of an HSDS node to process a DISTINCT clause.

```
iSQL> select distinct i4 from t1;
i4
-----
1
2
3
4
0
5 rows selected.

-----
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 4 )
DISTINCT ( ITEM_SIZE: 24, ITEM_COUNT: 5, BUCKET_COUNT: 1024,
           ACCESS: 5, SELF_ID: 1, REF_ID: 0 )
SCAN ( TABLE: T1, FULL SCAN, ACCESS: 16384, SELF_ID: 0 )
-----
```

Used for UNION Clause

An HSDS node can be used to process a UNION clause.

The example below shows the use of an HSDS node to eliminate redundancies in order to process a UNION clause.

```
iSQL> select i3, i4 from t1 union select i3, i4 from t2;
i3      i4
-----
1        1
2        2
3        3
4        4
0        0
5 rows selected.

-----
PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 8 )
DISTINCT ( ITEM_SIZE: 24, ITEM_COUNT: 5, BUCKET_COUNT: 16384,
           ACCESS: 5, SELF_ID: 4, REF_ID: 2 )
VIEW ( ACCESS: 32768, SELF_ID: 2 )
BAG-UNION
PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 8 )
SCAN ( TABLE: T1, FULL SCAN, ACCESS: 16384, SELF_ID: 0 )
PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 8 )
SCAN ( TABLE: T2, FULL SCAN, ACCESS: 16384, SELF_ID: 1 )
-----
```

Used for Subquery Key Range Processing

The example below shows the use of an HSDS node in subquery key range processing to eliminate redundancies when processing a UNION clause. The HSDS node is used to remove repeat T2.i4 values.

```

iSQL> select i1 from t1 where i1 in ( select i4 from t2 );
i1
-----
1
2
3
4
4 rows selected.

-----
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 4 )
SCAN ( TABLE: T1, INDEX: IDX1, ACCESS: 4, SELF_ID: 1 )
::SUB-QUERY BEGIN
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 4 )
  DISTINCT ( ITEM_SIZE: 24, ITEM_COUNT: 5, BUCKET_COUNT: 1024,
    ACCESS: 10, SELF_ID: 4, REF_ID: 2 )
  VIEW ( ACCESS: 16384, SELF_ID: 3 )
    PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 4 )
      SCAN ( TABLE: T2, FULL SCAN, ACCESS: 16384, SELF_ID: 2 )
::SUB-QUERY END
-----

```

11.3.4.5 VMTR (MATERIALIZATION) Node

VMTR nodes are used to create temporary storage tables used by views. This kind of node has one child node and uses a temporary table to save intermediate results.

The following VMTR node information can be observed by setting EXPLAIN PLAN to ON (or ONLY):

Table 11-27 VMTR Node Information

| Item | Description | Remarks |
|-----------------|-------------------------------------|---------|
| MATERIALIZATION | The name of the execution node | |
| ITEM_SIZE | The size of the record to be stored | |
| ITEM_COUNT | The number of records to be stored | |

For an example of the use of the VMTR node, please refer to the description of the VSCN node.

11.3.4.6 STOR Node

The STOR execution node is used for the temporary storage of part of the results of a query. This kind of node has one child node and uses a temporary table to save intermediate results.

The following STOR node information can be observed by setting EXPLAIN PLAN to ON (or ONLY):

Table 11-28 STOR Node Information

| Item | Description | Remarks |
|-------|--------------------------------|---------|
| STORE | The name of the execution node | |

11.3 Analyzing Execution Plans

| Item | Description | Remarks |
|-----------------|--|--|
| ITEM_SIZE | The size of a stored record | |
| ITEM_COUNT | The number of records that are stored | |
| DISK_PAGE_COUNT | The number of disk pages that the temporary table comprises | No information is provided for temporary memory tables |
| ACCESS | The number of records that are accessed | |
| SELF_ID | The identifier of the execution node | |
| REF_ID | A node identifier used to determine whether to refresh a temporary table | |

STOR nodes are used for various purposes. The following is a description of the execution plan tree for each purpose.

Used for Processing Joins

STOR nodes can be used to process joins.

A STOR node is usually used in cases involving Cartesian products in which there are no join conditions. If this kind of node is used to process joins, it does not itself process the join conditions.

The example below shows the use of a STOR node for a Cartesian product. The node stores the returned results in the *t1* table to prevent the redundant use of the index.

```
ISQL> select count(*) from t1, t2 where t1.i1 < 5 and t2.i3 = 1;
COUNT
-----
13108
1 row selected.

PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 8 )
GROUP-AGGREGATION ( ITEM_SIZE: 24, GROUP_COUNT: 1,
                     BUCKET_COUNT: 1, ACCESS: 1,
                     SELF_ID: 5, REF_ID: 2 )

JOIN
SCAN ( TABLE: T2, FULL SCAN, ACCESS: 16384, SELF_ID: 3 )
STORE ( ITEM_SIZE: 16, ITEM_COUNT: 4, ACCESS: 13108,
        SELF_ID: 4, REF_ID: 2 )
SCAN ( TABLE: T1, INDEX: IDX1, ACCESS: 4, SELF_ID: 2 )
```

11.3.4.7 LMST (LIMIT-SORT) Node

A so-called “LMST node” performs limited sort operations according to the relational model. This kind of node has a single child node, and uses a temporary table to save intermediate results.

The following LMST node information can be observed by setting EXPLAIN PLAN to ON (or ONLY):

Table 11-29 LMST Node Information

| Item | Description | Remarks |
|-------------|--|---------|
| LIMIT-SORT | The name of the execution node | |
| ITEM_SIZE | The size of the record to be sorted for storing | |
| ITEM_COUNT | The number of records that are processed | |
| STORE_COUNT | The number of records that are stored | |
| ACCESS | The number of records that are accessed | |
| SELF_ID | The identifier of the execution node | |
| REF_ID | A node identifier used to determine whether to refresh a temporary table | |

LMST nodes are used for various purposes. The following is a description of the execution plan tree for each purpose.

Used for ORDER BY Clause

An LMST node can be used to process an ORDER BY clause that contains a LIMIT clause.

The example below shows the use of an LMST node to process an ORDER BY clause. The execution plan shows that only 3 records needed to be stored in order to sort 16384 records.

```
iSQL> select * from t1 order by i4 limit 3:
T1.I1    T1.I2    T1.I3    T1.I4
-----
15       15       0        0
10       10       0        0
5        5        0        0
3 rows selected.

PROJECT ( COLUMN_COUNT: 4, TUPLE_SIZE: 16 )
  LIMIT-SORT ( ITEM_SIZE: 16, ITEM_COUNT: 16384,
               STORE_COUNT: 3, ACCESS: 3,
               SELF_ID: 1, REF_ID: 0 )
    SCAN ( TABLE: T1, FULL SCAN, ACCESS: 16384, SELF_ID: 0 )
```

Used to Process a Subquery

An LMST node can be used to process a SELECT statement within a subquery.

The example below shows that the LMST node sorts and stores only some of the records in order to execute the subquery. By storing only those t2.i4 values that are required, the LMST node helps reduce the cost of the comparison operation.

11.3 Analyzing Execution Plans

```
iSQL> select i1 from t1 where i1 <=ANY ( select i4 from t2 );
i1
-----
1
2
3
4
4 rows selected.
-----
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 4 )
SCAN ( TABLE: T1, FULL SCAN, ACCESS: 16384, SELF_ID: 1 )
::SUB-QUERY BEGIN
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 4 )
  LIMIT-SORT ( ITEM_SIZE: 16, ITEM_COUNT: 16384,
              STORE_COUNT: 2, ACCESS: 32768,
              SELF_ID: 4, REF_ID: 2 )
  VIEW ( ACCESS: 16384, SELF_ID: 3 )
    PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 4 )
      SCAN ( TABLE: T2, FULL SCAN, ACCESS: 16384, SELF_ID: 2 )
    ::SUB-QUERY END
-----
```

11.3.5 Binary Non-Materialization Nodes

Binary Non-Materialization Nodes have two child nodes and do not require any additional storage space in order to perform their functions.

Below, the function of each kind of Binary Non-Materialization Node will be explained, along with how to analyze them in an execution plan.

11.3.5.1 JOIN Node

A so-called "JOIN node" performs join operations according to the relational model. This kind of node has two child nodes, does not generate any intermediate results, and controls the flow of execution of child nodes.

The following JOIN node information can be observed by setting EXPLAIN PLAN to ON (or ONLY):

Table 11-30 JOIN Node Information

| Item | Description | Remarks |
|------|--------------------------------|---------|
| JOIN | The name of the execution node | |

JOIN nodes are used to perform almost all normal join operations.

Below, examples of the execution plan trees for the following types of joins will be presented:

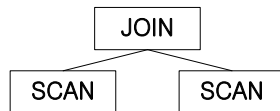
- Full nested loop join
- Full store nested loop join
- Index nested loop join

- One-pass sort join
- Multi-pass sort join
- One-pass hash join
- Multi-pass hash join

In the following examples, the same query is processed using various join methods. The execution plan and plan tree information are shown for each join method.

On the left is a diagram that represents a portion of the execution plan tree, and on the right is the actual execution plan information.

Execution Plan Tree for Full Nested Loop Join



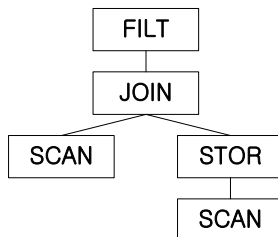
```

iSQL> select count(*) from t1, t2 where t1.i1 = t2.i1 and t1.i3 = 1 and t2.i4 = 1;
COUNT
-----
3277
1 row selected.
-----
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 8 )
GROUP-AGGREGATION ( ITEM_SIZE: 24, GROUP_COUNT: 1,
                    BUCKET_COUNT: 1, ACCESS: 1,
                    SELF_ID: 4, REF_ID: 3 )

JOIN
SCAN ( TABLE: T1, INDEX: IDX3, ACCESS: 3277, SELF_ID: 2 )
SCAN ( TABLE: T2, FULL SCAN, ACCESS: 53690368, SELF_ID: 3 )
  
```

In the execution plan shown above, the join condition is processed by the SCAN node on the right, which repeatedly scans table *t2* in its entirety.

Execution plan tree of full store nested loop join



```

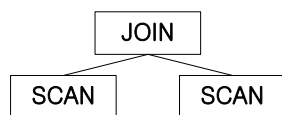
iSQL> select count(*) from t1, t2 where t1.i1 = t2.i1 and t1.i3 = 1 and t2.i4 = 1;
COUNT
-----
3277
1 row selected.
-----
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 8 )
GROUP-AGGREGATION ( ITEM_SIZE: 24, GROUP_COUNT: 1,
                    BUCKET_COUNT: 1, ACCESS: 1,
                    SELF_ID: 5, REF_ID: 3 )

FILTER
JOIN
SCAN ( TABLE: T1, INDEX: IDX3, ACCESS: 3277, SELF_ID: 2 )
STORE ( ITEM_SIZE: 16, ITEM_COUNT: 3277, ACCESS: 10738729,
        SELF_ID: 4, REF_ID: 3 )
SCAN ( TABLE: T2, FULL SCAN, ACCESS: 16384, SELF_ID: 3 )
  
```

In the execution plan shown above, the join condition is processed by the FILT node, which is above the JOIN node. Table *T2* is scanned only once, and the results are saved. The results are used repeatedly for subsequent searches, so that table *T2* does not need to be scanned again.

Execution Plan Tree for Index Nested Loop Join

11.3 Analyzing Execution Plans



```

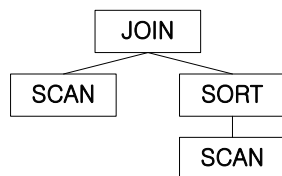
iSQL> select count(*) from t1, t2 where t1.i1 = t2.i1 and t1.i3 = 1 and t2.i4 = 1;
COUNT
-----
3277
1 row selected.

PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 8 )
GROUP-AGGREGATION ( ITEM_SIZE: 24, GROUP_COUNT: 1,
                     BUCKET_COUNT: 1, ACCESS: 1,
                     SELF_ID: 4, REF_ID: 3 )

JOIN
SCAN ( TABLE: T1, INDEX: IDX3, ACCESS: 3277, SELF_ID: 2 )
SCAN ( TABLE: T2, INDEX: IDX11, ACCESS: 3277, SELF_ID: 3 )
  
```

In the execution plan shown above, the join condition is processed by the SCAN node on the right, using an index.

Execution Plan Tree for One-pass Sort Join



```

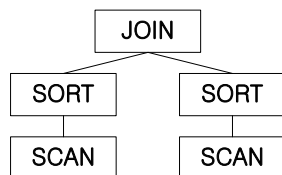
iSQL> select count(*) from t1, t2 where t1.i1 = t2.i1 and t1.i3 = 1 and t2.i4 = 1;
COUNT
-----
3277
1 row selected.

PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 8 )
GROUP-AGGREGATION ( ITEM_SIZE: 24, GROUP_COUNT: 1,
                     BUCKET_COUNT: 1, ACCESS: 1,
                     SELF_ID: 5, REF_ID: 3 )

JOIN
SCAN ( TABLE: T1, INDEX: IDX3, ACCESS: 3277, SELF_ID: 2 )
Sort ( ITEM_SIZE: 16, ITEM_COUNT: 3277, ACCESS: 3277,
       SELF_ID: 4, REF_ID: 3 )
SCAN ( TABLE: T2, FULL SCAN, ACCESS: 16384, SELF_ID: 3 )
  
```

In the execution plan shown above, the join condition is processed by the SORT node on the right using the sorted data.

Execution Plan Tree for Multi-Pass Sort Join



```

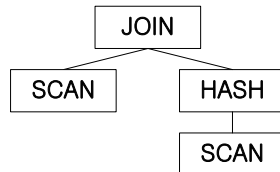
iSQL> select count(*) from t1, t2 where t1.i1 = t2.i1 and t1.i3 = 1 and t2.i4 = 1;
COUNT
-----
3277
1 row selected.

PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 8 )
GROUP-AGGREGATION ( ITEM_SIZE: 24, GROUP_COUNT: 1,
                     BUCKET_COUNT: 1, ACCESS: 1,
                     SELF_ID: 6, REF_ID: 3 )

JOIN
Sort ( ITEM_SIZE: 16, ITEM_COUNT: 3277, ACCESS: 3277,
       SELF_ID: 4, REF_ID: 2 )
SCAN ( TABLE: T1, INDEX: IDX3, ACCESS: 3277, SELF_ID: 2 )
Sort ( ITEM_SIZE: 16, ITEM_COUNT: 3277, ACCESS: 3277,
       SELF_ID: 5, REF_ID: 3 )
SCAN ( TABLE: T2, FULL SCAN, ACCESS: 16384, SELF_ID: 3 )
  
```

In the execution plan shown above, the join condition is processed by the SORT node on the right using the sorted data, but a SORT node is also created on the left.

Execution Plan Tree for One-Pass Hash Join



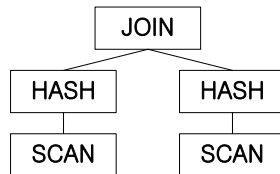
```

iSQL> select count(*) from t1, t2 where t1.i1 = t2.i1 and t1.i3 = 1 and t2.i4 = 1;
COUNT
-----
3277
1 row selected.
-----
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 8 )
GROUP-AGGREGATION ( ITEM_SIZE: 24, GROUP_COUNT: 1,
                     BUCKET_COUNT: 1, ACCESS: 1,
                     SELF_ID: 5, REF_ID: 3 )

JOIN
SCAN ( TABLE: T1, INDEX: IDX3, ACCESS: 3277, SELF_ID: 2 )
HASH ( ITEM_SIZE: 24, ITEM_COUNT: 3277,
       BUCKET_COUNT: 1024, ACCESS: 3277,
       SELF_ID: 4, REF_ID: 3 )
SCAN ( TABLE: T2, FULL SCAN, ACCESS: 16384, SELF_ID: 3 )
-----
  
```

In the execution plan shown above, the join condition is processed by the HASH node on the right using hashed data.

Execution Plan Tree for Multi-Pass Hash Join



```

iSQL> select count(*) from t1, t2 where t1.i1 = t2.i1 and t1.i3 = 1 and t2.i4 = 1;
COUNT
-----
3277
1 row selected.
-----
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 8 )
GROUP-AGGREGATION ( ITEM_SIZE: 24, GROUP_COUNT: 1,
                     BUCKET_COUNT: 1, ACCESS: 1,
                     SELF_ID: 6, REF_ID: 3 )

JOIN
HASH ( ITEM_SIZE: 24, ITEM_COUNT: 3277,
       BUCKET_COUNT: 1024, ACCESS: 3277, SELF_ID: 4, REF_ID: 2 )
SCAN ( TABLE: T1, INDEX: IDX3, ACCESS: 3277, SELF_ID: 2 )
HASH ( ITEM_SIZE: 24, ITEM_COUNT: 3277,
       BUCKET_COUNT: 1024, ACCESS: 3277, SELF_ID: 5, REF_ID: 3 )
SCAN ( TABLE: T2, FULL SCAN, ACCESS: 16384, SELF_ID: 3 )
-----
  
```

In the execution plan shown above, the join condition is processed by the HASH node on the right using sorted data, but another HASH node is also created on the left.

11.3.5.2 MGJN (MERGE-JOIN) Node

The so-called “MGJN node” performs merge-join operations according to the relational model. This kind of node has two child nodes, does not create intermediate results, and controls the flow of execution of child nodes. The child nodes of an MGJN node are one of SCAN, SORT, or MGJN nodes.

The following MGJN node information can be observed by setting EXPLAIN PLAN to ON (or ONLY):

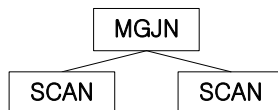
Table 11-31 MGJN Node Information

| Item | Description | Remarks |
|------------|--------------------------------|---------|
| MERGE-JOIN | The name of the execution node | |

A MGJN node is used to perform common join operations. The node either sorts both the left and right child nodes, or processes the results in the order in which they are sorted.

There are 9 types of merge joins, depending on the type of child node. Execution plans for the two most common types are shown below:

Merge Join using Index



```

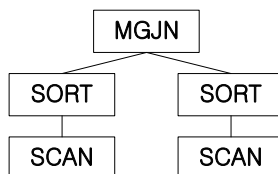
iSQL> select count(*) from t1, t2 where t1.i1 = t2.i1 and t1.i3 = 1 and t2.i4 = 1;
COUNT
-----
3277
1 row selected.

PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 8 )
GROUP-AGGREGATION ( ITEM_SIZE: 24, GROUP_COUNT: 1,
                    BUCKET_COUNT: 1, ACCESS: 1,
                    SELF_ID: 5, REF_ID: 3 )

MERGE-JOIN
SCAN ( TABLE: T1, INDEX: IDX1, ACCESS: 16384, SELF_ID: 2 )
SCAN ( TABLE: T2, INDEX: IDX11, ACCESS: 16384, SELF_ID: 3 )
  
```

In the above execution plan, the join condition is processed by the MGJN node, and all indexes defined for columns that are referenced in the join condition are used, regardless of whether the table is an outer table or an inner table.

Merge Join using Sort



```

iSQL> select count(*) from t1, t2 where t1.i1 = t2.i1 and t1.i3 = 1 and t2.i4 = 1;
COUNT
-----
3277
1 row selected.

PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 8 )
GROUP-AGGREGATION ( ITEM_SIZE: 24, GROUP_COUNT: 1,
                    BUCKET_COUNT: 1, ACCESS: 1,
                    SELF_ID: 7, REF_ID: 3 )

MERGE-JOIN
SORT ( ITEM_SIZE: 16, ITEM_COUNT: 3277, ACCESS: 3277,
      SELF_ID: 4, REF_ID: 2 )
SCAN ( TABLE: T1, FULL SCAN, ACCESS: 16384, SELF_ID: 2 )
SORT ( ITEM_SIZE: 16, ITEM_COUNT: 3277, ACCESS: 3277,
      SELF_ID: 5, REF_ID: 3 )
SCAN ( TABLE: T2, FULL SCAN, ACCESS: 16384, SELF_ID: 3 )
  
```

In the above execution plan, the join condition is processed by the MGJN node, and the results are first sorted using columns that are referenced in the join condition.

11.3.5.3 LOJN (LEFT-OUTER-JOIN) Node

The so-called “LOJN node” performs LEFT OUTER JOIN operations according to the relational model.

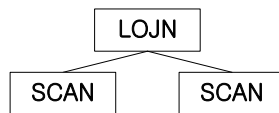
This kind of node has two child nodes, does not generate intermediate results, and controls the flow of execution of child nodes.

The following LOJN node information can be observed by setting EXPLAIN PLAN to ON (or ONLY):

Table 11-32 LOJN Node Information

| Item | Description | Remarks |
|-----------------|--------------------------------|---------|
| LEFT-OUTER-JOIN | The name of the execution node | |

As with normal joins, the LOJN node can be used with most join types, as was illustrated in the examples of the use of the JOIN node. In this section, only a simple execution plan tree illustrating the use of a LOJN node is shown.



```

iSQL> select count(*) from t1 left outer join t2 on t1.i1 = t2.i1
      where t1.i3 = 1 and t2.i4 = 1;
COUNT
-----
3277
1 row selected.

PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 8 )
GROUP-AGGREGATION ( ITEM_SIZE: 24, GROUP_COUNT: 1,
                    BUCKET_COUNT: 1, ACCESS: 1,
                    SELF_ID: 4, REF_ID: 3 )

FILTER
LEFT-OUTER-JOIN
SCAN ( TABLE: T1, FULL SCAN, ACCESS: 16384, SELF_ID: 2 )
SCAN ( TABLE: T2, INDEX: IDX11, ACCESS: 3277, SELF_ID: 3 )
  
```

11.3.5.4 FOJN (FULL-OUTER-JOIN) Node

The so-called “FOJN node” performs FULL OUTER JOIN operations according to the relational model. This kind of node has two child nodes, does not generate intermediate results, and controls the flow of execution of the child nodes.

