Altibase Application Development

SQL Reference

Release 6.1.1

May 23, 2012

ALTIBASE
PERFORMANCE SOLUTIONS

Altibase Application Development SQL Reference

Release 6.1.1

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

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

Contents

Preface

About This Manual

This manual explains how to use SQL (Structured Query Language) statements with an Altibase database.

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 5.5.1 will be used as the database server.

Organization

This manual is organized as follows:

- Chapter1: Introduction

  This chapter provides an overview of the structure and organization of SQL in ALTIBASE HDB.
- Chapter2: Data Definition Statements

  This chapter explains each of the Data Definition Language (DDL) statements that are available in ALTIBASE HDB.
- Chapter3: Data Manipulation Statements

  This chapter explains each of the Data Manipulation Language (DML) statements that are available in ALTIBASE HDB.
- Chapter4: Data Control Statements

  This chapter explains each of the Data Control Language (DCL) statements that are available in ALTIBASE HDB.
- Chapter5: Set Operators

This chapter explains the set operators that can be used in SQL statements in ALTIBASE HDB.

- Chapter6: SQL Functions

  This chapter describes the functions that are supported in ALTIBASE HDB.

- Chapter7: Arithmetic Operators

  This chapter describes the arithmetic operators that can be used in SQL statements in ALTIBASE HDB.

- Chapter8: SQL Conditions

  This chapter describes the conditional operators that can be used in conditional clauses in SQL statements in ALTIBASE HDB.

Documentation Conventions

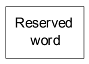
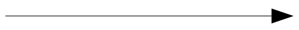
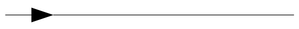
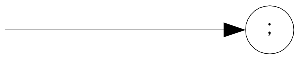
This section describes the conventions used in this manual. Understanding these conventions will make it easier to find information in this manual and in the other manuals in the series.

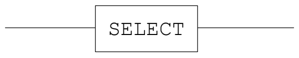
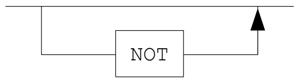
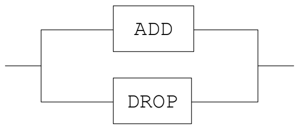
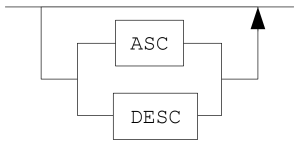
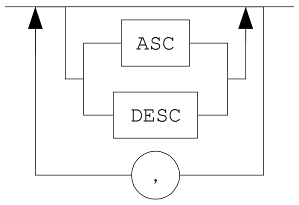
There are two sets of conventions:

- Syntax Diagram Conventions
- Sample Code Conventions

Syntax Diagram Conventions

In this manual, the syntax of commands is described using diagrams composed of the following elements:

Element	Description
	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.

Element	Description
	Indicates a mandatory element.
	Indicates an optional element.
	Indicates a mandatory element comprised of options. One, and only one, option must be specified.
	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.

Convention	Meaning	Example
[]	Indicates an optional item.	`VARCHAR [(size)] [[FIXED |] VARIABLE]`

Convention	Meaning	Example
{ }	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);`

Italics	Statement elements in italics indicate variables and special values specified by the user.	`SELECT * FROM table_name;` CONNECT *userID/password*;
Lower Case Letters	Indicate program elements set by the user, such as table names, column names, file names, etc.	`SELECT e_lastname FROM employees;`
Upper Case Letters	Keywords and all elements provided by the system appear in upper case.	`DESC SYSTEM_.SYS_INDICES_;`

Related Reading

For additional technical information, please refer to the following manuals:

- Installation Guide
- Administrator's Manual
- Replication Manual
- Precompiler User's Manual
- ODBC Reference
- Application Program Interface User's Manual
- iSQL User's Manual
- Utilities Manual
- Error Message Reference

Online Manuals

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

Altibase Welcomes Your Comments

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 that you are using
- Any comments that you have about the manual
- Your full name, address, and phone number

Write to us at the following e-mail address: support@altibase.com

For immediate assistance with technical issues, please contact the Altibase Customer Support Center.

We always appreciate your comments and suggestions.

1 Introduction

1.1 Overview of SQL in ALTIBASE HDB

SQL (Structured Query Language) is a language for defining data objects and managing, manipulating and searching the data in a database. This section describes the characteristics of SQL in ALTIBASE HDB.

1.1.1 Features

- Superior Query Performance

  In ALTIBASE HDB, because the system catalog information for most queries does not change between the time of preparation (SQL prepare) and the time of execution, an optimized execution plan tree is made in advance, at the time point of performing preparation tasks, so that it can be used repeatedly to realize greatly improved execution speed when the query is actually executed. This method is useful in situations where Data Manipulation Language (DML) statements are frequently executed but Data Definition Language (DDL) statements are seldom executed once the initial database scheme has been created.

- Support for the SQL-92 Standard

  The ALTIBASE HDB implementation of SQL fully complies with the SQL-92 specification so that users who are already familiar with SQL can easily get started using ALTIBASE HDB without having to upgrade their skill sets.

- Powerful Subquery Support

  Subqueries are usually used in formulae and IN clauses within SELECT statements, CREATE TABLE ~ AS SELECT statements, and INSERT~ AS SELECT statements. Most subqueries used in this way return multiple columns and multiple records.

  In ALTIBASE HDB, when the result of a subquery is a single value (a single record and a single column), because the subquery can be used instead of a constant and because subqueries can be executed quickly, it is better to use the subquery than multiple SQL statements not containing the subquery. Therefore, subqueries are useful for realizing complicated applications.

- A Wide Variety of System Functions

  In addition to the SQL-92 specification, ALTIBASE HDB supports a wide variety of very useful system functions.

1.1.2 Query Optimizer and Execution Plans

Efficient SQL statements make for better system performance. Therefore, a thorough understanding of how ALTIBASE HDB processes SQL statements is necessary in order to write optimal SQL statements. This is especially true for systems that process large volumes of transactions.

In ALTIBASE HDB, the tasks involved in processing an SQL statement can be broadly divided into preparation tasks and execution tasks.

1. Preparation Step

   The preparation step comprises analyzing the syntax of an SQL statement, checking its valid-ity, optimizing it, and then creating an execution plan and an execution plan tree. During the course of these tasks, meta tables are consulted, information about tables, indexes, etc. is read, and an optimized execution plan is created.

Therefore, if preparation and execution tasks are executed separately in a client application, rather than using direct execution, the execution plan tree created during the preparation tasks can be used without change as long as the meta information has not changed since the preparation tasks. For example, if an index that existed when a statement was prepared no longer exists when it is time to execute the statement, the execution plan tree that was optimized on the basis of the index when the statement was prepared is invalid, and thus cannot be used.

2. Execution Step

   The execution task comprises actually executing the query statement, following the execution plan that was created during preparation work. When a SQL statement uses host variables that are changed to different values every time it is executed, the preparation task is performed only once, and the tasks of setting the value of the variables and executing the statement are performed multiple times.

This manual is a reference for the syntax and usage of SQL in ALTIBASE HDB. For more information about SQL performance and optimizing SQL statements, please refer to the “SQL Tuning” chapter in the Administrator's Manual.

1.1.3 Comments Syntax

In ALTIBASE HDB, the following two types of comment delimiters can be used in SQL statements:

- /* */

  Using the same syntax as in the C language, the start of a comment is indicated with '/*' and the end with '*/'. This kind of comment can occupy multiple lines.

- --

  Use two hyphen '-' characters to indicate the start of a single-line comment.

1.1.4 Objects in ALTIBASE HDB

The data objects provided in ALTIBASE HDB are classified as either schema objects or non-schema objects as follows:

1.1.4.1 Schema Objects

- Constraint
- Index
- Sequence
- Synonym
- Table

- Stored Procedure
- View
- Trigger
- Database Link

1.1.4.2 Non-schema Objects

- User
- Replication
- Tablespace
- Directory

1.1.5 Rules for Object Names

1.1.5.1 Object Names

In an SQL statement, the name of an object can be represented using a quoted identifier or a non-quoted identifier.

- A quoted identifier begins and ends with double quotation marks ("). If an object is named using a quoted identifier when the object is created, then the quoted identifier must be used when the object is referred to.
- A nonquoted identifier is not surrounded by any delimiters.

The following rules apply to both quoted and nonquoted identifiers:

1. The maximum length of an object name is 40 bytes.
2. Nonquoted identifiers can contain the upper-case letters from A to Z, the lower-case letters from a to z, the numeric characters from 0 to 9, the underscore character ('_') and the dollar-sign character ('$'). Additionally, the first letter of an object name must be a letter or the underscore character('_'). However, nonquoted identifiers cannot begin with 'V$', 'X$', or 'D$'.

   Quoted identifiers can contain any character, punctuation marks and spaces. However, they cannot contain double quotation marks.
3. ALTIBASE HDB reserved words cannot be used as object names. (A list of reserved words in ALTIBASE HDB is found below.)
4. Two objects cannot have the same name within the same namespace.

   The following schema objects share one namespace:

   - Tables
   - Views
   - Sequences

- Synonyms
- Stored procedures

Each of the following schema objects has its own namespace:

- Constraints
- Indexes
- Triggers
- Database Link objects

Because tables and views share the same namespace, a table and a view in the same schema cannot have the same name. However, because tables and indexes are in different namespaces, a table and an index in the same schema can have the same name.

Each of the following non-schema objects also has its own namespace:

- Users
- Replication objects
- Tablespaces
- Directory objects

Because these objects do not belong to schemas, each of them has its own namespace in a database.

5. Nonquoted identifiers are not case-sensitive. ALTIBASE HDB internally changes them to upper-case letters. Quoted identifiers are case-sensitive.

   Because the following names are changed into the same name in ALTIBASE HDB, they cannot be used for different objects in the same namespace:

   employees, EMPLOYEES, "EMPLOYEES"

For more information about the objects provided with ALTIBASE HDB, please refer to the *Administrator's Manual*.

1.1.5.2 Passwords

The passwords that users use to connect to ALTIBASE HDB are governed by restrictions similar to those for object names. The only characters that can be used for passwords are the upper-case letters from A to Z, the lower-case letters from a to z, the numeric characters from 0 to 9, the underscore character ('_') and the dollar-sign character ('$'). Additionally, ALTIBASE HDB reserved words cannot be used as object names. The maximum password length is 8 bytes or 11 bytes, depending on the operating system. In Solaris x86 2.8 and above and Windows, the maximum password length is 11 bytes. In other operating systems, the maximum password length is 8 bytes.

1.1.5.3 Reserved Words

The following words are reserved words in ALTIBASE HDB, and cannot be used as database object names or passwords. This needs to be borne in mind when creating database objects and writing SQL statements.

```
ADD                 ALL                 ALTER
AND                 ANY                 AS
```

ASC	BEGIN	BETWEEN
BY	CASCADE	CASE
CHECK	CLOSE	COLUMN
COMMIT	CONNECT	CONSTANT
CONSTRAINT	CONSTRAINTS	CONTINUE
CREATE	CUBE	CURSOR
CYCLE	DATABASE	DECLARE
DEFAULT	DELETE	DESC
DISCONNECT	DISTINCT	DROP
ELSE	ELSEIF	ELSIF
END	ESCAPE	EXCEPTION
EXEC	EXECUTE	EXISTS
EXIT	EXTENTSIZE	FALSE
FETCH	FIXED	FOR
FOREIGN	FROM	FULL
FUNCTION	GET	GOTO
GRANT	GROUP	GROUPING
HAVING	IDENTIFIED	IF
IN	INDEX	INNER
INSERT	INTERSECT	INTO
IS	ISOLATION	JOIN
LEFT	LEVEL	LIKE
LIMIT	LOCALUNIQUE	LOCK
LOOP	MAXROWS	MINUS
MODE	NATIVE	NO
NOCYCLE	NOT	NULL
OFF	OFFLINE	ON
OPEN	OR	ORDER
OTHERS	OUT	OUTER
PRIMARY	PRIOR	PRIVILEGES
PROCEDURE	RAISE	READ
REFERENCES	RENAME	REPLACE
REPLICATION	RESTRICT	RETURN
REVERSE	REVOKE	RIGHT
ROLLBACK	ROLLUP	ROW
ROWCOUNT	ROWTYPE	SAVEPOINT
SELECT	SEQUENCE	SESSION
SET	SETS	SOME
SQLCODE	SQLERRM	START
STEP	SYNONYM	SYSTEM
TABLE	TABLESPACE	TEMPORARY
THEN	TO	TRANSACTION
TRIGGER	TRUE	TRUNCATE
UNION	UNIQUE	UNTIL
UPDATE	USER	VALUES
VARIABLE	VIEW	WAIT
WHEN	WHERE	WHILE
WITH	WORK	SWRITE

1.2 SQL Statements Classification

Every SQL statement supported in ALTIBASE HDB falls into one of the following categories:

- Data Definition Language (DDL)
- Data Manipulation Language (DML)
- Data Control Language (DCL)

Each of these SQL statements will be briefly introduced in this chapter.

1.2.1 Data Definition Language (DDL)

DDL is a set of commands used in database systems that allow users to define databases, data types and structures, and data constraints.

SQL statement	Description
ALTER DATABASE	Change a database definition
ALTER INDEX	Change the definition of an index
ALTER QUEUE	
ALTER REPLICATION	Change, enable or disable an existing replication object
ALTER SEQUENCE	Change the definition of a sequence
ALTER TABLE	Change the definition of a table
ALTER TABLESPACE	Change the definition of a tablespace
ALTER TRIGGER	Change the definition of a trigger
ALTER USER	Change a user's password
ALTER VIEW	Rebuild an existing view
CREATE DATABASE	Create a database
CREATE DIRECTORY	Create a directory object for handling files in stored procedures
CREATE INDEX	Create an index for a table
CREATE QUEUE	Create a queue
CREATE REPLICATION	Create a replication object
CREATE SEQUENCE	Create a sequence
CREATE SYNONYM	Create an alias for an object
CREATE TABLE	Create a table
CREATE DISK TABLESPACE	Create a disk tablespace

SQL statement	Description
CREATE MEMORY TABLESPACE	Create a memory tablespace
CREATE VOLATILE TABLESPACE	Create a volatile tablespace
CREATE TEMPORARY TABLESPACE	Create a temporary tablespace
CREATE TRIGGER	Create a trigger
CREATE USER	Create a database user
CREATE VIEW	Create a view
DROP DATABASE	Remove a database
DROP DIRECTORY	Remove a directory object from the database
DROP INDEX	Remove an index from the database
DROP QUEUE	Remove a queue
DROP REPLICATION	Remove a replication object from the database
DROP SEQUENCE	Remove a sequence object from the database
DROP SYNONYM	Remove a synonym object from the database
DROP TABLE	Completely remove a table and all of its data from the database
DROP TABLESPACE	Remove a tablespace from the database
DROP TRIGGER	Remove a trigger from the database
DROP USER	Remove a user, and optionally removes the user’s objects and data, from the database
DROP VIEW	Remove a view from the database
GRANT	Grant privileges to database users
RENAME TABLE	Change the name of a table
REVOKE	Take privileges away from a database user
TRUNCATE TABLE	Delete all rows from a table

When any of the above DDL statements, which change system metadata, are executed, all transactions that are currently underway in the same session as the session in which the DDL statements are executed are immediately committed. Then the DDL statement is executed. That is, each DDL SQL statement is handled as a single transaction. In other words, even when AUTOCOMMIT has been set to OFF, under which conditions DML statements are normally not committed automatically when they are executed, when any of the above DDL SQL statements is executed, all previously executed DML transactions are automatically committed. That is, DML transactions that were executed before a DDL SQL statement is executed cannot be rolled back after the DDL SQL statement has completed execution.

1.2.2 Data Manipulation Language (DML)

DML (Data Manipulation Language) statements are used to manipulate data within existing database tables. They differ from DDL statements in that they are not automatically committed when AUTOCOMMIT has been set to OFF. Therefore, as long as AUTOCOMMIT has been set to OFF, a transaction comprising multiple DML statements can be executed and subsequently rolled back.

SQL statement	Description
DELETE	Remove data from a table
INSERT	Insert data into a table
LOCK TABLE	Lock a table in a specified lock mode
SELECT	Retrieve data from objects
UPDATE	Change the data in a table
MOVE	Move data from one table to another
ENQUEUE	Insert a message into a queue
DEQUEUE	Retrieve a message from a queue and remove it from the queue

1.2.3 Data Control Language (DCL)

1.2.3.1 System Control Statements

SQL statement	Description
ALTER SYSTEM	Change database properties, perform checkpointing, and back up the database

1.2.3.2 Session Control Statements

SQL statement	Description
ALTER SESSION	Change the properties for a session

1.2.3.3 Transaction Control Statements

SQL statement	Description
COMMIT	Complete a transaction normally
ROLLBACK or ROLLBACK TO SAVE-POINT *savepoint_name*	Undo previous tasks, or undo previous tasks back to the time point specified in *savepoint_name*
SAVEPOINT *savepoint_name*	Set a marker in a transaction
SET TRANSACTION	Begin a read-only or read-write transaction, or change the ISOLATION LEVEL settings of a transaction

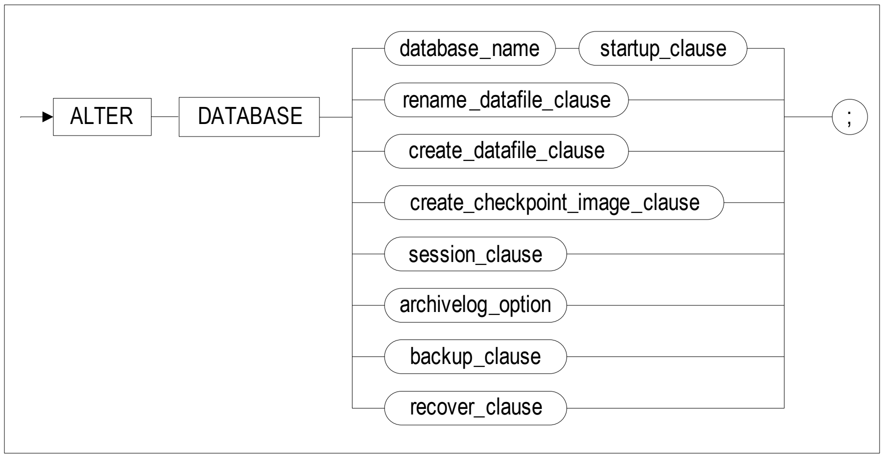
Session control statements and transaction control statements only affect individual transactions; they do not affect other tasks.

2 Data Definition Statements

2.1 ALTER DATABASE

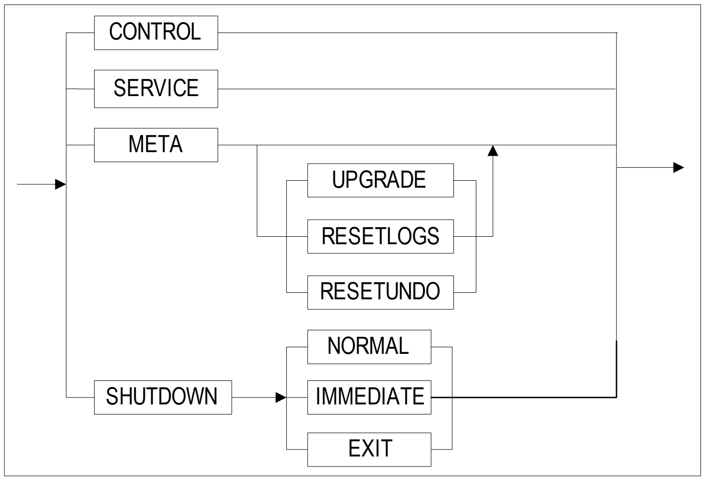
2.1.1 Syntax

2.1.1.1 alter_database ::=

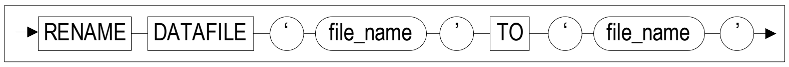


startup_clause ::=, *rename_datafile_clause ::=*, *create_datafile_clause ::=*, *create_checkpoint_image_clause ::=*, *session_clause ::=*, *archivelog_option ::=*, *backup_clause ::=*, *recover_clause ::=*

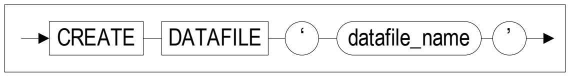
2.1.1.2 startup_clause ::=



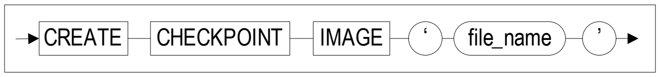
2.1.1.3 rename_datafile_clause ::=



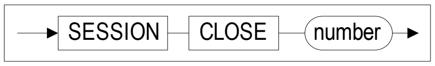
2.1.1.4 create_datafile_clause ::=



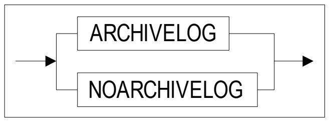
2.1.1.5 create_checkpoint_image_clause ::=



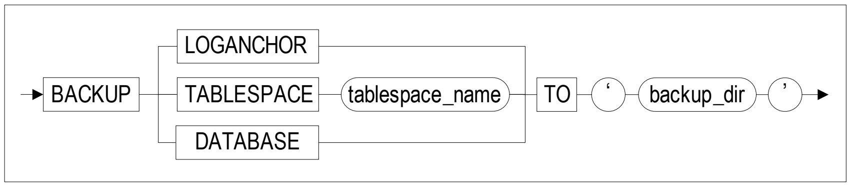
2.1.1.6 session_clause ::=



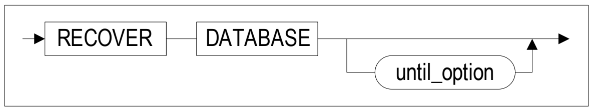
2.1.1.7 archivelog_option ::=



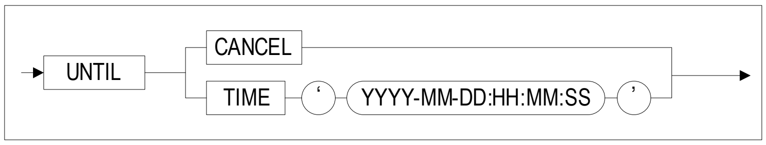
2.1.1.8 backup_clause ::=



2.1.1.9 recover_clause ::=



2.1.1.10 until_option ::=



2.1.2 Prerequisites

The ALTER DATABASE statement can only be executed in a startup phase preceding the SERVICE phase by the SYS user after connecting in -sysdba mode. The exception is when using the SESSION CLOSE option, in which case it is not necessary to connect in -sysdba mode in order to execute this statement.

2.1.3 Description

This statement is used to modify, maintain, or restore an existing database.

2.1.3.1 database_name

This element is used to specify the name of the database to be managed.

2.1.3.2 startup_clauses

This element is used to specify the name of the startup phase in which to start up ALTIBASE HDB.

For reference, this command can also be used to change the status of the database from an earlier startup phase to a later startup phase in the order described below.

CONTROL

This option is used to start the database in the CONTROL phase. When the database is started in this phase, database media recovery can be performed. Tablespaces can also be discarded in this phase. For more information about the various ALTIBASE HDB startup phases, please refer to the *Administrator's Manual*.

To proceed to the phase immediately following the CONTROL phase, the META phase, execute this statement as follows:

```
ALTER DATABASE database_name META;
```

META

This option is used to start the database in the META phase. While proceeding to this phase from the previous phase, the CONTROL phase, the database meta data are loaded. To proceed to the next phase, execute the following statement:

```
ALTER DATABASE database_name SERVICE;
```

SERVICE

This option is used to start the database in the SERVICE phase. When the database is started in this phase, all memory and disk tables are loaded, and extended services such as replication and SNMP can be started. If the database can be successfully started in this phase, it means that any required recovery has been performed, and that the system is in a state in which service is being provided normally.

META UPGRADE

This option is used to start the database in the META UPGRADE phase. When the database is started up to this phase, the meta data are upgraded, meaning that all recovery-related tasks have been completed.

To proceed to the next phase, execute the following statement:

ALTER DATABASE *database_name* SERVICE;

META RESETLOGS

This task is required to ensure normal startup of the server after incomplete recovery has been performed in the CONTROL phase. Logs that are no longer necessary once incomplete recovery has been performed are deleted while proceeding to this phase.

To proceed to the next phase, execute the following statement:

ALTER DATABASE *database_name* SERVICE;

META RESETUNDO

In this phase, the SYS_TBS_DISK_UNDO tablespace is initialized, but the size of the tablespace file is not changed. Before executing this statement, check the integrity of the database, ensure that disk garbage collection has been performed, and shut down the database normally.

SHUTDOWN NORMAL

The server waits until all client connections have been disconnected normally before shutting down normally.

SHUTDOWN IMMEDIATE

The server forcibly disconnects all clients and then shuts down normally.

SHUTDOWN EXIT

This option is used to kill the ALTIBASE HDB server forcibly. 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.

The `shutdown normal` and `shutdown immediate` commands can be executed only during the service phase; in the other startup phases, only `shutdown exit` is available.

For more information about ALTIBASE HDB startup phases and about starting up the database, please refer to the *Administrator's Manual*.

2.1.3.3 RENAME DATAFILE

This command is used to change a reference to a data file within ALTIBASE HDB so that it points to a data file that has a different name or is located in a different directory. The data file specified in TO '*datafile_path*' must exist. This command can only be executed in the CONTROL phase.

datafile_path must be an absolute path.

For reference, the 2.8 ALTER TABLESPACE statement is used to move memory tablespace checkpoint image files.

2.1.3.4 CREATE DATAFILE

When a disk data file has been lost, this command is used to create a data file with reference to the log anchor data. After this statement is executed, complete media recovery is performed to restore the data file. This statement is available only during the CONTROL phase.

datafile_path, which is where the data file will be created, must be an absolute path. For reference, the 2.8 ALTER TABLESPACE statement is used to create memory tablespace checkpoint image files.

2.1.3.5 CREATE CHECKPOINT IMAGE

When a memory checkpoint image file has been lost, this command is used to create a checkpoint image file with reference to the log anchor data. After this statement is executed, complete media recovery can be performed in order to restore the memory checkpoint image file.

Because the checkpoint image file is created in the checkpoint path specified for memory tablespaces, it is not necessary to specify the path; only the name of the file to be created need be provided.

This statement is available only during the CONTROL phase.

<Query> Recreate the lost checkpoint image file *MEM-TBS-1*.

```
iSQL> ALTER DATABASE CREATE CHECKPOINT IMAGE 'MEM-TBS-1';
```

2.1.3.6 SESSION CLOSE

This statement is used to forcibly terminate a session. Specify the session_id of the session to terminate after SESSION CLOSE. When this statement is executed, any transactions associated with the session are rolled back.

Note: It is impossible to terminate a session immediately if the session is waiting to obtain a lock.

2.1.3.7 archivelog_option

The database can be switched between archivelog mode and noarchivelog mode in the CONTROL phase.

2.1.3.8 BACKUP LOGANCHOR

When the database is operating in archivelog mode, this statement is used to back up log anchor files online without interrupting service.

2.1.3.9 BACKUP TABLESPACE

When the database is operating in archivelog mode, this statement is used to back up the specified tablespace to the backup directory without interrupting service.

2.1.3.10 BACKUP DATABASE

When the database is operating in archivelog mode, this statement is used to back up all memory and disk tablespaces and log anchor files without interrupting service.

2.1.3.11 RECOVER DATABASE

This is used to perform complete media recovery. The log files in the archive log directory are read for use in recovering the data files that were affected by media errors to the current point in time.

2.1.3.12 RECOVER DATABASE UNTIL TIME

This is used to perform incomplete media recovery to a specified point in time. The log files in the archive log directory are read for use in recovering the data files that were affected by media errors to the specified point in time.

2.1.3.13 RECOVER DATABASE UNTIL CANCEL

This is used to perform incomplete media recovery to the most recent point in time at which the logs in archive log files are valid. The log files in the archive log directory are read for use in recovering the data files that were affected by media errors to that point in time.

2.1.3.14 AltiLinker_clause

This is used to shut down or restart the AltiLinker process, which mediates data exchange between Database Link and a remote server.

LINKER START

This is used to start the AltiLinker process. Note that there must not already be any currently running AltiLinker processes.

LINKER STOP

This statement is used to terminate the AltiLinker process. However, there must not be any transactions using Database Link at the time of termination. If there are any transactions using Database Link, the attempt to terminate Altilinker will fail.

2.1.4 Examples

<Query> Start up a database called *mydb* so that it provides service normally.

```
iSQL> ALTER DATABASE mydb SERVICE;
```

<Query> Switch the database to archivelog mode.

```
iSQL> ALTER DATABASE ARCHIVELOG;
```

<Query> Start up the database after incomplete recovery.

```
iSQL> ALTER DATABASE mydb META RESETLOGS;
```

<Query> Initialize the SYS_TBS_DISK_UNDO tablespace.

```
iSQL> ALTER DATABASE mydb META RESETUNDO;
```

<Query> Back up the SYS_TBS_DISK_DATA tablespace to the `/altibase_backup` directory.

```
iSQL> ALTER DATABASE BACKUP TABLESPACE
  SYS_TBS_DISK_DATA TO '/altibase_backup/';
```

<Query> Restore the database to Feb. 16, 2008, 12:00 PM from a previous backup.

```
iSQL> ALTER DATABASE RECOVER DATABASE UNTIL TIME '2008-02-16:12:00:00';
```

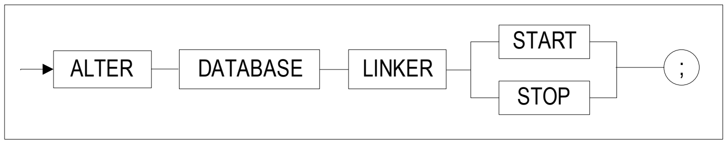
<Query> Restore the database from a previous backup to reflect the entire contents of the log file #20000, which immediately precedes the log file #20001, which was lost.

```
iSQL> ALTER DATABASE RECOVER DATABASE UNTIL CANCEL;
```

2.2 ALTER DATABASE LINKER

2.2.1 Syntax

2.2.1.1 altiLinker_clause ::=



2.2.2 Description

2.2.2.1 AltiLinker_clause

This is used to shut down or restart the AltiLinker process, which mediates data exchange between Database Link and a remote server.

LINKER START

This is used to start the AltiLinker process. Note that there must not already be any currently running AltiLinker processes.

LINKER STOP

This statement is used to terminate the AltiLinker process. However, there must not be any transactions using Database Link at the time of termination. If there are any transactions using Database Link, the attempt to terminate Altilinker will fail.

2.2.3 Examples

<Query> Start up AltiLinker for Database Link.

```
iSQL> ALTER DATABASE LINKER START;
```

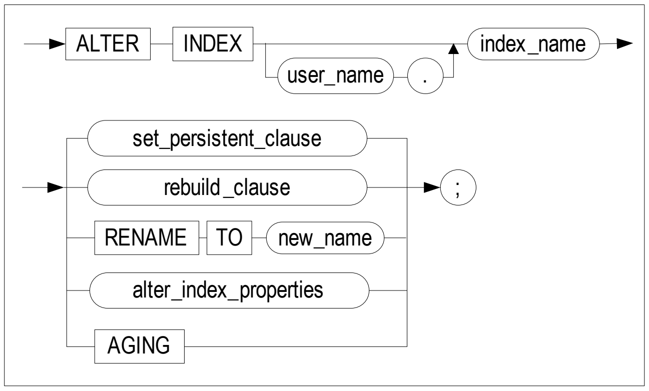
<Query> Stop AltiLinker.

```
iSQL> ALTER DATABASE LINKER STOP;
```

2.3 ALTER INDEX

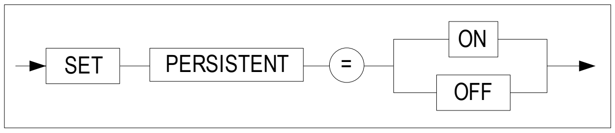
2.3.1 Syntax

2.3.1.1 alter_index ::=

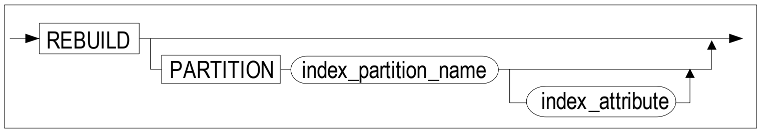


set_persistent_clause ::=, *rebuild_clause ::=*, *alter_index_properties::=*

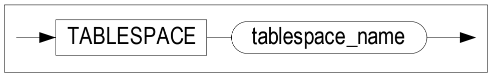
2.3.1.2 set_persistent_clause ::=



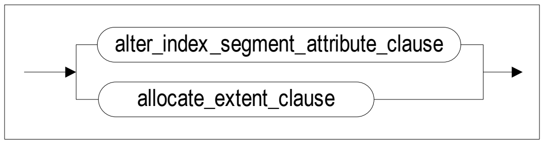
2.3.1.3 rebuild_clause ::=



2.3.1.4 index_attribute ::=

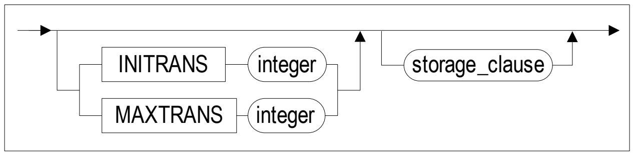


2.3.1.5 alter_index_properties::=

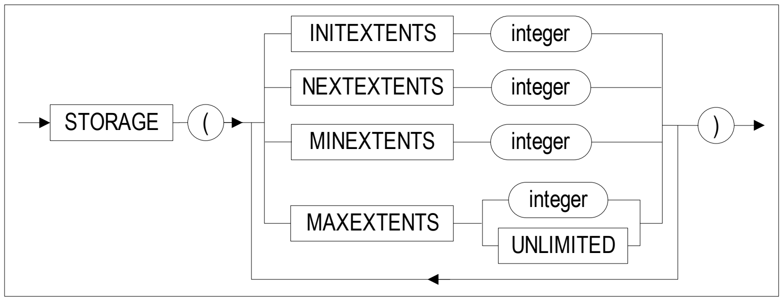


alter_index_segment_attribute_clause::=, *allocate_extent_clause::=*

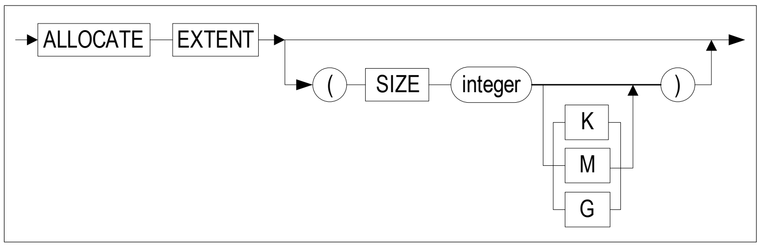
2.3.1.6 alter_index_segment_attribute_clause::=



2.3.1.7 storage_clause::=



2.3.1.8 allocate_extent_clause::=



2.3.2 Prerequisites

Only the SYS user, the owner of the schema containing the index, and users having the ALTER ANY INDEX system privilege can execute the ALTER INDEX statement.

2.3.3 Description

The ALTER INDEX statement is used to change or rebuild an existing index.

2.3.3.1 user_name

This is used to specify the name of the owner of the index to be altered.

If omitted, ALTIBASE HDB will assume that the index belongs to the schema of the user connected via the current session.

2.3.3.2 index_name

This is used to specify the name of the index to be altered.

2.3.3.3 SET PERSISTENT clause

This statement is used to change a non-persistent index into a persistent index, or to change a persistent index into a non-persistent index. For more information about persistent indexes, please refer to the description of the CREATE INDEX statement.

2.3.3.4 rebuild_clause

This statement is used to rebuild an existing index or one of its partitions.

index_attribute

This is used to specify the tablespace where the rebuilt index partition will be stored.

2.3.3.5 RENAME

This is used to specify the name of index to be changed.

2.3.3.6 AGING

This is used to record a transaction commit SCN in an index page and delete old versions of nodes. This statement is only available for disk-based indexes.

2.3.3.7 alter_index_segment_attribute_clause

- INITRANS Clause

  This is used to change the initial number of TTS (Touched Transaction Slots).

- MAXTRANS Clause

  This is used to change the maximum number of TTS (Touched Transaction Slots).

2.3.3.8 storage_clause

This is used to set parameters for managing extents in segments.

- INITEXTENTS Clause

The INITEXTENTS parameter in the ALTER INDEX statement is ignored.

- NEXTEXTENTS Clause

  This determines the number of extents that are added to the segment every time the segment is increased in size.

- MINEXTENTS Clause

  This sets the minimum number of extents in a segment.

- MAXEXTENTS Clause

  This sets the maximum number of extents in a segment.

2.3.3.9 allocate_extent_clause

This is used to explicitly allocate extents to the index segment. Set SIZE to the total size of extents that are to be added to the index segment. If the disk tablespace comprises several data files, extents are distributed equally between them.

2.3.4 Restrictions

Indexes based on replication target tables cannot be altered. The only exceptions to this are the 'ALTER INDEX SET PERSISTENT = ON/OFF' and 'ALTER INDEX REBUILD PARTITION' statements, which can be executed on replication target tables.

2.3.5 Examples

2.3.5.1 Changing Index Persistency

<Query> Change the index *emp_idx1* into a persistent index.

```
iSQL> ALTER INDEX emp_idx1 Set PERSISTENT = ON;
Alter success.
```

<Query> Change the index *const1* into a persistent index.

```
iSQL> ALTER INDEX const1 SET PERSISTENT = ON;
Alter success.
```

2.3.5.2 Rebuilding an Index Partition

<Query> Rebuild the index partition *IDX_P5* in the tablespace *TBS1*.

```
iSQL> ALTER INDEX IDX1 REBUILD PARTITION IDX_P5 TABLESPACE TBS1;
```

2.3.5.3 Changing the Index Name

<Query> Change the name of *emp_idx1* to *emp_idx2*.

```
iSQL> ALTER INDEX emp_idx1 RENAME TO emp_idx2;
```

2.3.5.4 Allocating extents to index

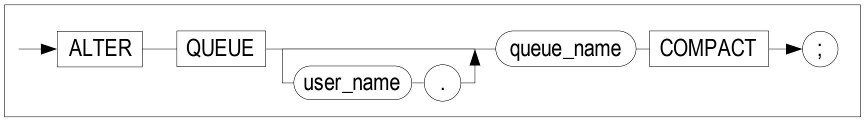
<Query> Allocate extents totaling 10MB in size to the index *LOCAL_IDX*, which is located in a disk tablespace.

```
iSQL> ALTER INDEX felt_idx ALLOCATE EXTENT ( SIZE 10M );
Alter success.
```

2.4 ALTER QUEUE

2.4.1 Syntax

2.4.1.1 alter_queue ::=



2.4.2 Description

This alters the definition of a queue.

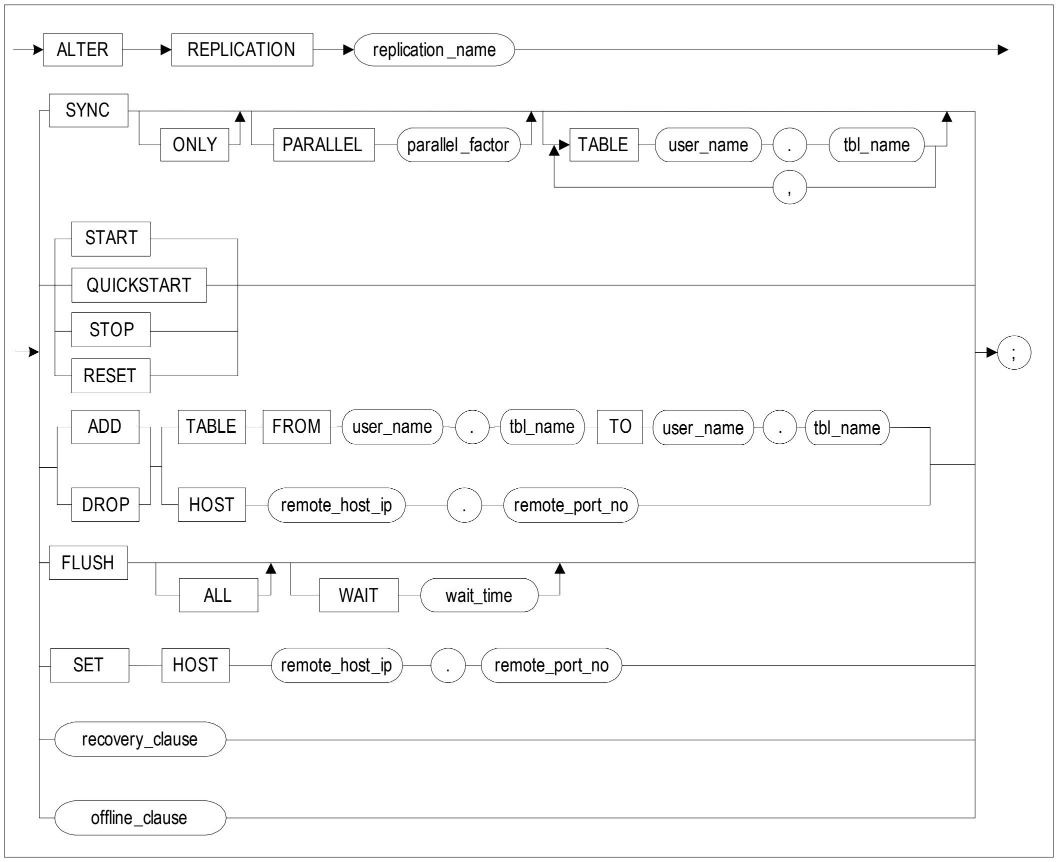
2.4.2.1 COMPACT

This command returns empty pages to the tablespace in which the queue is located. No data are actually moved.

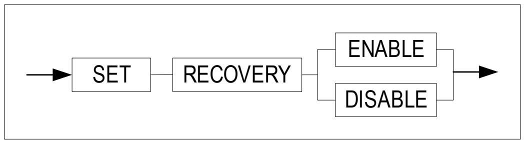
2.5 ALTER REPLICATION

2.5.1 Syntax

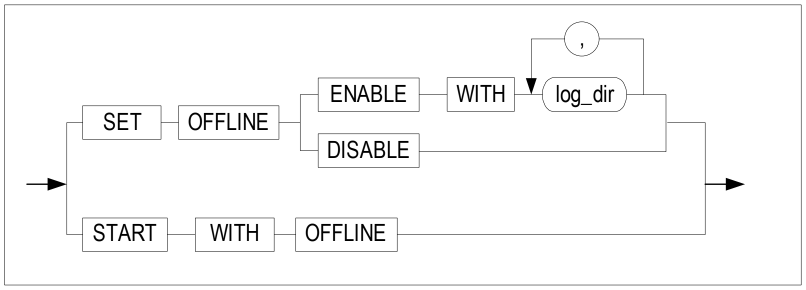
2.5.1.1 alter_replication ::=



2.5.1.2 recovery_clause ::=



2.5.1.3 offline_clause ::=



2.5.2 Description

After a replication object has been created using the CREATE REPLICATION statement, its operation is controlled in various ways using this statement. For more information about replication, please refer to the *Replication Manual*.

2.5.2.1 replication_name

This is used to specify the name of the replication object.

2.5.2.2 SYNC

This is used to send all data in replication target tables on the local server to the corresponding tables on the remote server and start replication.

SYNC ONLY

This is used to send all data in replication target tables on the local server to the corresponding tables on the remote server. It does not initiate a replication Sender thread.

PARALLEL parallel_factor

The *parallel_factor* option does not need to be specified. If it is omitted, it is taken as 1 by default. The maximum value of *parallel_factor* is the number of CPUs * 2. It cannot be set higher than this, even if a higher maximum value is specified. If 0 (zero) or a negative value is specified, an error will be returned.

TABLE user_name.table_name

This is used to specify which of the replication tables on the local server to synchronize using the SYNC parameter. If this clause is specified, replication starts from the time point up to which replication was last performed after the specified tables are synchronized. If the TABLE clause is omitted, replication starts from the current position in the logs after all of the replication tables are synchronized.

2.5.2.3 START

Replication starts from the time point up to which replication was last performed.

2.5.2.4 QUICKSTART

Replication starts from the current point in time.

2.5.2.5 START/QUICKSTART RETRY

When STARTing or QUICKSTARTing replication using the RETRY option, even if handshaking fails, a Sender thread is created on the local server. When handshaking between the local server and the remote server subsequently succeeds, replication starts.

When this option is used, `iSQL` will report handshaking success even if the first handshake attempt fails. Therefore, the user has to check the result of execution of this command by checking the trace logs or performance views.

When starting replication without the RETRY option, if the first handshaking attempt fails, an error is raised and execution stops. Note that the use of the RETRY option is not supported in EAGER mode.

2.5.2.6 STOP

Stops the replication.

2.5.2.7 RESET

This command resets replication information (such as the restart SN). It can only be executed while replication is stopped. It is an alternative to executing the DROP REPLICATION command followed by the CREATE REPLICATION command.

2.5.2.8 ADD TABLE

This is used to add a table to a replication object. A table can be added to a replication object only when replication is stopped.

where_clause

The use of a condition clause during replication specifies that only the logs corresponding to records that meet conditions will be used as replication targets. This clause takes the form WHERE user_name.table_name.column_name {< | > | <> | >= | <= | = | !=} value [{AND | OR} ...]. Note that this clause can be used only with the ALTER REPLICATION ... ADD TABLE command.

For more information, please refer to the description of the where_clause ::= in the SELECT statement.

user_name

This is the name of the owner of a replication target table.

tbl_name

This is the name of the replication target table.

2.5.2.9 DROP TABLE

This is used to remove a table from a replication object. A table can be removed from a replication object only when replication is stopped.

2.5.2.10 FLUSH

This is used to instruct the current session to wait, for the number of seconds specified in *wait_time*, for the replication Sender thread to send information about changed data, up to the current log (the log at the time the FLUSH statement is executed), to the other server. If this is used together with the ALL option, the current session is instructed to wait until the information about changed data in the most recent log, rather than the current log, has been transferred to the other server.

2.5.2.11 SET HOST

This sets a particular host as the current one. It can be changed while replication is stopped.

2.5.3 Considerations

There are several points that users working with replication must keep in mind before using replication. Before executing an ALTER REPLICATION command, please refer to the *Replication Manual*.

2.5.4 Examples

Start execution of the *repl1* replication object

- <Query> Send the data on the local server data to the remote server, and start replication.

```
iSQL> ALTER REPLICATION rep1 SYNC;
Alter success.
```

- <Query> Start the *rep1* replication from the time point at which replication was most recently executed:

```
iSQL> ALTER REPLICATION rep1 START;
Alter success.
```

- <Query> Start the replication.

```
iSQL> ALTER REPLICATION rep1 QUICKSTART;
Alter success.
```

Remove the replication target table *employees* from a replication object named *rep1*.

```
iSQL> ALTER REPLICATION rep1 STOP;
Alter success.
iSQL> ALTER REPLICATION rep1 DROP TABLE FROM sys.employees TO sys.employees;
Alter success.
```

Add the table *employees* to the replication object *rep1*.

```
iSQL> ALTER REPLICATION rep1 STOP;
Alter success.
iSQL> ALTER REPLICATION rep1 ADD TABLE FROM sys.employees TO sys.employees;
Alter success.
```

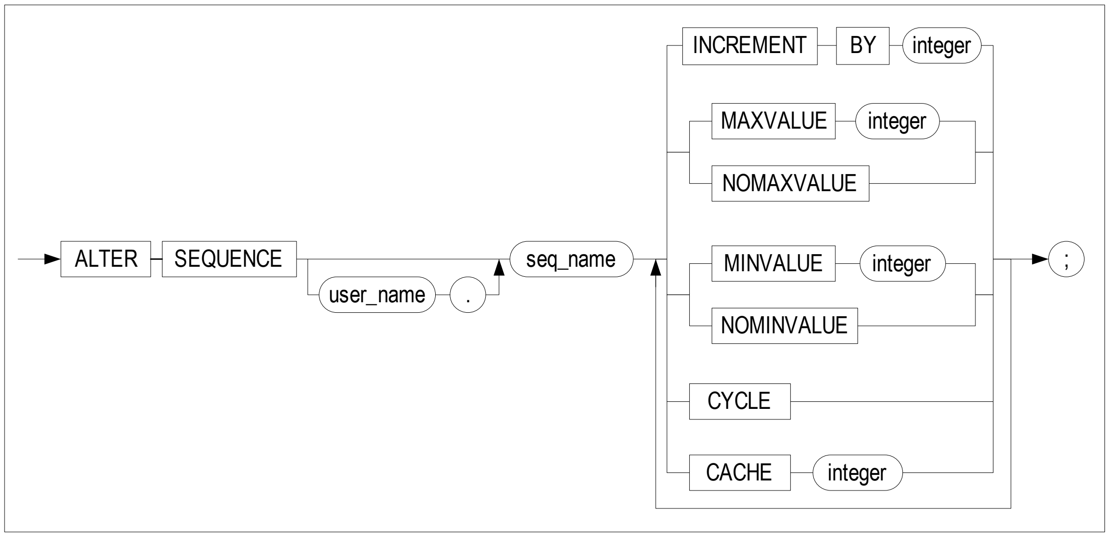
Stop the replication object *rep1*.

```
iSQL> ALTER REPLICATION rep1 STOP;
Alter success.
```

2.6 ALTER SEQUENCE

2.6.1 Syntax

2.6.1.1 alter_sequence ::=



2.6.2 Prerequisites

Only the SYS user, the owner of the schema to which the sequence belongs, users having the ALTER object privilege on the sequence, and users having the ALTER ANY SEQUENCE system privilege can execute this statement.

2.6.3 Description

After a sequence has been created using the CREATE SEQUENCE statement, this statement is used to change the definition of the sequence, including whether values are cached, the increment, minimum and maximum values, and the behavior of the sequence. For more information, please refer to the description of the CREATE SEQUENCE statement.

2.6.3.1 user_name

This is used to specify the name of the owner of the sequence to be changed. If omitted, ALTIBASE HDB will assume that the sequence belongs to the schema of the user connected via the current session.

2.6.3.2 seq_name

This is used to specify the name of the sequence to be altered.

2.6.3.3 INCREMENT BY

This is used to specify the interval between sequence numbers.

2.6.3.4 MAXVALUE

This is used to specify the maximum value that the sequence can generate.

2.6.3.5 MINVALUE

This is used to specify the minimum value of the sequence.

2.6.3.6 CYCLE

This is used to allow a sequence to continue to output values after it reaches the limit specified by MAXVALUE or MINVALUE. In the case of an ascending sequence, the minimum value will be output once its maximum value has been reached, whereas the opposite is true for a descending sequence: once it reaches its minimum value, it outputs its maximum value.

2.6.3.7 CACHE

This is used to specify the number of sequence values that are cached in memory so that they can be accessed more quickly. The first time the sequence is referred to, the cache is populated, and whenever values are subsequently requested from the sequence, they are retrieved from the cached values. 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. The number of sequence values that are created and cached at this time is set using the CREATE SEQUENCE statement.

2.6.4 Restriction

When changing the definition of an existing sequence, the START WITH clause cannot be used, because the sequence has already been created.

For more information about sequences, please refer to the description of the CREATE SEQUENCE statement.