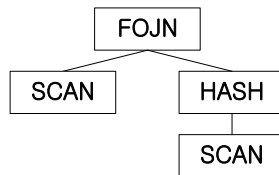
The following FOJN node information can be observed by setting EXPLAIN PLAN to ON (or ONLY):

Table 11-33 FOJN Node Information

| Item | Description | Remarks |
|-----------------|--------------------------------|---------|
| FULL-OUTER-JOIN | The name of the execution node | |

As with normal joins, the FOJN node can be used with most join types, as was described in the examples of the use of the FOJN node. In this section, only a simple execution plan tree illustrating the use of a FOJN node is shown.

11.3 Analyzing Execution Plans



```

iSQL> select count(*) from t1 full outer join t2 on t1.i1 = t2.i1
      where t1.i3 = 1 and t2.i4 = 1;
COUNT
-----
3277
1 row selected.

PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 8 )
GROUP-AGGREGATION ( ITEM_SIZE: 24, GROUP_COUNT: 1,
                    BUCKET_COUNT: 1, ACCESS: 1,
                    SELF_ID: 5, REF_ID: 3 )

FILTER
FULL-OUTER-JOIN
SCAN ( TABLE: T1, FULL SCAN, ACCESS: 16384, SELF_ID: 2 )
HASH ( ITEM_SIZE: 24, ITEM_COUNT: 3277, BUCKET_COUNT: 1024,
      ACCESS: 3277, SELF_ID: 4, REF_ID: 3 )
SCAN ( TABLE: T2, FULL SCAN, ACCESS: 16384, SELF_ID: 3 )
  
```

A storage NODE is created on the right due to the characteristics of the FULL OUTER JOIN node. The ON condition is processed by the HASH node.

11.3.5.5 AOJN (ANTI-OUTER-JOIN) Node

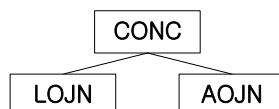
The so-called “AOJN node” is the physical entity that performs ANTI OUTER JOIN operations according to the relational model. This kind of node has two child nodes, does not generate intermediate results, and controls the flow of execution of the child nodes.

The following AOJN node information can be observed by setting EXPLAIN PLAN to ON (or ONLY):

Table 11-34 AOJN Node Information

| Item | Description | Remarks |
|-----------------|--------------------------------|---------|
| ANTI-OUTER-JOIN | The name of the execution node | |

The AOJN node is used only to process FULL OUTER JOINs. As shown below, this kind of node can be used when indexes have been defined for all of the columns referenced in an ON join condition.



```

iSQL> select count(*) from t1 full outer join t2 on t1.i1 = t2.i1
      where t1.i3 = 1 and t2.i4 = 1;
COUNT
-----
3277
1 row selected.

PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 8 )
GROUP-AGGREGATION ( ITEM_SIZE: 24, GROUP_COUNT: 1,
                    BUCKET_COUNT: 1, ACCESS: 1,
                    SELF_ID: 4, REF_ID: 3 )

FILTER
CONCATENATION
LEFT-OUTER-JOIN
SCAN ( TABLE: T1, FULL SCAN, ACCESS: 22938, SELF_ID: 2 )
SCAN ( TABLE: T2, INDEX: IDX11, ACCESS: 22938, SELF_ID: 3 )
ANTI-OUTER-JOIN
SCAN ( TABLE: T2, FULL SCAN, ACCESS: 22938, SELF_ID: 3 )
SCAN ( TABLE: T1, INDEX: IDX1, ACCESS: 22938, SELF_ID: 2 )
  
```


As shown in the above example, when processing a FULL OUTER JOIN, an AOJN node and a LOJN execution node always have a CONC execution node as their parent node. The join condition in the ON clause is processed by both the LOJN and AOJN nodes.

11.3.5.6 CONC (Concatenation) Node

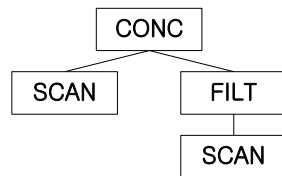
The so-called “CONC node” is the physical entity that performs concatenation operations according to the relational model. This kind of node has two child nodes, does not generate intermediate results, and controls the flow of execution of the child nodes.

The following CONC node information can be observed by setting EXPLAIN PLAN to ON (or ONLY):

Table 11-35 CONC Node Information

| Item | Description | Remarks |
|---------------|--------------------------------|---------|
| CONCATENATION | The name of the execution node | |

CONC nodes are used to process FULL OUTER JOIN and DNF conditions. Because an example of the use of a CONC node to process a FULL OUTER JOIN condition was already presented in the description of the AOJN node, what follows is an example of the use of a CONC node to process a DNF condition.



```

ISQL> select sum(i3) from t1 where i1 = 1000 or i2 = 100;
SUM(I3)
-----
0
1 row selected.
-----
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 8 )
GROUP-AGGREGATION ( ITEM_SIZE: 24, GROUP_COUNT: 1,
                    BUCKET_COUNT: 1, ACCESS: 1,
                    SELF_ID: 3, REF_ID: 2 )
CONCATENATION
SCAN ( TABLE: T1, INDEX: IDX1, ACCESS: 1, SELF_ID: 2 )
FILTER
SCAN ( TABLE: T1, INDEX: IDX2, ACCESS: 1, SELF_ID: 2 )
-----
  
```

The (i1 = 1000) condition is processed by the SCAN node on the left, while the (i2 = 100) condition is processed by the SCAN node on the right. With this method, both the IDX1 and IDX2 indexes are used, and the CONC node is used to combine the results. The FILT node is used to eliminate repeat results from the result set returned by the SCAN node on the left.

11.3.5.7 BUNI (BAG-UNION) Node

The so-called “BUNI node” is the physical entity that performs UNION ALL operations according to the relational model. This kind of node has two child nodes, does not generate intermediate results, and controls the flow of execution of the child nodes.

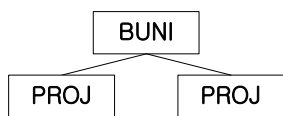
The following BUNI node information can be observed by setting EXPLAIN PLAN to ON (or ONLY):

11.3 Analyzing Execution Plans

Table 11-36 BUNI Node Information

| Item | Description | Remarks |
|-----------|--------------------------------|---------|
| BAG-UNION | The name of the execution node | |

The example below show how a BUNI node is executed.



```
isQL> select i3, sum(i2) from t1 group by i3
      union all
      select i3, avg(i2) from t2 group by i3;
i3      SUM(i2)
-----
1       158807
2       162084
3       165361
4       168638
0       155530
1       48.4610925
2       49.4610925
3       50.4610925
4       51.4610925
0       47.47558
10 rows selected.

PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 26 )
VIEW ( ACCESS: 10, SELF_ID: 3 )
BAG-UNION
PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 16 )
  GROUP-AGGREGATION ( ITEM_SIZE: 32, GROUP_COUNT: 5,
                     BUCKET_COUNT: 1024, ACCESS: 5,
                     SELF_ID: 4, REF_ID: 1 )
    SCAN ( TABLE: T1, FULL SCAN, ACCESS: 16384, SELF_ID: 1 )
PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 26 )
  GROUP-AGGREGATION ( ITEM_SIZE: 72, GROUP_COUNT: 5,
                     BUCKET_COUNT: 1024, ACCESS: 5,
                     SELF_ID: 5, REF_ID: 2 )
    SCAN ( TABLE: T2, FULL SCAN, ACCESS: 16384, SELF_ID: 2 )
```

In the above example, the BUNI execution node processes the UNION ALL clause simply by outputting the results of both of the queries.)

11.3.6 Binary Materialization Nodes

Binary Materialization Nodes have two child nodes, and require specially allocated storage space in order to execute their functions.

Below, the function of each kind of Binary Materialization Node will be explained, along with how to analyze them in an execution plan.

11.3.6.1 SITS (SET-INTERSECT) Node

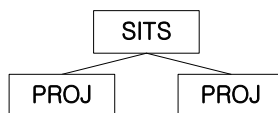
A so-called "SITS node" performs SET-INTERSECT operations according to the relational model. This kind of node has two child nodes, and stores and processes intermediate results in order to intersect two result sets.

The following SITS node information can be observed by setting EXPLAIN PLAN to ON (or ONLY):

Table 11-37 SITS Node Information

| Item | Description | Remarks |
|-----------------|--|---|
| SET-INTERSECT | The name of the execution node | |
| ITEM_SIZE | The stored size of a record to be intersected | |
| ITEM_COUNT | The number of records to be intersected | |
| BUCKET_COUNT | The number of hash buckets | |
| DISK_PAGE_COUNT | The number of disk pages that the temporary table comprises | No information is provided for temporary memory tables. |
| ACCESS | The number of records that are accessed | |
| SELF_ID | The identifier of the execution node | |
| L_REF_ID | The identifier of the node corresponding to the query on the left, used to determine whether to refresh the corresponding temporary table | |
| R_REF_ID | The identifier of the node corresponding to the query on the right, used to determine whether to refresh the corresponding temporary table | |

The example below shows how a SITS node is used to process an INTERSECT clause. After repeat records are eliminated from the result returned by the query on the left, the node searches for records that intersect with the data returned by the query on the right.



```

iSQL> select i1, i3 from t1 intersect select i3, i1 from t2:
i1      i3
-----
1        1
2        2
3        3
4        4
4 rows selected.

PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 8 )
VIEW ( ACCESS: 4, SELF_ID: 2 )
SET-INTERSECT ( ITEM_SIZE: 24, ITEM_COUNT: 16384,
                  BUCKET_COUNT: 8192, ACCESS: 4,
                  SELF_ID: 4, L_REF_ID: 0, R_REF_ID: 1 )
PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 8 )
SCAN ( TABLE: T1, FULL SCAN, ACCESS: 16384, SELF_ID: 0 )
PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 8 )
SCAN ( TABLE: T2, FULL SCAN, ACCESS: 16384, SELF_ID: 1 )
  
```

11.3.6.2 SDIF (SET-DEFERENCE) Node

The so-called “SDIF node” performs MINUS operations according to the relational model. This kind of node has two child nodes, and stores and processes intermediate results in order to obtain a dif-

11.3 Analyzing Execution Plans

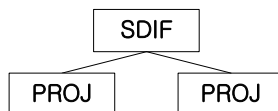
ference set.

The following SDIF node information can be observed by setting EXPLAIN PLAN to ON (or ONLY):

Table 11-38 SDIF Node Information

| Item | Description | Remarks |
|-----------------|--|---|
| SET-DIFFERENCE | The name of the execution node | |
| ITEM_SIZE | The stored size of a record for the difference set | |
| ITEM_COUNT | The number of saved records | |
| BUCKET_COUNT | The number of hash buckets | |
| DISK_PAGE_COUNT | The number of disk pages that the temporary table comprises | No information is provided for temporary memory tables. |
| ACCESS | The number of records that are accessed | |
| SELF_ID | The identifier of the execution node | |
| L_REF_ID | The identifier of the node corresponding to the query on the left, used to determine whether to refresh the corresponding temporary table | |
| R_REF_ID | The identifier of the node corresponding to the query on the right, used to determine whether to refresh the corresponding temporary table | |

The example below shows the use of an SDIF node to process a MINUS clause. After eliminating repeat records from the results returned by the query on the left, the node searches for records that do not intersect with the data returned by the query on the right.



```
iSQL> select i1, i3 from t1 minus select i1, i3 from t2;
i1      i3
-----
No rows selected.

PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 8 )
VIEW ( ACCESS: 0, SELF_ID: 2 )
  SET-DIFFERENCE ( ITEM_SIZE: 24, ITEM_COUNT: 16384,
                   BUCKET_COUNT: 8192, ACCESS: 0,
                   SELF_ID: 4, L_REF_ID: 0, R_REF_ID: 1 )
    PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 8 )
      SCAN ( TABLE: T1, FULL SCAN, ACCESS: 16384, SELF_ID: 0 )
    PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 8 )
      SCAN ( TABLE: T2, FULL SCAN, ACCESS: 16384, SELF_ID: 1 )
```

11.3.7 Multiple Non-materialization Nodes

Multiple Non-Materialization Nodes have two or more child nodes and do not require their own

storage space in order to perform their functions.

11.3.7.1 PCRD Node

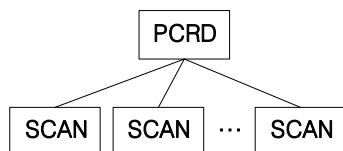
The purpose of the PCRD (Partition-Coordinator) Node is to manage the scanning of each partition in a partitioned table. This kind of node has multiple child nodes and filters partitions.

The following PCRD node information can be observed by setting EXPLAIN PLAN to ON (or ONLY):

Table 11-39 PCRD Node Information

| Item | Description | Remarks |
|-----------------------|---|---------|
| PARTITION-COORDINATOR | The name of the execution node | |
| TABLE | The name of the table to be accessed | |
| PARTITION | The number of partitions to be accessed | |
| ACCESS | The actual number of records that were accessed | |
| SELF_ID | The identifier of the node | |

An example of the execution of a PCRD node is shown below. The results of partition scanning conducted by the child node are delivered to a higher (parent) node.



```

iSQL> SELECT COUNT(*) FROM PDT_RANGE WHERE FI < 30;
COUNT
-----
3
1 row selected.

PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 8 )
GROUP-AGGREGATION ( ITEM_SIZE: 24, GROUP_COUNT: 1,
  BUCKET_COUNT: 1, ACCESS: 1, SELF_ID: 6, REF_ID: 2 )
  PARTITION-COORDINATOR ( TABLE: PDT_RANGE, PARTITION:
  3/4, ACCESS: 3, SELF_ID: 2 )
    SCAN ( PARTITION: P3, FULL SCAN, ACCESS: 1,
    DISK_PAGE_COUNT: 2, SELF_ID: 3 )
    SCAN ( PARTITION: P2, FULL SCAN, ACCESS: 1,
    DISK_PAGE_COUNT: 2, SELF_ID: 4 )
    SCAN ( PARTITION: P1, FULL SCAN, ACCESS: 1,
    DISK_PAGE_COUNT: 2, SELF_ID: 5 )
  
```

11.4 Using Optimizer Hints

Hints are used when the user wants to explicitly change the execution plan for an SQL statement.

There is no end to the variety of SQL statements that can be authored by users. Furthermore, different execution plans might be created to process the same query depending how the data are organized. The optimizer creates an efficient execution plan in most cases, but cannot come up with the optimum plan in every case.

To overcome this weakness, users can use hints to explicitly alter execution plans and improve performance. However, it is not appropriate to use hints to alter the execution plans for all queries, because hints need to be rewritten every time an index is created or the data are reorganized, which is burdensome. Therefore, hints should be used only for those queries that critically affect system performance.

11.4.1 Hint Processing Policy

User-defined hints are processed in accordance with the following policy.

If the syntax of a user-defined hint is error-free and it is possible to obey the hint, the hint will automatically be obeyed. Conversely, hints containing syntax errors or those that cannot be executed will be disregarded. When hints are applied, there is no process by which relative weights or priorities are assigned to user-defined hints: it is assumed that the user has already taken all available information into consideration and has determined the most efficient execution plan.

If hints contradict one another, the hint that is applied is determined according to the type of hint. The hint types are listed below in descending order of priority:

- Optimization Strategy
- Normalization Type
- Join Order
- Joining Method
- Intermediate Result Storage Medium
- Hash Bucket Count
- Group Processing Method
- Duplicate Removal Method
- View Optimization Method
- Access Method

11.4.2 Types of Hints

Optimization Strategy

These hints are used to tell the optimizer whether to perform rule-based optimization or cost-based

optimization.

- **RULE:** Create an execution plan without considering the cost of execution.
- **COST:** Take the cost of execution into consideration when creating an execution plan.

Use the RULE hint to dictate that the same execution plan is always to be used for a given query. However, use the COST hint if variation in the amount of data greatly influences the estimated cost.

If neither hint is used, cost-based optimization is performed by default.

Normalization Type

This hint allows the normalization method to be set for individual SQL statements.

- **CNF:** Normalization using conjunctive normal form

In conjunctive normal form, the AND operator is the highest-level logical operator and the OR operator is a lower-level operator. When the CNF hint is used, ALTIBASE HDB converts the SELECT statement into conjunctive normal form while creating the execution plan. However, sometimes the use of CNF results in the creation of very complicated conditional clauses, resulting in excessive consumption of system resources. In such cases, ALTIBASE HDB does not use CNF even if this hint is specified.

- **DNF:** Normalization using disjunctive normal form

In disjunctive normal form, the OR operator is the highest-level logical operator, while the AND operator is a lower-level operator. The use of the DNF hint creates an execution plan that converts the SELECT statement into disjunctive normal form, so that each conditional clause is processed separately using a respective index.

Note that if the SQL statement does not contain any OR clauses, this hint will be ignored. Additionally, depending on the properties of the conditional clauses, conversion into DNF can result in the creation of a large number of conditional clauses, entailing excessive exhaustion of system resources. In such cases, ALTIBASE HDB does not use DNF even if this hint is specified.

If neither of the two hints is used, the normalization method is selected on the basis of cost estimation.

Join Order

This hint is used to set the join order.

- **ORDERED:** The join order is determined according to the order that joins appear in the FROM clause of the SQL statement.

If this hint is not used, the join order is determined on the basis of cost estimation.

Joining Method

This hint is used to set the joining method.

- **USE_NL:** Use the nested loop joining method

The USE_NL hint instructs the optimizer to use a nested loop join to join the specified tables.

11.4 Using Optimizer Hints

- **USE_SORT:** Use the sort-based joining method

The **USE_SORT** hint instructs the optimizer to use a sort join to join the specified tables. Note that if no sorting predicate exists, a nested loop join will be used to join the tables.

- **USE_HASH:** Use the hash-based joining method

The **USE_HASH** hint instructs the optimizer to use a hash join to join the specified tables. Note that if no hashing predicate exists, a nested loop join will be used to join the tables.

- **USE_MERGE:** Use the merge joining method

The **USE_MERGE** hint instructs the optimizer to use a sort merge join to join the specified tables. Note, however, that if no sorting predicate exists, a nested loop join will be used to join the tables.

The hints pertaining to joining methods are processed as described below in order to determine the joining method:

- The optimizer checks whether it is possible to create a join using each of the specified tables as the inner table in the order in which they are specified.

For example, **USE_NL(T1, T2)** directs the optimizer to check whether it is possible to create the join using table *T1* as the outer table and table *T2* as the inner table (*T1* => *T2*).

- The optimizer then checks whether it is possible to create a join using table *T1* as the inner table.

That is, if the join cannot be created in the specified order, the optimizer then checks whether the join can be created in the opposite order. In the above example, it checks whether the join can be created using table *T2* as the outer table and table *T1* as the inner table (*T2* => *T1*).

- If a join cannot be created in either case, another joining method is chosen on the basis of cost estimation.

- Conflict with **ORDERED** hint

Given the following query:

```
SELECT /*+ ORDERED USE_NL(T2, T1) */ FROM T1, T2 WHERE T1.i1 = T2.i1;
```

If the table order specified in an **ORDERED** hint and that specified in an **USE_NL** hint contradict each other, the **ORDERED** hint takes priority.

- If more than one joining method hint is specified for the same table, one of those hints is chosen on the basis of cost estimation.

```
USE_NL(T1, T2) USE_HASH(T2, T1)
```

Intermediate Result Storage Medium

These hints are used to specify the storage medium in which to store temporary tables that contain intermediate results.

- **TEMP_TBS_MEMORY:** Specifies that memory temporary tables are used to store all intermediate results generated while processing the query.

- **TEMP_TBS_DISK:** Specifies that disk temporary tables are used to store all intermediate results generated while processing the query.

TEMP_TBS_MEMORY is used to maximize performance in the case where the intermediate result set is small, whereas TEMP_TBS_DISK is used when the intermediate result set is expected to be large, despite the decrease in performance.

Hash Bucket Count

These hints are used to set the number of buckets for execution nodes that use the hashing method.

The optimizer uses hashing to process statements such as GROUP BY, UNION, INTERSECT, MINUS, DISTINCT, HASH JOIN and aggregate functions. If the number of hash buckets that are allocated is suitable for the number of records to be processed, the query processing speed will be improved. In cost-based optimization, a suitable number of hash buckets is determined internally based on the number of records, however, the following hints can be used to set the number of hash buckets as desired.

- **HASH BUCKET COUNT:** Changes the number of hash buckets for HASH and HSDS nodes.
- **GROUP BUCKET COUNT:** Changes the number of hash buckets for GRAG and AGGR nodes.
- **SET BUCKET COUNT:** Changes the number of hash buckets for SITS and SDIF nodes.

Group Processing Method

These hints are used to specify the method with which GROUP BY clauses are processed.

- **GROUP_HASH:** Use hash processing
- **GROUP_SORT:** Use the sort order for processing

Duplicate Removal Method

These hints are used to specify the method with which DISTINCT clauses are processed.

- **DISTINCT_HASH:** Use hash processing
- **DISTINCT_SORT:** Use the sort order for processing

View Optimization Method

For a WHERE clause located outside a view, these hints determine whether to process the conditions of the WHERE clause inside the view.

- **NO_PUSH_SELECT_VIEW:** Specifies that the WHERE clause's conditions are not to be moved inside the view for processing.
- **PUSH_SELECT_VIEW:** Specifies that those conditions of a WHERE clause that can be moved inside the view are to be moved into the view for processing.

Push Predicate Method

For a WHERE clause located outside a view, these hints determine whether to process the join conditions of the WHERE clause inside the view.

- **PUSH_PRED:** Specifies that the join conditions of a WHERE clause that pertain to the view are

11.4 Using Optimizer Hints

to be moved into the view for processing.

Access Method

These hints are used to control how tables are accessed.

- **FULL SCAN:** Specifies that a full table scan is to be performed without using an index, even if an index is available.
- **INDEX:** Specifies that an index scan is to be performed using one of the listed indexes.
- **INDEX ASC:** Specifies that an ascending index scan is to be performed using one of the listed indexes.
- **INDEX DESC:** Specifies that a descending index scan is to be performed using one of the listed indexes.
- **NO INDEX:** Specifies that none of the listed indexes are to be used in the optimization process.