2.6.5 Examples

- <Query> Change the definition of a sequence so that the minimum and maximum values are 0 and 100, respectively, and so that the increment value is 1.

```
iSQL> ALTER SEQUENCE seq1
 INCREMENT BY 1
 MINVALUE 0
 MAXVALUE 100;
Alter success.
```

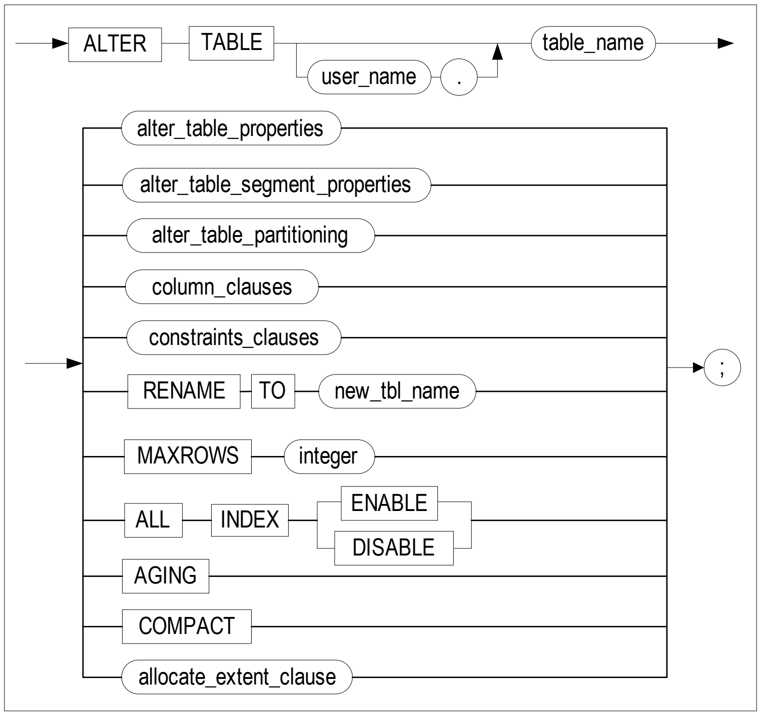
- <Query> Change the definition of a sequence so that its minimum and maximum values are unlimited.

```
iSQL> ALTER SEQUENCE seq2
 NOMAXVALUE
 NOMINVALUE;
Alter success.
```

2.7 ALTER TABLE

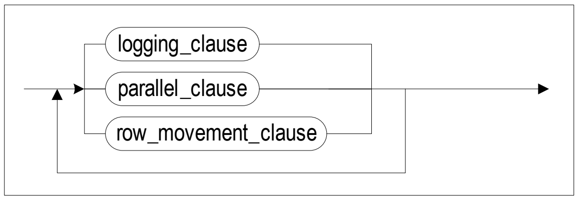
2.7.1 Syntax

2.7.1.1 alter_table::=

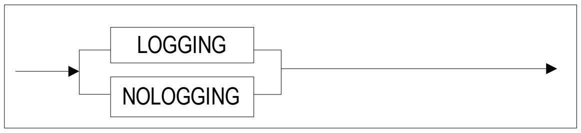


alter_table_properties::=, *alter_table_segment_properties::=*, *alter_table_partitioning::=*, *column_clauses::=*, *constraints_clauses::=*, *allocate_extent_clause::=*

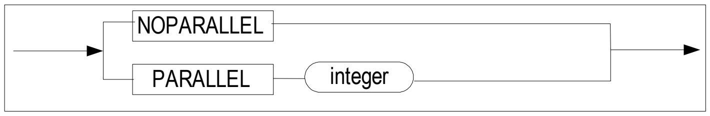
2.7.1.2 alter_table_properties::=



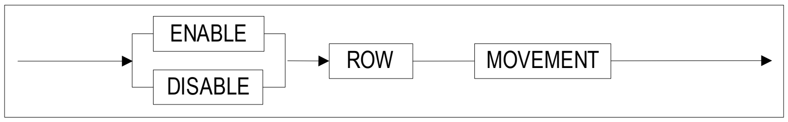
2.7.1.3 logging_clause::=



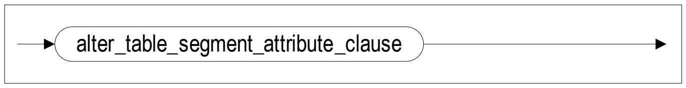
2.7.1.4 parallel_clause::=



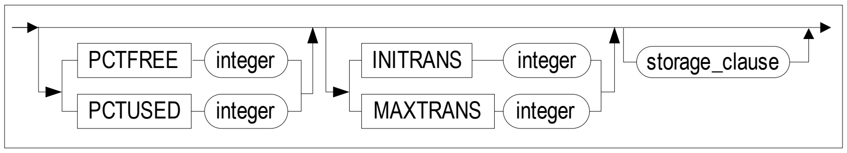
2.7.1.5 row_movement_clause::=



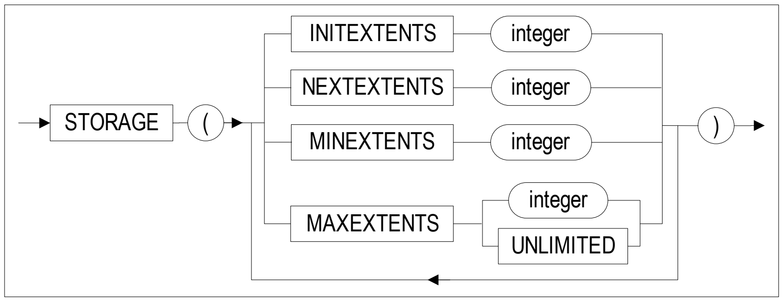
2.7.1.6 alter_table_segment_properties::=



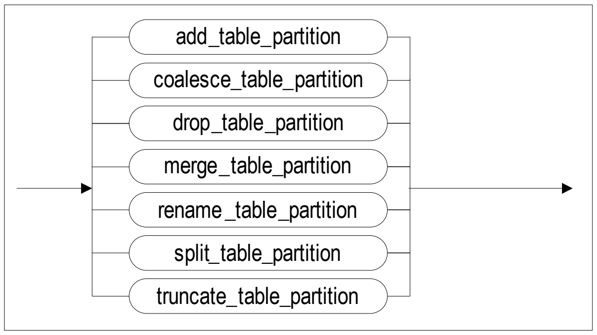
2.7.1.7 alter_table_segment_attribute_clause::=



2.7.1.8 storage_clause::=

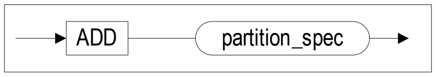


2.7.1.9 alter_table_partitioning::=



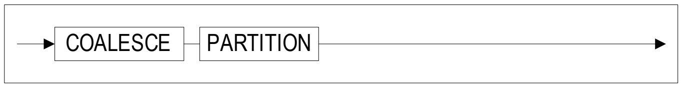
add_table_partition ::=, *coalesce_table_partition ::=*, *drop_table_partition ::=*, *merge_table_partition ::=*, *rename_table_partition ::=*, *split_table_partition ::=*, *truncate_table_partition ::=*

2.7.1.10 add_table_partition ::=

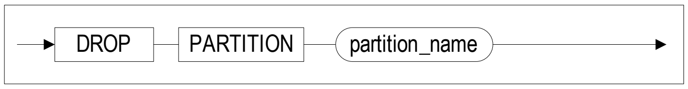


partition_spec ::=

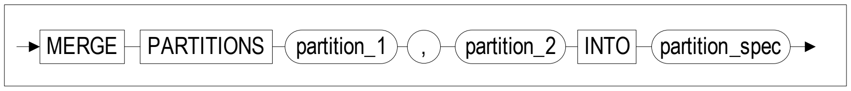
2.7.1.11 coalesce_table_partition ::=



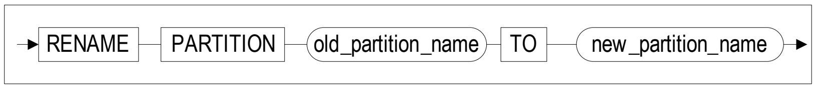
2.7.1.12 drop_table_partition ::=



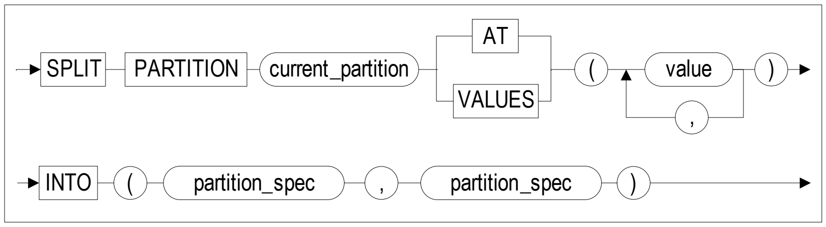
2.7.1.13 merge_table_partition ::=



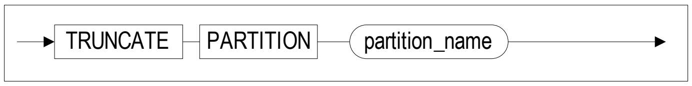
2.7.1.14 rename_table_partition ::=



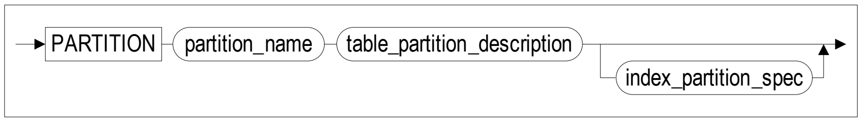
2.7.1.15 split_table_partition ::=



2.7.1.16 truncate_table_partition ::=

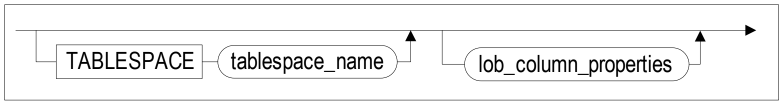


2.7.1.17 partition_spec ::=



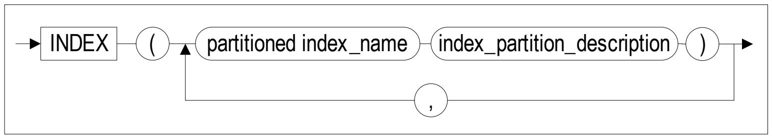
table_partition_description ::=, *index_partition_spec ::=*

2.7.1.18 table_partition_description ::=

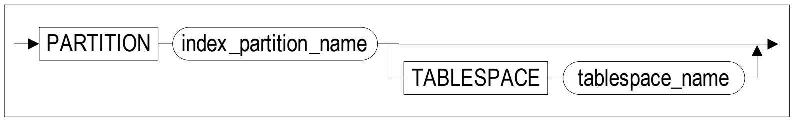


lob_column_properties ::=

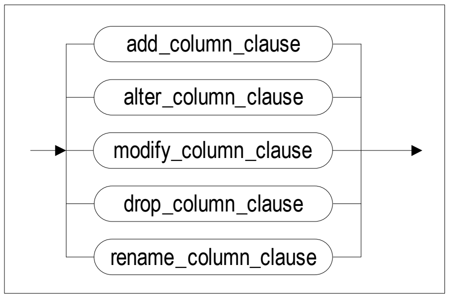
2.7.1.19 index_partition_spec ::=



2.7.1.20 index_partition_description ::=

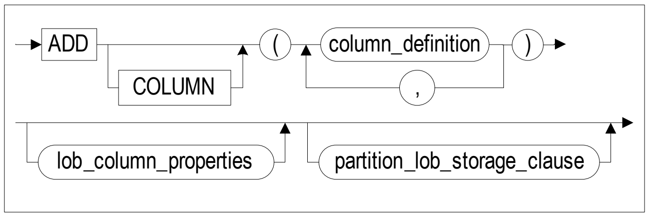


2.7.1.21 column_clauses::=



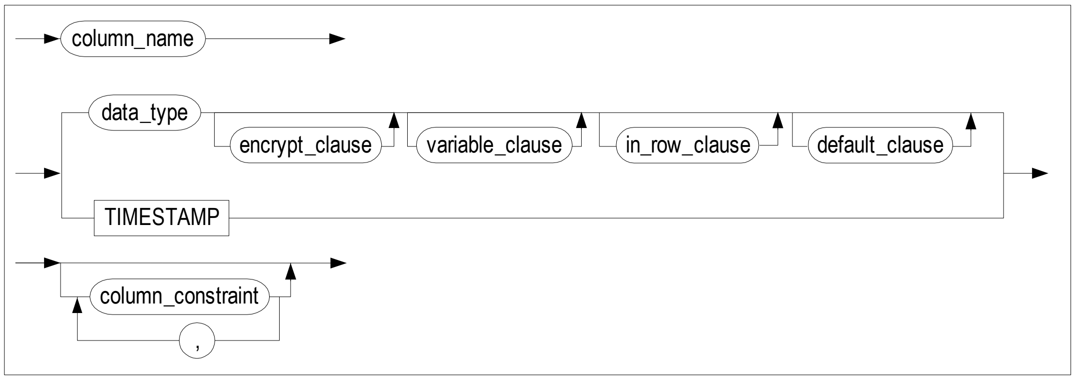
add_column_clauses::=, *alter_column_clause ::=*, *modify_column_clause::=*, *drop_column_clause::=*, *rename_column_clause::=*

2.7.1.22 add_column_clauses::=



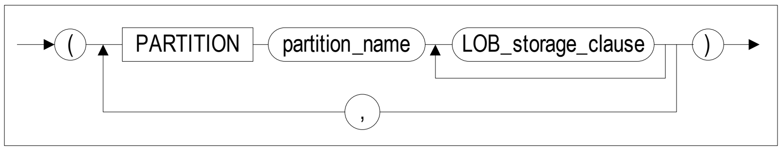
column_definition ::=, *lob_column_properties ::=*, *partition_lob_storage_clause ::=*

2.7.1.23 column_definition ::=



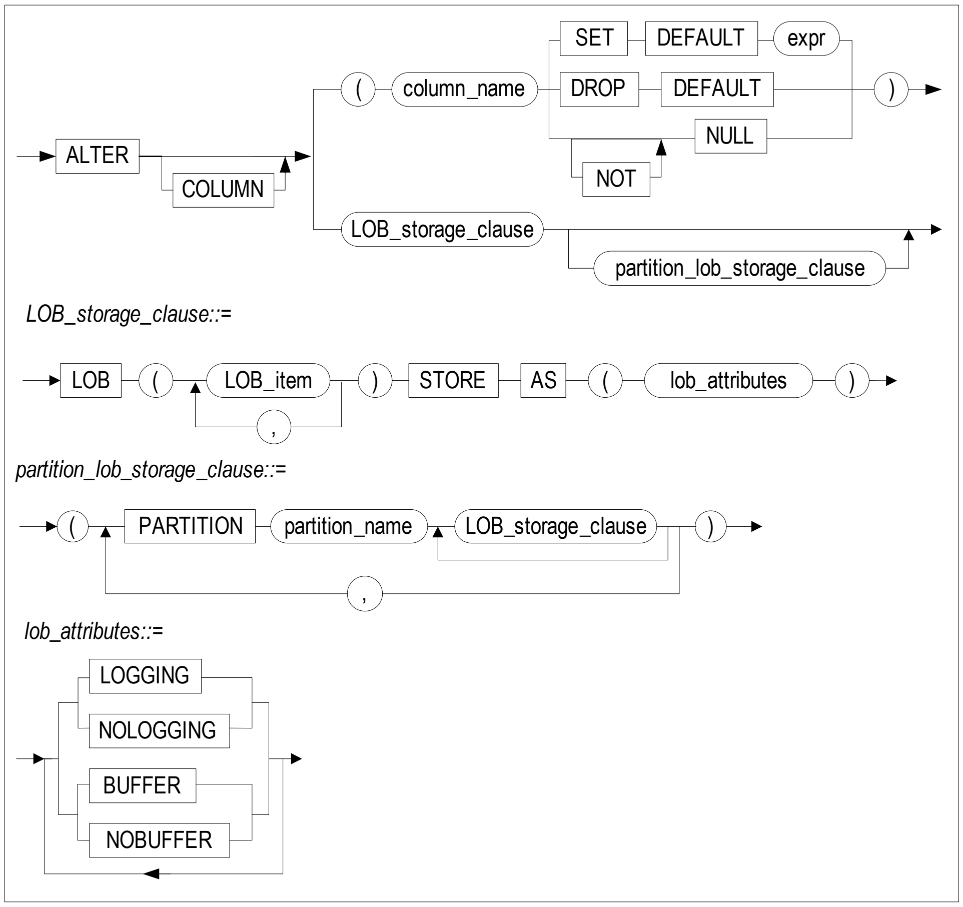
column_constraint ::=

2.7.1.24 partition_lob_storage_clause ::=

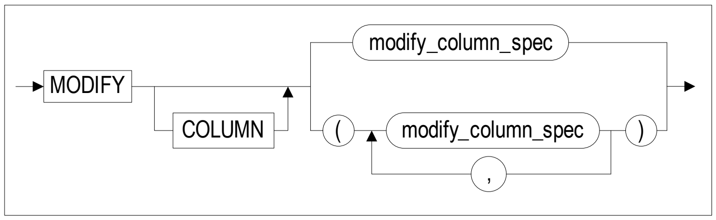


LOB_storage_clause ::=

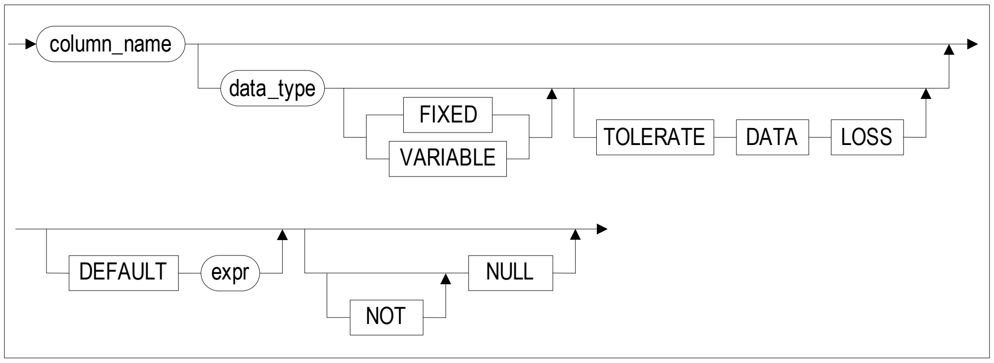
2.7.1.25 alter_column_clause ::=



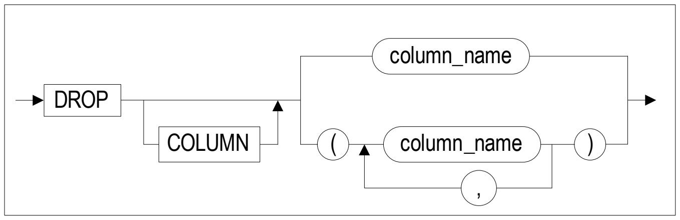
2.7.1.26 modify_column_clause::=



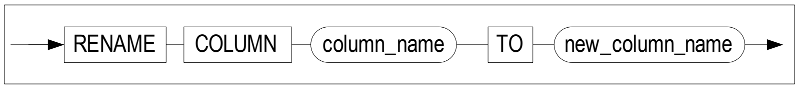
2.7.1.27 modify_column_spec::=



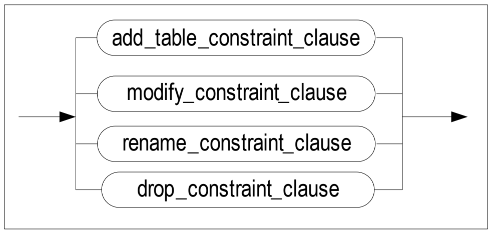
2.7.1.28 drop_column_clause::=



2.7.1.29 rename_column_clause::=

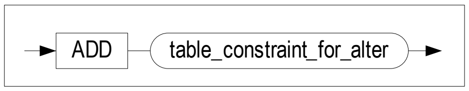


2.7.1.30 constraints_clauses::=



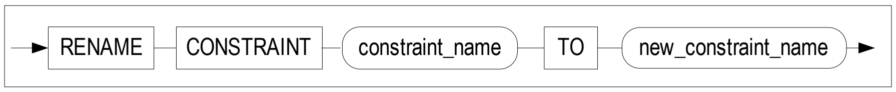
add_table_constraint_clause ::=, *modify_constraint_clause::=*, *rename_constraint_clause ::=*, *drop_constraint_clause ::=*

2.7.1.31 add_table_constraint_clause ::=

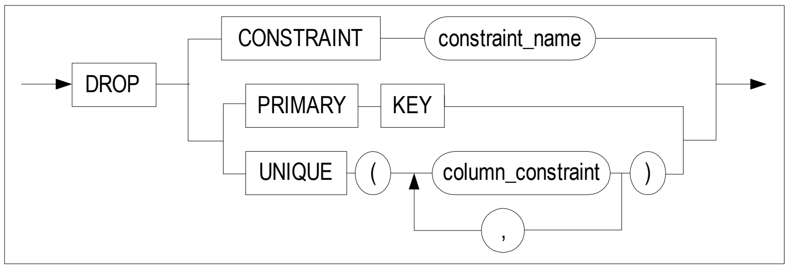


table_constraint_for_alter::=

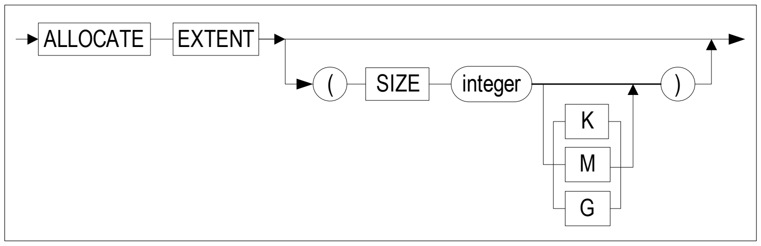
2.7.1.32 rename_constraint_clause ::=



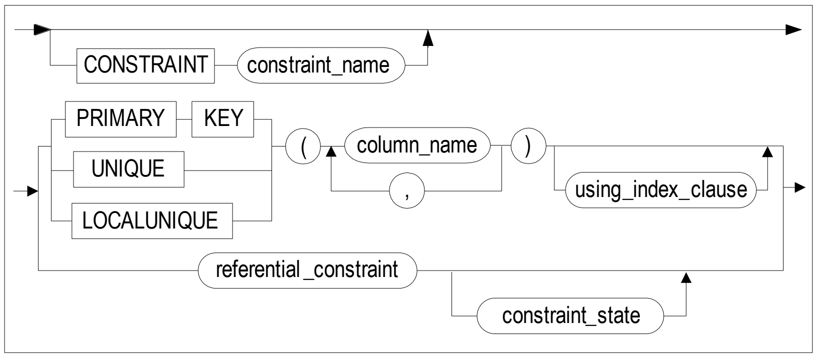
2.7.1.33 drop_constraint_clause ::=



2.7.1.34 allocate_extent_clause::=

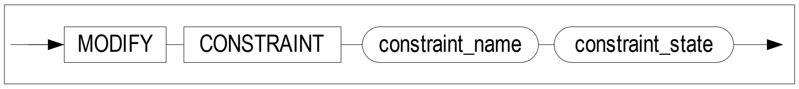


2.7.1.35 table_constraint_for_alter::=

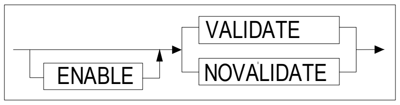


using_index_clause ::=, *referential_constraint ::=*, *constraint_state::=*

2.7.1.36 modify_constraint_clause::=



2.7.1.37 constraint_state::=



2.7.2 Prerequisites

Only the SYS user, the owner of the schema to which the table belongs, users having the ALTER object privilege for the table, and users having the ALTER ANY TABLE system privilege can alter table definitions.

2.7.3 Description

The ALTER TABLE statement is an SQL statement that is used to change the definition of a specified table. Execution of this statement modifies the meta information for the table.

The ALTER TABLE statement can also be used to change the attributes of partitioned tables. The clauses related to partitioned tables are the ADD, COALESCE, DROP, MERGE, RENAME, SPLIT and TRUNCATE clauses.

The following table shows whether each statement can be used with range-, list- and hash-partitioned tables.

Table 2-1 Operations Supported for Use with Different Partitioning Methods

	Range-Partitioned Tables	List-Partitioned Tables	Hash-Partitioned Tables
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

2.7.3.1 user_name

This is used to specify the name of the owner of the table to be altered. If omitted, ALTIBASE HDB will assume that the table belongs to the schema of the user connected via the current session.

2.7.3.2 tbl_name

This is used to specify the name of the table to alter.

2.7.3.3 COMPACT

This indicates whether empty pages that do not contain any data are to be returned to the tablespace. ALTIBASE HDB doesn't actually move any data when compacting tables. This command is supported for use only with memory tables and volatile tables.

2.7.3.4 alter_table_segment_attribute_clause

PCTFREE Clause

This is used to change the percentage of free space that is reserved for future use when updating records that have already been saved in pages.

An ALTER TABLE statement containing the *alter_table_segment_attribute_clause*, which is used to change segment attributes, can be executed while ALTIBASE HDB is running. However, the changes will not be immediately applied in all of the pages in the segment; rather, each table page will be changed individually the next time the page is accessed.

PCTUSED Clause

This is used to change the threshold below which the amount of used space in a page must decrease in order for the page to return to the state in which records can be inserted.

INITRANS Clause

This is used to change the initial number of TTS (Touched Transaction Slots).

MAXTRANS Clause

This is used to change the maximum number of TTS (Touched Transaction Slots).

2.7.3.5 storage_clause

This is used to set parameters for managing extents in segments.

INITEXTENTS Clause

The INITEXTENTS parameter in the ALTER TABLE statement is ignored.

NEXTEXTENTS Clause

This determines the number of extents that are added to the segment every time the segment is increased in size.

MINEXTENTS Clause

This sets the minimum number of extents in a segment.

MAXEXTENTS Clause

This sets the maximum number of extents in a segment.

2.7.3.6 add_table_partition

This clause is used to add a partition to a partitioned table. It can be used with hash-partitioned tables. If local indexes have already been defined for the existing partitions, then another local index is automatically created for the newly added partition. When the index is created, the name of the index is automatically determined by the system, and the index is stored in the same tablespace as the new partition.

2.7.3.7 partition_spec

This is used to specify the name of the partition and the tablespace in which the partition will be stored. The name of the tablespace can be omitted. If it is omitted, the data pertaining the partition are stored in the tablespace in which the table is located. Furthermore, if an index has been defined for the table, the tablespace in which the index partition is stored can be specified.

2.7.3.8 table_partition_description

This is used to specify the tablespace in which each partition is stored and the attributes of LOB columns, if any.

If the tablespace clause is omitted, the data are stored in the default tablespace for the table. In the same way, if the tablespace statement for a LOB column is omitted, the LOB column data are stored in the tablespace in which the partition is stored. For more information on using tablespaces, please refer to the explanation of table_partition_description in 2.21 CREATE TABLE.

2.7.3.9 index_partition_spec

When the ALTER TABLE statement is executed with the SPLIT PARTITION, MERGE PARTITION or ADD PARTITION clause, a new partition is created. At this time, this clause can be used to specify the tablespace in which to store the index partition that is created along with the table partition.

2.7.3.10 coalesce_table_partition

This can only be used with hash partitions. It is used to coalesce hash partitions and reorganize the data. When partitions are coalesced, the last partition is chosen, its contents are distributed among the remaining partitions, and it is dropped.

2.7.3.11 drop_table_partition

This is used to remove a partition. The data in the partition are deleted, together with any local indexes. In order to avoid deleting the data, MERGE the partition with another partition before executing DROP on it.

2.7.3.12 merge_table_partition

This is used to merge two partitions into one. Use the INTO clause to specify the name of the new partition. The name can be the same as the name of one of the two partitions being merged, or can be a new name not belonging to any existing table partitions.

When merging range partitions, the partitions are merged into the partition having the higher upper limit.

When merging list partitions, the partitions are merged into a partition having the union of the key values of the two partitions.

When a partition is merged with the default partition, the domain of the default partition is increased to encompass the domain of the merged partition, and only the default partition remains.

If any local indexes have been defined for the table, the local indexes of the merged partitions are deleted.

If the table contains a LOB column, its attributes can be specified separately.

If no tablespace is specified, the new partition is stored in the default tablespace for the table, even if the original partition having the same name as the name of the newly created partition was stored in another tablespace.

2.7.3.13 rename_table_partition

This is used to rename a partition.

2.7.3.14 split_table_partition

This is used to split a partition into two partitions.

The AT clause can only be used with range partitions. It is used to specify a partition key value, on the basis of which a partition is split into two. This value must be larger than the partition key value for the partition immediately preceding it, and smaller than the partition key value for the partition before it was split.

The VALUES clause can only be used with list partitions. It is used to specify a list of values to separate from the list of values for the existing partition. The values specified using the VALUES clause must be present in the list of values for the existing partition; however, not all of the values for the existing partition can be specified in the VALUES list.

The INTO clause is used to specify the names and tablespaces for the two partitions resulting from the SPLIT operation.

If any local indexes have been defined for the table, the local index partition is also split, along with the data partition.

If the table contains a LOB column, the attributes for the LOB column can be set separately.

2.7.3.15 truncate_table_partition

This is used to delete all of the data in a partition.

2.7.3.16 add_column_clause

This is used to add a new column to the table.

2.7.3.17 partition_lob_storage_clause

When a LOB column is added to a partitioned table, this clause is used to set the tablespace in which each of the LOB column partitions is stored.

2.7.3.18 alter_column_clause

These clauses are used to change the default value for an existing column.

2.7.3.19 modify_column_clause

This is used to change the data type of an existing column. The following table shows which data types can be changed into which data types.

△ means that the data type change might result in the loss of non-NULL data. To acknowledge the possibility of data loss and proceed with the data type change anyway, use the TOLERATE DATA LOSS option. When changing data into date type data, ALTIBASE HDB does so according to the DEFAULT_DATE_FORMAT property.

After Modification Before Modification	char	varchar	nchar	nvarchar	clob	bigint	double	float	integer	number	numeric	real	smallint	date	blob	byte	nibble	bit	varbit	geometry
char	o	o	o	o		△	△	△	△	△	△	△	△	△						
varchar	o	o	o	o		△	△	△	△	△	△	△	△	△						
nchar	o	o	o	o		△	△	△	△	△	△	△	△	△						
nvarchar	o	o	o	o		△	△	△	△	△	△	△	△	△						
clob																				
bigint	o	o	o	o		o	△	△	o	△	△	△	o							
double	o	o	o	o		△	o	△	△	△	△	△	△							
float	o	o	o	o		△	△	△	△	△	△	△	△							
integer	o	o	o	o		o	△	△	o	△	△	△	o							
number	o	o	o	o		△	△	△	△	△	△	△	△							
numeric	o	o	o	o		△	△	△	△	△	△	△	△							
real	o	o	o	o		△	△	△	△	△	△	o	△							
smallint	o	o	o	o		o	△	△	o	△	△	△	o							

After Modification Before Modification	char	varchar	nchar	nvarchar	clob	bigint	double	float	integer	number	numeric	real	smallint	date	blob	byte	nibble	bit	varbit	geometry
date	△	△	△	△										o						
blob																				
byte																o				
nibble																	o			
bit																		o	o	
varbit		o																o	o	
geometry																				o

2.7.3.20 drop_column_clause

This is used to delete one or more columns.

2.7.3.21 rename_column_clause

This is used to rename a column.

2.7.3.22 column_definition

- DEFAULT

  When a new column is added, if the DEFAULT option is not specified, the value for each row in the new column will be NULL. However, if the DEFAULT option is specified, when the column is added to a table containing existing rows, it will be populated with the DEFAULT value.

- TIMESTAMP

  This is used to add a timestamp column.

2.7.3.23 column_constraints

This is used to specify constraints for the column.

- NULL/NOT NULL

  This is used to specify whether NULL values are allowed in the column. If NULL values are to be disallowed for a column, then the column can be added using the ALTER TABLE command only if a default value is also specified. In other words, in order to add a new column to a table, the column must either allow NULL values or have a default value specified.

- SET PERSISTENT

  For detailed information, please refer to the explanation of the 2.16 CREATE INDEX command.

- USING INDEX TABLESPACE tablespace_name

  This is used to specify the tablespace in which to store the index for the constraint.

* Many of the clauses in the ALTER TABLE statement have the same function as those in the CREATE TABLE statement. For more information about those clauses, please also refer to 2.21 CREATE TABLE.

2.7.3.24 constraints_clauses

This clause is used to add a constraint to a table, delete an existing constraint, or change the name of an existing constraint.

add_table_constraints_clause

This clause is used to add a constraint to a table.

rename_constraints_clause

This clause is used to change the name of an existing constraint.

drop_constraints_clause

This clause is used to delete an existing constraint.

- DROP CONSTRAINT

  This is used to remove a constraint.

- DROP PRIMARY KEY

  This is used to remove the primary key from the table.

- DROP UNIQUE

  This is used to drop a UNIQUE constraint from one or more columns.

2.7.3.25 RENAME TO

This is used to change the name of the table.

2.7.3.26 MAXROWS

This is used to change the maximum number of records that the table can contain, which was set when the table was created. For more information, please refer to 2.21 CREATE TABLE.

2.7.3.27 ENABLE/DISABLE

This clause is used to activate or deactivate all of the indexes in the table specified using *tbl_name*. The performance of the server can be improved by minimizing the time taken to build indexes, either when the database is started[1] or while the database is providing service.

For example, when using iLoader to load large amounts of data into a database (or relocate them to a new table), if many indexes exist in the table in which the data are to be saved, it will take a lot of time[1] to load the data due to the operations that must be performed on the indexes. In such cases, disabling indexes and then enabling them again after inserting a large amount of records can minimize the time required to build indexes, leading to improved performance.

2.7.3.28 AGING

This is used to physically eliminate previous versions of records that have already been logically deleted from the table.

2.7.3.29 allocate_extent_clause

This is used to explicitly allocate extents to table segments. The SIZE attribute determines the total size of extents to be allocated to the table segment. If the size specified here is not an exact multiple of the size of one extent, then the number of extents that are allocated is rounded up. If the disk tablespace comprises several data files, the extents are distributed among them equally.

2.7.4 Considerations

The definition of a table that has been designated as a replication target table cannot be altered. This means that it is impossible to change the data type of a column in a replication target table because changing the data type would mean altering the table definition.

If the table has only one partition, COALESCE/DROP TABLE PARTITION cannot be used.

The DROP PARTITION and MERGE PARTITION clauses cannot be used with hash-partitioned tables. Instead, the COALESCE PARTITION clause should be used. Moreover, the SPLIT PARTITION clause cannot be used with hash-partitioned tables.

For range-partitioned tables, the partitions to merge must be adjacent to each other.

If the table contains a primary key or unique key, called the referenced key, that is referenced by any foreign key constraints in other tables, the definition of the table cannot be changed.

Columns cannot be added to or removed from a table such that the total number of columns in the table is reduced to 0 (zero) or increased beyond 1024, which is the maximum allowable number of columns in a table. If a table contains one or more columns with the VARIABLE option, the maximum allowable number of columns in the table might be less than 1024, depending on the value specified in IN ROW SIZE clause.

A table can have only one primary key defined for it.

For a foreign key constraint, the foreign key and the unique or primary key being referred to in another table must have the same number of columns, and respective columns must have the same

1. To maximize performance when the system is restarted, a command specifying that indexes are built using parallel processing can be used. Additionally, indexes can be set to PERSISTENT, meaning that they are written to disk when the server is shut down normally.
1. The time required to build an index for a table that contains a large amount of data is proportional to the number of indexes that have been defined for the table. Although it is not possible to build multiple indexes for the same table simultaneously, the index building time can be minimized by building them one by one using parallel processing.

data type.

The data type of a column that is the basis of a foreign key constraint cannot be changed. This is not permitted, either for foreign keys or for unique or primary keys that are referred to by foreign keys in other tables, because data values might be changed when the data type is changed.

The maximum number of indexes that can be created in one table is 64. The maximum number of primary keys and unique keys combined that can exist in one table is 64.

2.7.5 Constraints

A timestamp constraint cannot be added to or removed from an existing column using the ADD/DROP CONSTRAINT clause.

When an INSERT or UPDATE statement is executed on a table containing a column with the TIMESTAMP constraint, the system time is inserted into that column by default. Therefore, the DEFAULT value cannot be changed or dropped using the ALTER TABLE SET/DROP DEFAULT statement. For more information, please refer to the description of the CREATE TABLE statement.

2.7.6 Example

2.7.6.1 Adding and Dropping Columns

<Query>Add the column shown below to the table *books*.

```
isbn: CHAR(10) PRIMARY KEY
edition: INTEGER DEFAULT 1

iSQL> ALTER TABLE books
 ADD COLUMN (isbn CHAR(10) PRIMARY KEY,
 edition INTEGER DEFAULT 1);
Alter success.
```

or

```
iSQL> ALTER TABLE books
 ADD COLUMN (isbn CHAR(10) CONSTRAINT const1
PRIMARY KEY, edition INTEGER DEFAULT 1);
Alter success.
```

<Query> Drop the *isbn* column from the table *books*.

```
iSQL> ALTER TABLE books
 DROP COLUMN isbn;
Alter success.
```

<Query> Drop the *isbn* and *edition* columns from the table *books*.

```
iSQL> ALTER TABLE books DROP COLUMN (isbn, edition);
Alter success.
```

<Query>Add a TIMESTAMP column to the table *books*.

```
iSQL> ALTER TABLE books
ADD COLUMN (due_date TIMESTAMP);
Alter success.
```

<Query> Drop the TIMESTAMP column from the table *books*.

```
iSQL> ALTER TABLE books
 DROP COLUMN due_date;
Alter success.
```

2.7.6.2 Adding and Dropping Constraints for Existing Columns

<Query> Add the UNIQUE constraint to the existing *bno* (book number) column in the table *books*.

```
iSQL> ALTER TABLE books
 ADD UNIQUE(bno);
Alter success.
```

or

```
iSQL> ALTER TABLE books
 ADD CONSTRAINT const1 UNIQUE(bno);
Alter success
```

<Query> Change the name of the constraint *const1* in the table *books*.

```
iSQL> ALTER TABLE books
 RENAME CONSTRAINT const1 TO const_unique;
Alter success
```

<Query> Drop the UNIQUE constraint from the *bno* column in the table *books*.

```
iSQL> ALTER TABLE books
 DROP UNIQUE(bno);
Alter success.
```

or

```
iSQL> ALTER TABLE books
 DROP CONSTRAINT const_unique;
Alter success
```

<Query> While adding a column to the table *inventory*, place the FOREIGN KEY constraint *fk_isbn*, which refers to the *isbn* column in the table *books*, on the new column.

isbn : CHAR(10)

```
iSQL> ALTER TABLE inventory
 ADD COLUMN(isbn CHAR(10) CONSTRAINT fk_isbn REFERENCES books(isbn));
Alter success.
```

<Query> Drop the constraint *fk_isbn* from the table *inventory*.

```
iSQL> ALTER TABLE inventory
 DROP CONSTRAINT fk_isbn;
Alter success.
```

<Query> Drop the primary key constraint from the table *books*.

```
iSQL> ALTER TABLE books
 DROP PRIMARY KEY;
Alter success.
```

<Query> Add a primary key constraint to the existing *bno* (book number) column in the table *books*, and ensure that the index can be used even if a system or media fault occurs (LOGGING).

```
iSQL> ALTER TABLE books
 ADD PRIMARY KEY (bno) USING INDEX PARALLEL 4;
Alter success.
```

Or

```
iSQL> ALTER TABLE book
 ADD PRIMARY KEY (bno) USING INDEX LOGGING
 PARALLEL 4;
Alter success.
```

<Query> Add a primary key constraint to the existing *bno* (book number) column in the table *books*. Create an index using the NOLOGGING option. Use the FORCE option so that the index is available even if the server dies.

```
iSQL> ALTER TABLE books
 ADD PRIMARY KEY (bno) USING INDEX NOLOGGING PARALLEL 4;
Alter success.
```

Or

```
iSQL> ALTER TABLE books
 ADD PRIMARY KEY (bno) USING INDEX NOLOGGING FORCE PARALLEL 4;
Alter success.
```

<Query> Add a primary key constraint to the existing *bno* (book number) column in the table *books*. Create an index using the NOLOGGING option. Use the NOFORCE option to prevent the index from being written to disk.

```
iSQL> ALTER TABLE books
 ADD PRIMARY KEY (bno) USING INDEX NOLOGGING NOFORCE PARALLEL 4;
Alter success.
```

2.7.6.3 Specifying the Tablespace for Individual Index Partitions

<Query> Add the *I2* column, having the LOCALUNIQUE constraint, to the partitioned table *T1*.

```
iSQL> ALTER TABLE T1 ADD COLUMN
(I2 INTEGER LOCALUNIQUE USING INDEX LOCAL
(
PARTITION P1_LOCALUNIQUE ON P1 TABLESPACE TBS3,
PARTITION P2_LOCALUNIQUE ON P2 TABLESPACE TBS2,
PARTITION P3_LOCALUNIQUE ON P3 TABLESPACE TBS1
)
);
```

2.7.6.4 Renaming a Column

This is used to change the name of one of the columns in a table. The new column name must not be the same as the name of any of the other columns in the table. When a column is renamed, the new column inherits all of the indexes and constraints that were originally defined for the column.

After renaming a column, any stored procedures that reference the column by its previous name become invalid. In order to be able to use such a stored procedure again, it will be necessary to rewrite the name of the column in the stored procedure.

<Query> Change the name of a column in the table *departments* from *dno* to *dcode*.

```
iSQL> ALTER TABLE departments
```

```
 RENAME COLUMN dno TO dcode;
Alter success.
```

2.7.6.5 Setting DEFAULT Values

<Query> Set the default value for the *sex* column in the table *employees* to "X".

```
iSQL> ALTER TABLE employees
 ALTER (sex SET DEFAULT 'X');
Alter success.
```

<Query>Change the *sex* column in the table *employees* so that it has no default value.

```
iSQL> ALTER TABLE employees
 ALTER (sex DROP DEFAULT);
Alter success.
```

2.7.6.6 Changing the Data Type of a Column

<Query> Set the data type of the *isbn* column in the *books* table to CHAR(20) and that of the *edition* column to BIGINT.

```
iSQL> ALTER TABLE books MODIFY COLUMN (isbn CHAR(20), edition BIGINT);
Alter success.
```

2.7.6.7 Changing the Name of a Table

<Query> Change the name of the table from *books* to *ebooks*.

```
iSQL> RENAME books TO ebooks;
Rename success.
```

or

```
iSQL> ALTER TABLE books
 RENAME TO ebooks;
Alter success.
```

2.7.6.8 Adding a Persistent Index

<Query> Add the *isbn* column, with a persistent index and having the primary key constraint, to the to the table *books*.

```
iSQL> ALTER TABLE books
 ADD COLUMN (isbn CHAR(10) CONSTRAINT const1
PRIMARY KEY SET PERSISTENT = ON, edition
INTEGER DEFAULT 1);
Alter success.
```

<Query> Add a persistent index to the column *isbn*, which has the primary key constraint, in the table *books*.

```
iSQL> ALTER TABLE books
 ADD COLUMN (isbn CHAR(10) CONSTRAINT const1
PRIMARY KEY, edition INTEGER DEFAULT 1);
Alter success.
```

```
iSQL> ALTER TABLE books
 DROP COLUMN isbn;
Alter success.

iSQL> ALTER TABLE books
 ADD COLUMN (isbn CHAR(10) CONSTRAINT const1
PRIMARY KEY SET PERSISTENT = ON, edition
INTEGER DEFAULT 1);
Alter success.
```

2.7.6.9 Change the Maximum Number of Rows for a Table

<Query> Set the maximum number of records for the table *departments* to 6000000.

```
iSQL> ALTER TABLE departments MAXROWS 6000000;
Alter success.
```

2.7.6.10 Activating and Deactivating Indexes

<Query> Disable all indexes in the table *orders*.

```
iSQL> ALTER TABLE orders ALL INDEX DISABLE;
Alter success.
```

2.7.6.11 Create Partitioned Tables

<Query> Create range-, list- and hash-partitioned tables.

```
iSQL> CREATE TABLE T1
(
I1 INTEGER,
I2 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> CREATE TABLE T2
(
I1 INTEGER,
I2 INTEGER
)
PARTITION BY LIST (I1)
(
PARTITION P1 VALUES (1,2,3,4),
PARTITION P2 VALUES (5,6,7,8),
PARTITION P3 VALUES DEFAULT
) TABLESPACE SYS_TBS_DISK_DATA;
Create success.

iSQL> CREATE TABLE T3
(
I1 INTEGER
)
PARTITION BY HASH (I1)
(
PARTITION P1,
PARTITION P2
) TABLESPACE SYS_TBS_DISK_DATA;
Create success.
```

2.7.6.12 Adding Partitions

<Query> Add a new partition to a hash-partitioned table.

```
ALTER TABLE T3 ADD PARTITION P3;
```

2.7.6.13 Coalescing Partitions

<Query> Coalesce the partitions in the hash-partitioned table *T3* so that only two hash partitions remain.

```
ALTER TABLE T3 COALESCE PARTITION;
```

2.7.6.14 Dropping Partitions

<Query> Delete partition *P2* from table *T1*.

```
ALTER TABLE T1 DROP PARTITION P2;
```

2.7.6.15 Merging Partitions

<Query> Merge the remaining partitions *P1* and *P3* in table T1 to form a new partition named *P_1_3*.

```
ALTER TABLE T1 MERGE PARTITIONS P1, P3 INTO PARTITION P_1_3;
```

2.7.6.16 Renaming Partitions

<Query> Change the name of a partition from *P1* to *P1_LIST*.

```
ALTER TABLE T2 RENAME PARTITION P1 TO P1_LIST;
```

2.7.6.17 Splitting Partitions

<Query> Split the default partition *P3* in the range-partitioned table *T1* on the basis of the value 350. This will create a partition named *P_200_350*, which holds values from 200 - 350, and change the name of the default partition to *P_OVER_350*.

```
ALTER TABLE T1 SPLIT PARTITION P3
AT ( 350 ) INTO ( PARTITION P_200_350, PARTITION P_OVER_350 );
```

<Query> Use a VALUES clause instead of an AT clause to split a list-partitioned table.

```
ALTER TABLE T2
SPLIT PARTITION P1_LIST VALUES ( 2, 4 )
INTO
(
PARTITION P_2_4 TABLESPACE TBS1,
PARTITION P_1_3 TABLESPACE TBS2
);
```

2.7.6.18 Truncating Partitions

<Query> Delete all data in partition *P5*.

```
ALTER TABLE T1 TRUNCATE PARTITION P5;
```

2.7.6.19 Using the row_movement_clause

<Query> Table T1 must be a partitioned table. If it is a non-partitioned table, an error will occur.

```
ALTER TABLE T1 ENABLE ROW MOVEMENT;
```

2.7.6.20 Allocating Extents to Tables

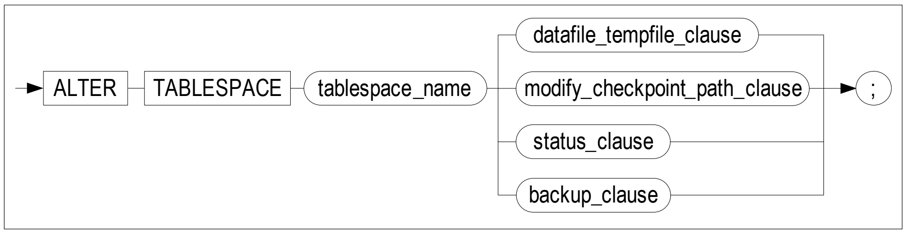
<Query> Allocate 10MB extents to the table *LOCAL_TBL*, which exists in disk tablespace.

```
ALTER TABLE LOCAL_TBL ALLOCATE EXTENT ( SIZE 10M );
```

2.8 ALTER TABLESPACE

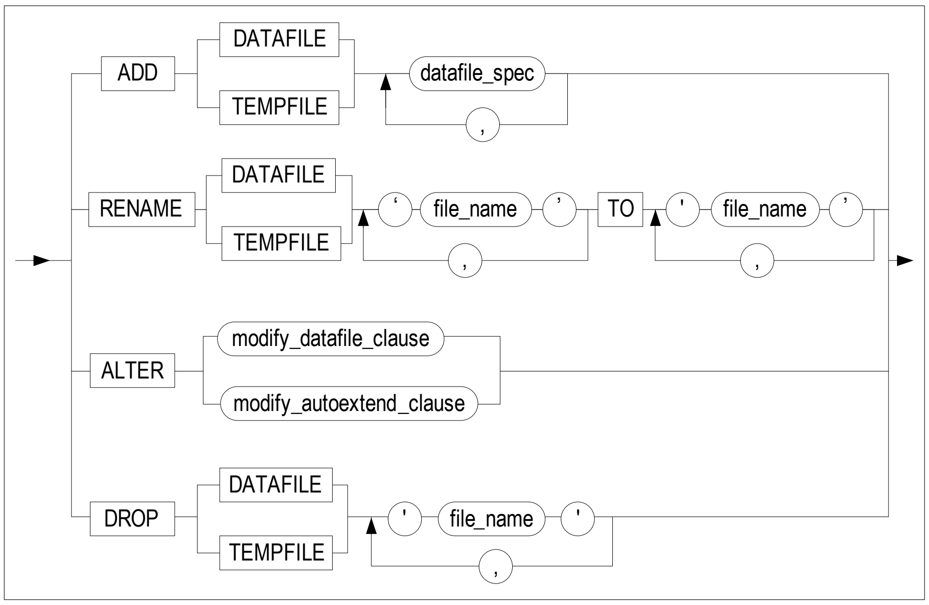
2.8.1 Syntax

2.8.1.1 alter_tablespace ::=



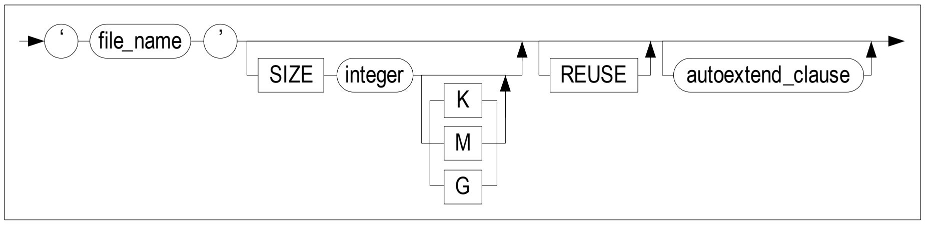
datafile_tempfile_clause ::=, *modify_checkpoint_path_clause ::=*, *status_clause ::=backup_clause ::=*

2.8.1.2 datafile_tempfile_clause ::=

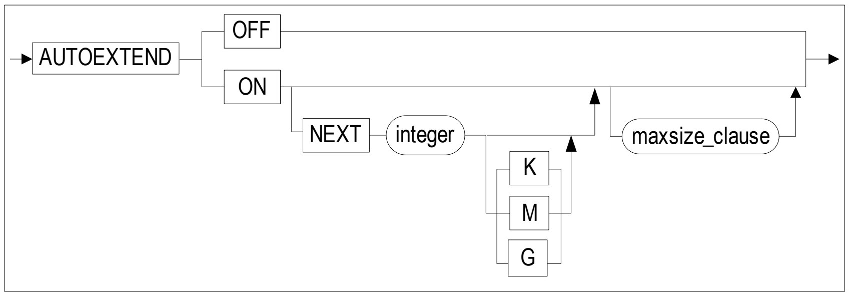


datafile_spec ::=, *modify_datafile_clause ::=*, *modify_autoextend_clause ::=*

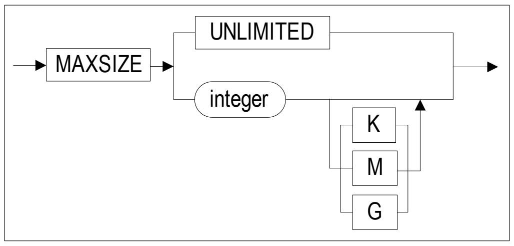
2.8.1.3 datafile_spec ::=



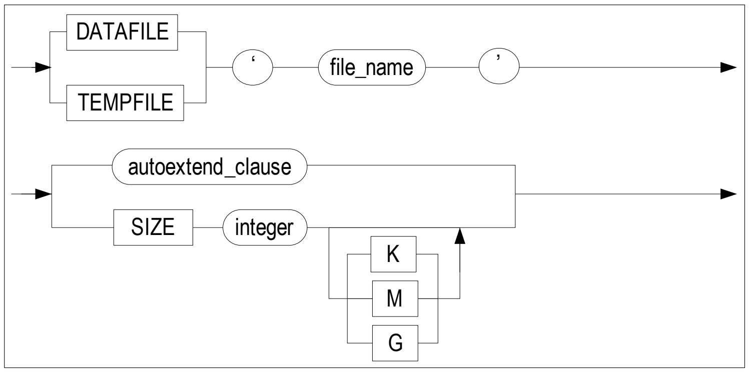
2.8.1.4 autoextend_clause ::=



2.8.1.5 maxsize_clause ::=

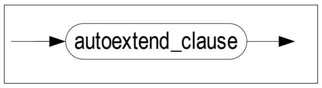


2.8.1.6 modify_datafile_clause ::=



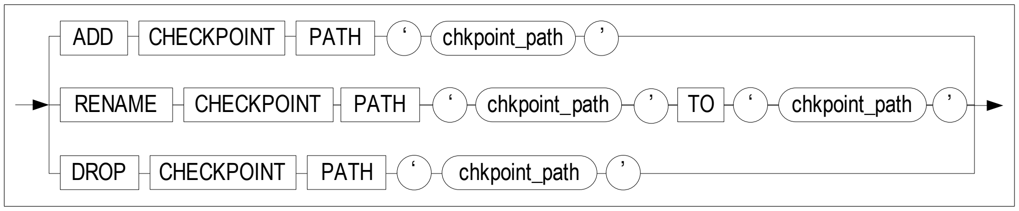
autoextend_clause ::=

2.8.1.7 modify_autoextend_clause ::=

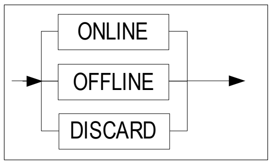


autoextend_clause ::=

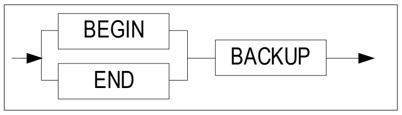
2.8.1.8 modify_checkpoint_path_clause ::=



2.8.1.9 status_clause ::=



2.8.1.10 backup_clause ::=



2.8.2 Prerequisites

The SYS user and users to whom the ALTER TABLESPACE system privilege has been granted can use all of the functionality of the ALTER TABLESPACE statement.

2.8.3 Description

The ALTER TABLESPACE statement is used to change the definition of a disk, temporary, memory or volatile tablespace. This statement is also used to change other tablespace attributes, including the data and temporary files associated with the tablespace, the checkpoint paths, settings related to automatic extension, and the status of the tablespace.

2.8.3.1 tablespace_name

This is used to specify the name of the tablespace to change.

2.8.3.2 datafile_tempfile_clause

This is used to add, delete or change a data file or temporary file.

For detailed information on the datafile_spec, maxsize_clause and autoextend_clause clauses, please refer to the description of the CREATE TABLESPACE statement.

ADD DATAFILE | TEMPFILE Clause

This is used to add data or temporary files to the corresponding tablespace.

RENAME DATAFILE | TEMPFILE Clause

This is used to change the name of the data or temporary files in a tablespace. More than one file can be renamed at one time. A file having the same name as TO *file_name* must have already been created.

modify_datafile_clause

This is used to change the size or the autoextend attribute of a data or temporary file in a disk tablespace.

modify_autoextend_clause

This is used to change the autoextension-related attributes for a memory or volatile tablespace, including whether it is automatically extended in size, the unit (increment) by which it is extended, and its maximum size.

DROP DATAFILE | TEMPFILE Clause

This is used to remove data or temporary files from a tablespace. More than one file can be removed at one time. Because this statement does not actually delete the physical files from the file system, they will need to be deleted (or otherwise managed) manually by the user.

2.8.3.3 modify_checkpoint_path_clause

This is used to add, change or delete a checkpoint image path. Operations related to checkpoint image paths can only be performed during the CONTROL phase.

ADD CHECKPOINT PATH Clause

This is used to add a new checkpoint path to a memory tablespace. The DBA can move any checkpoint image files from the existing checkpoint paths to the new checkpoint path as desired. Because ALTIBASE HDB looks for checkpoint image files in all checkpoint paths when a memory tablespace is loaded, checkpoint image files can be stored in any of the checkpoint paths for the tablespace.

When checkpointing takes place after a new checkpoint path has been added, the new checkpoint image files are distributed evenly among all of the checkpoint paths, including the new checkpoint path.

If the specified checkpoint path does not exist, or if the user who started up the ALTIBASE HDB server does not have write permissions for the checkpoint path, an error will be raised. Therefore, the DBA must manually create the checkpoint path to be added in the file system and add write permissions to the path for the user.

RENAME CHECKPOINT PATH Clause

This is used to change an existing checkpoint path for a memory tablespace to the path specified in the clause following "TO". The DBA must manually create or rename the actual checkpoint path in the file system. If the specified checkpoint path does not exist, or if the user who started up the ALTIBASE HDB server does not have suitable permissions for the checkpoint path, an error will be raised.

DROP CHECKPOINT PATH Clause

This is used to delete an existing checkpoint path from a memory tablespace. It is the DBA's responsibility to physically move the checkpoint image files in the directory corresponding to the deleted checkpoint path to the remaining checkpoint paths in the tablespace. Because ALTIBASE HDB looks for checkpoint image files in all checkpoint paths when a memory tablespace is loaded, checkpoint image files can be stored in any of the valid checkpoint paths for the tablespace.

The physical path corresponding to the dropped checkpoint path must be manually deleted from the file system. A memory tablespace must have at least one checkpoint path. If an attempt is made to delete the only checkpoint path remaining for a memory space, an error will be raised.

2.8.3.4 status_clause

This is used to change the status of a disk or memory tablespace to ONLINE, OFFLINE or DISCARD.

OFFLINE

When a disk tablespace is taken offline, the contents of all data pages in the buffer corresponding the tablespace are written to data files, and the pages in the buffer pool are invalidated.

In the case of a memory tablespace, the contents of the data pages are written to checkpoint image files, and the page memory is released.

All memory that has been allocated to indexes in the tablespace is released, and the indexes for the tables in the tablespace become unavailable. Furthermore, the tables in the tablespace become unavailable until the tablespace is brought back online.

ONLINE

When a disk tablespace is brought online, all of its data files become accessible, and the tables in the tablespace become available again.

When a memory tablespace is brought online, memory is assigned to all data pages again, and the contents of checkpoint image files are loaded back into the memory pages.

If a referenced tablespace is OFFLINE, an attempt to bring the tablespace ONLINE will succeed, but it may be impossible to access the tables that are stored in the tablespace.

A so-called "referenced tablespace" is, in the case of a disk tablespace, a tablespace containing a table that is associated with indexes, BLOB/CLOB columns, etc. that are stored in one or more other tablespaces, or containing a partitioned table having some of its partitions stored in one or more other tablespaces.

DISCARD

This is used during the CONTROL phase to switch the status of a disk or memory tablespace to DISCARD.