If the names of indexes defined on the basis of tables that are used within a view are specified in any of the above hints, hints can be given for views, just as they can for normal tables.

Many hints can be used to control the access method. These hints are processed according to the following policy:

- If any hints contradict each other, the hint that appears first will be applied, and subsequent hints will be disregarded.

Example: `INDEX(T1, IDX1) NO INDEX(T1, IDX1)`

- If the hints do not contradict each other, the most efficient of the specified access methods will be chosen on the basis of cost estimation.

Example: `FULL SCAN(T1), INDEX(T1, IDX1)`

When an access method hint is used together with a join method hint, the hints are processed separately.

Example: `USE_HASH(T1, T2), INDEX(T2, IDX2)`

12 Using Explain Plan

This chapter comprises the following sections:

- [Overview](#)
- [Reading Plan Trees](#)
- [Plan Nodes](#)

12.1 Overview

The EXPLAIN PLAN statement makes it possible to examine the execution plan set by the optimizer for an SQL statement. When EXPLAIN PLAN is set to ON or ONLY and an SQL statement is executed, the optimizer determines an execution plan for the statement and returns data describing the plan to the client at which the SQL statement was executed. Simply execute the SQL statement after EXPLAIN PLAN has been set appropriately.

12.1.1 Plan Trees Defined

Query optimization greatly affects how fast an SQL statement can be executed.

In ALTIBASE HDB, EXPLAIN PLAN explains how the server executes a query that has been optimized. That is, EXPLAIN PLAN is used to improve the performance of SQL statements and to help determine execution plans (hereinafter referred to as "plan trees").

12.1.2 Understanding Plan Trees

Various optimization methods (join method, join order, access path, etc.) can be used while still ensuring that the same results are obtained when executing SQL statements that scan many tables for data. ALTIBASE HDB determines a suitable execution on the basis of the following factors:

- The presence of any indexes that can be used
- The order of tables and columns within the SQL statement
- The optimization method that is used

Setting the EXPLAIN PLAN property appropriately makes it possible to view the plan tree for an SQL statement. Users can check the plan tree to clearly understand how ALTIBASE HDB will execute an SQL statement.

12.1.3 Analyzing Plan Trees

Because EXPLAIN PLAN allows a plan tree to be created without actually executing the SQL statement, it is useful not just for displaying how the SQL statement will be executed, but for comparing different plan trees with the ultimate goal of improving the performance of the SQL statement.

The plan tree displays the following information:

- The execution plan created by the SQL optimizer
- The properties of objects (e.g. tables and indexes)
- Any indexes that are used
- The join methods that are used
- The optimized join order

12.1.4 Analyzing Performance

The procedure for verifying that the performance of an SQL statement has improved is as follows:

- After modifying an SQL statement to optimize it, execute the new SQL statement and compare the performance with that obtained using previous versions of the statement.
- Create a new plan tree and compare it with previous plan trees.
- Double-check the properties of objects (e.g. tables and indexes) to make sure that they are correct.

12.1.5 How to Display the Plan Tree

In iSQL, first set the EXPLAIN PLAN property to a suitable value by executing this statement:

```
ALTER SESSION SET EXPLAIN PLAN = {ON|OFF|ONLY};
```

Then execute the actual SQL statement. Depending on which of the following EXPLAIN PLAN property options is set, the plan tree can be viewed in text form:

- ON

Display the plan tree, the number of times that individual records were accessed,, and the amount of memory occupied by tuples after executing the query.

- OFF

Do not display the plan tree.

- ONLY

Do not actually execute the query; just display the plan tree.

```
iSQL> ALTER SESSION SET EXPLAIN PLAN = ON;
iSQL> SELECT e_firstname, e_lastname
  FROM employees
  WHERE emp_job = 'PROGRAMMER';
E_FIRSTNAME      E_LASTNAME
-----
Ryu              Momoi
Elizabeth        Bae
2 rows selected.

-----
PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 44 )
SCAN ( TABLE: EMPLOYEES, FULL SCAN, ACCESS: 20, SELF_ID: 2 )
-----
```

```
iSQL> ALTER SESSION SET EXPLAIN PLAN = OFF;
Alter success.
iSQL> SELECT e_firstname, e_lastname
  FROM employees
  WHERE emp_job = 'PROGRAMMER';
E_FIRSTNAME      E_LASTNAME
-----
Ryu              Momoi
Elizabeth        Bae
2 rows selected.
```

12.1 Overview

```
iSQL> ALTER SESSION SET EXPLAIN PLAN = ONLY;
Alter success.
iSQL> SELECT e_firstname, e_lastname
FROM employees
WHERE emp_job = 'PROGRAMMER';
E_FIRSTNAME          E_LASTNAME
-----
No rows selected.
-----
PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 44 )
SCAN ( TABLE: EMPLOYEES, FULL SCAN, ACCESS: ??, SELF_ID: 2 )
-----
```

When EXPLAN PLAN is set to ONLY, because the query is not actually executed after the execution plan is created, the values of items that can be determined only after the query is actually executed, such as the ACCESS item, are displayed as two question marks ("??"), as can be seen above.

12.2 Reading Plan Trees

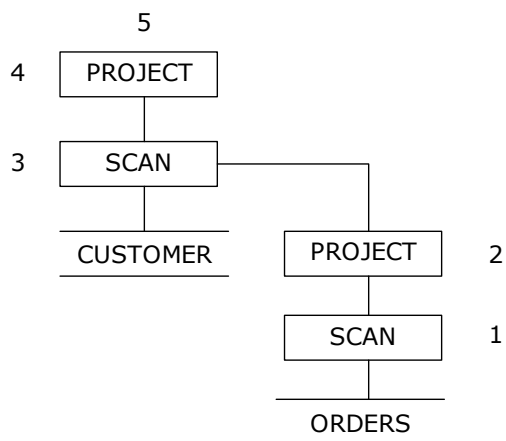
A plan tree is made up of multiple plan nodes with relationships defined therebetween. So-called "child nodes" are shown immediately below parent nodes, and are indented one space further. Additionally, subqueries are shown between the delimiters "::SUB-QUERY BEGIN" and "::SUB-QUERY END".

```
iSQL> SELECT c.cname
FROM customer c
WHERE c.cno IN
(SELECT o.cno
FROM orders o
WHERE o.ono = 12310001);
CNAME
```

DKKIM

1 row selected.

```
-----
4  PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 22 )
3  SCAN ( TABLE: CUSTOMER, FULL SCAN, ACCESS: 20, SELF_ID: 2 )
   ::SUB-QUERY BEGIN
   .
   .
2  PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 16 )
1  SCAN ( TABLE: ORDERS, INDEX: __SYS_IDX_ID_69, ACCESS: 20, SELF_ID: 3 )
   ::SUB-QUERY END
-----
```



1. The order number (ono) in the orders table is scanned using an index. The number of times that records in the orders table are accessed using the index is 20. In order to scan the orders table on the basis of a column for which an index was not defined, it would be necessary to scan the entire orders table ("full scan") in order to find records that matched the condition. That is, users use the plan nodes to get the cost of performing a full scan with that of scanning using an index, compares the costs with each other, and selects the most cost efficient plan.
2. The cno column is selected from the orders table to create relation1, which has one column.
3. The entire customer table is scanned in order to retrieve rows that satisfy the condition "c.cno = o.cno". The access count is equal to the number of records (20) in the customer table.

12.2 Reading Plan Trees

4. The cname column is selected from the customers table to create relation2.
5. relation2 is output.

12.3 Plan Nodes

A plan tree is an execution plan created for a specific SQL statement. Plan nodes are the nodes constituting a plan tree.

12.3.1 Types of Plan Nodes

Plan nodes are classified according to the number of child nodes they possess and according to whether they are materialized, that is, whether they physically store intermediate results.

12.3.1.1 Classification According to the Number of Child Nodes

- One-Child Node
This kind of plan node has one child node or no child nodes.
- Binary Node (Two-Child Node)
This kind of plan node has two child nodes.
- Multi-Child Node
This kind of plan node has two or more child nodes.

12.3.1.2 Classification According to Materialization

- Non-Materialized Node
This kind of plan node processes one record at a time without storing intermediate results.
- Materialized Node
This kind of plan node stores the results of execution of all child nodes as intermediate results.

Table 12-1 Types of Plan Nodes

| Node | | Non-Materialized | | Materialized | |
|-----------|-------|------------------|-------------|-------------------|-------------------|
| | | Name | Meaning | Name | Meaning |
| One-child | Basic | SCAN | Selection | SORT | Sort |
| | | FILTER | Filter | HASH | Hash |
| | | PROJECT | Projection | | |
| | Group | GROUPING | Group By | GROUP-AGGREGATION | Group Aggregation |
| | | AGGREGATION | Aggregation | | |
| | | COUNT | Count(*) | | |

12.3 Plan Nodes

| Node | | Non-Materialized | | Materialized | |
|-------------|--------------|-----------------------|-----------------------|-----------------|----------------------|
| | | Name | Meaning | Name | Meaning |
| | View | VIEW | View | MATERIALIZATION | View Materialization |
| | | VIEW-SCAN | View Scan | | |
| | Others | HIERARCHICAL QUERY | Hierarchy | DISTINCT | Hash Distinct |
| | | | | LIMIT-SORT | Limit Sort |
| Two-child | Join | JOIN | Join | | |
| | | MERGE-JOIN | Merge Join | | |
| | | LEFT-OUTER-JOIN | Left Outer Join | | |
| | | FULL-OUTER-JOIN | Full Outer Join | | |
| | | ANTI-OUTER-JOIN | Anti Outer Join | | |
| | Set function | BAG-UNION | Bag Union | SET-INTERSECT | Set Intersect |
| | | | | SET-DIFFERENCE | Set Difference |
| | Others | CONCATENATION | Concatenation | | |
| Multi-child | Others | PARTITION-COORDINATOR | Partition-Coordinator | | |

12.3.2 AGGREGATION

12.3.2.1 Definition

This kind of node handles so-called "distinct aggregation". When an aggregate operation is to be performed on a column on which the DISTINCT operation is to be performed first, the node that removes duplicate values from the target column is executed in a different way from the GROUP-AGGREGATION node.

12.3.2.2 Format

```
AGGREGATION ( ITEM_SIZE: item_size, GROUP_COUNT: group_count )
```

| | |
|--------------------|-------------------------|
| <i>item_size</i> | Size of the stored item |
| <i>group_count</i> | Total group count |

12.3.2.3 Example

Display the total number of departments and the average salary of all employees.

```
iSQL> SELECT COUNT(DISTINCT dno), AVG(salary) FROM employees;
COUNT(DISTINCT DNO)  AVG(SALARY)
-----
8                      1836.64706
1 row selected.
-----
PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 31 )
AGGREGATION ( ITEM_SIZE: 72, GROUP_COUNT: 1 )
  SCAN ( TABLE: EMPLOYEES, FULL SCAN, ACCESS: 20, SELF_ID: 1 )
-----
```

12.3.3 ANTI-OUTER-JOIN

12.3.3.1 Definition

This node is created so that FULL-OUTER-JOINS can be processed more quickly using an index, and is a binary node.

12.3.3.2 Format

ANTI-OUTER-JOIN

12.3.3.3 Example

Output the name and number of the department and the number of the product for all locations, including the department locations and the locations where products are stored.

```
iSQL> CREATE INDEX dep_idx2 ON department(dep_location);
Create success.
iSQL> CREATE INDEX gds_idx1 ON goods(goods_location);
Create success.
iSQL> SELECT d.dno, d.dname, g.gno
  FROM department d FULL OUTER JOIN goods g
    ON d.dep_location = g.goods_location;
DNO      DNAME      GNO
-----
.
.
.
38 rows selected.
-----
PROJECT ( COLUMN_COUNT: 3, TUPLE_SIZE: 46 )
CONCATENATION
  LEFT-OUTER-JOIN
    SCAN ( TABLE: DEPARTMENT D, FULL SCAN, ACCESS: 38, SELF_ID: 1 )
    SCAN ( TABLE: GOODS G, INDEX: GDS_IDX1, ACCESS: 38, SELF_ID: 2 )
  ANTI-OUTER-JOIN
    SCAN ( TABLE: GOODS G, FULL SCAN, ACCESS: 38, SELF_ID: 2 )
    SCAN ( TABLE: DEPARTMENT D, INDEX: DEP_IDX2, ACCESS: 38, SELF_ID: 1 )
-----
iSQL> DROP INDEX dep_idx2;
Drop success.
iSQL> DROP INDEX gds_idx1;
```

12.3 Plan Nodes

Drop success.

12.3.4 BAG-UNION

12.3.4.1 Definition

This node performs bag union operations, and has two or more child nodes. It is equivalent to the UNION ALL SQL statement.

12.3.4.2 Format

BAG-UNION

12.3.4.3 Example

Display the name and salary of all sales representatives and all employees whose salary is greater than 2000000.

```
iSQL> SELECT e_firstname, e_lastname, emp_job, salary
FROM employees
WHERE emp_job = 'SALES REP'
UNION ALL
SELECT e_firstname, e_lastname, emp_job, salary
FROM employees
WHERE salary > 2000;
```

| E_FIRSTNAME | E_LASTNAME | EMP_JOB | SALARY |
|-------------|------------|------------|--------|
| SANDRA | HAMMOND | SALES REP | 1890 |
| ALVAR | MARQUEZ | SALES REP | 1800 |
| WILLIAM | BLAKE | SALES REP | |
| FARHAD | GHOEBANI | PL | 2500 |
| ELIZABETH | BAE | PROGRAMMER | 4000 |
| ZHEN | LIU | WEBMASTER | 2750 |
| YUU | MIURA | PM | 2003 |
| WEI-WEI | CHEN | MANAGER | 2300 |

8 rows selected.

```
-----
PROJECT ( COLUMN_COUNT: 4, TUPLE_SIZE: 70 )
VIEW ( ACCESS: 8, SELF_ID: 4 )
BAG-UNION
PROJECT ( COLUMN_COUNT: 4, TUPLE_SIZE: 70 )
SCAN ( TABLE: EMPLOYEES, FULL SCAN, ACCESS: 20, SELF_ID: 2 )
PROJECT ( COLUMN_COUNT: 4, TUPLE_SIZE: 70 )
SCAN ( TABLE: EMPLOYEES, FULL SCAN, ACCESS: 20, SELF_ID: 3 )
-----
```

12.3.5 CONCATENATION

12.3.5.1 Definition

This node is used for quick processing of an OR condition (DNF). This node is a binary node, and executes the left child and right child in sequence. This node also performs concatenation for Full Outer Joins.

12.3.5.2 Format

CONCATENATION

12.3.5.3 Example

Please refer to the ANTI-OUTER-JOIN node example.

12.3.6 COUNT

12.3.6.1 Definition

This node is used to quickly process a COUNT(*) operation.

12.3.6.2 Format

1. In the case where an index is used:

```
COUNT (TABLE: tbl_name, INDEX: index_name, ACCESS: acc_num,
SELF_ID: node_id, REF_ID: ref_id )
```

2. In the case where an index is not used:

```
COUNT (TABLE: tbl_name, FULL SCAN, ACCESS: acc_num, SELF_ID:
node_id, REF_ID: ref_id )
```

| | |
|-------------------|---|
| <i>table_name</i> | table name |
| FULL SCAN | Index not used |
| <i>index_name</i> | Index used |
| <i>acc_num</i> | Number of times that individual records were accessed |
| <i>node_id</i> | ID in the tuple set (Taken as the node ID) |
| <i>ref_id</i> | Reference node ID |

12.3.6.3 Example

Calculate the total number of employees.

```
iSQL> SELECT COUNT(*) rec_count FROM employees;
REC_COUNT
-----
20
1 row selected.
-----
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 8 )
COUNT ( TABLE: EMPLOYEES, FULL SCAN, ACCESS: 1, SELF_ID: 1, REF_ID: 1 )
-----
```

12.3 Plan Nodes

12.3.7 DISTINCT

12.3.7.1 Definition

This is a materialized node that executes DISTINCT operations.

12.3.7.2 Format

1. In the case where intermediate results are cached in memory:

```
DISTINCT ( ITEM_SIZE: item_size, ITEM_COUNT: item_count,  
          BUCKET_COUNT: bucket_count, ACCESS: acc_num, SELF_ID: self_id,  
          REF_ID: ref_id )
```

2. In the case where intermediate results are stored on disk:

```
DISTINCT ( ITEM_SIZE: item_size, ITEM_COUNT: item_count,  
          DISK_PAGE_COUNT: page_count, ACCESS: acc_num, SELF_ID: self_id,  
          REF_ID: ref_id )
```

| | |
|---------------------|---|
| <i>item_size</i> | Size of the stored item |
| <i>item_count</i> | Number of stored items |
| <i>bucket_count</i> | Number of hash buckets |
| <i>page_count</i> | Number of pages |
| <i>acc_num</i> | Number of times that individual records were accessed |
| <i>self_id</i> | Node ID |
| <i>ref_id</i> | Reference node ID |

12.3.7.3 Example

Display the name of the customer who ordered product C111100001.

```
iSQL> SELECT DISTINCT customer.cname  
      FROM customer  
      WHERE customer.cno IN  
            (SELECT orders.cno  
             FROM orders  
             WHERE orders.gno = 'C111100001');  
CNAME  
-----  
YSKIM  
DKKIM  
IJLEE  
CHLEE  
4 rows selected.  
-----  
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 22 )  
  DISTINCT ( ITEM_SIZE: 24, ITEM_COUNT: 4, BUCKET_COUNT: 1024, ACCESS: 4,  
            SELF_ID: 6, REF_ID: 2 )  
    SCAN ( TABLE: CUSTOMER, INDEX: __SYS_IDX_ID_124, ACCESS: 4, SELF_ID: 2 )  
      ::SUB-QUERY BEGIN  
        PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 8 )  
          DISTINCT ( ITEM_SIZE: 24, ITEM_COUNT: 4, BUCKET_COUNT: 1024, ACCESS: 8,  
                    SELF_ID: 5, REF_ID: 3 )
```

```

VIEW ( ACCESS: 4, SELF_ID: 4 )
  PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 8 )
    SCAN ( TABLE: ORDERS, INDEX: ODR_IDX3, ACCESS: 4, SELF_ID: 3 )
  ::SUB-QUERY END
-----

```

12.3.8 FILTER

12.3.8.1 Definition

This node performs selection, that is, it checks the results returned by subnodes against some conditions.

12.3.8.2 Format

```
FILTER
```

12.3.8.3 Example

For all products of which two or more were ordered, display the product number and the quantity ordered.

```

iSQL> SELECT gno, COUNT(*)
FROM orders
GROUP BY gno
HAVING COUNT(*) > 2;
GNO          COUNT
-----

```

```

A111100002  3
C111100001  4
D111100008  3
E111100012  3
4 rows selected.
-----

```

```

PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 24 )
  FILTER
    AGGREGATION ( ITEM_SIZE: 24, GROUP_COUNT: 16 )
    GROUPING
      SCAN ( TABLE: ORDERS, INDEX: ODR_IDX3, ACCESS: 30, SELF_ID: 2 )
-----

```

12.3.9 FULL-OUTER-JOIN

12.3.9.1 Definition

This performs Full Outer Join operations, and is a binary node.

12.3.9.2 Format

```
FULL-OUTER-JOIN
```

12.3 Plan Nodes

12.3.9.3 Example

Output the name and number of the department and the number of the product for all locations, including the department locations and the locations where products are stored.

```
iSQL> INSERT INTO department VALUES(6002, 'headquarters', 'CE0002', 100);
1 row inserted.
```

```
iSQL> SELECT d.dno, d.dname, g.gno
FROM department d FULL OUTER JOIN goods g
ON d.dep_location = g.goods_LOCATION;
```

| DNO | DNAME | GNO |
|------|-----------------------------|------------|
| | | A111100001 |
| | | A111100002 |
| | | B111100001 |
| | | C111100001 |
| | | C111100002 |
| | | D111100001 |
| | | D111100002 |
| | | D111100003 |
| | | D111100004 |
| | | D111100005 |
| | | D111100006 |
| | | D111100007 |
| | | D111100008 |
| | | D111100009 |
| | | D111100010 |
| | | D111100011 |
| | | E111100001 |
| | | E111100002 |
| | | E111100003 |
| | | E111100004 |
| 6002 | headquarters | E111100005 |
| | | E111100006 |
| | | E111100007 |
| | | E111100008 |
| | | E111100009 |
| | | E111100010 |
| | | E111100011 |
| | | E111100012 |
| | | E111100013 |
| | | F111100001 |
| 1001 | RESEARCH DEVELOPMENT DEPT 1 | |
| 1002 | RESEARCH DEVELOPMENT DEPT 2 | |
| 1003 | SOLUTION DEVELOPMENT DEPT | |
| 4001 | MARKETING DEPT | |
| 2001 | QUALITY ASSURANCE DEPT | |
| 3001 | CUSTOMER SUPPORT DEPT | |
| 3002 | PRESALES DEPT | |
| 4002 | BUSINESS DEPT | |

38 rows selected.

```
-----
PROJECT ( COLUMN_COUNT: 3, TUPLE_SIZE: 46 )
FULL-OUTER-JOIN
  SCAN ( TABLE: GOODS G, FULL SCAN, ACCESS: 38, SELF_ID: 2 )
  HASH ( ITEM_SIZE: 24, ITEM_COUNT: 9, BUCKET_COUNT: 1024, ACCESS: 38,
SELF_ID: 3, REF_ID: 1 )
  SCAN ( TABLE: DEPARTMENT D, FULL SCAN, ACCESS: 38, SELF_ID: 1 )
-----
```


12.3.10 GROUP-AGGREGATION

12.3.10.1 Definition

This is a materialized node that performs group aggregation operations.

12.3.10.2 Format

1. In the case where intermediate results are cached in memory:

```
GROUP-AGGREGATION ( ITEM_SIZE: item_size, GROUP_COUNT:
  group_count, BUCKET_COUNT: bucket_count, ACCESS: acc_num,
  SELF_ID: self_id, REF_ID: ref_id )
```

2. In the case where intermediate results are stored on disk:

```
GROUP-AGGREGATION ( ITEM_SIZE: item_size, GROUP_COUNT:
  group_count, DISK_PAGE_COUNT: page_count, ACCESS: acc_num,
  SELF_ID: self_id, REF_ID: ref_id )
```

| | |
|---------------------|---|
| <i>item_size</i> | Size of the stored item |
| <i>group_count</i> | Total group count |
| <i>bucket_count</i> | Number of hash buckets |
| <i>page_count</i> | Number of pages |
| <i>acc_num</i> | Number of times that individual records were accessed |
| <i>self_id</i> | Node ID |
| <i>ref_id</i> | Reference node ID |

12.3.10.3 Example

Display the total amount of wages paid to each position within each department. (Here, the GROUP BY clause is used with multiple columns.)

```
iSQL> SELECT dno, emp_job, COUNT(emp_job) num_emp, SUM(salary) sum_sal FROM
employees GROUP BY dno, emp_job;
```

```
DNO  EMP_JOB    NUM_EMP SUM_SAL
```

```
-----
```

```
.
```

```
.
```

```
.
```

```
16 rows selected.
```

```
-----
```

```
PROJECT ( COLUMN_COUNT: 4, TUPLE_SIZE: 55 )
```

```
  GROUP-AGGREGATION ( ITEM_SIZE: 56, GROUP_COUNT: 16, BUCKET_COUNT: 1024,
  ACCESS: 16, SELF_ID: 2, REF_ID: 1 )
```

```
    SCAN ( TABLE: EMPLOYEES, FULL SCAN, ACCESS: 20, SELF_ID: 1 )
```

```
-----
```

12.3 Plan Nodes

12.3.11 GROUPING

12.3.11.1 Definition

This node checks whether the data retrieved from the subordinate node belongs to the same group as the previous data.

12.3.11.2 Format

GROUPING

12.3.11.3 Example

Display the number of customers managed by each employee and the total number of products purchased by each employee's customers.

```
iSQL> SELECT eno, COUNT(DISTINCT cno), SUM(qty) FROM orders GROUP BY eno;
ENO          COUNT(DISTINCT CNO)  SUM(QTY)
-----
12             8                17870
19             6                25350
20             8                13210
3 rows selected.

PROJECT ( COLUMN_COUNT: 3, TUPLE_SIZE: 24 )
AGGREGATION ( ITEM_SIZE: 32, GROUP_COUNT: 3 )
  GROUPING
    SCAN ( TABLE: ORDERS, INDEX: ODR_IDX1, ACCESS: 30, SELF_ID: 1 )
```

12.3.12 HASH

12.3.12.1 Definition

This is a materialized node that performs hash searching.

12.3.12.2 Format

1. In the case where intermediate results are stored in memory:

```
HASH ( ITEM_SIZE: item_size, ITEM_COUNT: item_count,
      BUCKET_COUNT: bucket_count, ACCESS: acc_num, SELF_ID: self_id,
      REF_ID: ref_id )
```

2. In the case where intermediate results are stored on disk:

```
HASH ( ITEM_SIZE: item_size, ITEM_COUNT: item_count,
      DISK_PAGE_COUNT: page_count, ACCESS: acc_num, SELF_ID: self_id,
      REF_ID: ref_id )
```

item_size Size of the stored item

| | |
|---------------------|---|
| <i>item_count</i> | Number of stored items |
| <i>bucket_count</i> | Number of hash buckets |
| <i>page_count</i> | Number of pages |
| <i>acc_num</i> | Number of times that individual records were accessed |
| <i>self_id</i> | ID in the tuple set. Taken as the node ID. |
| <i>ref_id</i> | Reference node ID |

12.3.12.3 Example

Display the name and the department name for all department managers.

```
iSQL> ALTER SESSION SET EXPLAIN PLAN = OFF;
Alter success.
iSQL> CREATE TABLE dept2 TABLESPACE sys_tbs_disk_data
AS SELECT * FROM department;
Create success.
iSQL> CREATE TABLE manager(
  eno INTEGER PRIMARY KEY,
  mgr_no INTEGER,
  mname VARCHAR(20),
  address VARCHAR(60))
TABLESPACE sys_tbs_disk_data;
Create success.
iSQL> INSERT INTO manager VALUES(2, 1, 'HJNO', '11 Inyoung Bldg. Nonhyun-dong
Kangnam-guSeoul, Korea');
1 row inserted.
iSQL> INSERT INTO manager VALUES(7, 2, 'HJMIN', '44-25 Youido-dong Young-
dungpo-gu Seoul, Korea');
1 row inserted.
iSQL> INSERT INTO manager VALUES(8, 7, 'JDLEE', '3101 N. Wabash Ave. Brook-
lyn, NY');
1 row inserted.
iSQL> INSERT INTO manager VALUES(12, 7, 'MYLEE', '130 Gongpyeongno Jung-gu
Daegu, Korea');
1 row inserted.
iSQL> ALTER SESSION SET EXPLAIN PLAN = ON;
Alter success.
iSQL> SELECT m.mname, d.dname
  FROM dept2 d, manager m
 WHERE d.mgr_no = m.mgr_no;
MNAME                DNAME
-----
JDLEE                BUSINESS DEPT
MYLEE                BUSINESS DEPT
2 rows selected.
-----
PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 54 )
  JOIN
    SCAN ( TABLE: DEPT2 D, FULL SCAN, ACCESS: 10, DISK_PAGE_COUNT: 64, SELF_ID:
1 )
    HASH ( ITEM_SIZE: 40, ITEM_COUNT: 4, DISK_PAGE_COUNT: 2, ACCESS: 2,
SELF_ID: 3, REF_ID: 2 )
    SCAN ( TABLE: MANAGER M, FULL SCAN, ACCESS: 4, DISK_PAGE_COUNT: 64,
SELF_ID: 2 )
-----
```

12.3 Plan Nodes

12.3.13 HIERARCHICAL QUERY

12.3.13.1 Definition

This node processes queries of hierarchically arranged data.

12.3.13.2 Format

1. In the case where an index is used:

```
HIER ( TABLE: table_name, FULL SCAN, ACCESS: acc_num, SELF_ID:
      node_id )
```

2. In the case where an index is not used:

```
HIER ( TABLE: table_name, INDEX: index_name, ACCESS: acc_num,
      SELF_ID: node_id )
```

| | |
|-------------------|---|
| <i>table_name</i> | Table name |
| FULL SCAN | Index not used |
| <i>index_name</i> | Index used |
| <i>acc_num</i> | Number of times that individual records were accessed |
| <i>node_id</i> | ID in the tuple set (taken as the node ID) |

12.3.13.3 Example

The hierarchical query statement used to acquire rows that are hierarchically related to the row for which the value in the id column is 0 is as follows:

```
isQL> alter session set explain plan = off;
Alter success.
isQL> CREATE TABLE hier_order(id INTEGER, parent INTEGER);
Create success.
isQL> INSERT INTO hier_order VALUES(0, NULL);
1 row inserted.
isQL> INSERT INTO hier_order VALUES(1, 0);
1 row inserted.
isQL> INSERT INTO hier_order VALUES(2, 1);
1 row inserted.
isQL> INSERT INTO hier_order VALUES(3, 1);
1 row inserted.
isQL> INSERT INTO hier_order VALUES(4, 1);
1 row inserted.
isQL> INSERT INTO hier_order VALUES(5, 0);
1 row inserted.
isQL> INSERT INTO hier_order VALUES(6, 0);
1 row inserted.
isQL> INSERT INTO hier_order VALUES(7, 6);
1 row inserted.
isQL> INSERT INTO hier_order VALUES(8, 7);
1 row inserted.
isQL> INSERT INTO hier_order VALUES(9, 7);
1 row inserted.
isQL> INSERT INTO hier_order VALUES(10, 6);
1 row inserted.
```

```

iSQL> alter session set explain plan = on;
Alter success.
iSQL> SELECT id, parent, level FROM hier_order START WITH id = 0 CONNECT BY
PRIOR id = parent ORDER BY level;

```

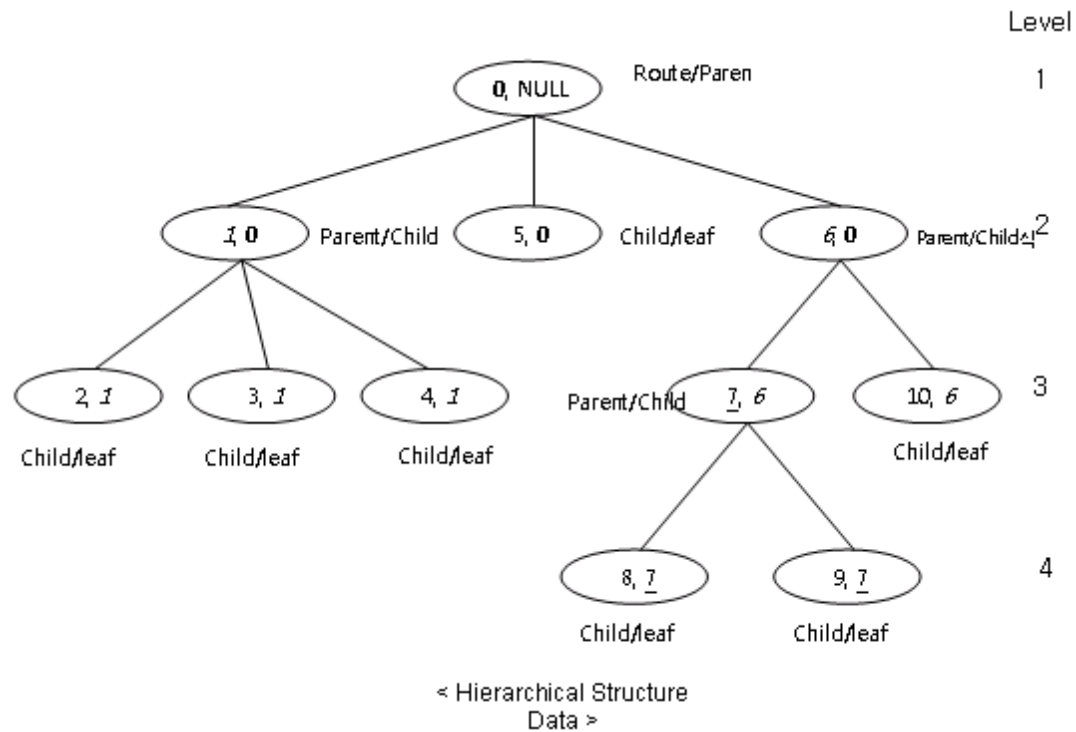
| ID | PARENT | LEVEL |
|----|--------|-------|
| 0 | | 1 |
| 6 | 0 | 2 |
| 5 | 0 | 2 |
| 1 | 0 | 2 |
| 10 | 6 | 3 |
| 4 | 1 | 3 |
| 7 | 6 | 3 |
| 3 | 1 | 3 |
| 2 | 1 | 3 |
| 8 | 7 | 4 |
| 9 | 7 | 4 |

11 rows selected.

```

-----
PROJECT ( COLUMN_COUNT: 3, TUPLE_SIZE: 16 )
SORT ( ITEM_SIZE: 24, ITEM_COUNT: 11, ACCESS: 11, SELF_ID: 5, REF_ID: 2 )
  HIER ( TABLE: HIER_ORDER, FULL SCAN, ACCESS: 132, SELF_ID: 2 )
    SCAN ( TABLE: HIER_ORDER, FULL SCAN, ACCESS: 132, SELF_ID: 2 )
-----

```



12.3.14 JOIN

12.3.14.1 Definition

This node calculates Cartesian products. Its right child evaluates JOIN conditions.

12.3 Plan Nodes

12.3.14.2 Format

JOIN

12.3.14.3 Example

Display the employee number, order number, product number, and number of units ordered for all orders corresponding to any employees whose last name is 'Marquez'.

```
iSQL> SELECT e.eno, ono, cno, gno, qty
FROM employees e, orders o
WHERE e.eno = o.eno
AND e.e_lastname = 'Marquez';
```

| ENO | ONO | CNO | GNO | QTY |
|-----|----------|-----|------------|-------|
| 19 | 11290100 | 11 | E11110000 | 1500 |
| 19 | 12100277 | 5 | D111100008 | 2500 |
| 19 | 12300001 | 1 | D111100004 | 1000 |
| 19 | 12300005 | 4 | D111100008 | 4000 |
| 19 | 12300010 | 16 | D111100010 | 2000 |
| 19 | 12310004 | 5 | E111100010 | 5000 |
| 19 | 12310008 | 1 | D111100003 | 100 |
| 19 | 12310011 | 15 | E111100012 | 10000 |
| 19 | 12310012 | 1 | C111100001 | 250 |

9 rows selected.

```
-----
PROJECT ( COLUMN_COUNT: 5, TUPLE_SIZE: 40 )
JOIN
  SCAN ( TABLE: EMPLOYEES E, FULL SCAN, ACCESS: 20, SELF_ID: 2 )
  SCAN ( TABLE: ORDERS O, INDEX: ODR_IDX1, ACCESS: 9, SELF_ID: 3 )
-----
```

12.3.15 LEFT-OUTER-JOIN

12.3.15.1 Definition

This kind of node evaluates left outer joins. It is a binary node.

12.3.15.2 Format

LEFT-OUTER-JOIN

12.3.15.3 Example

Display all department numbers and the names of all of the employees in each department. (Note that 5001, the number of a department without any staff, is also output.)

```
iSQL> INSERT INTO departments VALUES(5001, 'Quality Assurance', 'Mokpo', 22);
1 row inserted.
iSQL> SELECT d.dno, e.e_firstname, e.e_lastname FROM departments d LEFT OUTER
JOIN employees e ON d.dno = e.dno ORDER BY d.dno;
```

| DNO | E_FIRSTNAME | E_LASTNAME |
|------|-------------|------------|
| 1001 | Ken | Kobain |
| 1001 | Wei-Wei | Chen |
| 1002 | Ryu | Momoi |

```

1002      Mitch      Jones
1003      Elizabeth  Bae
1003      Zhen       Liu
1003      Yuu        Miura
1003      Jason      Davenport
2001      Takahiro   Fubuki
3001      Aaron      Foster
3002      Chan-seung Moon
3002      Farhad     Ghorbani
4001      Xiong      Wang
4001      Curtis     Diaz
4001      John       Huxley
4002      Gottlieb   Fleischer
4002      Sandra     Hammond
4002      Alvar      Marquez
4002      William    Blake
5001
20 rows selected.
-----
PROJECT ( COLUMN_COUNT: 3, TUPLE_SIZE: 46 )
  LEFT-OUTER-JOIN
    SCAN ( TABLE: DEPARTMENTS D, INDEX: __SYS_IDX_ID_137, ACCESS: 9, SELF_ID: 1
  )
    SCAN ( TABLE: EMPLOYEES E, INDEX: EMP_IDX1, ACCESS: 20, SELF_ID: 2 )
-----

```

12.3.16 LIMIT-SORT

12.3.16.1 Definition

This kind of node performs so-called "limit sorting". Because not all data need to be stored, the data are sorted using only a set amount of storage space.

12.3.16.2 Format

```

LIMIT-SORT ( ITEM_SIZE: item_size, ITEM_COUNT: item_count,
STORE_COUNT: store_count, ACCESS: acc_num, SELF_ID: self_id, REF_ID:
ref_id )

```

| | |
|--------------------|---|
| <i>item_size</i> | Size of the stored item |
| <i>item_count</i> | Number of stored items |
| <i>store_count</i> | Number of sorted items |
| <i>acc_num</i> | Number of times that individual records were accessed |
| <i>self_id</i> | ID in the tuple set (taken as the node ID) |
| <i>ref_id</i> | Reference node ID |

12.3.16.3 Example

Sort all employees, first by department number and then by wage, and then output the name, department number and salary of the first 10 employees. (Here, the sort order is determined by the list of columns in the ORDER BY clause.)

```

iSQL> SELECT e_firstname, e_lastname, dno, salary
FROM employees
ORDER BY dno, salary DESC

```

12.3 Plan Nodes

```
LIMIT 10;
E_FIRSTNAME      E_LASTNAME      DNO      SALARY
-----
Wei-Wei          Chen            1001      2300
Ken              Kobain          1001      2000
Ryu              Momoi           1002      1700
Mitch            Jones           1002      980
Elizabeth        Bae             1003      4000
Zhen             Liu             1003      2750
Yuu              Miura           1003      2003
Jason            Davenport       1003      1000
Takahiro         Fubuki          2001      1400
Aaron            Foster          3001      1800
10 rows selected.
-----
PROJECT ( COLUMN_COUNT: 4, TUPLE_SIZE: 55 )
  LIMIT-SORT ( ITEM_SIZE: 16, ITEM_COUNT: 20, STORE_COUNT: 10, ACCESS: 10,
  SELF_ID: 1, REF_ID: 0 )
    SCAN ( TABLE: EMPLOYEES, FULL SCAN, ACCESS: 20, SELF_ID: 0 )
-----
```

12.3.17 MATERIALIZATION

12.3.17.1 Definition

This node is used so that views can be processed quickly. Records created by a subordinate VIEW node are stored in a virtual table, that is, a view.

12.3.17.2 Format

```
MATERIALIZATION ( ACCESS: acc_num, SELF_ID: self_id )
```

| | |
|----------------|---|
| <i>acc_num</i> | Number of times that individual records were accessed |
| <i>self_id</i> | Node ID |

12.3.17.3 Example

Display the name, department number, and wage of each employee whose wage is higher than the average wage in that employee's department, but lower than the average wage in the department having the highest average wage.

```
iSQL> CREATE VIEW v1 AS
(SELECT dno, AVG(salary) avg_sal
FROM employees GROUP BY dno);
Create success.
iSQL> SELECT e_firstname, e_lastname, e.dno, e.salary
FROM employees e, v1
WHERE e.dno = v1.dno
AND e.salary > v1.avg_sal
AND e.salary < (SELECT MAX(avg_sal)
FROM v1);
E_FIRSTNAME      E_LASTNAME      DNO      SALARY
-----
Wei-Wei          Chen            1001      2300
Ryu              Momoi           1002      1700
John             Huxley          4001      1900
Sandra           Hammond          4002      1890
```



```

Alvar                      Marquez                      4002          1800
5 rows selected.
-----
PROJECT ( COLUMN_COUNT: 4, TUPLE_SIZE: 55 )
  JOIN
    VIEW-SCAN ( VIEW: V1, ACCESS: 9, SELF_ID: 3 )
      MATERIALIZATION ( ITEM_SIZE: 40, ITEM_COUNT: 9 )
        VIEW ( ACCESS: 9, SELF_ID: 8 )
          PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 25 )
            AGGREGATION ( ITEM_SIZE: 72, GROUP_COUNT: 9 )
              GROUPING
                SCAN ( TABLE: EMPLOYEES, INDEX: EMP_IDX1, ACCESS: 20, SELF_ID: 2 )
            SCAN ( TABLE: EMPLOYEES E, INDEX: EMP_IDX1, ACCESS: 19, SELF_ID: 1 )
          ::SUB-QUERY BEGIN
            PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 23 )
              STORE ( ITEM_SIZE: 32, ITEM_COUNT: 1, ACCESS: 14, SELF_ID: 12, REF_ID: 5 )
                VIEW ( ACCESS: 1, SELF_ID: 11 )
                  PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 23 )
                    GROUP-AGGREGATION ( ITEM_SIZE: 40, GROUP_COUNT: 1, BUCKET_COUNT: 1,
ACCESS: 1, SELF_ID: 10, REF_ID: 5 )
                  VIEW-SCAN ( VIEW: V1, ACCESS: 9, SELF_ID: 5 )
                ::SUB-QUERY END
          -----

```

12.3.18 MERGE-JOIN

12.3.18.1 Definition

Performs the Merge Join function.

12.3.18.2 Format

MERGE-JOIN

12.3.18.3 Example

Display the number, last name, department number and department name of all employees who joined the company before 1 Jan, 2010.

```

iSQL> SELECT e.eno, e_lastname, d.dno, dname
FROM employees e, departments d
WHERE e.dno = d.dno
      AND TO_CHAR(join_date, 'YYYY-MM-DD HH:MI:SS') < '2010-01-01 00:00:00';
ENO          E_LASTNAME          DNO          DNAME
-----
5            Ghorbani            3002          PRESALES DEPT
8            Wang                4001          MARKETING DEPT
18           Huxley                4001          MARKETING DEPT
7            Fleischer            4002          BUSINESS DEPT
12           Hammond            4002          BUSINESS DEPT
20           Blake                4002          BUSINESS DEPT
6 rows selected.
-----
PROJECT ( COLUMN_COUNT: 4, TUPLE_SIZE: 60 )
  MERGE-JOIN
    SCAN ( TABLE: DEPARTMENTS D, INDEX: __SYS_IDX_ID_137, ACCESS: 9, SELF_ID: 3
)
    SCAN ( TABLE: EMPLOYEES E, INDEX: EMP_IDX1, ACCESS: 19, SELF_ID: 2 )

```

12.3 Plan Nodes

12.3.19 PROJECT

12.3.19.1 Definition

This node performs projection and outputs designated columns.

12.3.19.2 Format

```
PROJECT ( COLUMN_COUNT: col_count, TUPLE_SIZE: tuple_size )
```

| | |
|-------------------|---|
| <i>col_count</i> | Number of columns |
| <i>tuple_size</i> | Actual size of the tuple returned by projection |

12.3.19.3 Example

Display the name and salary of all employees after increasing the salary of all employees by 10%.

```
iSQL> SELECT e_firstname, e_lastname, salary * 1.1 FROM employees;
E_FIRSTNAME      E_LASTNAME      SALARY * 1.1
-----
.
.
.
20 rows selected.
-----
PROJECT ( COLUMN_COUNT: 3, TUPLE_SIZE: 67 )
SCAN ( TABLE: EMPLOYEES, FULL SCAN, ACCESS: 20, SELF_ID: 2 )
-----
```

12.3.20 SCAN

12.3.20.1 Definition

This node performs selection and scans a table for records that meet a condition using Key Range and Filter processing.