The tables, indexes and BLOB/CLOB columns in a discarded tablespace are unusable. Furthermore, discarded tablespaces are ignored when RESTART RECOVERY is executed and when unused data are removed from the database while the database is being started up.

Once a tablespace has been discarded, the only command that can be executed on it is DROP TABLESPACE. It cannot be brought back ONLINE.

2.8.3.5 backup clause

This statement is used to indicate that online backup ("hot backup"), in which the data files for a disk or memory tablespace are copied, is to be performed, or is complete.

BEGIN BACKUP

This is used to indicate that online backup is to be performed on all of the data files that constitute a tablespace. During the tablespace backup operation, which takes place between this statement and the END BACKUP statement, transactions are not prevented from accessing the tablespace.

The user must execute BEGIN BACKUP before performing online backup. Furthermore, it is possible to indicate that online backup is to be performed for multiple tablespaces at the same time. However, disk temporary tablespaces cannot be backed up online.

END BACKUP

This is used to indicate that online backup of a disk or memory tablespace is complete. The user must execute the END BACKUP statement immediately after online backup is completed.

2.8.4 Considerations

The ALTER TABLESPACE statement can be used to add data files and change tablespace attributes only when in online mode, and can be used to rename data files only during the CONTROL phase (STARTUP CONTROL).

status_clause cannot be used with temporary or volatile tablespaces.

2.8.5 Examples

<Query> Add the data file *tbs2.user*, which is 64 MB in size, to the *user_disk_tbs* tablespace. When more space is needed, automatically increase the size of the file in 512 kB increments.

```
iSQL> ALTER TABLESPACE user_disk_tbs
 ADD DATAFILE '/tmp/tbs2.user' SIZE 64M
 AUTOEXTEND ON NEXT 512K;
Alter success.
```

<Query 2> To distribute disk I/O when checkpointing, add the *'/home/path'* checkpoint path to the *user_memory_tbs* tablespace. Additionally, change the tablespace attributes such that it increases in

size in 256 MB increments and cannot grow larger than 1 GB. (Note that although table attributes can be changed during the SERVICE phase, checkpoint paths can be added only during the CONTROL phase.)

```
iSQL(sysdba)> startup control;
iSQL(sysdba)> ALTER TABLESPACE user_memory_tbs ADD CHECKPOINT PATH '/home/
path';
Alter success.

iSQL> ALTER TABLESPACE user_memory_tbs ALTER AUTOEXTEND OFF;
iSQL> ALTER TABLESPACE user_memory_tbs ALTER AUTOEXTEND ON NEXT 256M MAXSIZE
1G;
Alter success.
```

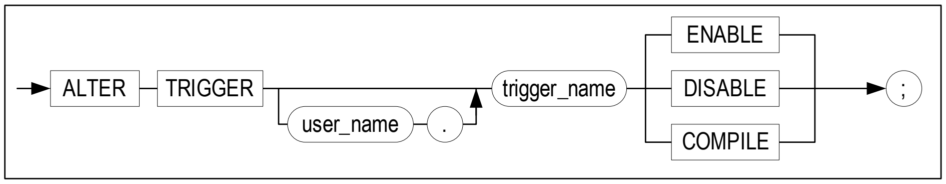
<Query 3> Change the attributes of the *user_volatile_tbs* tablespace such that it increases in size in 256 MB increments and cannot grow larger than 1 GB.

```
iSQL> ALTER TABLESPACE user_volatile_tbs ALTER AUTOEXTEND ON NEXT 256M MAX-
SIZE 1G;
Alter success.
```

2.9 ALTER TRIGGER

2.9.1 Syntax

2.9.1.1 alter_trigger ::=



2.9.2 Prerequisites

Only the SYS user, the owner of the schema containing the trigger, and users having the ALTER ANY TRIGGER system privilege can execute the ALTER TRIGGER statement.

2.9.3 Description

This statement is used to enable, disable, or compile a specified database trigger.

2.9.3.1 user_name

This is used to specify the name of the owner of the trigger to be altered. If it is omitted, ALTIBASE HDB will assume that the trigger to be altered belongs to the schema of the user connected via the current session.

2.9.3.2 trigger_name

This is used to specify the name of the trigger to alter.

2.9.3.3 ENABLE

This is used to enable the trigger.

2.9.3.4 DISABLE

This is used to disable the trigger.

2.9.3.5 COMPILE

This is used to explicitly compile the trigger, regardless of whether or not it is valid. Explicit recompiling helps reduce the load on the system incurred by automatically compiling invalid triggers when they need to fire.

2.9.4 Example

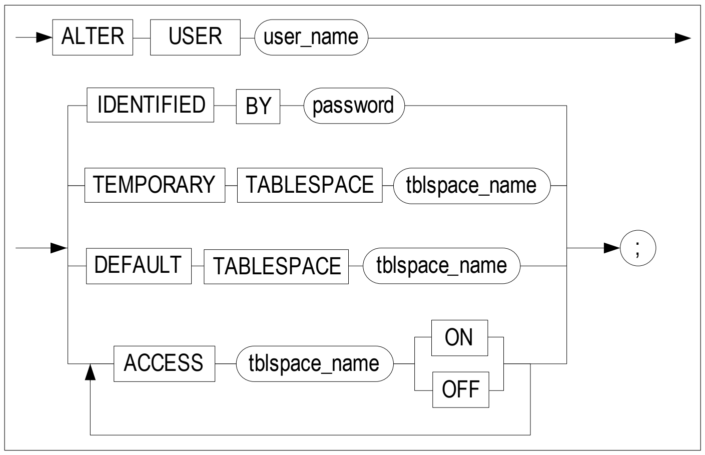
<Query>This is used to disable a trigger so that it does not fire automatically. More examples involving the *del_trigger* trigger can be found in the explanation of the CREATE TRIGGER statement.

```
iSQL> ALTER TRIGGER del_trigger DISABLE;
Alter success.
```

2.10 ALTER USER

2.10.1 Syntax

2.10.1.1 alter_user ::=



2.10.2 Prerequisites

Only the SYS user and users having the ALTER USER system privilege can execute the ALTER USER statement. However, individual users can change their own passwords without this privilege.

2.10.3 Description

This statement is used to change a user's password and the tablespaces associated with the user.

2.10.3.1 IDENTIFIED clause

This is used to specify a new password for the user. The new password is specified after the BY element.

Because the other commands are the same as the corresponding commands executed using the CREATE USER statement, please refer to the description of the CREATE USER statement.

2.10.4 Precautions

When changing the password for the SYS user, who can log on in SYSDBA mode, after the password is changed using the ALTER USER statement, it must be changed one more time by running the `altipasswd` utility in a console window of the operating system (Unix shell or DOS terminal window). For more information on the altipasswd utility, please refer to the *Utilities Manual.*

2.10.5 Examples

<Query> Change the password of the user *Tom* to ufd4xh9.

```
iSQL> ALTER USER Tom
 IDENTIFIED BY ufd4xh9;
Alter success.
```

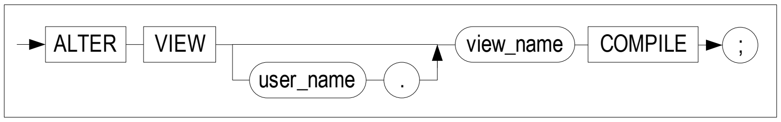
<Query> Change a user's default tablespace.

```
iSQL> ALTER USER Tom
 DEFAULT TABLESPACE tom_data;
Alter success.
```

2.11 ALTER VIEW

2.11.1 Syntax

2.11.1.1 alter_view ::=



2.11.2 Prerequisites

Only the SYS user, the owner of the schema containing the view, and users having the ALTER ANY TABLE system privilege can execute the ALTER VIEW statement.

2.11.3 Description

This is used to explicitly recompile a view that has become invalid. As an example, when one of the tables on which the view is based has been changed using the ALTER TABLE statement, it will be necessary to explicitly recompile the view in order to use it.

2.11.3.1 user_name

This is used to specify the name of the owner of the view to recompile. If omitted, ALTIBASE HDB will assume that the view belongs to the schema of the user connected via the current session.

2.11.3.2 view_name

This is used to specify the name of the view to recompile.

When recompiling the view, ALTIBASE HDB reads the view creation statement and compiles the view again, so any errors that arose when the view was originally created will be raised again when the ALTER VIEW statement is executed. If the FORCE option was used when creating the view originally, the view may still be in an invalid state after the ALTER VIEW statement has executed successfully.

The ALTER VIEW statement cannot be used to change the definition of an existing view. To change a view's definition, use the CREATE VIEW statement with the OR REPLACE option.

2.11.4 Examples

<Query> After changing the definition of the table *employees*, on which the view *avg_sal* is based, recompile the view. (The definition of *avg_sal* can be found in the explanation of the CREATE VIEW statement.)

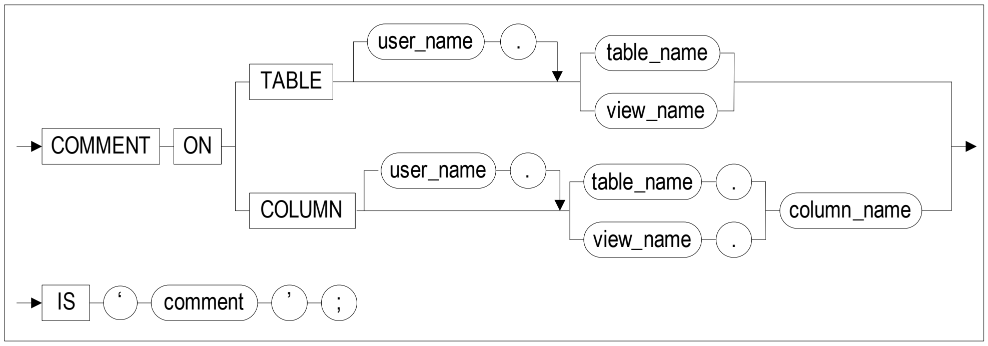
```
iSQL> ALTER TABLE employees
```

```
 ADD COLUMN (email VARCHAR(20));
Alter success.
iSQL> ALTER VIEW avg_sal COMPILE;
Alter success.
iSQL> SELECT * FROM avg_sal;
AVG_SAL.DNO AVG_SAL.EMP_AVG_SAL
-----------------------------------
1001        2150
1002        1340
1003        2438.25
2001        1400
3001        1800
3002        2500
4001        1550
4002        1396.66667
            1500
9 rows selected.
```

2.12 COMMENT

2.12.1 Syntax

2.12.1.1 comment_on ::=



2.12.2 Prerequisite

Only the SYS user, the owner of the schema to which the table (or view) belongs, users having the ALTER object privilege for the table (or view), and users having the ALTER ANY TABLE system privilege can use the COMMENT statement to write comments.

2.12.3 Description

This statement is used to write or modify comments for a specified table, view or column.

2.12.3.1 user_name

This is the name of the owner of the object on which the COMMENT statement is executed. If it is omitted, ALTIBASE HDB will assume that the object being commented on belongs to the schema of the user connected via the current session.

2.12.3.2 table_name, view_name

This is used to specify the name of the table or view being commented on.

2.12.3.3 column_name

This is used to specify the name of the column being commented on.

2.12.3.4 comment

This is the actual content of the comment. The comment can be up to 4000 bytes long. To delete an existing comment, execute the COMMENT statement with nothing between the single quotation marks (’’).

2.12.4 Example

<Query> Add comments to the *books* table, which is owned by the user *library*, and to one of its columns.

```
iSQL> COMMENT ON TABLE library.books IS 'Table of Book Info';
Comment Created.

iSQL> COMMENT ON COLUMN library.books.title IS 'Title of Book';
Comment Created.
```

<Query> Read the comments on the *books* table, which is owned by the user *library*, and on its columns.

```
iSQL> SET VERTICAL ON;
iSQL> SELECT * FROM system_.sys_comments_ WHERE user_name = ’LIBRARY’ AND
table_name = ’BOOKS’;
SYS_COMMENTS_.USER_NAME : LIBRARY
SYS_COMMENTS_.TABLE_NAME : BOOKS
SYS_COMMENTS_.COLUMN_NAME : TITLE
SYS_COMMENTS_.COMMENTS : title of book

SYS_COMMENTS_.USER_NAME : LIBRARY
SYS_COMMENTS_.TABLE_NAME : BOOKS
SYS_COMMENTS_.COLUMN_NAME :
SYS_COMMENTS_.COMMENTS : table of book info

2 rows selected.
```

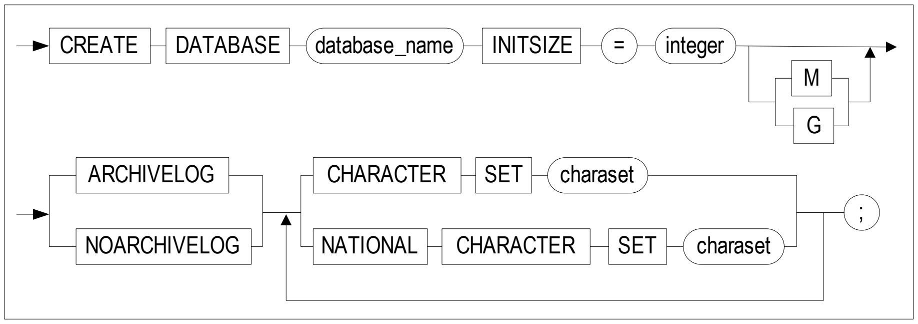
<Query> Delete the comments from the *books* table and the *title* column in that table.

```
iSQL> COMMENT ON TABLE library.books IS ’’;
Comment created.
iSQL> COMMENT ON COLUMN library.books.title IS ’’;
Comment created.
```

2.13 CREATE DATABASE

2.13.1 Syntax

2.13.1.1 create_database ::=



2.13.2 Prerequisites

Because this statement can only be executed during the PROCESS phase, the only user who can execute this SQL statement is the SYS user, and only when running in -sysdba mode.

2.13.3 Description

This statement is used to create a database. When a database is created, various system tablespaces, including the dictionary tablespace, the undo tablespace, and temporary tablespaces are created. The names of these system tablespaces are defined by the system.

Note that user-defined tablespaces cannot be created during database creation; rather, they are subsequently added by users.

The database character set and national character set must be specified when the database is created.

2.13.3.1 database_name

This is used to specify the name of the database to be created. The database name specified here must be the same as that specified using the DB_NAME property in the properties file. If they are not the same, an error will be raised.

2.13.3.2 INITSIZE clause

The initial size of a memory database is specified here, using an expression such as “128M“ (i.e. 128 megabytes) or “4G“ (i.e. 4 gigabytes). If only a number is specified here, the size unit is assumed to be megabytes by default.

The system tablespaces related to a disk database are automatically created when the CREATE DATA-

BASE statement is executed.

The default values for system tablespaces are determined by reading the following properties from the altibase.properties file.

- SYS_DATA_TBS_EXTENT_SIZE, SYS_TEMP_TBS_EXTENT_SIZE, SYS_UNDO_TBS_EXTENT_SIZE
- SYS_DATA_FILE_INIT_SIZE, SYS_TEMP_FILE_INIT_SIZE, SYS_UNDO_FILE_INIT_SIZE
- SYS_DATA_FILE_MAX_SIZE, SYS_TEMP_FILE_MAX_SIZE, SYS_UNDO_FILE_MAX_SIZE
- SYS_DATA_FILE_NEXT_SIZE, SYS_TEMP_FILE_NEXT_SIZE, SYS_UNDO_FILE_NEXT_SIZE

2.13.3.3 ARCHIVELOG | NOARCHIVELOG

This is used to specify whether the database will initially run in archivelog mode or noarchivelog mode. Archivelog mode is used to prepare for media recovery, whereas noarchivelog mode does not support media recovery.

For more information about ALTIBASE HDB backup and recovery, please refer to the *Administrator's Manual*.

2.13.3.4 charset

This is used to specify the database character set and national character set.

- Available Database Character Sets

  US7ASCII

  KO16KSC5601

  MS949

  BIG5

  GB231280

  UTF8

  SHIFTJIS

  EUCJP

- Available National Character Sets

  UTF8

  UTF16

2.13.4 Examples

<Query> Create a database named *mydb* that is initially 10MB in size, and for which the database character set is KSC5601 and the national character set is UTF16.

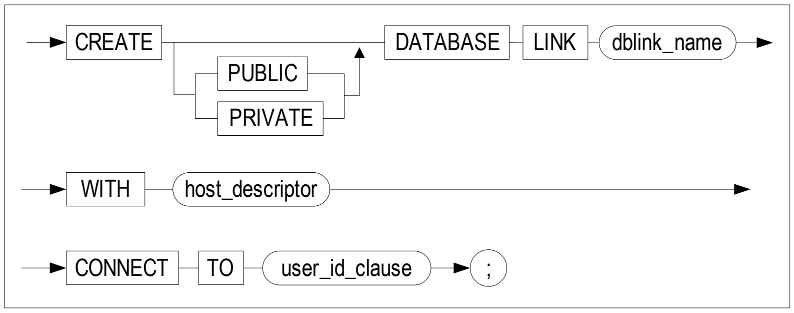
```
$ isql -s localhost -u sys -p manager -sysdba
..
iSQL(sysdba)> STARTUP PROCESS;
Trying Connect to Altibase.. Connected with Altibase.

TRANSITION TO PHASE: PROCESS
Command execute success.
iSQL(sysdba)> CREATE DATABASE mydb INITSIZE=10M NOARCHIVELOG CHARACTER SET
KSC5601 NATIONAL CHARACTER SET UTF16;
.
.
Create success.
```

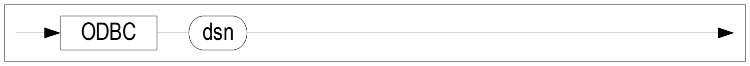
2.14 CREATE DATABASE LINK

2.14.1 Syntax

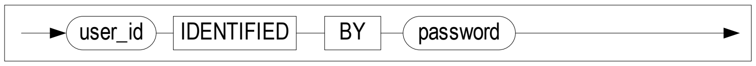
2.14.1.1 create_database_link ::=



2.14.1.2 host_descriptor ::=



2.14.1.3 user_id_clause ::=



2.14.2 Prerequisites

Only the SYS user and users having the CREATE DATABASE LINK system privilege can create a Database Link instance.

Since Database Link uses ODBC to connect to a remote server, the ODBC Driver Manager and the ALTIBASE HDB ODBC driver must be installed on the local server. Additionally, the DSN for the remote server must be set in the ODBC environment file on the local server. For more information on ODBC settings, please refer to the *Getting Started*.

2.14.3 Description

This command is used to create a new instance of Database Link having the specified name.

2.14.3.1 PUBLIC|PRIVATE

This is used to specify the whether the Database Link instance that will be created is PUBLIC or PRIVATE. If this is set to PUBLIC, the created Database Link instance will be accessible by all users. If it is set to PRIVATE, the created Database Link instance will only be accessible by the user who created it. If this is not specified, the Database Link instance is PRIVATE by default.

2.14.3.2 dblink_name

This is used to specify the name of the Database Link instance to be created.

2.14.3.3 dsn

This is used to specify the ODBC Data Source Name (DSN), which is required for access via ODBC.

2.14.3.4 user_id_clause (user_id/password)

This is used to specify the id and password of a database user for the remote server with which a connection is to be established. However, if a user id and password are specified in odbc.ini, those credentials will take precedence over the ones specified in this statement. Therefore, it is recommended that the DSN for Database Link should not include the user id and password settings.

Note also that the user specified here must have sufficient privileges to access the objects in question on the remote server when connecting via Database Link. If not, an error related to access privileges will be raised.

2.14.4 Examples

<Query 1>Suppose that there is the DSN *altibase_odbc* for connecting to a remote database server and that the Database Link user id and password are *user1/user1*. Create a Database Link instance named *link1* that can be used only by the user who created it.

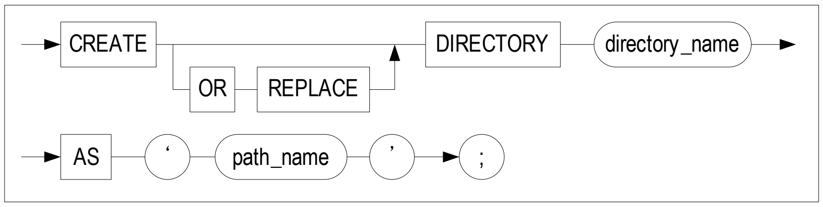
```
iSQL> CREATE PRIVATE DATABASE LINK link1
 WITH ODBC altibase_odbc
 CONNECT TO user1 IDENTIFIED BY user1;
```

<Query 2> Suppose that there is the DSN *altibase_odbc* for connecting to a remote database server and that the user id and password for use with Database Link are *user1/user1*. Create a Database Link instance named *link2*, which can be used by all users in the system.

```
iSQL> CREATE PUBLIC DATABASE LINK link2
 WITH ODBC altibase_odbc
 CONNECT TO user1 IDENTIFIED BY user1;
```

2.15 CREATE DIRECTORY

2.15.1 Syntax



2.15.2 Prerequisites

Only the SYS user and users having the CREATE ANY DIRECTORY system privilege can execute this statement.

2.15.3 Description

The fact that stored procedures are able to manipulate files means that it is possible to use SQL to read from and write to text files in the host's file system. This feature makes it possible to perform a wide variety of tasks, including leaving messages in files about the execution of stored procedures, writing reports and query results in files, and inserting data read from files into database tables.

The CREATE DIRECTORY statement is used to create a database object that corresponds to a directory that contains files that are manipulated by stored procedures in this way.

The owner of a DIRECTORY object created using the CREATE DIRECTORY statement is always the SYS user. The user who actually created the DIRECTORY object is only granted read and write privileges, including the WITH GRANT OPTION, for the object.

The CREATE DIRECTORY statement records directory data in the SYS_DIRECTORIES_ meta table, but does not actually create the new directory in the file system. Therefore, the user must explicitly create the directory in the actual file system.

2.15.3.1 OR REPLACE

This option allows an existing DIRECTORY object to be replaced when a new DIRECTORY object is created with the same name. Note that the actual directory in the file system is not deleted.

2.15.3.2 directory_name

This is used to specify the name of the database object representing the directory.

2.15.3.3 path_name

This is used to specify the absolute path of the directory in the file system as a character string.

2.15.4 Examples

- <Query> Create a directory object named *alti_dir1* in the folder `/home/altibase/altibase_home/psm_msg`.

  ```
  iSQL> create directory alti_dir1 as
  '/home/altibase/altibase_home/psm_msg';
  Create success.
  ```

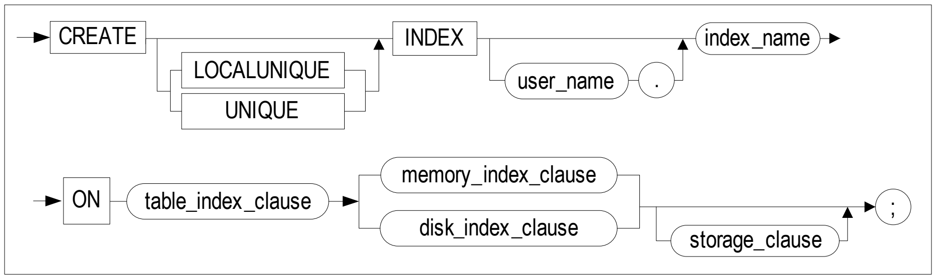
- <Query>Create a directory object named *alti_dir1* in the folder `/home/altibase/altibase_home/psm_result`. If a DIRECTORY object named *alti_dir1* already exists in the database, replace it with this new one.

  ```
  iSQL> create or replace directory alti_dir1 as
  '/home/altibase/altibase_home/psm_result';
  Create success.
  ```

2.16 CREATE INDEX

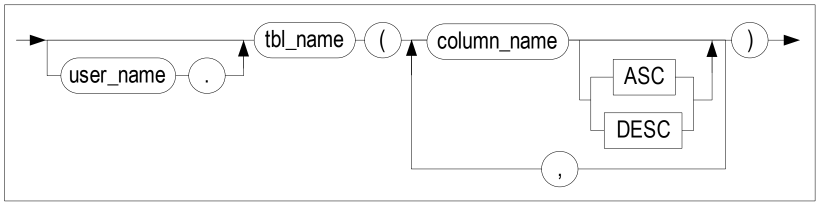
2.16.1 Syntax

2.16.1.1 create_index::=

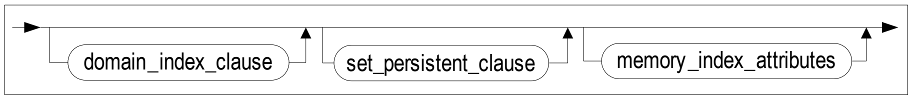


memory_index_clause::=, *disk_index_clause::=*, *storage_clause::=*

2.16.1.2 table_index_clause::=

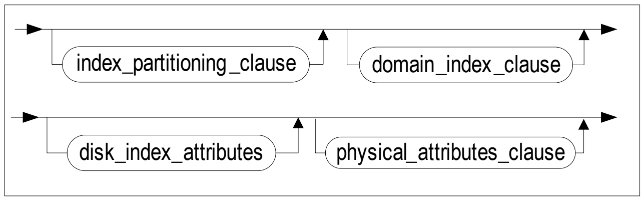


2.16.1.3 memory_index_clause::=



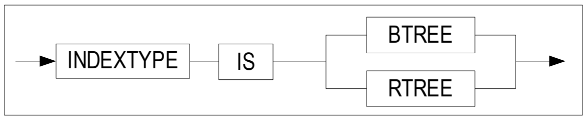
domain_index_clause::=, *set_persistent_clause::=*, *memory_index_attributes::=*

2.16.1.4 disk_index_clause::=

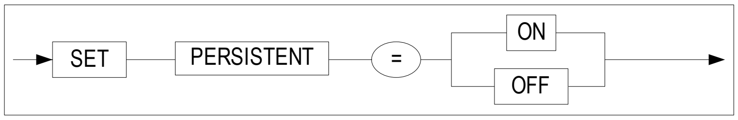


index_partitioning_clause::=, *domain_index_clause::=*, *disk_index_attributes::=*, *physical_attributes_clause::=*

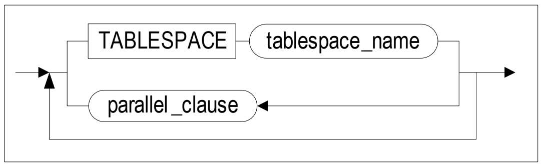
2.16.1.5 domain_index_clause::=



2.16.1.6 set_persistent_clause::=

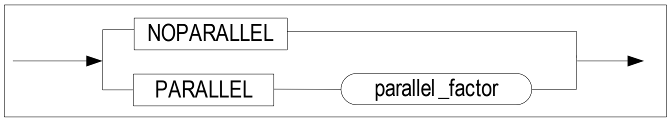


2.16.1.7 memory_index_attributes::=

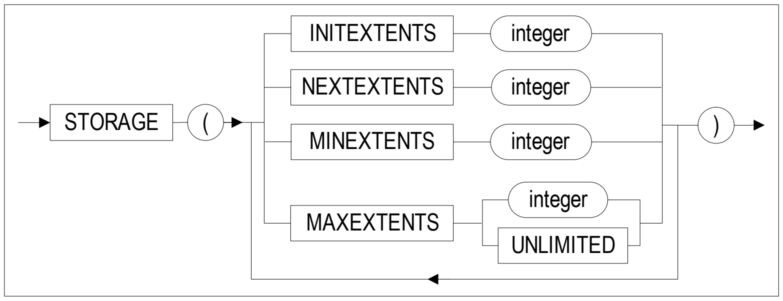


parallel_clause::=

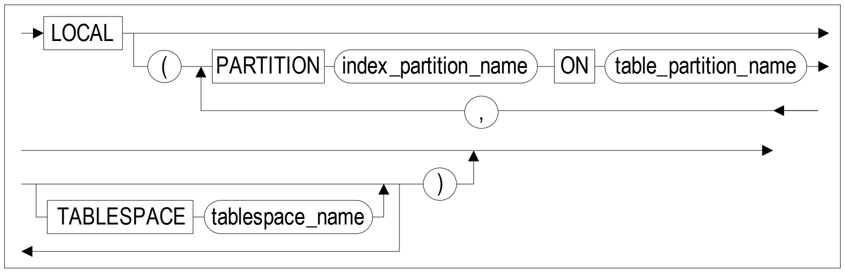
2.16.1.8 parallel_clause::=



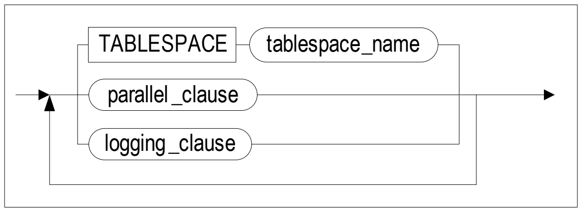
2.16.1.9 storage_clause::=



2.16.1.10 index_partitioning_clause::=

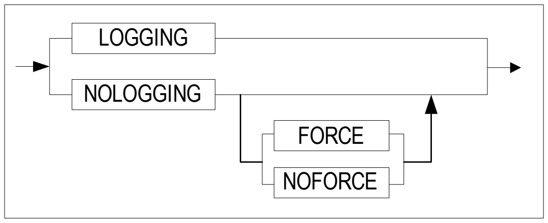


2.16.1.11 disk_index_attributes::=

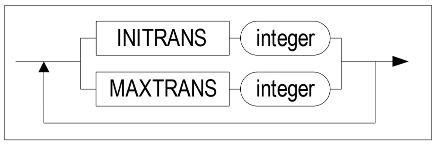


parallel_clause::=, *logging_clause::=*

2.16.1.12 logging_clause::=



2.16.1.13 physical_attributes_clause::=



2.16.2 Prerequisites

Only the SYS user, users having the CREATE INDEX system privilege, and users having sufficient privileges to modify index objects in the table to which the index is to be added can execute this statement.