12.3.20.2 Format

1. In the case where intermediate results are stored in memory:

```
SCAN ( TABLE: table_name, FULL SCAN, ACCESS: acc_num, SELF_ID:
node_id )
```

```
SCAN ( TABLE: table_name, INDEX: index_name, ACCESS: acc_num,
SELF_ID: node_id )
```

2. In the case where intermediate results are stored on disk:

```
SCAN ( TABLE: table_name, FULL SCAN, ACCESS: acc_num,
DISK_PAGE_COUNT: page_count, SELF_ID: node_id )
```

```
SCAN ( TABLE: table_name, INDEX: index_name, ACCESS: acc_num,
DISK_PAGE_COUNT: page_count, SELF_ID: node_id )
```

| | |
|-------------------|--|
| <i>table_name</i> | Name of the table |
| FULL SCAN | No index is used. |
| <i>index_name</i> | Name of the index to use. |
| <i>page_count</i> | Number of pages |
| <i>node_id</i> | ID in the tuple set (taken as the node ID) |

12.3.20.3 Example

Example1) Display the names, department numbers and birthdays of all employees who were born before January 1, 1980.

```
iSQL> SELECT e_firstname, e_lastname, dno, birth
FROM employees
WHERE birth > '800101';
```

| E_FIRSTNAME | E_LASTNAME | DNO | BIRTH |
|-------------|------------|------|--------|
| Aaron | Foster | 3001 | 820730 |
| Gottlieb | Fleischer | 4002 | 840417 |
| Xiong | Wang | 4001 | 810726 |
| Sandra | Hammond | 4002 | 810211 |
| Mitch | Jones | 1002 | 801102 |
| Jason | Davenport | 1003 | 901212 |

6 rows selected.

```
PROJECT ( COLUMN_COUNT: 4, TUPLE_SIZE: 54 )
SCAN ( TABLE: EMPLOYEES, FULL SCAN, ACCESS: 20, SELF_ID: 2 )
```

Example2} Output the names, department numbers, and birthdays of all employees who were born before January 1, 1980. (Use an index.)

```
iSQL> CREATE INDEX emp_idx2 ON employees(birth);
Create success.
```

```
iSQL> SELECT e_firstname, e_lastname, dno, birth
FROM employees
WHERE birth > '800101';
```

| E_FIRSTNAME | E_LASTNAME | DNO | BIRTH |
|-------------|------------|------|--------|
| Mitch | Jones | 1002 | 801102 |
| Sandra | Hammond | 4002 | 810211 |
| Xiong | Wang | 4001 | 810726 |
| Aaron | Foster | 3001 | 820730 |
| Gottlieb | Fleischer | 4002 | 840417 |
| Jason | Davenport | 1003 | 901212 |

6 rows selected.

```
PROJECT ( COLUMN_COUNT: 4, TUPLE_SIZE: 54 )
SCAN ( TABLE: EMPLOYEES, INDEX: EMP_IDX2, ACCESS: 6, SELF_ID: 2 )
```

12.3.21 SET-DIFFERENCE

12.3 Plan Nodes

12.3.21.1 Definition

This node performs set difference operations. This node is a binary node and a materialized node. This node is used to process the MINUS SQL operation.

12.3.21.2 Format

1. In the case where intermediate results are stored in memory:

```
SET-DIFFERENCE ( ITEM_SIZE: item_size, ITEM_COUNT: item_count,  
BUCKET_COUNT: bucket_count, ACCESS: acc_num, SELF_ID: self_id,  
L_REF_ID: l_ref_id, R_REF_ID: r_ref_id )
```

2. In the case where intermediate results are stored on disk:

```
SET-DIFFERENCE ( ITEM_SIZE: item_size, ITEM_COUNT: item_count,  
DISK_PAGE_COUNT: page_count, ACCESS: acc_num, SELF_ID: self_id,  
L_REF_ID: l_ref_id, R_REF_ID: r_ref_id )
```

| | |
|---------------------|---|
| <i>item_size</i> | Size of the stored item |
| <i>item_count</i> | Number of stored items |
| <i>bucket_count</i> | Number of hash buckets |
| <i>page_count</i> | Number of pages |
| <i>acc_num</i> | Number of times that individual records were accessed |
| <i>self_id</i> | Node ID |
| <i>l_ref_id</i> | Left reference node ID. If the node is independent, the left child data do not need to be stored again. |
| <i>r_ref_id</i> | Right reference node ID. If the node is independent, the right child data do not need to be stored again. |

12.3.21.3 Example

Display the product numbers of products that have not been ordered.

```
isQL> SELECT gno FROM goods  
MINUS  
SELECT gno FROM orders;  
GNO  
-----  
.  
.  
.  
14 rows selected.  
-----  
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 12 )  
  VIEW ( ACCESS: 14, SELF_ID: 2 )  
    SET-DIFFERENCE ( ITEM_SIZE: 32, ITEM_COUNT: 30, BUCKET_COUNT: 1024, ACCESS:  
14, SELF_ID: 4, L_REF_ID: 0, R_REF_ID: 1 )  
      PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 12 )  
        SCAN ( TABLE: GOODS, FULL SCAN, ACCESS: 30, SELF_ID: 0 )  
      PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 12 )
```

```
SCAN ( TABLE: ORDERS, FULL SCAN, ACCESS: 30, SELF_ID: 1 )
```

12.3.22 SET-INTERSECT

12.3.22.1 Definition

This node performs set intersection. This node is a binary node and a materialized node. This node is used to process the INTERSECT SQL operation.

12.3.22.2 Format

1) In the case where intermediate results are stored in memory:

```
SET-INTERSECT ( ITEM_SIZE: item_size, ITEM_COUNT: item_count,
BUCKET_COUNT: bucket_count, ACCESS: acc_num, SELF_ID: self_id,
L_REF_ID: l_ref_id, R_REF_ID: r_ref_id )
```

2) In the case where intermediate results are stored on disk:

```
SET-INTERSECT ( ITEM_SIZE: item_size, ITEM_COUNT: item_count,
DISK_PAGE_COUNT: page_count, ACCESS: acc_num, SELF_ID: self_id,
L_REF_ID: l_ref_id, R_REF_ID: r_ref_id )
```

| | |
|---------------------|---|
| <i>item_size</i> | Size of the stored item |
| <i>item_count</i> | Number of stored items |
| <i>bucket_count</i> | Number of hash buckets |
| <i>page_count</i> | Number of pages |
| <i>acc_num</i> | Number of times that individual records were accessed |
| <i>self_id</i> | Node ID |
| <i>l_ref_id</i> | Left reference node ID. If the node is independent, the left child data do not need to be stored again. |
| <i>r_ref_id</i> | Right reference node ID. If the node is independent, the right child data do not need to be stored again. |

12.3.22.3 Example

Output a list of all items in the *goods* table that have been ordered at least once.

```
iSQL> SELECT gno FROM goods INTERSECT SELECT gno FROM orders;
GNO
```

```
-----
A111100002
E111100001
D111100008
D111100004
C111100001
E111100002
D111100002
D111100011
D111100003
D111100010
E111100012
```

12.3 Plan Nodes

```
F111100001
E111100009
E111100010
E111100007
E111100013
16 rows selected.
-----
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 12 )
  VIEW ( ACCESS: 16, SELF_ID: 2 )
    SET-INTERSECT ( ITEM_SIZE: 32, ITEM_COUNT: 30, BUCKET_COUNT: 1024, ACCESS:
16, SELF_ID: 4, L_REF_ID: 0, R_REF_ID: 1 )
      PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 12 )
        SCAN ( TABLE: GOODS, FULL SCAN, ACCESS: 30, SELF_ID: 0 )
      PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 12 )
        SCAN ( TABLE: ORDERS, FULL SCAN, ACCESS: 30, SELF_ID: 1 )
-----
```

12.3.23 SORT

12.3.23.1 Definition

This is a materialized node that performs the sorting function.

12.3.23.2 Format

1. In the case where intermediate results are stored in memory:

```
SORT ( ITEM_SIZE: item_size, ITEM_COUNT: item_count, ACCESS:
acc_num, SELF_ID: self_id, REF_ID: ref_id )
```

2. In the case where intermediate results are stored on disk:

```
SORT ( ITEM_SIZE: item_size, ITEM_COUNT: item_count,
DISK_PAGE_COUNT: page_count, ACCESS: acc_num, SELF_ID: self_id,
REF_ID: ref_id )
```

| | |
|-------------------|---|
| <i>item_size</i> | Size of the stored item |
| <i>item_count</i> | Number of stored items |
| <i>page_count</i> | Number of pages |
| <i>acc_num</i> | Number of times that individual records were accessed |
| <i>self_id</i> | ID in the tuple set (Taken as the node ID) |
| <i>ref_id</i> | ID of the node that is referenced, that is, the node on the basis of which dependency checking is performed |

* Dependency is the basis for determining the validity of the data stored by a materialized node. If the reference node is a dependent node, the result must be determined again, whereas if it is an independent node, the result will be the same, and thus need not be determined again.

12.3.23.3 Example

Display the name, title, hiring date, and wage for all employees whose wage is less than \$1500 USD per month, sorted by wage in descending order.

```
iSQL> SELECT e_firstname, e_lastname, emp_job, salary
```

```

FROM employees
WHERE salary < 1500
ORDER BY 4 DESC;
E_FIRSTNAME      E_LASTNAME      EMP_JOB      SALARY
-----
Takahiro          Fubuki          PM           1400
Curtis            Diaz            planner       1200
Jason            Davenport       webmaster     1000
Mitch            Jones           PM            980
Gottlieb          Fleischer       manager       500
5 rows selected.
-----
PROJECT ( COLUMN COUNT: 4, TUPLE_SIZE: 70 )
  SORT ( ITEM_SIZE: 16, ITEM_COUNT: 5, ACCESS: 5, SELF_ID: 3, REF_ID: 2 )
    SCAN ( TABLE: EMPLOYEES, FULL SCAN, ACCESS: 20, SELF_ID: 2 )
-----

```

12.3.24 VIEW

12.3.24.1 Definition

This node is used to process inline views. This node is used to make the columns in a projection appear to be a virtual record that comprises adjacent columns.

12.3.24.2 Format

```
VIEW ( ACCESS: acc_num, SELF_ID: self_id )
```

| | |
|----------------|---|
| <i>acc_num</i> | Number of times that individual records were accessed |
| <i>self_id</i> | Node ID |

12.3.24.3 Example

Display the name, wage, and department for every employee whose wage is higher than the average wage in his/her department, as well as the average wage in that department.

```

iSQL> SELECT e.e_firstname, e.e_lastname, e.salary, e.dno, v1.salavg
FROM employees e,
  (SELECT dno, AVG(salary) salavg
   FROM employees
   GROUP BY dno) v1
WHERE e.dno = v1.dno
AND e.salary > v1.salavg;
E_FIRSTNAME      E_LASTNAME      SALARY      DNO      SALAVG
-----
Wei-Wei          Chen            2300        1001      2150
Ryu              Momoi           1700        1002      1340
Elizabeth        Bae             4000        1003      2438.25
Zhen             Liu             2750        1003      2438.25
John             Huxley          1900        4001      1550
Sandra           Hammond         1890        4002      1396.66667
Alvar            Marquez         1800        4002      1396.66667
7 rows selected.
-----
PROJECT ( COLUMN_COUNT: 5, TUPLE_SIZE: 79 )
  JOIN
    VIEW ( ACCESS: 9, SELF_ID: 3 )

```

12.3 Plan Nodes

```
PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 25 )
AGGREGATION ( ITEM_SIZE: 72, GROUP_COUNT: 9 )
GROUPING
SCAN ( TABLE: EMPLOYEES, INDEX: EMP_IDX1, ACCESS: 20, SELF_ID: 2 )
SCAN ( TABLE: EMPLOYEES E, INDEX: EMP_IDX1, ACCESS: 19, SELF_ID: 1 )
-----
```

12.3.25 VIEW-SCAN

12.3.25.1 Definition

Scans a virtual table (that is, a view) created by a MATERIALIZATION node.

12.3.25.2 Format

```
VIEW-SCAN ( VIEW: view_name, SELF_ID: self_id )
```

| | |
|------------------|-----------|
| <i>view_name</i> | View name |
| <i>self_id</i> | Node ID |

12.3.25.3 Example

Please refer to the example in the description of the MATERIALIZATION Node.

12.3.26 PARTITION-COORDINATOR

12.3.26.1 Definition

This node manages the scanning of each partition of a partitioned table. It has many child nodes and performs partition filtering.

12.3.26.2 Format

```
PARTITION-COORDINATOR(TABLE: table_name, PARTITION: partition_acc_cnt,  
ACCESS: acc_num, SELF_ID: self_id )
```

| | |
|--------------------------|-------------------------------|
| <i>table_name</i> | Table name |
| <i>partition_acc_cnt</i> | Number of partitions accessed |
| <i>acc_num</i> | Number of records accessed |
| <i>self_id</i> | Node ID |

12.3.26.3 Example

```
iSQL> CREATE TABLE t1 ( i1 INTEGER )  
PARTITION BY RANGE ( i1 )  
(  
    PARTITION p1 VALUES LESS THAN ( 100 ),  
    PARTITION p2 VALUES LESS THAN ( 200 ),
```



```

    PARTITION p3 VALUES DEFAULT
) TABLESPACE SYS_TBS_DISK_DATA;
Create success.

iSQL> INSERT INTO t1 VALUES ( 50 );
1 row inserted.
iSQL> INSERT INTO t1 VALUES ( 60 );
1 row inserted.
iSQL> INSERT INTO t1 VALUES ( 150 );
1 row inserted.
iSQL> INSERT INTO t1 VALUES ( 160 );
1 row inserted.
iSQL> alter session set explain plan = on;
Alter success.

iSQL> SELECT COUNT(*) FROM t1 WHERE i1 < 100;
COUNT
-----
2
1 row selected.
-----
PROJECT ( COLUMN_COUNT: 1, TUPLE_SIZE: 8 )
  GROUP-AGGREGATION ( ITEM_SIZE: 24, GROUP_COUNT: 1, BUCKET_COUNT: 1, ACCESS:
1, SELF_ID: 4, REF_ID: 2 )
    PARTITION-COORDINATOR ( TABLE: T1, PARTITION: 1/3, ACCESS: 2, SELF_ID: 2 )
      SCAN ( PARTITION: P1, FULL SCAN, ACCESS: 2, DISK_PAGE_COUNT: 64, SELF_ID:
3 )
-----

```

12.3 Plan Nodes

13 SQL Plan Cache

This chapter provides a conceptual overview of the SQL Plan Cache in ALTIBASE HDB, and describes its features.

This chapter contains the following sections:

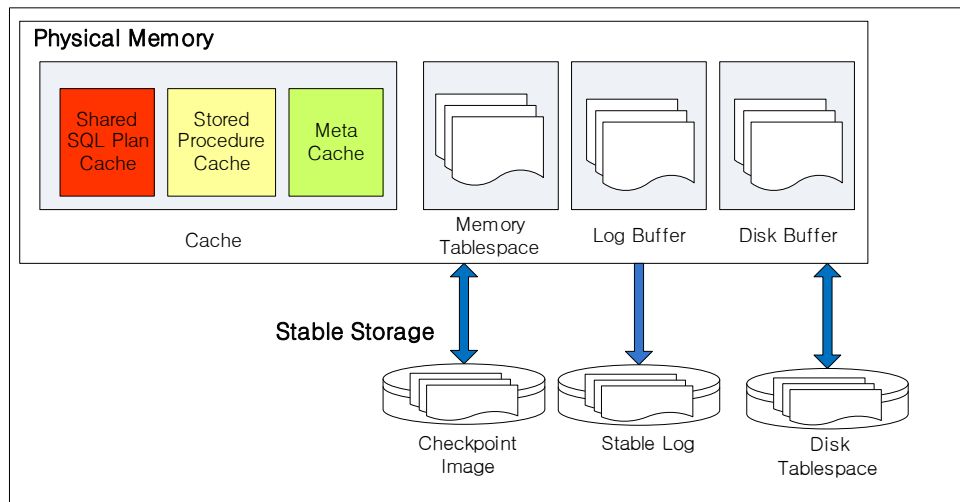
- [Overview](#)
- [Features](#)
- [Related Properties](#)

13.1 Overview

The purpose of the SQL Plan Cache is to share SQL execution plans, which are required when SQL statements are executed. Whenever a shared SQL plan is executed, information about the conditions under which it was executed can be reused.

13.1.1 Structure of the SQL Plan Cache

Figure 13-1 Structure of the SQL Plan Cache



ALTIBASE HDB has cache areas that are shared by all sessions. These cache areas comprise the SQL Plan Cache, the Stored Procedure Cache and the Meta Cache.

Whenever new SQL execution plans are created, they are stored in the SQL Plan Cache so that they can be shared by all sessions.

Whenever new stored procedure execution plans are created, they are stored in the Stored Procedure Cache so that they can be shared.

Meta data, that is, information about database objects, is stored in the Meta Cache so that it can be accessed quickly.

13.1.2 Advantages

The use of the ALTIBASE HDB SQL Plan Cache confers the following advantages:

- Improved performance of direct execution (Direct execution is the most basic way to execute a statement. An application builds a character string containing a SQL statement and submits it for execution using the `SQLExecDirect` function.)

In environments in which queries are directly executed, because plans that have already been prepared can be shared and reused, the cost of preparing execution plans can be reduced, thereby realizing improved performance.

- A drastic reduction in the use of so-called "prepare memory" in prepared execution environments (Prepared execution is a way to reduce the parsing and compiling overhead associated with repeatedly executing a SQL statement.)

Additionally, even in environments which exclusively follow the prepare/execute model, the SQL Plan Cache confers not only the benefit of reducing prepare costs, but also the benefit of reducing the total amount of prepare memory that is required.

13.1.3 Statements that use the SQL Plan Cache

For a detailed explanation of each statement, please refer to the *SQL Reference*.

- SELECT (SELECT FOR UPDATE)
- INSERT (INSERT SELECT)
- UPDATE
- DELETE
- MOVE
- ENQUEUE
- DEQUEUE

13.2 Features

The features of the ALTIBASE HDB SQL Plan Cache are as follows:

- It uses a plan-sharing-oriented check-in method.
- Its replacement policy considers not only which plans were most recently referred to, but also the frequency with which plans are referred to.
- It can only be used for identical SQL statements, in order to eliminate parsing costs.

The so-called "check-in method", which is oriented towards sharing plans, works by registering execution plans in the SQL Plan Cache. In greater detail, new shared execution plans can continue to be registered in the SQL Plan Cache as long as the capacity specified using the `SQL_PLAN_CACHE_SIZE` property is not exceeded, after which they can be used. If the capacity specified in `SQL_PLAN_CACHE_SIZE` is exceeded when a new plan is registered, one or more old and unused plans are found and deleted, after which new plans can be registered.

However, if an attempt to register a new execution plan causes the size specified in `SQL_PLAN_CACHE_SIZE` to be exceeded, and all currently registered execution plans are still frequently referred to, it will be impossible to register new execution plans. Whenever an attempt to register an execution plan in the SQL Plan Cache fails, the value of `CACHE_IN_FAIL_COUNT` in the `V$SQL_PLAN_CACHE` performance view is incremented.

The default value of the `SQL_PLAN_CACHE_SIZE` property is 64MB, but the user can set it arbitrarily.

Additionally, because the SQL Plan Cache's plan replacement policy takes into account not only the most recently referenced execution plans but also the most frequently referenced ones, the most popular Plan Cache objects are protected when there is no more free space in the cache.

However, please bear in mind that in order for it to be possible to use the shared plan cache, not only the syntax of SQL statements but also the values of parameters must be identical. The reason that the SQL Plan Cache was implemented in this way was to completely eliminate parsing costs. For example, the following two statements will be treated as different in the SQL Plan Cache:

```
INSERT INTO T1 VALUES (1,2);
INSERT INTO T1 VALUES (2,3);
```

In such cases, better performance can be realized by writing queries that make it possible to use the Plan Cache, as shown below:

```
INSERT INTO T1 VALUES (?, ?)
```

13.3 Related Properties

In order to maximize the efficiency of use of the SQL Plan Cache, it will be necessary to tweak the values of some of the properties in the `altibase.properties` file. The properties that are related to the use of the SQL Plan Cache are as follows. For detailed information about each property, please refer to the *General Reference*.

- `SQL_PLAN_CACHE_BUCKET_CNT`
- `SQL_PLAN_CACHE_HOT_REGION_LRU_RATIO`
- `SQL_PLAN_CACHE_PREPARED_EXECUTION_CONTEXT_CNT`
- `SQL_PLAN_CACHE_SIZE`

13.3 Related Properties

14 Communication Layer

This chapter describes the methods and protocols involved in establishing a connection between a client application and an ALTIBASE HDB database server.

14.1 Communication Protocols

A network protocol is a set of rules that governs the communication between computers on a network. This section describes the communication protocols that can be used by an ALTIBASE HDB database server and client applications.

- [TCP/IP](#)
- [Unix Domain Socket](#)
- [Shared Memory](#)

14.1.1 TCP/IP

Transmission Control Protocol/Internet Protocol (TCP/IP) is an industry standard suite of networking protocols, and was used to construct the global Internet. TCP is a protocol for exchanging data reliably and directly between two network hosts. IP is a protocol used for communicating data across packet-switched networks.

ALTIBASE HDB supports both Internet Protocol Version 4 (IPv4) and Internet Protocol Version 6 (IPv6).

IPv6 addresses the problem currently afflicting IPv4, which is the exhaustion of addresses for connecting computers or hosts on the Internet. IPv6 has a very large address space, because an IPv6 address consists of 128 bits, as compared to 32 bits in IPv4.

For more information about IPv6, please refer to the Internet Protocol Version 6 (IPv6) Specification, RFC 2460 (<http://tools.ietf.org/html/rfc2460>).