2.16.3 Description

This statement is used to create an index on the basis of one or more of the columns in a table. When a partitioned index (i.e. local index) is created, the LOCALUNIQUE keyword can be optionally specified.

A partitioned index is classified as either a prefixed index or a non-prefixed index, depending on the relationship between the partition key and the index columns. If the leftmost index partition key is the same column as the leftmost index column, it is a prefixed index. If they are different columns, it is a non-prefixed index.

2.16.3.1 user_name

This is used to specify the name of the owner of the index to create. If omitted, ALTIBASE HDB will create the index in the schema of the user connected via the current session.

2.16.3.2 index_name

This is used to specify the name of the index to create.

2.16.3.3 UNIQUE

This keyword indicates that duplicate values are not allowed.

2.16.3.4 LOCALUNIQUE

This keyword is useful when creating a partitioned index. It indicates that the UNIQUE constraint must be satisfied within each index partition of a partitioned (local) index.

2.16.3.5 ASC/DESC

The use of the ASC or DESC keywords respectively specifies that each column of the index is to be sorted in ascending or descending order.

2.16.3.6 index_partitioning_clause

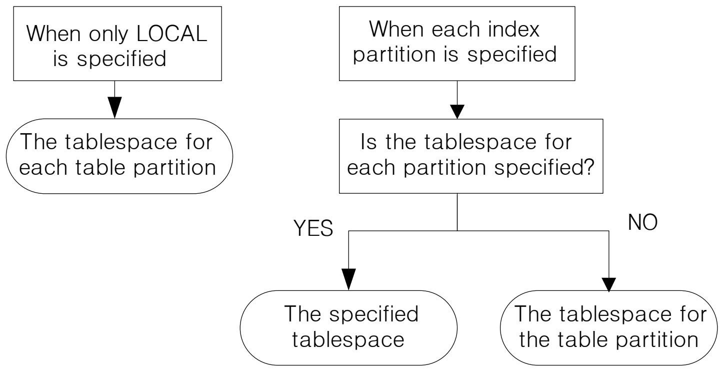
This is used to specify that the index to be created is a partitioned index.

The easiest way to create a partitioned index is simply to specify the LOCAL keyword when creating the index. Alternatively, the attributes of the index partition to be created for each table partition can be specified in greater detail.

If only the LOCAL keyword is specified, an index partition is created for each table partition, and the names of the partitions are automatically determined by the system. Index partitions are successively named SYS_IDX1, SYS_IDX2, etc.

The attributes of index partitions can be expressly specified for some or all table partitions. When the attributes of only some index partitions are specified, the attributes of the index partitions for the remaining table partitions are determined automatically as outlined above.

If no tablespace is specified when creating a partitioned index, the process for determining the tablespace in which each of the index partitions is stored is as shown in the following diagram:


When only LOCAL is specified
The tablespace for each table partition
When each index partition is specified
Is the tablespace for each partition specified?
YES
NO
The specified tablespace
The tablespace for the table partition

2.16.3.7 BTREE

This is used to specify that the index is a B^+-tree index, which is useful in situations where ranges are often searched. An index can be either a B^+-tree or an R-tree index. If the `INDEXTYPE IS` clause is omitted, the index will be a B^+-tree index by default.

2.16.3.8 RTREE

This is used to specify that the index is an R-tree index, which is useful for processing multidimensional data.

2.16.3.9 SET PERSISTENT

This option is used to specify whether or not to save a memory index to disk when the server is shut down normally.

The default behavior is for the server not to save memory indexes when shut down normally, in which case all indexes for memory tablespaces must be rebuilt when the server is started. However, if SET PERSISTENT is set to ON, memory indexes are saved in special files on disk when the server is shut down normally. This saves time when the server is started, because the indexes are read from these files, thus obviating the time required to rebuild them.

The default is OFF. Set PERSISTENT to ON to stipulate that indexes are to be saved to disk when the server is shut down.

2.16.3.10 physical_attributes_clause

- INITRANS Clause

  This is used to set the initial number of TTS (Touched Transaction Slots). The default is 8.

- MAXTRANS Clause

  This is used to set the maximum number of TTS (Touched Transaction Slots). The default is 30.

2.16.3.11 TABLESPACE clause

This is used to specify the name of the tablespace in which the index is to be stored. If this clause is omitted, ALTIBASE HDB stores the index in the default tablespace for the owner of the schema to which the index belongs. However, when an index is created for a memory table, even if the tablespace is specified, this clause is ignored because memory indexes are not stored in any tablespace.

2.16.3.12 parallel_clause

This is a hint for setting the number of threads used to create an index, with the aim of realizing a performance improvement. ALTIBASE HDB determines the optimal number of index creation threads in consideration of the user-defined *parallel_factor*, which is specified using a hint, the size of the tables for which the indexes are being created, and the amount of available memory at the time the index is created.

The value of *parallel_factor* can be set within the range from 0 to 512. The default is the number of CPUs in the host on which ALTIBASE HDB is running. Because the number of index creation threads is determined using the above optimized determination method, it is safe to omit *parallel_factor*.

If *parallel_factor* is not set, or is set to 0, the value of the INDEX_BUILD_THREAD_COUNT property in the altibase.properties file, which has the same meaning as *parallel_factor*, is used instead. If the INDEX_BUILD_THREAD_COUNT property has not been set either, the number of CPUs is used as a hint to set the optimum number of index creation threads.

If *parallel_factor* is set to a value that is greater than the number of CPUs, or is set to a value greater than 512, the user-defined value is ignored and the number of CPUs is used as a hint for setting the optimum number of threads.

2.16.3.13 logging_clause

The LOGGING or NOLOGGING clause can be used to enable or disable logging when an index is created for a disk table. Logging is enabled by default, meaning that information about the creation of the index is logged when an index is created.

The FORCE and NOFORCE options are used to determine whether to forcibly store a disk index to disk immediately after the index is created.

For more details about *logging_clause*, please refer to the section pertaining to indexes in the “Objects and Privileges“ chapter of the *Administrator's Manual*.

2.16.3.14 storage_clause

This is used to set parameters for managing extents in segments.

- INITEXTENTS Clause

  This sets the number of extents that are allocated by default when a segment is created. The default is 1.

- NEXTEXTENTS Clause

  This sets the number of extents that are added to the segment every time the segment is increased in size. The default is 1.

- MINEXTENTS Clause

  This sets the minimum number of extents in a segment. The default is 1.

- MAXEXTENTS Clause

  This set the maximum number of extents in a segment. If this isn't specified, there is no maximum limit on the number of extents in a segment.

2.16.4 Considerations

In the case of an index for a partitioned table, i.e. a partitioned index, the tablespace in which each local index is stored is defined separately in *index_partitioning_clause*. *disk_index_attributes* cannot be used to specify the tablespace for an entire partitioned index. Additionally, a local index can only be a B^+-tree index.

In the event of a system or media fault, the consistency of an index that was created using the NOLOGGING (FORCE/NOFORCE) option cannot be guaranteed. In this case, the error “The index is inconsistent” will be raised. To fix this error, locate the inconsistent index, drop it, and create it again. The consistency of an index can be checked using the V$DISK_BTREE_HEADER performance view.

An index cannot be created on the basis of a LOB column.

2.16.5 Examples

- <Query 1> Create the index *emp_idx2* on the column *eno* in ascending order and on the column *dno* in descending order.

  ```
  iSQL> CREATE INDEX emp_idx2
   ON employees (eno ASC, dno DESC);
  Create success.
  ```

- <Query 2> Create a unique index named *emp_idx2* for the *dno* column in the *employees* table in descending order. (This is possible when there are no records in the *employees* table, or when there are only unique values in the column *dno*.)

  ```
  iSQL> CREATE UNIQUE INDEX emp_idx2
   ON employees (dno DESC);
  Create success.
  ```

- <Query 3> Create the B^+-tree index *emp_idx3* in ascending order for the *eno* column in the *employees* table. Because a primary key already exists for the *eno* column of the *employees* table, it must first be deleted before the index *emp_idx3* can be created. If it is not deleted first, the following error will be raised:

  ```
  ERR-31051 : Duplicate key columns in an index

  iSQL> ALTER TABLE employees
   DROP PRIMARY KEY;
  Alter success.
  iSQL> CREATE INDEX emp_idx3
   ON employees (eno ASC)
   INDEXTYPE IS BTREE;
  Create success.
  ```

- <Query 4>Create the index *emp_idx1* as a persistent B$^+$-tree index in the table *employees* using the column *eno* in descending order and the column *dno* in ascending order.

```
iSQL> CREATE INDEX emp_idx1
 ON employees (eno DESC, dno ASC)
 INDEXTYPE IS BTREE SET PERSISTENT = ON;
Create success.
```

- <Query 5> Create the index *idx1* in the *user_data* tablespace on the basis of the *i1* column in the *table_user* table.

```
iSQL> CREATE INDEX idx1
 ON table_user (i1)
 TABLESPACE user_data;
Create success.
```

- <Query 6> Create the index *idx2* in the *user_data* tablespace on the basis of the *i1* column in the *table_user* table using the parallel option.

```
iSQL> CREATE INDEX idx2
 ON table_user (i1)
 TABLESPACE user_data PARALLEL 4;
Create success.
```

- <Query 7> Create a local index, that is, an index in which the partitions correspond to respective table partitions, based on *product_id*. Allow the partition names to be determined automatically.

```
CREATE INDEX prod_idx ON products(product_id) LOCAL;
```

- <Query 8> Create a local index, specifying the attributes for each index partition.

```
CREATE INDEX prod_idx ON products(product_id)
LOCAL
(
  PARTITION p_idx1 ON p1 TABLESPACE tbs_disk1,
  PARTITION p_idx2 ON p2 TABLESPACE tbs_disk2,
  PARTITION p_idx3 ON p3 TABLESPACE tbs_disk3
);
```

- <Query 9>Create a local index, specifying the attributes for only some of the index partitions. The attributes for the other partitions are determined automatically.

```
CREATE INDEX prod_idx ON products(product_id)
LOCAL
(
  PARTITION p_idx1 ON p1 TABLESPACE tbs_disk1,
  PARTITION p_idx3 ON p3 TABLESPACE tbs_disk3
);
```

- <Query 10>Create the index *idx1* in the table *employees* based on the employee number column (*eno*). Enable logging to ensure the availability of the index in the event of a system or media fault. Assume that the table *employees* is in a disk tablespace.

```
iSQL> CREATE INDEX idx1
 ON employees (eno);
Create success.
```

or

```
iSQL> CREATE INDEX idx1
```

```
 ON employees (eno) LOGGING ;
Create success.
```

- <Query 11>Create the index *idx1* in the table *employees* with the NOLOGGING option using the employee number column (*eno*) in ascending order and the department number column (*dno*) in ascending order. Ensure that the index is available in the event of a system fault after the index is created (FORCE). Assume that the table *employees* is in a disk tablespace.

```
iSQL> CREATE INDEX idx1
 ON employees (eno ASC, dno ASC)
 NOLOGGING;
Create success.
```

or

```
iSQL> CREATE INDEX idx1
 ON employees (eno ASC, dno ASC)
 NOLOGGING FORCE;
Create success.
```

- <Query 12> Create the index *idx1* in the table *employees* with the NOLOGGING option using the employee number column (*eno*) in ascending order and the department number column (*dno*) in ascending order. Specify that the index is not to be written to disk (NOFORCE). Assume that the table *employees* is in a disk tablespace.

```
iSQL> CREATE INDEX idx1
 ON employees (eno ASC, dno ASC)
 NOLOGGING NOFORCE;
Create success.
```

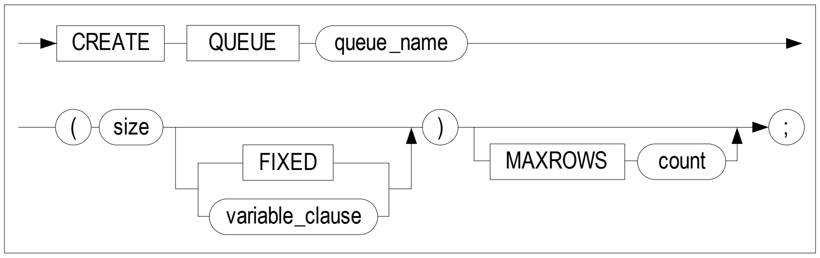
- <Query 13> Create the index *LOCAL_IDX* in the disk tablespace *USERTBS* for the table *LOCAL_TBL*. Allocate one extent to the index when it is created and specify that 2 extents are to be added whenever it is necessary to increase the size of the index segment, and also that there is no upper limit to the total number of extents in the index segment.

```
iSQL> CREATE INDEX LOCAL_IDX ON LOCAL_TBL ( I1 )
 TABLESPACE USERTBS
 STORAGE ( INITEXTENTS 1 NEXTEXTENTS 2 MAXEXTENTS UNLIMITED );
Create success.
```

2.17 CREATE QUEUE

2.17.1 Syntax

2.17.1.1 create_queue ::=



2.17.2 Description

This statement is used to create a queue and specify the maximum length of a message that can be inserted into the queue. It is also possible to specify the maximum number of records that the associated queue table can contain.

2.17.2.1 queue_name

This is used to specify the name of the queue. The maximum possible length of the queue name is 32 bytes.

2.17.2.2 size

This is used to set the maximum size (in bytes) of a message to be stored in the queue. This value can be set within the range from 1 to 32,000 bytes.

2.17.2.3 FIXED|variable_clause

This is used to specify how messages are saved. (For more information, please refer to the *General Reference*).

2.17.2.4 MAXROWS count

This is used to set the maximum number of records that can be stored in a queue table. This value can be set within the range from 1 to 4294967295 (or 2^{32}-1). When not specified, it defaults to the maximum value of 4294967295.

2.17.3 Considerations

When a queue is created, an object having the name *queue_name* + "_NEXT_MSG_ID" is created in the database. Therefore, if any existing table, view, sequence, synonym or stored procedure has the same name as the queue to be created, or has the name *queue_name* + "_NEXT_MSG_ID", the CREATE QUEUE statement will return an error.

2.17.4 Example

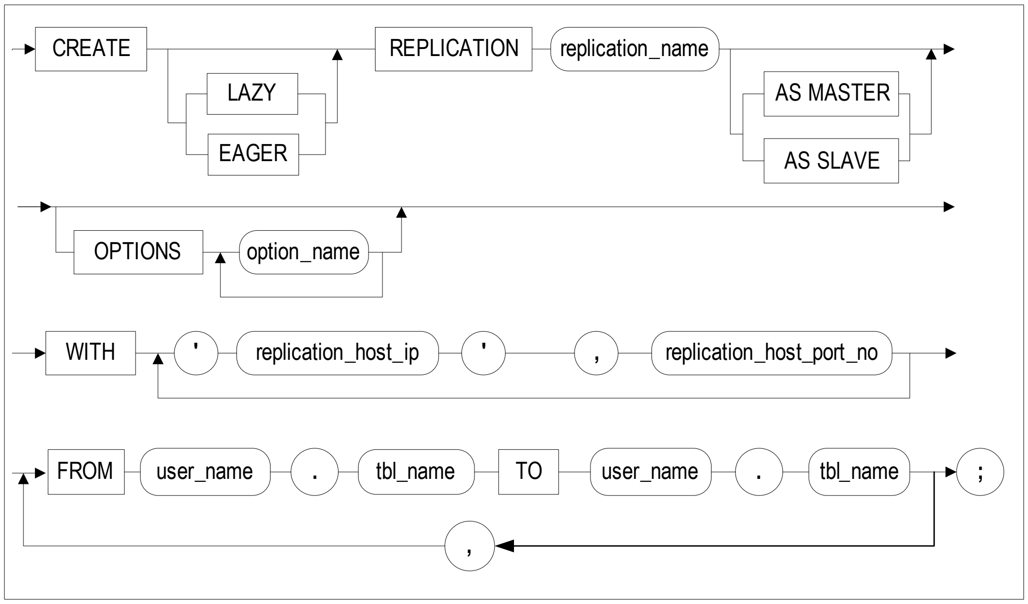
<Query>Create the queue *Q1*, stipulating that the maximum message length is 40 bytes and that the maximum number of records is 1,000,000.

```
CREATE QUEUE Q1(40) MAXROWS 1000000;
```

2.18 CREATE REPLICATION

2.18.1 Syntax

2.18.1.1 create_replication ::=



2.18.2 Description

This statement is used to create a replication object, set the connection between a local server and one or more remote servers, and establish replication therebetween. A single replication object can be used to connect to a maximum of 32 different remote servers. Replication takes place between tables on a 1:1 basis; that is, a table is matched to only one corresponding table.

In order to resolve conflicts, the AS MASTER or AS SLAVE clause can be specified in the statement. Doing so stipulates that a master-slave scheme is to be used to resolve conflicts. For more detailed information about replication conflict resolution, please refer to Chapter 2 of the *Replication Manual*.

- *replication_name*

  This is used to specify the name of the replication object. The replication object name must be the same on both the local server and the remote server.

- *option_name*

  This is used to optionally specify the RECOVERY and OFFLINE functions for the replication object. These extra features are for use in data recovery and when performing offline replication, respectively. For more information, please refer to Chapter 3 of the *Replication Manual*.

- *remote_host_ip*

This is used to specify the IP address of the remote server.

- *remote_host_port_no*

  This is used to specify the port number used by the Receiver thread on the remote server. It is identical to the REPLICATION _PORT_NO property in the altibase.properties file on the remote server.

- *user_name*

  This is used to specify the name of the owner of the table to be replicated.

- *tbl_name*

  This is used to specify the name of the table to be replicated.

2.18.3 Examples

<Query> Create replication *rep1* according to the following conditions:

- The IP address of the local server is 192.168.1.60.
- The replication port number on the local server is 25524.
- The IP address of the remote server is 192.168.1.12.
- The replication port number on the remote server is 35524.

Create the replication object to replicate the *employees* and *departments* tables between the servers.

On the local server:

```
iSQL> CREATE REPLICATION rep1
 WITH '192.168.1.12',35524
 FROM sys.employees TO sys.employees,
 FROM sys.departments TO sys.departments;
Create success.
```

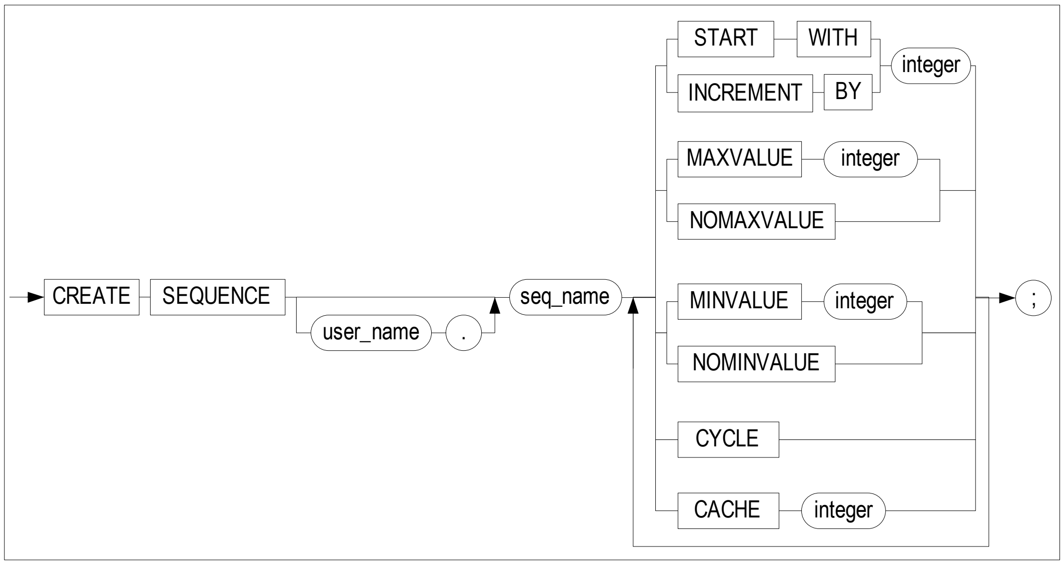
On the remote server:

```
iSQL> CREATE REPLICATION rep1
 WITH '192.168.1.60',25524
 FROM sys.employees TO sys.employees,
 FROM sys.departments TO sys.departments;
Create success.
```

2.19 CREATE SEQUENCE

2.19.1 Syntax

2.19.1.1 create_sequence ::=



2.19.2 Prerequisites

Only the SYS user and users having the CREATE SEQUENCE system privilege can execute this statement. Additionally, it is necessary to be the SYS user or have the CREATE ANY SEQUENCE system privilege in order to create a sequence in another user's schema.

2.19.3 Description

This statement is used to define a new sequence having the specified name and automatically populate the sequence.

2.19.3.1 user_name

This is used to specify the name of the owner of the sequence to be created. If it is omitted, ALTIBASE HDB will create the sequence in the schema of the user who is connected via the current session.

2.19.3.2 seq_name

This is used to specify the name of the sequence to be created.

2.19.3.3 START WITH

This is the initial value of the sequence. This can be set to any value between MINVALUE and MAX-VALUE inclusive. If this value is omitted and the value for INCREMENT BY is more than 0, the default value is the same as the minimum value of the sequence. If this value is omitted and the value for INCREMENT BY is less than 0, the default value is the same as the maximum value of the sequence.

2.19.3.4 INCREMENT BY

This is the value by which the sequence increments. The default value is 1. The absolute of this value must be less than the difference between MAXVALUE and MINVALUE.

2.19.3.5 MAXVALUE

This is the maximum value of the sequence. This can be set to any value between -9223372036854775807 and 9223372036854775806. If the value for INCREMENT BY is more than 0, the default value is 9223372036854775806. If the value for INCREMENT BY is less than 0, the default value is -1.

2.19.3.6 MINVALUE

This is the minimum value of the sequence. This can be set to any value between -9223372036854775807 and 9223372036854775806. If the value for INCREMENT BY is more than 0, the default value is 1. If the value for INCREMENT BY is less than 0, the default value is -9223372036854775807.

2.19.3.7 CYCLE

This clause is used to ensure that the sequence will continue to generate values when it reaches the value specified using MAXVALUE or MINVALUE. 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.

2.19.3.8 CACHE

A specified number of sequence values can be created in advance and cached in memory so that they can be accessed more quickly. The cache is populated when a key value is first requested from a new sequence, and is accessed every time another key value is 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. When a sequence is created, the default CACHE value is 20.

Note: Please note that the sequence_name.CURRVAL value of a newly created sequence cannot be accessed. In order to access the sequence_name.CURRVAL value for a newly created sequence, it is first necessary to access the sequence_name.NEXTVAL value.

2.19.4 Examples

The following SQL statements show how to define sequences and view sequence values and information.

```
iSQL> CREATE TABLE seqtbl(i1 INTEGER);
Create success.
iSQL> CREATE OR REPLACE PROCEDURE proc1
AS
BEGIN
 FOR i IN 1 .. 10 LOOP
 INSERT INTO seqtbl VALUES(i);
 END LOOP;
END;
/
Create success.
iSQL> EXEC proc1;
Execute success.
```

<Query> Create a sequence named *seq1* that begins at 13, increments by 3, and has a minimum value of 0 and no maximum value.

```
iSQL> CREATE SEQUENCE seq1
 START WITH 13
 INCREMENT BY 3
 MINVALUE 0 NOMAXVALUE;
Create success.

iSQL> INSERT INTO seqtbl VALUES(seq1.NEXTVAL);
1 row inserted.
iSQL> INSERT INTO seqtbl VALUES(seq1.NEXTVAL);
1 row inserted.
iSQL> SELECT * FROM seqtbl;
SEQTBL.I1
--------------
1
2
3
4
5
6
7
8
9
10
13
16
12 rows selected.
```

<Query> Change *seq1* so that it increments by 50 and starts over at the minimum value if it reaches the maximum value of 100.

```
iSQL> ALTER SEQUENCE sys.seq1
 INCREMENT BY 50
 MAXVALUE 100
 CYCLE;
Alter success.

iSQL> INSERT INTO sys.seqtbl VALUES(seq1.NEXTVAL);
1 row inserted.
iSQL> INSERT INTO sys.seqtbl VALUES(seq1.NEXTVAL);
1 row inserted.
iSQL> INSERT INTO sys.seqtbl VALUES(seq1.NEXTVAL);
1 row inserted.
iSQL> INSERT INTO sys.seqtbl VALUES(seq1.NEXTVAL);
1 row inserted.
iSQL> SELECT * FROM sys.seqtbl;
SEQTBL.I1
--------------
```

```
1
2
3
4
5
6
7
8
9
10
13
16
66
0
50
100
16 rows selected.
```

<Query> View the current value of *seq1*, which will cause a new value to be generated.

```
iSQL> SELECT seq1.CURRVAL FROM dual;
SEQ1.CURRVAL
-----------------------
100
1 row selected.
```

<Query> Change the value in column *i1* to the next value of the sequence, which is 0.

```
iSQL> UPDATE SEQTBL SET i1 = seq1.NEXTVAL;
16 rows updated.
```

<Query> View the current value of *seq1*.

```
iSQL> SELECT seq1.CURRVAL FROM dual;
SEQ1.CURRVAL
-----------------------
0
1 row selected.
```

<Query> Change *seq1* so that the specified number of sequence values (25) are cached for faster access.

```
iSQL> ALTER SEQUENCE seq1
INCREMENT BY 2
MAXVALUE 200
CACHE 25;
Alter success.

iSQL> CREATE OR REPLACE PROCEDURE proc2
AS
BEGIN
 FOR i IN 1 .. 30 LOOP
 INSERT INTO seqtbl VALUES(seq1.NEXTVAL);
 END LOOP;
END;
/
Create success.

iSQL> EXEC proc2;
Execute success.
iSQL> SELECT * FROM seqtbl;
SEQTBL.I1
--------------
0
```

```
50
100
0
50
100
0
50
100
0
50
100
0
50
100
0
2
4
6
8
10
12
14
.
.
.
58
60
46 rows selected.
```

When connected to a database as the SYS user, output information on all sequences.

```
iSQL> SELECT * FROM SEQ;
USER_NAME
--------------------------------------------
SEQUENCE_NAME      CURRENT_VALUE      INCREMENT_BY
------------------------------------------------
MIN_VALUE          MAX_VALUE          CYCLE              CACHE_SIZE
------------------------------------------------
SYS
SEQ1               60                 2
0                  200                YES                25
1 row selected.
```

The following SQL statements show how to define sequences and view sequence values and information using multiple user accounts.

```
iSQL> CONNECT sys/manager;
Connect success.
iSQL> CREATE USER user1 IDENTIFIED BY user1;
Create success.
iSQL> CREATE USER user2 IDENTIFIED BY user2;
Create success.
iSQL> CONNECT user1/user1;
Connect success.
iSQL> CREATE SEQUENCE seq1 MAXVALUE 100 CYCLE;
Create success.
iSQL> CREATE SEQUENCE seq2;
Create success.
```

<Query> Output information on all sequences created by *user1*.

```
iSQL> SELECT * FROM SEQ;
SEQUENCE_NAME CURRENT_VALUE INCREMENT_BY
```

```
-----------------------------------------------
MIN_VALUE MAX_VALUE CYCLE CACHE_SIZE
-----------------------------------------------
SEQ1 1 1
1 100 YES 20
SEQ2 1 1
1 9223372036854775806 NO 20
2 rows selected.
iSQL> CONNECT user2/user2;
Connect success.
iSQL> CREATE SEQUENCE seq1 INCREMENT BY -30;
Create success.
iSQL> CREATE SEQUENCE seq2 INCREMENT BY -10 MINVALUE -100;
Create success.
iSQL> CONNECT sys/manager;
Connect success.
iSQL> CREATE SEQUENCE seq2 START WITH 20 INCREMENT BY 30;
Create success.
iSQL> CREATE SEQUENCE seq3 CACHE 40;
Create success.
```

<Query> When connected to the database as the SYS user, output information on all sequences.

```
iSQL> SELECT * FROM SEQ;
USER_NAME
-------------------------------------------
SEQUENCE_NAME      CURRENT_VALUE         INCREMENT_BY
-----------------------------------------------
MIN_VALUE          MAX_VALUE             CYCLE              CACHE_SIZE
-----------------------------------------------
SYS
SEQ1               60                    2
0                  200                   YES                25
SYS
SEQ2               20                    30
1                  9223372036854775806   NO                 20
SYS
SEQ3               1                     1
1                  9223372036854775806   NO                 40
USER1
SEQ1               1                     1
1                  100                   YES                20
USER1
SEQ2               1                     1
1                  9223372036854775806   NO                 20
USER2
SEQ1               -1                    -30
-9223372036854775806 -1                  NO                 20
USER2
SEQ2               -1                    -10
-100               -1                    NO                 20
7 rows selected.
```

<Query> Use the following statements to view information on sequence objects.