14.1.1.1 IPv6 Address Notation

IPv6 addresses are denoted by eight groups of hexadecimal quartets, separated by colons in between them.

The following is an example of a valid IPv6 address:

```
2001:cdba:0000:0000:0000:0000:3257:9652
```

Any four-digit group of zeroes within an IPv6 address may be reduced to a single zero, or may even be omitted altogether. Therefore, the following IPv6 addresses are equivalent notations:

```
2001:cdba:0000:0000:0000:0000:3257:9652
2001:cdba:0:0:0:0:3257:9652
2001:cdba::3257:9652
```

The URL for the above address will be of the form:

```
http://[2001:cdba:0000:0000:0000:0000:3257:9652]/
```

ALTIBASE HDB supports the standard IPv6 address notation specified by RFC2732. When connecting to a database server, the IPv6 address must be enclosed between a left square bracket([) and a right square bracket(]).

The following are examples of valid IPv6 addresses in ALTIBASE HDB:

```
[::1]
[2002:c0a8:101:1:216:e6ff:fed2:7aea]

$ isql -s [2002:c0a8:101:1:216:e6ff:fed2:7aea] -u sys
```

In the case of a link-local address that begins with FE80, a zone index is appended to the address, separated by a percent sign (%). The zone index is the index for the interface to which the link-local address is assigned.

In Linux systems, a link-local address needs to be qualified with a zone index in order to connect to an ALTIBASE HDB server. (The exception is JDBC applications, for which this is not required.) An example of such a zone index is shown below:

```
[fe80::221:86ff:fe94:f51f%eth0]
$ isql -s [fe80::221:86ff:fe94:f51f%eth0] -u sys
```

14.1.1.2 IP Stack

A host may have one of a variety of different protocol stacks¹ installed. There are three types of IP hosts, which differ based on their ability to support the two protocols.

| | |
|----------------|---|
| IP4-only host | A host having only an IPv4 stack installed. An IPv4-only host cannot understand IPv6 addresses. |
| IPv6/IPv4 host | A host having a dual stack installed, thus supporting both IPv4 and IPv6. |
| IPv6-only host | A host having only an IPv6 stack installed. An IPv6-only host does not support IPv4. |

14.1.1.3 IPv6 Client/Server Connectivity

"Network connectivity" refers to establishing a connection and communication between two or more computers over a network.

The following table shows the protocol versions that can be used for communication between a server and client for different combinations of hosts having different protocol stacks. Supported (v6) means that the client/server hosts have protocol stacks that support IPv6, and that they can connect to other hosts using the IPv6 interface.

| | IPv4-only Server | Dual-Stack Server | IPv6-only Server |
|-------------------|------------------|--------------------|------------------|
| IPv4-only Client | Supported (v4) | Supported (v4) | Not supported |
| Dual-Stack Client | Supported (v4) | Supported (v4, v6) | Supported (v6) |
| IPv6-only Client | Not supported | Supported (v6) | Supported (v6) |

Note:

- *ALTIBASE HDB does not support the dual-stack server on Windows NT version 5.2 and earlier versions of Windows that do not provide the dual-stack protocol.*

1. The term "protocol stack" refers to the software implementation of a networking protocol suite.

14.1 Communication Protocols

- *ALTIBASE HDB does not support the IPv6-only server on SPARC SunOS.*

14.1.1.4 IPv6 Support in ALTIBASE HDB

Support for IPv6 by the components of ALTIBASE HDB release 5.5.1 was outlined above in the table in the previous section, [IPv6 Client/Server Connectivity](#).

- Server

To support IPv6, the `NET_CONN_IP_STACK` property in the `altibase.properties` file must be set to 1 or 2.

For detailed information about that property, please refer to the *General Reference*.

- Client

To connect using IPv6, the `DSN` attribute must be set to an IPv6 address, or the `DSN` attribute must be set to a host name and the `PREFER_IPV6` attribute must be set to `TRUE`.

For a given host name, ALTIBASE HDB clients attempt to connect to all IP addresses returned by a call to `getaddrinfo()` until a successful connection is established, or until all addresses have been attempted. If more than one IP address is returned, ALTIBASE HDB clients attempt to establish a connection to each of those IP addresses in an order determined in consideration of the `PREFER_IPV6` attribute. If the `PREFER_IPV6` attribute is not set, or if it is set to `FALSE`, an attempt is first made to connect to any IPv4 addresses that were returned. If this attempt fails, the client then attempts to connect to any IPv6 addresses that were returned. If the `PREFER_IPV6` attribute is set to `TRUE`, an attempt is first made to connect to any IPv6 addresses that were returned. If this attempt fails, the client then attempts to connect to any IPv4 addresses that were returned.

For more information about the `PREFER_IPV6` attribute, please refer to the *ODBC Reference*, the *API User's Manual*, and the manuals for the respective utilities.

14.1.2 Unix Domain Socket

On UNIX platforms, when both the client and database server are installed on a single machine, Unix domain sockets can be used for communication. Using UNIX domain sockets realizes better performance than when using TCP/IP. To use UNIX domain sockets, the `CONNTYPE` for ODBC/CLI and the `ISQL_CONNECTION` environment for the ALTIBASE HDB utilities must be set.

For more information, please refer to the *ODBC Reference* and to the manuals for the respective utilities.

14.1.3 Shared Memory

Using shared memory is a method of realizing inter-process communication (IPC), i.e. a way of exchanging data between simultaneously running processes. When both the client and database server are installed on a single machine, the client application may enjoy improved performance if this method is implemented. Using shared memory provides the best performance, but a lot of additional memory is required. In order to use shared memory for communication, it is first necessary to:

- Configure the server property in the `altibase.properties` file. Please refer to the *Admin-*

istrator's Manual.

- Set the `CONNTYPE` for ODBC/CLI and the `ISQL_CONNECTION` environment for utilities such as iSQL and iLoader. Please refer to the *ODBC Reference* and to the manuals for the respective utilities.

In Windows, TCP/IP is used to connect to a database server via IPC. When connecting in this way, clients attempt to connect according to the following procedure:

- If the `PREFER_IPV6` attribute is not set, or if the `PREFER_IPV6` attribute is set to `FALSE`, the client first tries to connect using the IPv4 address for the local host (127.0.0.1). If this attempt fails, the client subsequently attempts to connect using the corresponding IPv6 address ([::1]).
- If the `PREFER_IPV6` attribute is set to `TRUE`, the client first tries to connect using the IPv6 address. If this attempt fails, the client then attempts to establish a connection using the IPv4 address.

15 Securing Data

This chapter covers how to use security modules to develop a database encryption strategy.

This chapter contains the following sections:

- [Overview](#)
- [How Security is Organized in ALTIBASE HDB](#)
- [Integrating a Security Module](#)
- [Starting Security Modules and Encrypting Data](#)

15.1 Overview

As the protection of information becomes an increasingly important issue, and with the increased awareness of the highly sensitive nature of personal and business information, information protection legislation is being enacted, which is creating the need for more sophisticated database security management functionality.

With the goal of protecting your database from both internal and external threats, ALTIBASE HDB provides security module integration functionality, whereby security modules can be integrated as required so that your database can be effectively protected.

This chapter will focus on how to implement security modules in an integrated manner with the aim of protecting your data.

The ALTIBASE HDB security module integration functionality provides powerful encryption management that integrates ALTIBASE HDB server and independent security modules without modifying client applications to provide a complete system for protecting personal information. ALTIBASE HDB supports the integration of trusted third-party security modules with the ALTIBASE HDB server to improve on vulnerabilities in database security, and provides an interface that can be used to effectively integrate security modules.

ALTIBASE HDB provides an architectural framework that allows security modules to be used to encrypt data, tightly control database access and perform auditing at the database level. All operations pertaining to security are performed by the security module that is integrated with ALTIBASE HDB server, rather than by the ALTIBASE HDB server itself.

Encryption applies to columns in tables. The data in encrypted columns are secured regardless of whether the data reside on disk or in a memory buffer.

Access control tasks are roughly categorized into two areas: determining what is to be protected and determining which users have the right to access protected objects.

Like encryption, access control also applies to columns in tables. Every user wishing to access a protected column must first have been granted access rights for the corresponding object.

Which items are protected, which users have the right to access protected data, and all encryption tasks are logged for auditing purposes.

The security-related features provided with ALTIBASE HDB are as follows:

- Encrypted data can be stored and managed either on disk or in memory
- Data to be output are decrypted according to security privileges
- Indexes are built in such a way that the chronological order of the original data is maintained
- Tables that contain encrypted columns can be replicated

15.2 How Security is Organized in ALTIBASE HDB

ALTIBASE HDB and a security module are independent of each other. Encryption keys, security policy and information about security privileges are managed in a security module that is not part of the database itself.

ALTIBASE HDB can operate normally even when it is not integrated with a security module. However, queries executed on encrypted columns will fail if no security modules are present.

ALTIBASE HDB is integrated with a security module both by setting security module-related properties appropriately and by executing SQL statements. ALTIBASE HDB evaluates the validity of the connection between the security module and ALTIBASE HDB server and guarantees that the connection is valid.

When ALTIBASE HDB evaluates the connection with the security module, it compares security-related information, such as the module name, version and information about encrypted columns, in the database with the corresponding information in the security module.

Columns can be encrypted without requiring that any changes be made to existing applications that connect to ALTIBASE HDB. Encrypted columns can be created and deleted using SQL. Other tasks can be accomplished without changing any of the queries that are used in the existing application.

In ALTIBASE HDB, the security-related functions of the main module are as follows:

- Integrating ALTIBASE HDB with security modules in accordance with environment variables and SQL statements while the database is running
- Supporting data structures and meta information for managing encrypted data
- Supporting extended SQL for security
- Supporting replication

The roles played by an external security module are as follows:

- Managing encryption algorithm settings, including the kind of algorithm that is used and the initialization vector, if applicable
- Managing settings for encrypted columns, including the encryption algorithm and encryption/decryption permissions
- Encrypting and decrypting data
- Managing access control settings, including access to specific IPs and access by specific users
- Auditing, including encryption/decryption logging and access control logging

15.3 Integrating a Security Module

The tasks that must be performed in order to integrate a security module will now be explained.

A single server can be integrated with only one security module. In order to integrate the security module, the security policy of the ECC algorithm (the ALTIBASE HDB ECC algorithm is a form of Order Preserving Encryption), which guarantees that the order of the encrypted data is the same as the order of the original data, the name of the module and the location of the module in the file system are set. The security module can then be integrated.

In ALTIBASE HDB, ECC is an acronym for Encrypted Comparison Code, and is a hash value that guarantees that the order of the encrypted data is the same as the order of the original data. An ECC is generated using a hashing algorithm that allows only one-way transformation to ensure that encrypted data cannot be converted back to their original form. Using an ECC allows comparison operations to be rapidly executed on encrypted columns within ALTIBASE HDB, without exposing the actual data to database administrators or users.

The term “ECC algorithm” denotes a hashing algorithm used to generate an ECC. External security modules support various ECC algorithms. However, only one ECC algorithm can be used at one time on one server.

The following steps are taken to integrate a security module with ALTIBASE HDB:

- Install the external security module
- Configure the ALTIBASE HDB environment
- Start the security module
- Create encrypted columns and convert existing columns into encrypted columns

Because the step of installing the external security module varies depending on the type of security module that is used, please refer to the documentation for the external security module to be installed. The remaining steps, that is, the steps of configuring the ALTIBASE HDB environment, starting the security module, and Creating encrypted columns or converting existing columns into encrypted columns, as well as stopping the security module and decrypting columns, will be explained in this document.

15.3.1 Configuring ALTIBASE HDB for Security

Set the path of the security module in the ALTIBASE HDB properties file, which is located at \$ALTIBASE_HOME/conf/altibase.properties, as follows:

```
SECURITY_MODULE_NAME = DAmo
SECURITY_MODULE_LIBRARY = libdamo.so
SECURITY_ECC_POLICY_NAME = OE_POL_A
```

Please bear in mind that the property values are case-sensitive. Set the value of the SECURITY_MODULE_NAME property appropriately to identify the security module being used.

Set the value of the SECURITY_MODULE_LIBRARY property to the name of the installed security module library file. The SECURITY_ECC_POLICY_NAME property must be set in order for ALTIBASE HDB to distinguish the security policy from other items internally.

The values of these properties can be set or changed using the ALTER SYSTEM statement while the

system is running. When the values of these properties are changed using the ALTER SYSTEM statement, the value set for the SECURITY_MODULE_LIBRARY must include the absolute path to the library file.

```
ALTER SYSTEM SET SECURITY_MODULE_NAME = 'DAmo';  
ALTER SYSTEM SET SECURITY_MODULE_LIBRARY =  
  '/altibase_home/damo/libdamo.so';  
ALTER SYSTEM SET SECURITY_ECC_POLICY_NAME = 'OE_POL_A';
```

15.4 Starting Security Modules and Encrypting Data

This section explains how to start security modules and encrypt data, and introduces the related statements.

15.4.1 Starting up the Security Module

Once all of the properties related to the security module have been set, it can be started up. The following is a brief description of the internal processes that occur when the security module is started:

1. Security module authentication

If mutual authentication fails, the security module cannot be used.

2. Initialization and validation of the security module

The security module's internal settings, license and the like are checked.

3. Examining ECC security policy

The ECC security policy, which is set using properties in the security module, is checked to determine whether or not it is valid.

The security module can be started up using the ALTER SYSTEM statement with the START SECURITY option. It will be necessary to access the system as an administrator with suitable privileges.

15.4.1.1 Example

1. Access an idle instance of ALTIBASE HDB as an administrator with suitable privileges.

```
iSQL> CONNECT sys/manager
```

2. Set security module-related properties as appropriate.

```
iSQL> ALTER SYSTEM SET SECURITY_MODULE_NAME = 'altibase';  
iSQL> ALTER SYSTEM SET SECURITY_MODULE_LIBRARY =  
'/altibase_home/lib/libsecurity.so';  
iSQL> ALTER SYSTEM SET SECURITY_ECC_POLICY_NAME = 'ecc_policy1';
```

3. Start the security module.

```
iSQL> ALTER SYSTEM START SECURITY;
```

4. Check that the security module is running correctly.

```
iSQL> SELECT * FROM SYSTEM_.SYS_SECURITY_  
MODULE_NAME MODULE_VERSION ECC_POLICY_NAME ECC_POLICY_CODE  
-----  
altibase 1.0 ecc_policy1 abcde12345
```

15.4.2 Stopping the Security Module

Just as when starting up the security module, it is necessary to access ALTIBASE HDB as an administrator with suitable privileges in order to stop the security module. Then, execute the following

statement:

```
iSQL> ALTER SYSTEM STOP SECURITY;
```

The inactive status of the security module can be verified using the following statement:

```
iSQL> SELECT * FROM SYSTEM_.SYS_SECURITY_;
MODULE_NAME MODULE_VERSION ECC_POLICY_NAME ECC_POLICY_CODE
-----
No rows selected.
```

Note: The security module can be shut down only if there are no encrypted columns.

15.4.3 Column Encryption

When it is necessary to secure confidential data in particular columns, those columns can be encrypted. Column encryption is supported for the CHAR and VARCHAR data types.

Columns can be encrypted at the time that they are created using the CREATE TABLE statement, or alternatively, columns in existing tables can be encrypted using the ALTER TABLE statement. In both cases, use the ENCRYPT USING clause to specify the security policy to use to encrypt the column data. Use the DESC statement to confirm that a column has been encrypted.

15.4.3.1 Command Syntax

```
CREATE TABLE table_name (column_name datatype [ENCRYPT USING
'policy_name'] );
```

15.4.3.2 Limitations on Encrypting Columns

- The data type of an encrypted column cannot be changed.

15.4.3.3 Example

- Query1> Specify the empID1 and ssn1 columns as encrypted columns when creating a table.

```
CREATE TABLE t1 (name1 varchar(5),
empID1 varchar(10) ENCRYPT USING 'policy_id',
ssn1 char(12) ENCRYPT USING 'policy_ssn');
```

- Query2> Check whether any encrypted columns exist in a table.

```
iSQL> DESC t1
-----
NAME TYPE IS NULL
-----
NAME1 VARCHAR(10) FIXED
EMPID1 VARCHAR(8) ENCRYPT FIXED
SSN CHAR(12) ENCRYPT FIXED
```

15.4.4 Encrypting Existing Columns

Regular columns can be converted into encrypted columns using the ALTER TABLE statement.

15.4 Starting Security Modules and Encrypting Data

15.4.4.1 Command Syntax

```
ALTER TABLE table_name MODIFY (column_name [ENCRYPT USING  
'policy_name']);
```

15.4.4.2 Restriction

1. Encrypted columns cannot be encrypted again.
2. The data type of an encrypted column cannot be changed.

15.4.4.3 Example

- Query> Convert the empID1 column in t1, an existing table, to an encrypted column using the policy_ssn policy.

```
ALTER TABLE t1 MODIFY (empID1 ENCRYPT USING 'policy_ssn');
```

15.4.5 Canceling Encryption

Encrypted columns can be converted back into regular columns using the ALTER TABLE statement with the MODIFY option.

15.4.5.1 Command Syntax

```
ALTER TABLE table_name MODIFY (column_name [DECRYPT]);
```

15.4.5.2 Example

Query> Convert the empID1 column in table t1 back into a regular column.

```
ALTER TABLE t1 MODIFY (empID1 DECRYPT);
```

16 Tuning ALTIBASE HDB

This chapter describes the following two features:

- [Log File Groups](#)
- [Group Commit](#)

16.1 Log File Groups

This section describes what Log File Groups (LFGs) are and how to use them.

16.1.1 The Log File Group Concept

The use of log file groups allows logs for multiple transactions to be simultaneously written to multiple log files, thereby improving system performance.

By default, ALTIBASE HDB maintains log files in a single directory, the location of which is specified using the LOG_DIR property. Furthermore, only one of those log files is written to at one time while the database server is active. Although logging is essential in order to ensure that transactions meet the Durability criterion (one of the four ACID properties of reliable transactions), using only a single file to log multiple transactions is sure to create a bottleneck.

In order to eliminate this bottleneck, ALTIBASE HDB supports the use of Log File Groups, which allows you to set multiple log file paths. Because ALTIBASE HDB has a limitation in that only one log file in a directory can be active, the simultaneous use of multiple log files is achieved by locating the active log files in different directories from one another.

A single directory containing multiple log files, one of which is active at any point in time, is the basic building block from which a logging system is built, and is known as a Log File Group (LFG). Hereinafter, the term “Log File Group” will be abbreviated to “LFG” for the sake of brevity.

The more frequently the contents of your database are added to and updated, and the more frequently transactions are committed, the more remarkable the performance improvement you will observe by increasing the number of LFGs.

16.1.2 The Constituent Elements of LFGs

A single LFG comprises the following constituent parts:

- Log File Path (LOG_DIR Property)

The LOG_DIR property specifies the path and directory in which log files are stored. This directory contains one or more log files. ALTIBASE HDB can only write to one of those log files.

- Archive Log File Path (ARCHIVE_DIR Property)

If the database is running in archivelog mode, when the log files located in the directory specified using the LOG_DIR property are finished being written to, they are copied to the directory specified using the ARCHIVE_DIR property. These archive log files are used for database backup and recovery.

The number of values specified for this property must be the same as the number specified using the LOG_DIR property. Moreover, when multiple values are specified for the LOG_DIR property, each of the directories specified using the ARCHIVE_DIR property is mapped to a corresponding directory specified using the LOG_DIR property. The directories are mapped in the order in which they are specified.

For each LFG, the three following threads are maintained:

- Log File Creation Thread

When the active log file in a single LFG becomes full, ALTIBASE HDB starts recording the logs in a new log file. If, at this point in time, ALTIBASE HDB had to create a new log file every time it became necessary to switch to a new log file, the transaction needing to write to the log file would have to wait for the new log file to be created, during which time it would not be able to perform any other tasks, thereby causing a problem.

This thread creates new log files in advance, i.e. before the time point at which they are required in order to continue logging.

For reference, log files that are no longer being used while checkpointing is underway are identified and deleted.

- Log Flush Thread

The log flush thread periodically writes recently recorded logs to disk. When a transaction is committed, ALTIBASE HDB checks whether this thread has already written the related log entries to disk, and only writes those that have not already been written.

That is, by periodically writing log entries to disk, this thread reduces the number of log entries that need to be written to disk at the time that a transaction is committed.

- Archive Log Thread

When one of the log files in an LFG fills up, logging continues in a new log file. At this time, the archive log thread copies the previous log file, to which logs were being written until just recently, but which has been filled up and is thus no longer being used, to an archive log file.

16.1.3 Example

There are 3 properties which are related to LFGs:

```
LOG_FILE_GROUP_COUNT
LOG_DIR
ARCHIVE_DIR
```

By default, the LOG_FILE_GROUP_COUNT property is set to 1, which means that ALTIBASE HDB is configured to use only one LFG. Additionally, separate paths are defined for the log directory and the archive directory using the LOG_DIR and ARCHIVE_DIR properties.

The following shows how the relevant portion of the altibase.properties file would appear in the case where only one LFG is in use:

```
LOG_FILE_GROUP_COUNT = 1 # LFG count
LOG_DIR = ?/logs # log file path
ARCHIVE_DIR = ?/arch_logs # archive log file path
```

The question mark ("?") indicates the \$ALTIBASE_HOME directory. Thus, the complete LOG_DIR path would be \$ALTIBASE_HOME/logs. After a database is created, examining the contents of the log file directory will reveal three log files, numbered from logfile0 to logfile2 as shown below. One of these three files is written to when the contents of the database are changed.

```
-rw--- 1 altibase dba 10485760 Jun 22 15:46 logfile0
-rw--- 1 altibase dba 10485760 Jun 22 15:46 logfile1
-rw--- 1 altibase dba 10485760 Jun 22 15:46 logfile2
```

The following shows how the relevant portion of the altibase.properties file would appear in the

16.1 Log File Groups

case where 3 LFGs are used. The LOG_FILE_GROUP_COUNT property has been set to 3 to reflect the number of LFGs. Correspondingly, three values have been set for each of the LOG_DIR and ARCHIVE_DIR properties: one log file path and one archive file path for each LFG.

```
LOG_FILE_GROUP_COUNT = 3 # LFG file count

LOG_DIR = ?/logs1 # log file path for LFG#0
LOG_DIR = ?/logs2 # log file path LFG#1
LOG_DIR = ?/logs3 # log file path LFG#2

ARCHIVE_DIR = ?/arch_logs1 # archive log file path LFG#0
ARCHIVE_DIR = ?/arch_logs2 # archive log file path LFG#1
ARCHIVE_DIR = ?/arch_logs3 # archive log file path LFG#2
```

The user must manually create the additional log and archive log directories. Here, the new directories are created in \$ALTIBASE_HOME, which is the ALTIBASE HDB home directory, just like the default directories in the case where there is only one LFG.