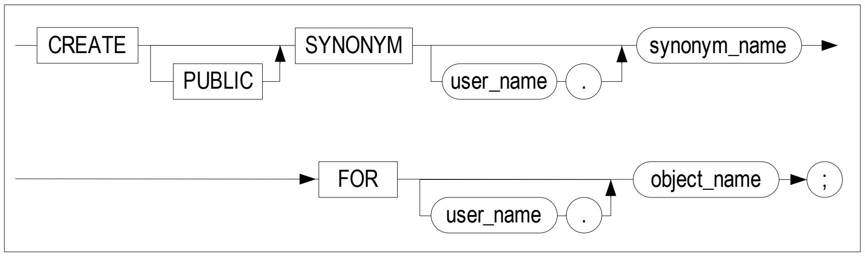
```
iSQL> select * from v$seq;
```

-> This command displays information about all sequence objects that have been created. Unlike `Select * from seq`, querying the performance view allows information about other users' sequences to be viewed. For more information on the performance view V$SEQ, please refer to the section of the Data Dictionary that explains performance views in the *General Reference*.

2.20 CREATE SYNONYM

2.20.1 Syntax

2.20.1.1 create_synonym ::=



2.20.2 Prerequisites

Only the SYS user and users having the CREATE SYNONYM system privilege can execute this statement. Additionally, it is necessary to be the SYS user or have the CREATE ANY SYNONYM system privilege in order to create a synonym in another user's schema, and it is necessary to be the SYS user or have the CREATE PUBLIC SYNONYM system privilege in order to create a public synonym.

2.20.3 Description

This command is used to create a synonym, which is an alternative name for a table, view, sequence, stored procedure, stored function, or another synonym.

Synonyms can be referred to using the following kinds of SQL statements:

DML statement	DDL statement

SELECT
INSERT
UPDATE
DELETE
MOVE
LOCK TABLE
	GRANT
	REVOKE

2.20.3.1 PUBLIC Synonyms vs. PRIVATE Synonyms

A PUBLIC synonym is accessible by all users, whereas a PRIVATE synonym can only be accessed by its owner. However only usershaving appropriate privileges on the underlying object can access the synonym.

To create a public synonym, specify PUBLIC in the statement. If this keyword is not provided, a PRIVATE synonym will be created by default.

2.20.3.2 user_name

If a user name is provided in front of the synonym name, that user will be the owner of the synonym.

When creating a PUBLIC synonym, do not specify the name of the owner.

When creating a PRIVATE synonym, it is possible to specify the name of the owner. If no user name is provided, the synonym will be created in the schema of the user connected via the current session.

2.20.3.3 synonym_name

If there is a table, view, sequence, stored procedure, stored function or another synonym that has the same name as the synonym to be created, an error will be raised. Because synonyms occupy the same namespace as these object types, the name for the synonym must be unique within the schema in which it is created.

2.20.3.4 FOR clause

This clause is used to specify the object for which the synonym will serve as an alias.

2.20.3.5 user_name

This is used to specify the owner of the object for which the synonym will function as an alias. If no user name is specified, ALTIBASE HDB will assume that the object belongs to the schema of the user connected via the current session.

2.20.3.6 object_name

This is used to specify the name of the object for which the synonym will serve as an alias.

If this object does not exist in the database, no error will be raised, and the synonym will be created successfully regardless. In other words, the schema object need not currently exist, and it is not necessary to have privileges for the object for which the synonym will function as an alias.

2.20.4 Privileges and Synonyms

To execute DML statements on synonyms, it is necessary to have DML execution privileges for the underlying object.

When DML execution privileges on a synonym are granted or revoked, the privileges are actually granted or revoked on the underlying objects.

Therefore, when the execution of a DML statement on a synonym results in an error, check the SYS_GRANT_SYSTEM_ or SYS_GRANT_OBJECT_ meta table to determine whether the user has been granted suitable privileges for the underlying object. If the user has not been granted suitable privileges, grant the privileges to the user. When granting privileges to the user, they can be granted either by referencing the object itself, or by referencing the synonym.

If the user already has appropriate privileges for the underlying object itself, it is necessary only to create the synonym. No additional privileges need to be granted for the synonym.

Privileges granted for an object by referencing a synonym for the object are not revoked even if the synonym is subsequently dropped. This is because the privileges are actually granted for the object underlying the synonym, not the synonym itself, even though the synonym was referenced when the privileges were granted.

2.20.5 Object Name Search Order

In order to determine which objects match the objects referenced in an SQL statement, any tables, views, sequences, stored procedures or stored functions having corresponding names are first searched for. If they don't exist, other synonym objects having the names are searched for. PRIVATE synonyms are examined before PUBLIC synonyms.

For example, whether objects having the names referenced in an SQL statement exist in the database would be determined in the order shown below.

- SELECT * FROM NAME

  1. Any tables or views having the name “NAME” are searched for.
  2. If no tables or views having the name “NAME” exist, any PRIVATE synonym objects having the name are searched for in the schema of the user connected via the current session.
  3. If no PRIVATE synonyms having the name exist, any PUBLIC synonym objects are searched for.

- SELECT * FROM USER.NAME

  1. Any tables or views having the name “NAME” are searched for in the “USER” schema.
  2. If no tables or views having the name “NAME” exist, any PRIVATE synonym objects having the name are searched for in the “USER” schema.
  3. If no PRIVATE synonyms having the name exist, no PUBLIC synonym objects are searched for. Instead, an error is returned.

2.20.6 Examples

<Query>Create a synonym called *my_dept* for the table *dept*, which is owned by the user *altibase*, in the current user's schema and execute some DML statements using the synonym.

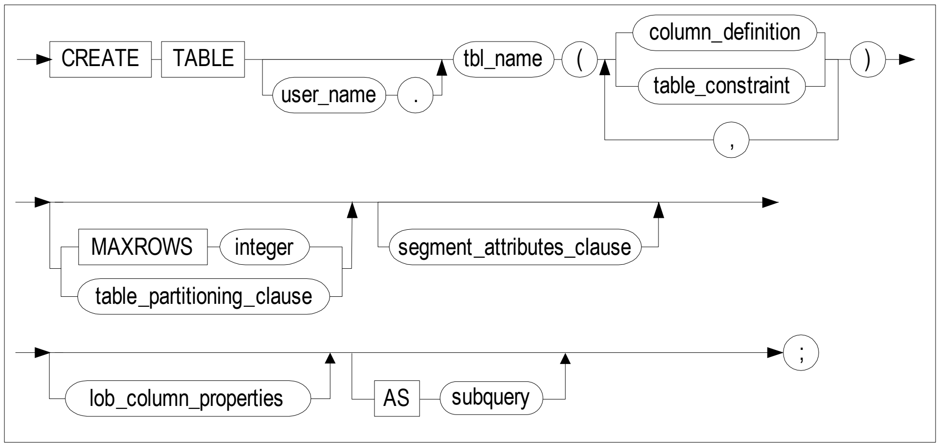
```
iSQL> CONNECT altibase/altibase;
Connect success.
iSQL> CREATE TABLE dept
      (
      id integer,
      name char(10),
      location varchar(40),
      member integer
      );
Create success.
iSQL> GRANT INSERT ON dept TO mylee;
Grant success.
iSQL> GRANT SELECT ON dept TO mylee;
Grant success.
iSQL> CONNECT mylee/mylee;
Connect success.
iSQL> CREATE SYNONYM mylee.my_dept FOR altibase.dept;
Create success.
```

```
iSQL> INSERT INTO my_dept VALUES (1,'rndn1',NULL,4);
1 row inserted.
iSQL> SELECT * FROM my_dept;
MY_DEPT.ID MY_DEPT.NAME MY_DEPT.LOCATION
------------------------------------------------------------------------
MY_DEPT.MEMBER
-----------------
1       rndn1
4
1 row selected.
```

2.21 CREATE TABLE

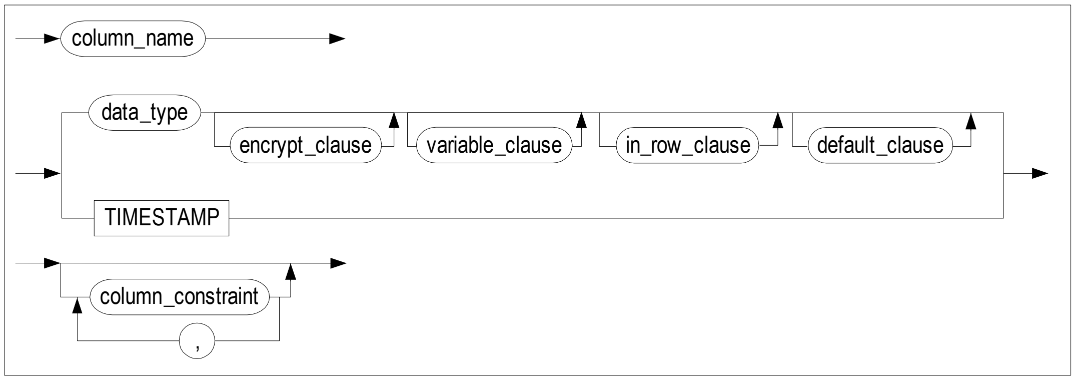
2.21.1 Syntax

2.21.1.1 create_table ::=



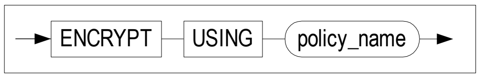
column_definition ::=, *table_constraint ::=*, *table_partitioning_clause ::=*, *segment_attributes_clause ::=*, *lob_column_properties ::=*, *subquery ::=*

2.21.1.2 column_definition ::=

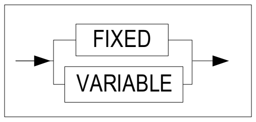


encrypt_clause ::=, *variable_clause ::=*, *in_row_clause ::=*, *default_clause ::=*, *column_constraint ::=*

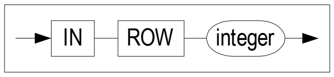
2.21.1.3 encrypt_clause ::=



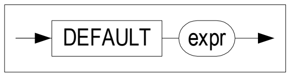
2.21.1.4 variable_clause ::=



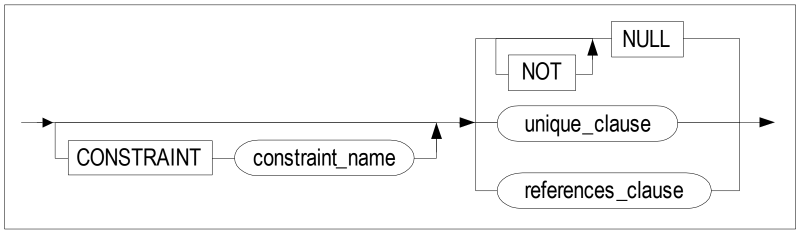
2.21.1.5 in_row_clause ::=



2.21.1.6 default_clause ::=

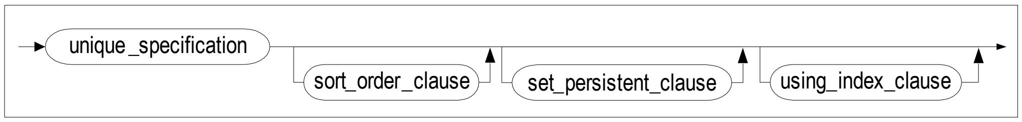


2.21.1.7 column_constraint ::=



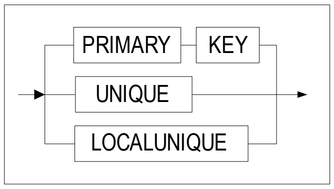
unique_clause ::=, *references_clause ::=*

2.21.1.8 unique_clause ::=

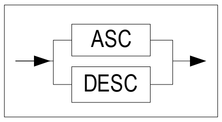


unique_specification ::=, *sort_order_clause ::=*, *set_persistent_clause ::=*, *using_index_clause ::=*

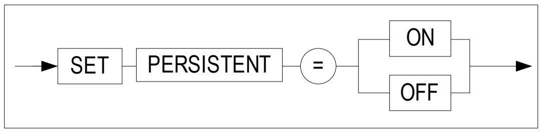
2.21.1.9 unique_specification ::=



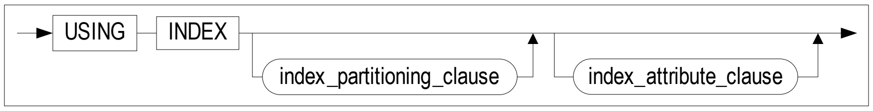
2.21.1.10 sort_order_clause ::=



2.21.1.11 set_persistent_clause ::=

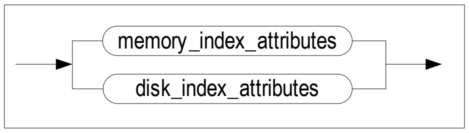


2.21.1.12 using_index_clause ::=



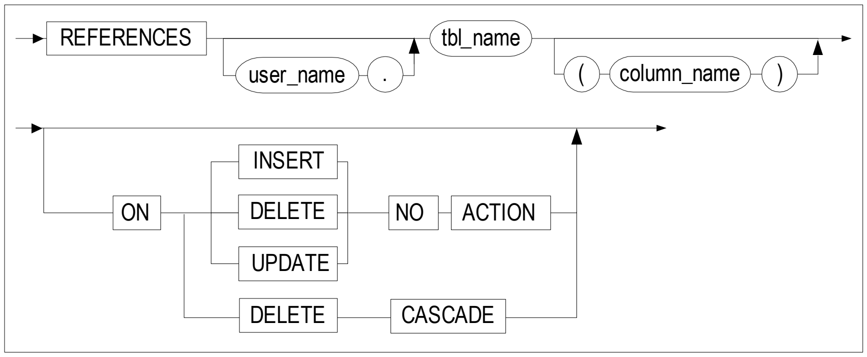
index_partitioning_clause::=, *index_attribute_clause ::=*

2.21.1.13 index_attribute_clause ::=

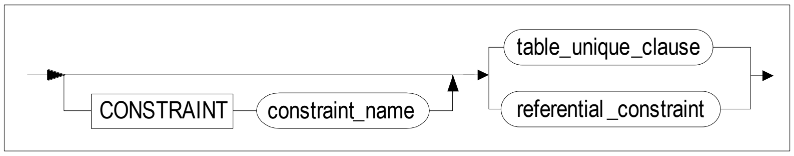


memory_index_attributes::=, *disk_index_attributes::=*

2.21.1.14 references_clause ::=

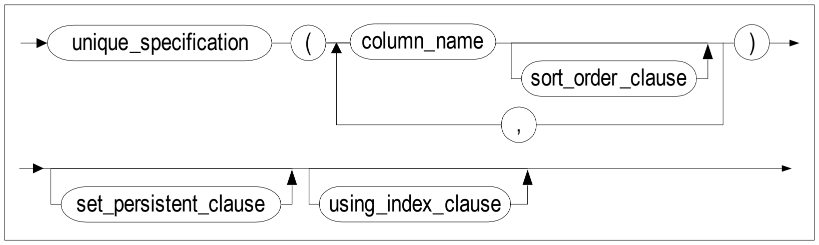


2.21.1.15 table_constraint ::=



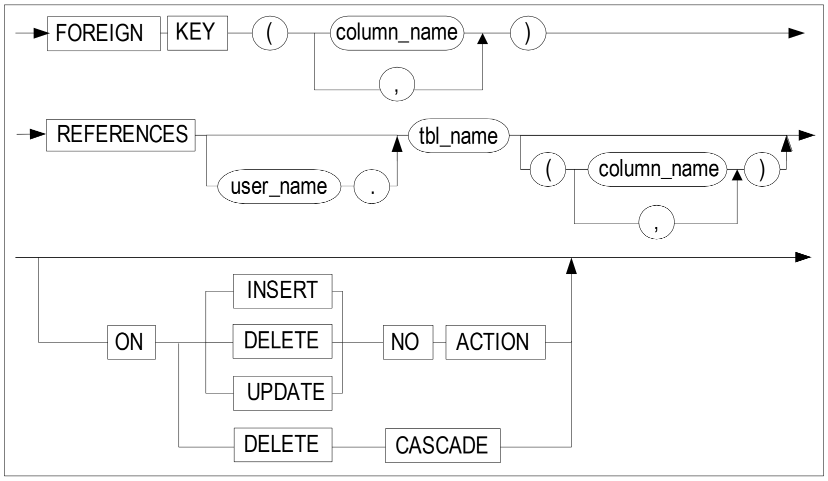
table_unique_clause ::=, *referential_constraint ::=*

2.21.1.16 table_unique_clause ::=

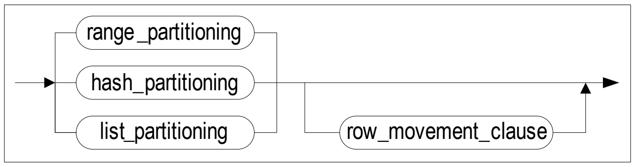


unique_specification ::=, *sort_order_clause ::=*, *set_persistent_clause ::=*, *using_index_clause ::=*

2.21.1.17 referential_constraint ::=

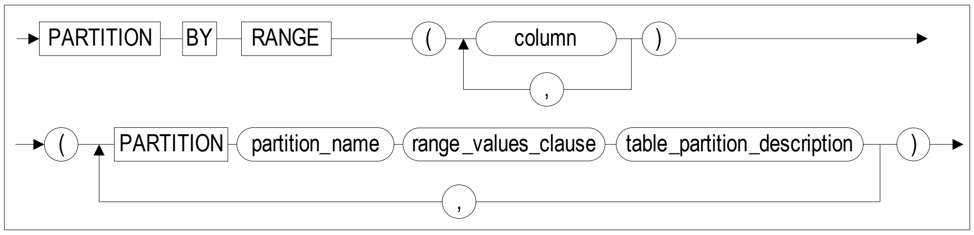


2.21.1.18 table_partitioning_clause ::=



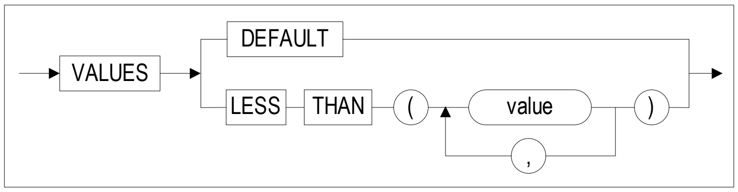
range_partitioning ::=, *hash_partitioning ::=*, *list_partitioning ::=*, *row_movement_clause ::=*

2.21.1.19 range_partitioning ::=

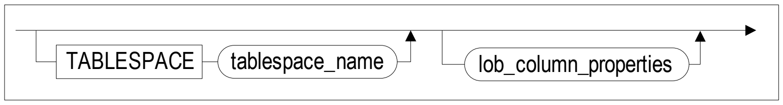


range_values_clause ::=, *table_partition_description ::=*

2.21.1.20 range_values_clause ::=

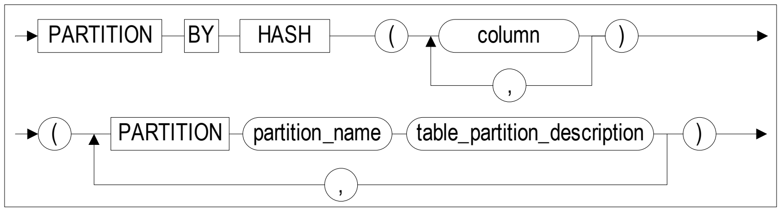


2.21.1.21 table_partition_description ::=



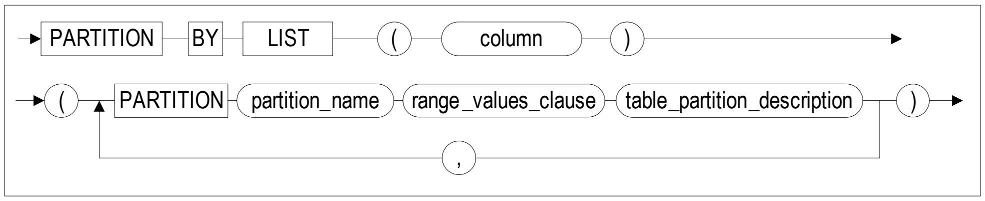
lob_column_properties ::=

2.21.1.22 hash_partitioning ::=



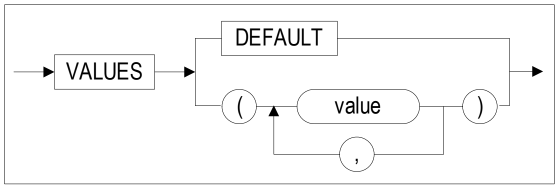
table_partition_description ::=

2.21.1.23 list_partitioning ::=

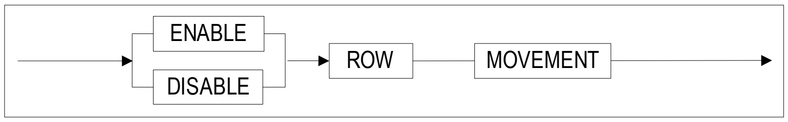


list_values_clause ::=, *table_partition_description ::=*

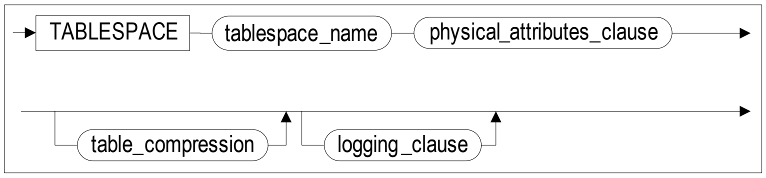
2.21.1.24 list_values_clause ::=



2.21.1.25 row_movement_clause ::=

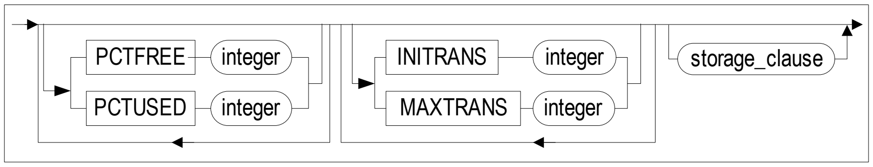


2.21.1.26 segment_attributes_clause ::=

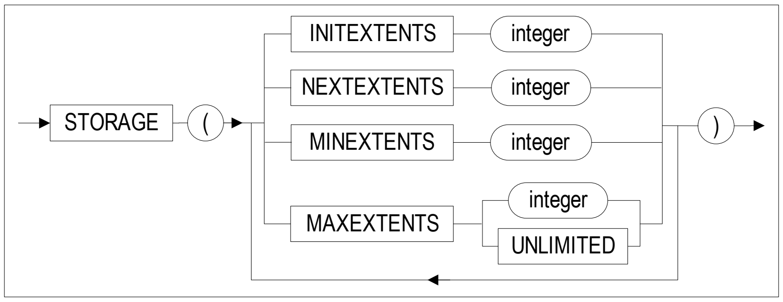


physical_attributes_clause ::=, *table_compression ::=*, *logging_clause ::=*

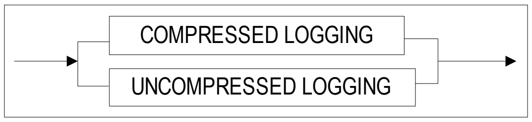
2.21.1.27 physical_attributes_clause ::=



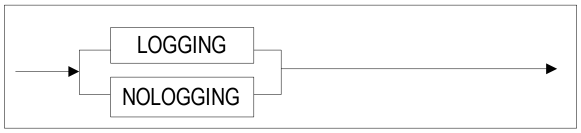
2.21.1.28 storage_clause::=



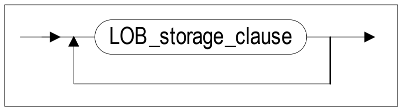
2.21.1.29 table_compression ::=



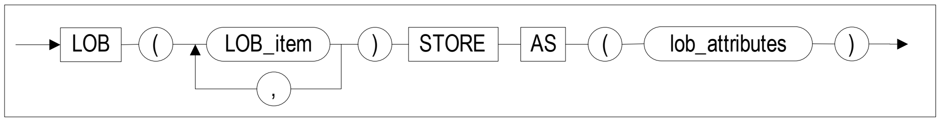
2.21.1.30 logging_clause ::=



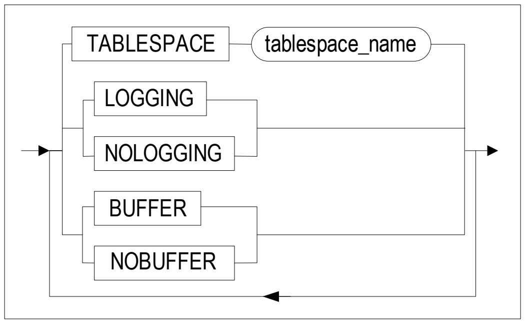
2.21.1.31 lob_column_properties ::=



2.21.1.32 LOB_storage_clause ::=



2.21.1.33 lob_attributes ::=



2.21.2 Prerequisites

Only the SYS user and users having the CREATE TABLE system privilege can execute the CREATE TABLE statement.