In the first log file group, LFG#0, logs are written to one of the log files in the first directory, which is \$ALTIBASE_HOME/logs1. The other log file groups write to the corresponding log files in the other directories.

In order to ensure transaction durability, when more than one LFG is maintained in this way, the transaction always waits until logs have been synchronized to disk every time a transaction is committed. That is, the server always operates as though COMMIT_WRITE_WAIT_MODE were set to 1.

16.1.4 Optimizing Log File Groups

When settings have been made so as to use two or more LFGs, it is advisable to set the log paths of each of the LFGs to different physical devices, as shown below. This prevents the bottleneck in which many logs are written to a single disk at the same time.

Configuring multiple LFGs such that they are located on different disks allows multiple disk I/O tasks to be simultaneously performed, thereby improving system performance still further.

```
LOG_FILE_GROUP_COUNT = 3 # LFG count is set to 3

LOG_DIR = /disk1/logs # log file path for LFG #0
LOG_DIR = /disk2/logs # log file path for LFG #1
LOG_DIR = /disk3/logs # log file path for LFG #2

ARCHIVE_DIR = /disk1/arch # archive log file path for LFG #0
ARCHIVE_DIR = /disk2/arch # archive log file path for LFG #1
ARCHIVE_DIR = /disk3/arch # archive log file path for LFG #2
```

In contrast, as shown below, configuring two or more LFGs on a single disk can allow disk I/O throughput to be more extensively used, thereby realizing better performance.

It is usually advantageous from the aspect of performance to locate two or more LFGs on the same disk in this way. However, system performance, disk I/O throughput, and system load strongly impact the determination of whether to configure a single disk to store only one LFG or to store two or more LFGs. Therefore, it is best to measure actual performance under real-life conditions when determining where to locate multiple LFGs.

```
LOG_FILE_GROUP_COUNT = 6 # LFG count is set to 6

LOG_DIR = /disk1/logs1 # log file path for LFG #0
LOG_DIR = /disk1/logs2 # log file path for LFG #1
```

```

LOG_DIR = /disk2/logs1 # log file path for LFG #2
LOG_DIR = /disk2/logs2 # log file path for LFG #3
LOG_DIR = /disk3/logs1 # log file path for LFG #4
LOG_DIR = /disk3/logs2 # log file path for LFG #5

ARCHIVE_DIR = /disk1/arch1 # archive log file path for LFG #0
ARCHIVE_DIR = /disk1/arch2 # archive log file path for LFG #1
ARCHIVE_DIR = /disk2/arch1 # archive log file path for LFG #2
ARCHIVE_DIR = /disk2/arch2 # archive log file path for LFG #3
ARCHIVE_DIR = /disk3/arch1 # archive log file path for LFG #4
ARCHIVE_DIR = /disk3/arch2 # archive log file path for LFG #5

```

16.1.5 Assigning Log File Group

If a database has only one LFG, all log entries are written into log files which are associated with that LFG. However, if a database has multiple LFGs, transaction logs must be distributed evenly among all LFGs. All of the logs for a particular transaction are written to the same LFG. This process is known as “Assigning LFGs to Transactions”.

Consider for example how LFGs are assigned to transactions in the case where there are four LFGs, namely LFG#0, LFG#1, LFG#2, and LFG#3.

- Transactions that change disk tables are assigned to the first LFG, which is LFG #0.
- Transactions that only affect memory tables are assigned cyclically to LFG#1, LFG#2, or LFG#3.
- If a memory transaction is changed such that it affects a disk table, thus becoming a disk transaction, ALTIBASE HDB dissociates the transaction from the LFG that is currently in use, and assigns it instead to LFG #0.
- Other kinds of transactions, such as those executed by the garbage collector, which looks for deleted records and recovers space, or checkpointing-related transactions, are also assigned to LFG #0.

16.1.6 Limits & Constraints

16.1.6.1 Limitations to LFG Functionality

Because all transactions that change disk tables, even if only one time, are assigned to the first LFG, the performance of disk transactions is not improved by increasing the number of LFGs. However, when the system is configured with two LFGs, the logs for disk transactions and memory transactions are separated and allocated to different LFGs, so the performance of disk transactions will be improved compared to when only one LFG is used.

16.1.6.2 Constraints Pertaining to the Use of LFGs

- The system operates as though COMMIT_WRITE_WAIT_MODE were set to 1. Even if this property is expressly set to 0, the server operates as though it were set to 1, and transactions wait until commit logs have been written to disk when committing transactions.
- Once a database has been created, the number of LFGs cannot be changed from its initial value. However, the LFG log file paths and archive log file paths can be changed after a database has been created.

16.1 Log File Groups

- Multiple LFGs cannot use the same log directory. In other words, the log file path of each LFG must be different from all others.
- Multiple LFGs cannot use the same archive log directory; that is, the archive log file path of each LFG must be different from all others.

16.2 Group Commit

In this section, the ALTIBASE HDB Group Commit feature, which is provided to improve transaction processing performance, will be explained.

16.2.1 The Group Commit Concept

Using Group Commit helps to reduce disk I/O overhead by executing multiple commit requests at the same time.

After a single transaction has been committed, the changes made by the transaction must not be lost, regardless of the circumstances. In order to ensure this, it must be verified that all of the logs generated by the transaction have been written to disk, and then the client application must be informed of this.

However, because the disk I/O performed during this process takes a long time compared to writing to memory, when disk I/O requests for multiple transactions are processed separately at the same time, system performance suffers.

The time taken to perform disk I/O has a greater effect on system performance than the actual volume of I/O, due to the number of times the disk is accessed. Therefore, grouping I/O tasks that would otherwise be performed separately and executing them together can have a positive impact on system performance.

Group Commit involves grouping disk I/O tasks related to logging multiple transactions that have yet to be committed and performing the I/O all at once in order to improve system performance.

16.2.2 How Group Commit Works

In order to perform all of the disk I/O required to commit multiple transactions at one time, ALTIBASE HDB memorizes the time at which the most recent disk I/O occurred, and, for a fixed period of time after that, does not permit disk I/O, but makes transactions wait until the next occurrence of disk I/O.

If the specified disk I/O wait time is too short, disk I/O occurs very frequently, with the result that performance is not very different from the case where Group Commit is not used. Conversely, if the specified wait time is too long, transactions wait needlessly long periods of time for disk I/O, with a consequent drop in performance.

For more detailed information about tuning the Group Commit disk I/O waiting time, please refer to the [16.2.7 Tuning Group Commit-Related Properties](#) section in this chapter.

When attempting to perform Group Commit disk I/O to write logs to disk in the process of committing a transaction, the following steps are performed, and the appropriate action is taken.

1. If the log entries to be written to disk have already been flushed to disk by other transactions, because the log entries have already been written to disk, and thus do not need to be written to disk again, no disk I/O is performed for the transaction.

Otherwise, number 2 is performed.

2. If the number of transactions in this LFG that change the contents of the database is less than the number specified using the LFG_GROUP_COMMIT_UPDATE_TX_COUNT property, disk I/O

16.2 Group Commit

is not performed at this time, but is postponed.

Otherwise, number 3 is performed.

3. If the amount of time that has elapsed since the time point at which the most recent disk I/O occurred has not surpassed the time interval set using the `LFG_GROUP_COMMIT_INTERVAL_USEC` property, disk I/O is not performed at that time, but is postponed.

Otherwise, number 4 is performed.

4. The time at which disk I/O occurs is noted, disk I/O is performed, and all pending logs are written to disk.

16.2.3 Unit of Operation for Group Commit

As was seen above for the Log File Group feature, the task of writing transaction logs to disk is executed separately for each log file group. Similarly, Group Commit is executed separately for each log file group, and different LFGs do not affect each other.

16.2.4 COMMIT_WRITE_WAIT_MODE and Group Commit

If `COMMIT_WRITE_WAIT_MODE` is set to 0, because the server does not wait until commit logs have been written to disk when a transaction is committed, durability is not guaranteed in the event of a power outage.

Group Commit is an optimization technique that groups disk I/O tasks for processing without compromising the durability of transactions. Therefore, it is meaningful only when `COMMIT_WRITE_WAIT_MODE` is set to 1. This is because, when a transaction is committed, the transaction waits until commit logs have been written to disk only when this property is set to 1.

16.2.5 Considerations

Group Commit only has a noticeable effect when it is used in an environment in which multiple transactions are committed at the same time. For example, if only one transaction needs to be committed in a system, that is, no other transactions are waiting to be committed, there are no transactions to be grouped for processing via a single disk I/O task. In such cases, committing and logging each transaction directly would result in better performance than using Group Commit.

As mentioned above, Group Commit is executed independently for each LFG. The decision of whether to use Group Commit is also made independently for each LFG on the basis of the number of uncommitted transactions in that LFG. For each LFG, ALTIBASE HDB maintains a separate count of the number of transactions that change the contents of the database.

The DBA must determine the optimal value for the `LFG_GROUP_COMMIT_UPDATE_TX_COUNT` property in consideration of system characteristics. The default value for this property is 80.

Group Commit improves performance by allowing as many transactions as possible to be committed in a given amount of time. However, because Group Commit works by grouping many transactions and committing them at the same time, the response time for individual transactions will be slightly longer than when not using Group Commit.

16.2.6 Statistics Related to Group Commit

ALTIBASE HDB provides statistics to help DBAs monitor Group Commit execution. Because Group Commit is executed independently for each LFG, the Group Commit-related statistics are viewed using the V\$LFG performance view.

The Group Commit-related statistics in the V\$LFG performance view are as follows:

- **UPDATE_TX_COUNT**

This statistic shows, in real time, the total number of transactions that change the database in the LFG. ALTIBASE HDB performs Group Commit for the LFG only when this value exceeds the value specified using the LFG_GROUP_COMMIT_UPDATE_TX_COUNT property.

- **GC_WAIT_COUNT**

This statistic shows the number of times that disk I/O was postponed since the most recent occurrence of disk I/O. This count increases by one whenever disk I/O cannot be performed because the amount of time specified using the LFG_GROUP_COMMIT_INTERVAL_USEC property has not passed.

- **GC_ALREADY_SYNC_COUNT**

When the contents of logs that must be written to disk in order to commit a transaction have already been written to disk by another transaction, no additional disk I/O needs to be performed in order to commit the transaction. In this situation, disk I/O is not performed, a commit response is sent, and this value is increased by 1.

- **GC_REAL_SYNC_COUNT**

The value of this property is increased by 1 whenever disk I/O is actually performed to write logs to disk. This value is incremented in either of the following two cases:

- If, at the time that logs are to be written to disk to commit a transaction, the UPDATE_TX_COUNT value in the V\$LFG performance view is less than the value specified for the LFG_GROUP_COMMIT_UPDATE_TX_COUNT property, Group Commit is not performed, so disk I/O is performed for that transaction.
- Under conditions in which Group Commit is active, when an amount of time equal to the value specified using the LFG_GROUP_COMMIT_INTERVAL_USEC property has passed since disk I/O was most recently performed, disk I/O is performed again.

16.2.7 Tuning Group Commit-Related Properties

To optimize the performance of Group Commit, it is necessary to consider both system performance and the system load under the conditions in which a lot of transactions are committed. This section describes how to optimize each of the properties related to Group Commit.

- **LFG_GROUP_COMMIT_UPDATE_TX_COUNT**

If this property is set too low, Group Commit will become active even if the number of transactions that change the database is not very high. In such cases, the number of disk I/O operations for writing logs increases, thereby compromising system performance.

On the other hand, if this property is set too high, even if the number of transactions that

16.2 Group Commit

change the database is high enough to warrant the use of Group Commit, this will be disregarded, and Group Commit will not be used.

When the database is congested with multiple transactions, the DBA is advised to monitor the UPDATE_TX_COUNT value in the V\$LFG performance view and set the LFG_GROUP_COMMIT_UPDATE_TX_COUNT property to a suitable value.

- **LFG_GROUP_COMMIT_INTERVAL_USEC**

If this property is set too low, disk I/O occurs very frequently, which has a negative impact on system performance. Under such circumstances, the value of GC_REAL_SYNC_COUNT in the V\$LFG performance view will increase very quickly, attributable to the increased frequency of disk I/O for committing transactions.

Conversely, performance will also suffer if this property is set too high, because the available disk I/O capacity cannot be utilized effectively. When this is the case, the value of GC_WAIT_COUNT in the V\$LFG performance view will increase very quickly, attributable to the increased number of waits to perform disk I/O.

To optimize the value of this property, measure the database performance in TPS (Transactions Per Second) while adjusting the value. Start with the default value, which is 1000 (one millisecond), double it, and measure again.

When the value of GC_REAL_SYNC_COUNT decreases gradually and performance is optimal, it means that this property has been optimized.

A decreasing value of GC_REAL_SYNC_COUNT does not itself mean that performance is optimal. As explained above, if this property is set too large, then transactions that need to be committed wait a needlessly long time for disk I/O, because disk I/O is not performed even when it is possible. In such cases, the optimal value of this property is found using a method similar to that described above, starting with the default value of 1000 and halving it (dividing it by 2) every time.

- **LFG_GROUP_COMMIT_RETRY_USEC**

If this property is set too low, the transactions that are waiting for disk I/O check more frequently whether the time to perform disk I/O has arrived. In such circumstances, the response time of individual transactions is shortened, but CPU usage increases due to the frequent checks of whether disk I/O is possible. Moreover, because many transactions enter a waiting state after checking whether disk I/O is possible, the value of GC_WAIT_COUNT in the V\$LFG performance view will increase more quickly than when this property is set to a high value.

On the other hand, if this property is set too high, CPU usage will be very low, because transactions do not check whether disk I/O is possible very frequently. However, because the amount of time that a transaction waits before checking whether disk I/O is possible is higher, individual transactions have to wait a longer time to receive a response after being committed.

To optimize this property, measure the average response time of individual transactions, and use a tool such as the "top" system monitor in Unix or the Task Manager in Windows to monitor the amount of CPU used by the ALTIBASE HDB process when this property is set to different values.

17 DB Diagnostic Monitoring

This chapter explains how to check and analyze the operational status of an Altibase database. This chapter contains the following sections:

- [Monitoring Database Servers](#)
- [Troubleshooting Procedures](#)

17.1 Monitoring Database Servers

Performance views are used to check the status of operation of the database. For a detailed description of every performance view available in ALTIBASE HDB, please refer to the *General Reference*.

The major entities to be monitored are as follows:

Session and Statements

Information on currently connected sessions can be checked using performance views while ALTIBASE HDB is running. Multiple statements can be assigned to a single session¹. Session properties can be set differently for each session

Database Information (Tables and Indexes)

Information about the entire database, as well as information about each table, index and tablespace, can be checked using performance views.

Memory Usage

Information on the memory areas used by ALTIBASE HDB while it is running can also be checked using performance views. This includes information about the amount of memory used for memory tablespace data (including old versions of records) storage, index storage space, temporary areas for use in processing queries, session information storage space, the memory buffer pool, and the like.

Replication Status

Finally, the status of replication can also be checked using performance views. This includes the status of threads related to replication, particularly the Sender and Receiver threads, as well as the status of replication data transmission.

1. A so-called “statement” in this context is an internally used object for processing a single SQL statement on a one-to-one basis.

17.2 Troubleshooting Procedures

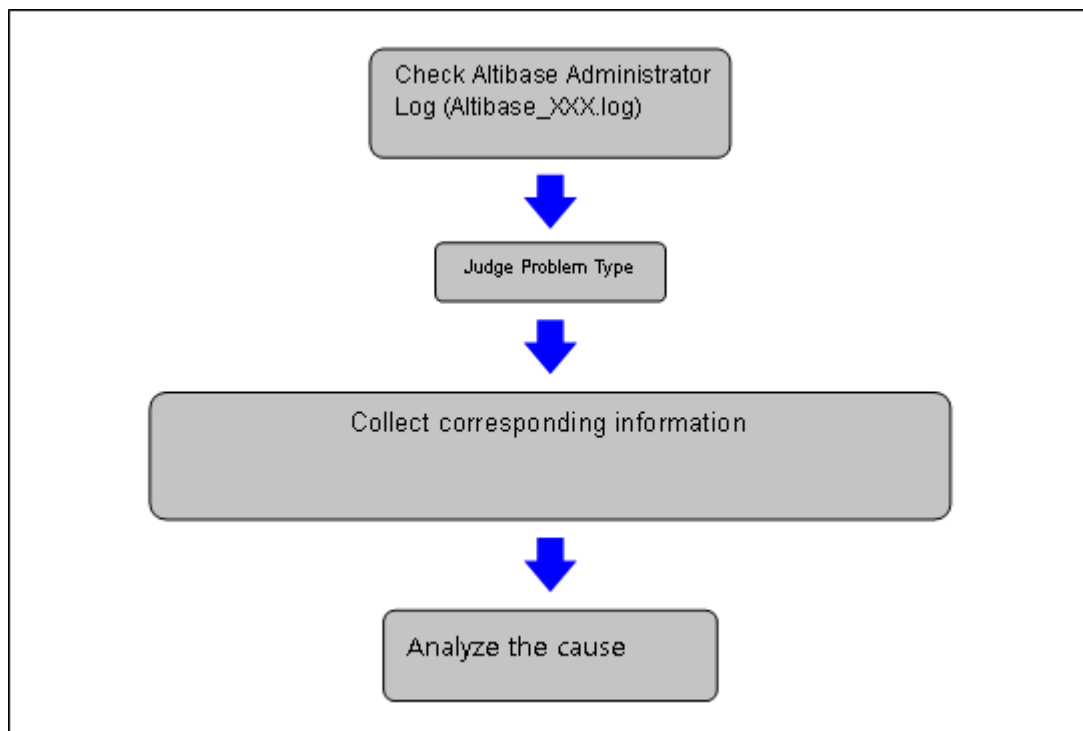
This section describes what should be checked and explains the analysis methods that are used to track various problems that may occur while ALTIBASE HDB is running. Because it is impossible to foresee all of the wide variety of problems that can occur in an actual operating environment, the problems you experience will probably not be exactly the same as those described here. Nevertheless, this section classifies the sorts of problems that fall within the range of what can reasonably be expected, and provides detailed information on how to respond to them.

The problems that are typically experienced can generally be thought of as falling into one of the following categories:

- Abnormal termination or failure to restart ALTIBASE HDB
- Poor server responsiveness
- Excessive disk usage
- Excessive memory usage
- Excessive CPU usage
- Replication-related problems
- Problems related to applications and query execution

The general troubleshooting procedure is as follows:

Figure 17-1 General Troubleshooting Procedure



17.2 Troubleshooting Procedures

The ALTIBASE HDB Administrator Logs are text logs that are created and maintained in the \$ALTIBASE_HOME/trc/ directory with the *.log filename extension. This directory contains the following trace log files:

- altibase_boot.log
- altibase_id.log
- altibase_mt.log
- altibase_qp.log
- altibase_rp.log
- and altibase_sm.log

17.2.1 Diagnosing Abnormal Server Termination and Restart Failure

17.2.1.1 Abnormal Termination

ALTIBASE HDB may shut down abnormally for either of the following reasons:

- Insufficient memory
- System OS in panic status

In the event of an abnormal shutdown, check the error messages left in the Administrator logs and make a judgment about the cause on the basis of those error messages. Unless the shutdown was caused by insufficient memory, consult an expert systems engineer for advice.

If the shutdown was caused by insufficient memory, a system call error message related to memory allocation, such as "Memory allocation failed" or "Unable to invoke the shmget() system function" will have been recorded in one or more of the Administrator Logs.

If this is the case, check the amount of memory that is currently in use to determine whether any needlessly large areas of memory are being used. If this is the case, release the memory and identify the cause of the excessive memory usage to prevent the reoccurrence of the problem. If there does not seem to be any specific cause of excessive memory usage, consider upgrading the system memory.

Memory-related problems will be discussed further in greater detail in the next section, "Memory Usage".

17.2.1.2 ALTIBASE HDB Restart Failure

Restart failures may be caused by any of the following:

- Another ALTIBASE HDB process exists, and is already using the same service port (specified using the PORT_NO property).
- Files required for startup or recovery are missing, or alternatively, files could not be accessed, attributable either to file permissions or to a problem with the file system.

If there is a file access error message in the admin log, verify the presence of all files, including

all log files, log anchor files, and data files, to determine whether the relevant file or files exist. If the error occurred even though the file exists and can be accessed, the file might be corrupted, in which case the database will need to be created again or recovery procedures will need to be performed.

- If ALTIBASE HDB is being used in shared memory mode, after abnormal termination, the data in shared memory is not compatible with the data in the database image file.

If restart failed due to shared memory compatibility problems, delete the current shared memory and restart ALTIBASE HDB.

Shared memory can be deleted using either `shmutil`, which is the ALTIBASE HDB shared memory management tool, or, in Unix, the `ipcrm` system command. Before deleting shared memory, information about the shared memory to be deleted can be reported using the `ipcs` Unix command.

- Insufficient system resources

If system startup failed due to insufficient system resources, identify which resource is lacking, check the actual amount of that resource that has been loaded and is available on the system, check the system kernel settings, and fix the area having the problem.

Most system resource-related problems that cause startup to fail are memory- or semaphore-related.

For memory-related problems, check the amount of memory that is available to a single process, the amount of available shared memory, the maximum size of a shared memory segment and the like in the system kernel settings.

When ALTIBASE HDB is used in shared memory mode, the `STARTUP_SHM_CHUNK_SIZE` property in the `altibase.properties` file setting must not be set to a greater value than the maximum size of a shared memory segment.

17.2.2 Diagnosing Poor Server Responsiveness

In cases where the server response time is very slow, it is easy to erroneously conclude that the server has become nonresponsive, whereas it is likely that the server is actually processing one or more time-consuming queries.

If the time taken to respond to a query request is very long, the cause is most likely either because an entire table is being scanned, or because memory swapping is taking place due to insufficient memory. In such cases, check the system information managed by the operating system and the information about the session in which the query was executed to determine whether the query is actually being processed. If swapping is taking place, due either to excessive CPU usage or to insufficient memory, it is likely that the query is being processed.

A more detailed description will be provided below in the section entitled “Diagnosing Problems related to Application and Query Execution”.

Another cause of nonresponsiveness is insufficient disk space. This possibility should be considered in cases where there is no response after data are changed or entered. In such cases, an error message indicating insufficient disk space is recorded in one or more of the Administrator Logs, and the nonresponsive status will persist until sufficient disk space has been acquired. A detailed description of how to solve problems related to insufficient disk space will be provided below in the section entitled “Diagnosing Problems related to Disk Usage”.

17.2 Troubleshooting Procedures

If an ALTIBASE HDB server becomes nonresponsive for any reason not described above, consult a specialized systems engineer.

17.2.3 Diagnosing Problems Related to Disk Usage

17.2.3.1 Insufficient Disk Space

If the amount of disk space becomes insufficient while ALTIBASE HDB is running, the system will stop operating, and will no longer be able to make changes to data. In such cases, the first task is to determine which of the file systems is lacking in disk space.

While ALTIBASE HDB is running, it uses disk space in the following ways:

- For log file storage
- For tablespace file storage

Active logs and archive logs are generated continuously while ALTIBASE HDB is running. Active log files are saved in the directory designated using the LOG_DIR property in the ALTIBASE HDB configuration file, altibase.properties. Additionally, when the database is operating in archive log mode, archive log files are automatically saved in the directory designated using the ARCHIVE_DIR property.

If, due to insufficient disk space, ALTIBASE HDB becomes unable to save any more active log files, ALTIBASE HDB stops operation. In such cases, because deleting log files or log anchor files would make recovery impossible, it is necessary to either extend the size of the file system or delete unnecessary files.

In the case of archive log files, if the ARCHIVE_FULL_ACTION property in the altibase.properties file is set to 0, when the system runs out of space for saving archive logs, it will continue to operate without saving them. If this property is set to 1, however, the system will stop operating until additional space is made available.

If the number of log files saved in the directory specified using the LOG_DIR property increases, resulting in a shortage of space for saving log files, first check the Administrator Log files to determine whether checkpointing is executing normally, and verify that the CHECKPOINT_INTERVAL_IN_SEC and CHECKPOINT_INTERVAL_IN_LOG properties in the ALTIBASE HDB configuration file are properly set. If checkpointing is executing normally but there are still log files that have been deleted or archived remaining in the directory specified using the LOG_DIR property, check the status of replication transmission. If replication transmission is slow, or if replication logs cannot be sent, log files will not be archived or deleted, and will thus accumulate in the directory specified using the LOG_DIR property, which can lead to a shortage of log storage space.

A more detailed description will be provided below in the section entitled “Diagnosing Problems related to Replication”.

Memory and system tablespaces are saved in the directory specified using the MEM_DB_DIR property in the altibase.properties file. If the shortage in disk space is tablespace-related, check this property and the amount of space available for storing user-created tablespace files. The amount of free space in the file system in which these tablespaces are stored must be at least as large as the amount by which these tablespaces are set to increase.

Because storage space for all memory tablespaces is used for checkpointing, the amount of free space must be greater than the size of all memory tablespaces existing in memory.

17.2.4 Diagnosing Problems related to Memory Usage

If ALTIBASE HDB runs out of memory while it is running, response times can greatly increase, and ALTIBASE HDB might shut down abnormally. In such cases, check the amount of memory that ALTIBASE HDB is using to determine whether it is reasonable, and eliminate any causes of unnecessary memory space usage, if any. If there do not seem to be any factors causing unnecessary memory space usage, consider providing additional memory.

The ways in which ALTIBASE HDB uses memory space can be broadly classified as follows:

- Memory tablespaces
- Temporary memory spaces
- Memory buffers

Both actual (current) memory table records and previous versions of those records, which are required in order to support MVCC, are stored in memory tablespaces.

Temporary memory space is used to store indexes for memory tables, space for sorting records when querying memory tables, information about sessions, and the like.

The memory buffer is used to sort disk table records and perform other kinds of operations on disk tables.

If excessive use of memory is suspected, for a set period monitor the number of statements that are created, the amount of temporary memory space that is used, memory tablespace usage, the size of memory table indexes and the like to determine whether they have increased, and use system monitoring utilities such as “ps” and “top” in Unix to determine whether the size of a process is increasing continuously.

When ALTIBASE HDB is initially started up, memory usage may be higher than normal for a period of time, during which temporary memory is allocated, multiple previous versions of records are reconstructed, and the amount of session information increases. This is normal.

However, if memory usage continues to rise after this period, a memory leak or similar problem may be the culprit. In such cases consult a specialized systems engineer.

17.2.5 Diagnosing Excessive CPU Usage

When ALTIBASE HDB exhibits sudden surges in CPU usage, the following scenarios should be considered as possible causes:

- A query of a memory table was processed without using an index.
- Excessive disk I/O occurred when a query of a disk table was processed.
- Swapping occurred due to insufficient memory.

Use a system performance monitoring tool to check the amount of memory that is being used. If memory swapping is occurring due to insufficient memory, it may be necessary to install additional memory. A more detailed description will be provided below in the section entitled “Diagnosing Problems related to Memory Usage”.

If the problem was caused not by insufficient memory but by a full scan of memory table or exces-

17.2 Troubleshooting Procedures

sive disk usage relating to a query of a disk table, the problem can be solved by tuning the corresponding query or modifying the table.

A detailed description will be provided below in the section entitled “Diagnosing Problems related to Application and Query Execution”.

17.2.6 Diagnosing Problems related to Replication

The following types of replication-related problems may occur:

- Failure to start replication
- Slow progress of replication
- Inconsistent number of records in the tables being replicated

If a replication transmission problem occurs, log files accumulate in the log storage space, which can decrease the amount of free space to the point where it is insufficient, ultimately resulting in a service interruption.

If a replication-related problem occurs, check the administrator log files for any replication-related error messages, and pass this information on to a specialized engineer.

If one of the systems in a replication environment develops a fault and remains faulty for a long time, it will be impossible to transmit replication data, decreasing the amount of available space in the log storage directory. Therefore, if it is taking a long time to solve the problem with the system in which the fault occurred, consider pausing replication and deleting the replication object to prevent problems with the system that is still running. In such cases, once the problem has been solved, it will subsequently be necessary to perform data recovery on the system that developed the fault. The data can be recovered either using the iLoader tool or by running replication in SYNC mode.

If it is difficult to delete the replication object, it will be necessary to continuously monitor the amount of available space in the log storage directory and add additional space if it becomes insufficient.

Similar measures should be taken in the event of a problem with the replication network connection, or in the event that replication connection is unsuccessful for a long time due to some replication-related error.

17.2.7 Diagnosing Problems related to Application and Query Execution

The problems that may occur while executing applications and queries fall generally into the following two categories:

- An application fails to connect to ALTIBASE HDB.
- After an application submits a query processing request to ALTIBASE HDB, the system stops operating, or the query times out.

17.2.7.1 ALTIBASE HDB Access Failure

If an application fails to access ALTIBASE HDB, determine whether it is possible to connect to ALTIBASE HDB using the iSQL ALTIBASE HDB utility. Because ALTIBASE HDB supports three different access methods (TCP/IP, Unix socket domain, and IPC), test the connection using the access method that is used by the application.

The access method can be configured by setting the ISQL_CONNECTION environment variable to one of TCP, UNIX, or IPC.

If it is possible to connect to ALTIBASE HDB using iSQL, the problem is related to internal access settings within the application, and can be solved by adjusting these settings. If connection via iSQL fails, either of the following scenarios may be the reason:

- The number of connected sessions exceeds the value specified using the MAX_CLIENT property in the altibase.properties file.
- If the access protocol is IPC, the number of sessions connected via IPC exceeds the value specified using the IPC_CHANNEL_COUNT property.

17.2.7.2 Nonresponsiveness or Session Disconnected due to Timeout

If no response to a query processing request is received due to slow query processing, any of the following situations may be the cause:

- Swapping is taking place because the amount of memory is insufficient.
- Performance is decreased due to a scan of an entire table.
- The system is waiting to acquire a lock on a table.

First, use a system performance monitoring tool and check the CPU usage and memory usage to identify whether the problem is caused by insufficient memory. If this is the case, please refer to the “Diagnosing Problems related to Memory Usage” section above.

If there is no memory shortage problem, but CPU usage is high, check all of the queries that are currently executing to determine whether any of them is performing a full scan of a table.

To view a list of the queries that are currently being processed, connect using iSQL and check the QUERY column in the V\$STATEMENT performance view.

Determine which of the queries that are currently executing is likely to be causing the problem, check the execution plan for that query, and tune the query if a problem is found.

For a detailed explanation of how to tune queries, please refer to “Chapter 11. SQL Tuning” in this manual.

If the problem does not seem to have been caused by either of the above two reasons, it is likely that the system is waiting to acquire a lock on a resource. Check the information on currently held locks in the V\$LOCK and V\$LOCK_WAIT performance views to verify whether any unnecessary locks are being continuously held in any sessions. If this is the case, forcibly terminating the session will solve the problem.

Appendix A. Trace Logs

Using Application Trace Logs

The following property in the `altibase.properties` file can be used to specify that various kinds of information pertaining to ALTIBASE HDB execution is written to the `altibase_boot.log` file. This information includes information about SQL statements being executed on an ALTIBASE HDB server, various kinds of error messages, SQL statement execution times, and the like.

The default value for this property is 0; to specify that information is to be written to a trace log, set the corresponding property to 1.

The value set in the property file can be overridden using the `ALTER SYSTEM` statement. For more information about this property, please refer to the *General Reference* and to Chapter 11, Section 11.3 “Analyzing Execution Plans” in this manual.

| TRCLOG | Description |
|-------------------------|---|
| TRCLOG_DETAIL_PREDICATE | When EXPLAIN PLAN is set to ON (or ONLY), information about the status of the predicate portion of the WHERE clause is also output. |

Appendix B. ALTIBASE HDB Limitations

Maximum ALTIBASE HDB Values

The following table lists the maximum possible values that various ALTIBASE HDB objects can have.

| Identification | Maximum Values of Objects | Remarks |
|---------------------------|--|--|
| DB_NAME Length | 128 | The maximum length of the database name |
| OBJECT Length | 40 | The maximum length of database object names |
| The Number of Tablespaces | 65,536 (Including System Tablespaces) | The maximum number of tablespaces in one DB |
| The Number of Data Files | 1,024 | The maximum number of data files in one tablespace |
| | 67,108,864 (65,536 * 1024) | The maximum number of data files in one DB |
| The Number of Users | 2,147,483,638 (Including System Users) | The maximum number of users in one DB |
| The Number of Tables | 2,097,151 (Including Meta Tables) | The maximum number of tables in one DB |
| The Number of Indexes | 64 | The maximum number of indexes per table |
| The Number of Columns | 1,024 | The maximum number of columns per table |
| | 32 | The maximum number of key columns per index |
| The Number of Rows | Unlimited | The maximum number of row per table |
| The Number of Partitions | 2,147,483,638 (In the Entire System) | The maximum number of partitions in one DB |

Maximum ALTIBASE HDB Values

| Identification | Maximum Values of Objects | Remarks |
|---------------------------|--------------------------------------|---|
| The Number of Constraints | 2,147,483,638 (In the Entire System) | The maximum number of constraints in one DB |

Index

A

- Access Method 236
- admin directory 20
- aexport 25
- AGGR Execution Node 286
- AGGREGATION node 326
- alaAPI.h 21
- Altering Tables 50
- Altering Undo Tablespace 102
- altibase 25
- Altibase monitoring 382
- Altibase.jar 22
- altibase_boot.log 23
- ALTIBASE_HOME 20
- altibase_id.log 23
- altibase_IPC.log 24
- altibase_mm.log 24
- altibase_mt.log 23
- altibase_qp.log 23
- altibase_rp.log 23
- altibase_sm.log 23
- altierr 25
- altimon 25
- altipasswd 25
- altiprofile 25
- ANTI-OUTR-JOIN node 327
- AOJN Execution Node 308
- apre 26
- Archive Thread 15
- arch_logs directory 20
- audit 25
- audit directory 20

B

- Background of Hybrid DBMS 2
- BAG-UNION node 328
- base table 62
- BCB 182
- BCB Mode 186
- bin directory 20
- Binary Materialization Node join 310
- Binary Non-materialization Node 302
- b-tree index 57
- Buffer Control Block 182
- Buffer Flush Thread 15
- Buffer Frame 182
- Buffer Manager 182
- Buffer Pool 9
- Buffer pool 182
- BUNI Execution Node 309

C

- CACHE 65
- Changing a Column in Encrypted form 369
- Checkpoint 10
- Checkpoint Thread 15
- checkServer 25
- Client-Server Structure 11
- CNF 315
- Cold Backup 115
- column constraint 55
- Column Encyption 369
- commit 167
- Complete Recovery 204
- CONC Execution Node 309
- CONCATENATION node 328
- Condition Clause 232
- conf directory 20
- constant filter 237
- constraint 44
- cost 315
- COUNT Execution Node 290
- COUNT node 329
- CPU Usage 387
- Creating a Database 32
- Creating Tables 49
- CYCLE 65

D

- Data File 17
- Data Page Structure 94
- Data tablespace 30
- database backup 198
- Database Link 45
- database mode 200
- database objects 44
- Database Recovery 203
- Database Startup 31
- DB Space 11
- dbz directory 20
- Deadlock Detection 10
- DECRYPT 370
- Decryption 370
- default partition 147
- Definition and Structure of Tablespace 82
- directory 45
- Direct-Path INSERT 11
- Disk Data File Attributes 104
- Disk Table 223
- disk table 47
- Disk Tablespace 94

- disk tablespace 83
- Disk Temporary File Attributes 105
- Disk Usage 386
- distinct 257
- DISTINCT node 330
- DISTINCT_HASH 317
- DISTINCT_SORT 317
- DNF 315
- Double Write 9
- DRDBMS 2
- Drop Tables 51
- dumpla 26
- dumpIf 26
- dump_stack.sh 26

E

- encrypting column 369
- Encryption Structure 365
- Encyption 369
- engine structure 8
- exclusive lock 169
- Executable Binary 25
- Execution Node 274
- Execution Plan 272, 320
- Execution plan manager 228
- execution plan tree 220
- executor 223
- EXPLAIN PLAN 272
- extent 85

F

- FILT Execution Node 282
- FILT Node Information 283
- filter 237
- FILTER node 331
- Fixed Size Mode of Undo Tablespace 126
- Flush Thread 186
- FOJN Execution Node 307
- Foreign Key Constraints 55
- FULL SCAN 318
- FULL-OUTER-JOIN node 331

G

- Garbage Collection Thread 15
- global index 144
- GRAG Execution Node 296
- grant 78
- Graph manager 228
- GRBY Execution Node 284
- GROUP BUCKET COUNT 317
- Group Commit 377
- group operation 255
- GROUP-AGGREGATION node 333

- GROUP_HASH 317
- GROUPING node 334
- GROUP_SORT 317

H

- HASH BUCKET COUNT 317
- HASH Execution Node 294
- HASH node 334
- hash partition pruning 262
- hash partitioning 160
- Hash Table 184
- hash-based join 253
- HIER Execution Node 291
- HIERARCHICAL QUERY node 336
- hint 314
- Hit Ratio 196
- HOT Backup 114
- How to Encrypt Data with Statements 368
- How to Run Security Module 366
- HSDS Execution Node 297
- Hybrid DBMS 4

I

- iload 26
- include directory 21
- Incomplete Recovery 204
- INCREMENT BY 65
- INDEX 318
- index 44, 57
- INDEX ASC 318
- Index Attributes 57
- Index creation options 59
- INDEX DESC 318
- Index Selection 236, 238
- install directory 21
- intent exclusive lock 169
- intent shared lock 169
- Interfaces 8
- IP Stack 359
- IPv6 Address 358
- isql 26

J

- join 250
- JOIN Execution Node 302
- JOIN node 337
- Join Order 247

K

- key filter 237
- key range 237
- killCheckServer 26

L

- Large Volume Memory Table 47
- LEFT-OUTER-JOIN node 338
- lib directory 21
- libapre.a 22
- libodbccli.ar 22
- limit 259
- LIMIT-SORT node 339
- list partition pruning 261
- list partitioning 156
- LMST Execution Node 300
- local index 144
- Lock 169
- Lock Compatibility 170
- log anchor file 31
- Log Anchor Files 16
- log file 31
- Log File Creation Thread 372
- Log Files 16
- Log Flush Thread 15, 372, 373
- Logging 9
- Logical backup 198
- Logical Structure 17
- logs directory 22
- LOJN Execution Node 306
- LRU List 184

M

- Main Thread 14
- Main-Module 6
- Managing Tablespaces 104
- MATERIALIZATION node 340
- materialized node 325
- Media Recovery 203
- Memory table 47, 223
- Memory tablespace 30
- memory tablespace 86
- Memory Tablespace Attributes 106
- Memory Usage 387
- merge join 254
- Message queue table 53
- MGJN Execution Node 305
- MINVALUE 65
- MMDBMS 3
- Monitoring
 - Memory Usage 382
 - Replication Status 382
 - Session and Statements 382
- msg directory 22
- Multi Version Concurrency Control 172
- Multiplexed Page Lists 48
- Multi-Version Concurrency Control 9
- MVCC 172

- MVCC of Disk Tablespace 175
- MVCC of Memory Tablespace 173

N

- NO INDEX 318
- Node for PCRD 313
- non-materialized node 325
- non-partitioned index 141
- non-partitioned object 138
- non-persistent index 58
- NO_PUSH_SELECT_VIEW 317
- normalization 231
- NOT NULL Constraint 55

O

- Object Privilege 77
- Offline backup 207
- optimization 227
- optimizer 224
- Optimizer Structure 227
- optimizing partition 259
- order by 258
- ORDERED 315

P

- page 85, 88
- page list 88
- partition filtering 263
- partition key 139
- partition pre-condition 147
- PARTITION-COORDINATOR 348
- partitioned index 141
- partitioned object 138, 140
- partitioned table 140
- partitioning 138
- PCTFREE 95
- PCTUSED 96
- persistent index 58
- Physical backup 198
- Physical Structure 16
- plan tree 320
- Predicate manager 228
- Primary Key Constraints 55
- privilege 76
- PROJ Execution Node 284
- PROJECT node 342
- projection 264
- properties 33
- push selection Method 233
- PUSH_SELECT_VIEW 317

Q

- Query Optimization Process 230

query processing 225
Query Processor 5
QUEUE table 53

R

Range Partition Pruning 260
range partitioning 149
Real-time Statistics 229
Record Level Lock Mode 170
Replicated Tables 48
Replication 10
replication 46
revoke 78
rollback 167
Row Chaining 98
Row Migration 98
Row Structure 97
r-tree index 57
rule 315

S

sample directory 23
savepoint 168
SCAN node 342
schema objects 44
SDIF Execution Node 311
Securing Data 364
segment 84
sequence 45, 64
server 26
Service Daemon Thread 15
Service Thread Pool 15
Session Management Thread 15
SET BUCKET COUNT 317
Set Operations 258
SET-DIFFERENCE node 343
SET-INTERSECT node 345
shared lock 169
Shared Memory 360
shared with intent exclusive lock 169
shmutil 26
shutdown abort 41
Shutdown ALTIBASE 40
shutdown immediate 40
Shutdown Server 33
SITS Execution Node 310
SORT Execution Node 292
SORT MERGE JOIN node 341
SORT node 346
sort-based join 252
SQL Tuning 220
sqlcli.h 21
sqltypes.h 21

sqlucode.h 21
START WITH 65
Startup Procedures 38
startup service 38
startup stage 39
STOR Execution Node 299
Storage Manager 5
stored function 68
Stored Procedure 10
stored procedure 68
store procedure or function 45
subquery 265
subquery filter 237
synonym 45, 67
SYS User 74
system privilege 76
system table 47
SYSTEM_ User 74

T

table 44
Table Compaction 10
table constraint 55
tablespace 46
tablespace backup 114
Tablespace Info 135
tablespace recovery 114
Tablespace State 103
Tablespaces
 Definition and Structure of Tablespace 81
Temporary tablespace 31
The Internal Structure of Altibase 7
TIMESTAMP Constraint 55
trace log 391
Transaction 166
Transaction Durability 178
Transaction Processing 9
Transaction Segment Management 100
trc directory 23
TRCLOG_DETAIL_PREDICATE 272, 391
trigger 45, 72
trigger restriction 72
Truncating Tables 51
Types of Graph 228
types of tablespace 90

U

ulpLibInterface.h 21
Unary Materialization Node 292
Unary Non-Materialization Node 278
Undo Record 99
Undo Tablespace 99
Undo tablespace 30

- Undo Tablespace Size Estimation 126
- UNIQUE KEY 55
- Unix Domain Socket 360
- USE_HASH 316
- USE_NL 315
- user 46, 74
- user table 47
- USER_MERGE 316
- USER_SORT 316

V

- view 44
- VIEW Execution Node 287
- VIEW Node 347
- VIEW-SCAN node 348
- VMTR Execution Node 299
- volatile tablespace 88
- Volatile Tablespace Attributes 108
- VSCN Execution Node 289