2.21.3 Description

This command is used to create a new table with the specified name.

2.21.3.1 user_name

This is used to set the owner of the table. If it is omitted, ALTIBASE HDB will create the table in the schema of the user connected via the current session.

2.21.3.2 tbl_name

This is used to specify the name of the table to be created.

2.21.3.3 column_definition

- DEFAULT

If no DEFAULT clause is specified for a column, the initial value for each row in the column is NULL.

- TIMESTAMP

  A TIMESTAMP column is handled like other data types in many respects. When the data type of a column is specified as TIMESTAMP in a CREATE TABLE statement, a TIMESTAMP value having a size of 8 bytes is generated internally. However, because the value of a TIMESTAMP column is determined by the system, no DEFAULT value can be expressly specified. Furthermore, only one TIMESTAMP column can be created for one table.

2.21.3.4 column_constraint

This is used to specify the constraint for a column when a new table is created. A constraint name can be expressly specified by the user. The LOCALUNIQUE constraint is intended for use with partitioned tables.

- PRIMARY KEY
- UNIQUE
- LOCALUNIQUE
- (NOT) NULL
- Referential Integrity
- PERSISTENT Indexing
- TIMESTAMP

PRIMARY KEY

The value(s) on which a primary key is based must be unique in the table. Additionally, none of the columns which a primary key is based can contain NULL values. Only one primary key can be defined in each table. A primary key can be created on the basis of up to 32 columns.

LOCALUNIQUE

This keyword specifies that each local index must satisfy the UNIQUE constraint.

UNIQUE

A UNIQUE constraint prohibits multiple rows from having the same value in the same column (or combination of columns). However, NULL values are allowed.

A unique constraint and a primary key constraint cannot both be defined for the same column or combination of columns in one table. Additionally, only one unique constraint can be defined for a column or combination of columns. However, these limitations do not pertain to other columns or combinations of columns within the same table. A unique constraint can be created for a combination of up to 32 columns.

NULL

This keyword specifies that the corresponding column can contain NULL values.

NOT NULL

This keyword specifies that the corresponding column cannot contain NULL values.

PERSISTENT

If SET PERSISTENT is not specified, the default is OFF. For more information, please refer to the explanation of the CREATE INDEX statement.

2.21.3.5 table_constraint

This is used to specify the constraint for combination of columns or one column. The following table constraints exist:

- PRIMARY KEY
- UNIQUE
- LOCALUNIQUE
- Referential Integrity

2.21.3.6 using_index_clause

This is used to specify the tablespace in which to store an index that is created to support a constraint.

If any of the PRIMARY KEY, UNIQUE or LOCALUNIQUE constraints are specified, the tablespace in which to store the index of the local index for each index partition can be specified. For more information, please refer to index_partitioning_clause::= in the description of the CREATE INDEX statement.

2.21.3.7 referential_constraint

This clause is used to define a foreign key. The referenced key, that is, the key that resides in another table and is referenced by a foreign key, must either have the UNIQUE constraint applied to it, or be the PRIMARY KEY for the table in which it resides. If the columns of a referenced key are not specified, the primary key for that table is automatically taken as the referenced key.

NO ACTION

This is the default behavior for checking referential integrity.

Normally, when an INSERT, UPDATE, or DELETE operation is performed on a so-called “parent table”, that is, a table that contains a referenced key, the operation is performed only after an integrity check is performed on any so-called “child tables”, that is, tables containing foreign keys that reference the referenced key. The NO ACTION option prevents parent rows from being altered if integrity checking fails, and outputs an error instead.

In this example, when an attempt is made to delete a department from the *departments* table, if the department code is referenced by a record in the *employees* table, the delete attempt will fail and an error will be raised.

```
CREATE TABLE employees (
ENO INTEGER PRIMARY KEY,
DNO INTEGER,
NAME CHAR(10),
FOREIGN KEY(DNO) REFERENCES
departments(DNO) ON DELETE NO ACTION );
```

ON DELETE CASCADE

This option stipulates that if a row in the parent table is deleted, all rows in child tables that have foreign keys that reference this row will also be deleted.

```
CREATE TABLE employees (
ENO INTEGER PRIMARY KEY,
DNO INTEGER, NAME CHAR(10),
FOREIGN KEY(DNO) REFERENCES
departments(DNO) ON DELETE CASCADE );
```

2.21.3.8 MAXROWS

This is used to specify the maximum number of records that can be entered into a table. If an attempt is made to insert records such that the total number of records would be more than that specified using MAXROWS, the insert attempt will fail and an error will be returned.

2.21.3.9 table_partitioning_clause

This is used to create a partitioned table. A partitioned table can be range-partitioned, hash-partitioned or list-partitioned. *row_movement_clause* can also be specified when a partitioned table is created.

2.21.3.10 range_partitioning

This specifies that the table will be partitioned based on ranges of partition key values. It is primarily used with the DATE data type. Because the table is partitioned based on user-specified values, there is no guarantee that the data will be uniformly distributed among the partitions. The range of each partition is determined by setting the maximum value of its range.

Any values exceeding all of the specified ranges, along with any NULL values, will be saved in the default partition. The default partition clause cannot be omitted. A partition key can be defined on the basis of multiple columns.

2.21.3.11 table_partition_description

The tablespace for a partition can be specified. Additionally, if the table contains one or more LOB columns, the attributes for each LOB column can be specified separately.

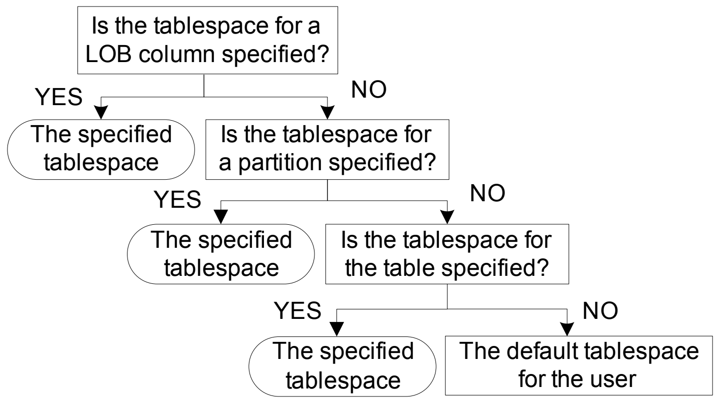
If the tablespace statement is omitted, the partition will be stored in the default tablespace for the table.

Additionally, if the tablespace in which to store a LOB column is not specified, the LOB data will be stored in the tablespace for the partition.

In the following example, the default tablespace for the user is *tbs_05*.

```
CREATE TABLE print_media_demo
(
 product_id INTEGER,
 ad_photo BLOB,
 ad_print BLOB,
 ad_composite BLOB
)
PARTITION BY RANGE (product_id)
(
 PARTITION p1 VALUES LESS THAN (3000) TABLESPACE tbs_01
 LOB (ad_photo) STORE AS (TABLESPACE tbs_02 ),
 PARTITION p2 VALUES DEFAULT
 LOB (ad_composite) STORE AS (TABLESPACE tbs_03)
) TABLESPACE tbs_04;
```

Partition p1 will be stored in the *tbs_01* tablespace because this was expressly specified. However, the *ad_photo* column for this partition will be stored in the *tbs_02* tablespace. Because no tablespace was specified for partition *p2*, which is the default partition, it will be stored in tablespace *tbs_04*, where table *T1* resides. If no tablespace for the table is specified either, it will be stored in the default tablespace, which is *tbs_05*. This is illustrated in the following diagram:


Is the tablespace for a LOB column specified?
YES
NO
The specified tablespace
Is the tablespace for a partition specified?
YES
NO
The specified tablespace
Is the tablespace for the table specified?
YES
NO
The specified tablespace
The default tablespace for the user

2.21.3.12 range_values_clause

This is used to specify the noninclusive upper limit for a range partition. This value must not be set to the same value as that of any other partition.

2.21.3.13 hash_partitioning

This specifies that the table will be partitioned based on hash values corresponding to partition key values. This partitioning scheme is suitable for situations in which the data must be distributed uniformly among the partitions. A partition key can be defined on the basis of multiple columns.

2.21.3.14 list_partitioning

This specifies that the table will be partitioned based on sets of values. The default partition cannot be omitted because any values not specified as belonging to another partition are automatically included in this partition.

When a new partition is defined, the values specified as belonging to that partition are removed from the default partition. This is because values cannot be specified as belonging to more than one partition. Additionally, the partition key for a list-partitioned table can be defined only on the basis of a single column.

2.21.3.15 list_values_clause

The list that defines each list partition must comprise at least one value. A value in one list must not be present in any other list.

2.21.3.16 row_movement_clause

When a record in a partitioned table is updated in a way that changes the data in a column on which the partition key is defined such that the record (row) must be moved to another partition, this clause determines whether to move the record automatically or raise an error. If this clause is omitted, the DISABLE ROW MOVEMENT option (i.e. raise an error) is set by default.

2.21.3.17 CREATE TABLE ... AS SELECT

When creating a table, to copy column attributes and data from other tables into the new table, use the CREATE TABLE ... AS SELECT statement. The number of columns in the new table cannot be set differently from the number of columns retrieved by the AS SELECT clause. Additionally, the data types of the new columns cannot be expressly set, as they are set the same as the original columns from which the data are retrieved.

If no column names are specified for the new table, the names of the original columns will be used as the column names for the new table. If the name of the search target is in the form of an expression, an alias must be provided. This alias will becomes the name of the column in the new table.

2.21.3.18 physical_attributes_clause

This clause is used to specify the tablespace, PCTFREE, PCTUSED, INITRANS, and MAXTRANS. If this clause is specified for a partitioned table, the PCTFREE and PCTUSED values will apply to all of the partitions in the table.

- TABLESPACE Clause

  This clause is used to set the tablespace in which to save the table. The tablespace can only be set using the CREATE TABLE statement; it cannot later be changed using the ALTER TABLE statement. If this clause is omitted, the table will be saved in the DEFAULT TABLESPACE of the user in whose schema the table is being created. The user's DEFAULT TABLESPACE was specified when the user was created. If no DEFAULT TABLESPACE has been specified for the user, the table will be created in the SYSTEM MEMORY DEFAULT TABLESPACE.

  If a UNIQUE or PRIMARY KEY constraint is specified in the CREATE TABLE statement, the index supporting the constraint will be saved in the tablespace in which the table is saved.

- PCTFREE Clause

  This is used to specify the amount of free space that is reserved for use in updating records that have already been saved in a page. Additional records can only be saved into the portion of the page that is not reserved in this way. This value represents the percentage of free space in the page.

For example, for a table in which PCTFREE is set to 20, records can only be inserted into 80% of the space in each page, and the remaining 20% of the page is reserved for use in updating existing records. This value is only meaningful for disk-based tables.

This option must be set to an integer value ranging from 0 to 99, representing the percentage. If this value is not set, the default PCTFREE value is 10. This option only applies to pages that have been assigned to tables.

- PCTUSED Clause

  This is the threshold below which the amount of used space in a page must decrease in order for the page to return to the state in which records can be saved in it again. When the amount of free space in a page falls below the percentage specified in PCTFREE, it becomes impossible to save new records in the page. At this time it is permissible only to update and delete existing records. Once subsequent update or delete operations reduce the percentage of used space in the page below the threshold specified by PCTUSED, it becomes possible to save new records in the page again.

  For example, assuming that PCTUSED has been set to 40, once the percentage of unused space in a page has decreased below the limit specified using PCTFREE (i.e. when the percentage of used space increases beyond 100 - PCTFREE), no more records are saved in that page until the percentage of used space falls to 39%. In other words, new records can be saved in the page only after the percentage of used space falls below 40%. This option only applies to disk-based tables.

  This option must be set to an integer value ranging from 0 to 99, representing the percentage. If this value is not set, the default PCTUSED value is 40. This option only applies to pages that have been assigned to tables.

- INITRANS Clause

  This clause is used to set the initial number of TTS (Touched Transaction Slots). The default value is 2.

- MAXTRANS Clause

  This clause is used to set the maximum number of TTS (Touched Transaction Slots), to which the number of TTS can increase. The default value is 120.

Note: PCTFREE and PCTUSED are used together to optimize performance as follows. In this example, assume that PCTFREE has been set to 20 and PCTUSED to 40.

20% of each page that is allocated to a table is reserved for use in updating existing records. New records can only be saved in the page until the remaining 80% of the space in the page has been filled.

At this point, no more new records can be saved in the page. The only operations that can be performed are update and delete operations on records that already exist in the page. 20% of the page has been reserved for update operation. If enough records are deleted for the amount of used space in the page to fall below 40%, it becomes possible to save new records in the page again.

The values of PCTFREE and PCTUSED are used in this way to determine in a cyclical manner how the space in pages is used.

2.21.3.19 storage_clause

This clause is used to set storage parameters for managing extents in segments.

- INITEXTENTS Clause

  This is used to set the number of extents that are initially allocated when a segment is created. If this is not specified, one extent is allocated by default.

- NEXTEXTENTS Clause

  This is used to set the number of extents that are added to a segment every time the segment is increased in size. If this is not specified, the default value is 1.

- MINEXTENTS Clause

  This is used to set the minimum number of extents in a segment. If this is not specified, the default value is 1.

- MAXEXTENTS Clause

  This is used to set the maximum number of extents in a segment. If this is not specified, there is no upper limit.

2.21.3.20 LOB_storage_clause

In a disk table, LOB column data can be stored in a tablespace other than that in which the table containing the LOB column is stored. However, in a memory table, LOB column data cannot be stored separately from the rest of the table; that is, they can only be stored in the same tablespace as the table.

2.21.4 Considerations

What follows are general considerations to keep in mind when creating tables:

If columns are created larger than their maximum allowable size or smaller than their minimum allowable size, an error occurs. The maximum and minimum sizes vary depending on the data type.

The maximum number of columns in one table is 1024.

A maximum of one primary key can be defined for a table.

For a foreign key constraint, the foreign key and the referenced key must have the same number of columns. For a foreign key constraint, corresponding columns in the foreign key and the referenced key must have the same data types.

The total number of indexes, primary keys and unique keys cannot exceed 64.

When executing a CREATE TABLE ... AS SELECT statement, if the names of the columns to create are specified, the number column names must be the same as the number of columns retrieved using the AS SELECT clause.

When executing a CREATE TABLE ... AS SELECT statement, when the column name is not specified in the CREATE TABLE statement and the name of the column to be retrieved is provided in the form of an expression, an alias name must be specified for the purpose of determining the name of the column in the new table.

The MAXROWS clause is not supported for use with partitioned tables.

For range- and hash-partitioned tables, up to 32 columns can be specified as partition key columns. (This is the same as the upper limit on the number of index columns when an index is created.)

In the event of a system or media fault, the consistency of an index that was created using the NOLOGGING (FORCE/NOFORCE) option cannot be guaranteed. After an index becomes inconsistent, the error message indicating that the index is inconsistent will be raised when the index is accessed. To fix this error, locate the inconsistent index, drop it, and create it again. The consistency of an index can be checked using the V$DISK_BTREE_HEADER performance view.

Just as when executing the CREATE INDEX statement, the tablespace in which a local partitioned index is saved cannot be specified.

2.21.5 Example

Create the following tables.

- Create a table named *employees*, containing columns for employee number, employee first and last name, position, telephone number, department number, salary, gender, birthday, hiring date, and status.

```
iSQL> CREATE TABLE employees(
  eno INTEGER PRIMARY KEY,
  e_lastname CHAR(20) NOT NULL,
  e_firstname CHAR(20) NOT NULL,
  emp_job VARCHAR(15),
  emp_tel CHAR(15),
  dno SMALLINT,
  salary NUMBER(10,2) DEFAULT 0,
  sex CHAR(1),
  birth CHAR(6),
  join_date DATE,
  status CHAR(1) DEFAULT 'H');
Create success.
```

- Create a table named *orders*, containing columns for order number, order date, salesperson, customer number, product number, quantity, estimated delivery date, and status.

```
iSQL> CREATE TABLE orders(
  ono BIGINT,
  order_date DATE,
  eno INTEGER NOT NULL,
  cno BIGINT NOT NULL,
  gno CHAR(10) NOT NULL,
  qty INTEGER DEFAULT 1,
  arrival_date DATE,
  processing CHAR(1) DEFAULT '0', PRIMARY KEY(ono, order_date));
Create success.
```

- Using CREATE TABLE ... AS SELECT

  Create a new table called *dept_1002* and copy the column attributes and data that meet the condition shown from the *employees* table.

```
iSQL> CREATE TABLE dept_1002
```

```
  AS SELECT * FROM employees
  WHERE dno = 1002;
Create success.
```

- Create a table that has a TIMESTAMP type column.

```
iSQL> CREATE TABLE tbl_timestamp(
  i1 TIMESTAMP CONSTRAINT const2 PRIMARY KEY,
  i2 INTEGER,
  i3 DATE,
  i4 BYTE(8));
Create success.
```

The attributes of the table *tbl_timestamp* are as shown below.

```
iSQL> DESC tbl_timestamp;
[ TABLESPACE : SYS_TBS_MEMORY ]
[ ATTRIBUTE ]
------------------------------------------------
NAME           TYPE           IS NULL
------------------------------------------------
I1             TIMESTAMP      FIXED NOT NULL
I2             INTEGER        FIXED
I3             DATE           FIXED
I4             BYTE(8)        FIXED

[ INDEX ]
------------------------------------------------
NAME      TYPE      IS UNIQUE      COLUMN
------------------------------------------------
CONST2    BTREE     UNIQUE         I1 ASC
[ PRIMARY KEY ]
------------------------------------------------
I1
```

If the DEFAULT keyword is used for a timestamp column when performing an INSERT or UPDATE operation, the system time at which the operation is performed will be written to the TIMESTAMP column.

```
iSQL> INSERT INTO tbl_timestamp VALUES(DEFAULT, 2, '02-FEB-01',
BYTE'A1111002');
1 row inserted.
iSQL> UPDATE tbl_timestamp SET i1 = DEFAULT, i2 = 102, i3 = '02-FEB-02',
i4 = BYTE'B1111002' WHERE i2 = 2;
1 row updated.
iSQL> SELECT * FROM tbl_timestamp;
I1                 I2          I3           I4
------------------------------------------------------------------
4E3778C900037AE9  102         02-FEB-2002  B111100200000000
1 row selected.
```

Similarly, if the user does not specify a TIMESTAMP value when performing an INSERT or UPDATE operation on a TIMESTAMP column, the system time at which the operation is performed will be used to perform the INSERT or UPDATE operation.

```
iSQL> INSERT INTO tbl_timestamp(i2, i3, i4) VALUES(4, '02-APR-01',
BYTE'C1111002');
1 row inserted.
iSQL> UPDATE tbl_timestamp SET i2=104, i3='02-APR-02', i4=BYTE'D1111002'
WHERE i2=4;
1 row updated.

iSQL> SELECT * FROM tbl_timestamp;
```

```
I1                I2          I3           I4
-----------------------------------------------------------------
4E3778C900037AE9  102         02-FEB-2002  B111100200000000
4E37794900083702  104         02-APR-2002  D111100200000000
2 rows selected.
```

- Create a table in an expressly specified tablespace.

<Query> Create table *tbl1* in the user *uare1*'s schema. (Assume that no default tablespace was specified when this user was created.)

```
iSQL> CONNECT uare1/rose1;
Connect success.
iSQL> CREATE TABLE tbl1(
  i1 INTEGER,
  i2 VARCHAR(3));
Create success.
```

Note: The table will be created in the system memory default tablespace when no default tablespace has been defined for the user.

<Query> Create the *books* and *inventory* tables in the *user_data* tablespace, which is the default tablespace for the user. The *books* table will contain columns for book number, book name, author, edition, publication year, price, and publication code, and can contain a maximum of two rows. The *inventory* table will contain columns for subscription number, book number, store code, purchase date, quantity, and a character to indicate whether the item has been paid for.

```
iSQL> CREATE TABLE books(
  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 success.

iSQL> CREATE TABLE inventory(
  subscriptionid CHAR(10) PRIMARY KEY,
  isbn CHAR(10) CONSTRAINT fk_isbn REFERENCES books(isbn),
  storecode CHAR(4),
  purchasedate DATE,
  quantity INTEGER,
  paid CHAR(1))
TABLESPACE user_data;
Create success.
```

or

```
iSQL> CREATE TABLE inventory(
  subscriptionid CHAR(10),
  isbn CHAR(10),
  storecode CHAR(4),
  purchasedate DATE,
  quantity INTEGER,
  paid CHAR(1),
PRIMARY KEY(subscriptionid),
CONSTRAINT fk_isbn FOREIGN KEY(isbn) REFERENCES books(isbn))
TABLESPACE user_data;
Create success.
```

- Specify the tablespace for each index partition.

  <Query> Create the partitioned table *T1* having the UNIQUE constraint on column *I1*.

```
CREATE TABLE T1
(
  I1 INTEGER UNIQUE USING INDEX LOCAL
  (
    PARTITION P1_UNIQUE ON P1 TABLESPACE TBS3,
    PARTITION P2_UNIQUE ON P2 TABLESPACE TBS2,
    PARTITION P3_UNIQUE ON P3 TABLESPACE TBS1
  )
)
PARTITION BY RANGE (I1)
(
  PARTITION P1 VALUES LESS THAN (100),
  PARTITION P2 VALUES LESS THAN (200) TABLESPACE MEM_TBS1,
  PARTITION P3 VALUES DEFAULT TABLESPACE MEM_TBS2
) TABLESPACE SYS_TBS_DISK_DATA;
```

- Range Partitioning

  <Query 1> Create the table *range_sales*, partitioning the year 2006 into respective quarters as shown below.

```
CREATE TABLE range_sales
(
  prod_id NUMBER(6),
  cust_id NUMBER,
  time_id DATE
)
PARTITION BY RANGE (time_id)
(
  PARTITION Q1_2006 VALUES LESS THAN (TO_DATE('01-APR-2006')),
  PARTITION Q2_2006 VALUES LESS THAN (TO_DATE('01-JUL-2006')),
  PARTITION Q3_2006 VALUES LESS THAN (TO_DATE('01-OCT-2006')),
  PARTITION Q4_2006 VALUES LESS THAN (TO_DATE('01-JAN-2007')),
  PARTITION DEF VALUES DEFAULT
) TABLESPACE SYS_TBS_DISK_DATA;
```

  <Query 2> Create a partitioned table, specifying the tablespace for some of the partitions.

```
CREATE TABLE T1
(
  I1 INTEGER,
  I2 INTEGER
)
PARTITION BY RANGE (I1)
(
  PARTITION P1 VALUES LESS THAN (100),
  PARTITION P2 VALUES LESS THAN (200) TABLESPACE TBS1,
  PARTITION P3 VALUES DEFAULT TABLESPACE TBS2
) TABLESPACE SYS_TBS_DISK_DATA;
```

  <Query 3> Create a partitioned table in which multiple columns are used as the partition key.

```
CREATE TABLE T1
(
  I1 DATE,
  I2 INTEGER
)
PARTITION BY RANGE (I1, I2)
(
  PARTITION P1 VALUES LESS THAN (TO_DATE('01-JUL-2006'), 100),
  PARTITION P2 VALUES LESS THAN (TO_DATE('01-JAN-2007'), 200),
  PARTITION P3 VALUES DEFAULT
) TABLESPACE SYS_TBS_DISK_DATA;
```

<Query 4> Create a partitioned table in which the data are moved automatically when required.

```
CREATE TABLE T1
(
  I1 INTEGER,
  I2 INTEGER
)
PARTITION BY LIST (I1)
(
  PARTITION P1 VALUES (100, 200),
  PARTITION P2 VALUES (150, 250),
  PARTITION P3 VALUES DEFAULT
) ENABLE ROW MOVEMENT TABLESPACE SYS_TBS_DISK_DATA;
```

- List Partitioning

  <Query> Create the table *list_customers*, which is list-partitioned on the basis of the *nls_territory* column into the *asia* partition for the values 'CHINA' and 'THAILAND', the *europe* partition for the values 'GERMANY', 'ITALY' and 'SWITZERLAND', the *west* partition for the value 'AMERICA', the *east* partition for the value 'INDIA', and the default partition for any other values.

```
CREATE TABLE list_customers
(
  customer_id NUMBER(6),
  cust_first_name VARCHAR(20),
  cust_last_name VARCHAR(20),
  nls_territory VARCHAR(30),
  cust_email VARCHAR(30)
)
PARTITION BY LIST (nls_territory)
(
  PARTITION asia VALUES ('CHINA', 'THAILAND'),
  PARTITION europe VALUES ('GERMANY', 'ITALY', 'SWITZERLAND'),
  PARTITION west VALUES ('AMERICA'),
  PARTITION east VALUES ('INDIA'),
  PARTITION rest VALUES DEFAULT
) TABLESPACE SYS_TBS_DISK_DATA;
```

- Hash Partitioning

  <Query > Create a table that is hash-partitioned into 4 partitions based on *product_id*.

```
CREATE TABLE hash_products
(
  product_id NUMBER(6),
  product_name VARCHAR(50),
  product_description VARCHAR(2000)
)
PARTITION BY HASH (product_id)
(
  PARTITION p1,
  PARTITION p2,
  PARTITION p3,
  PARTITION p4
) TABLESPACE SYS_TBS_DISK_DATA;
```

<Query> Create a table in which the LOB data are stored in separate tablespaces; specifically, in which the LOB data in the *image1* column are stored in the *lob_data1* tablespace and the LOB data in the *image2* column are stored in the *lob_data2* tablespace.

```
CREATE TABLE lob_products
(
  product_id integer,
  image1 BLOB,
  image2 BLOB
) TABLESPACE SYS_TBS_DISK_DATA
LOB(image1) STORE AS ( TABLESPACE lob_data1 )
LOB(image2) STORE AS ( TABLESPACE lob_data2 );
```

- Creating a table in which the extents in the segment are managed

<Query> Create the table *local_tbl* in the *usertbs* disk tablespace. Allocate 10 extents to the table when it is created and specify that 1 extent is to be added whenever the size of the table needs to be increased.

```
iSQL> CREATE TABLE local_tbl ( i1 INTEGER, i2 VARCHAR(32) )
   TABLESPACE usertbs
   STORAGE ( INITEXTENTS 10 NEXTEXTENTS 1 );
Create success.
```

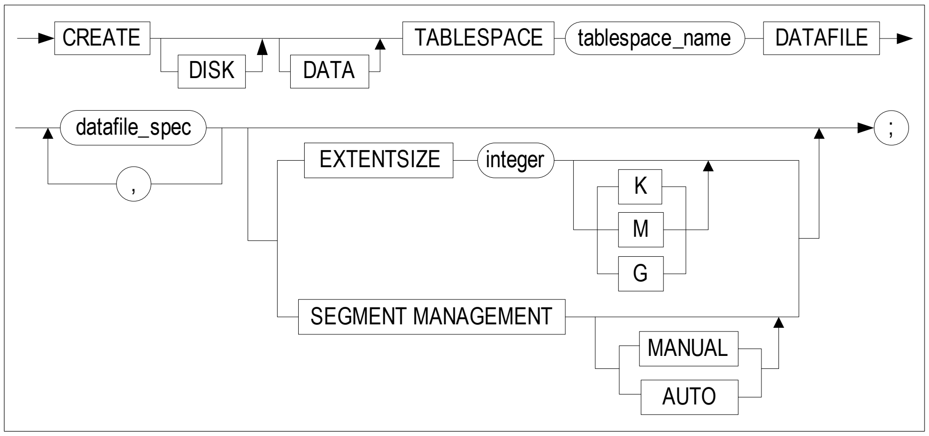
<Query> Create the table *local_tbl* in the *usertbs* disk tablespace. Specify that the minimum number of extents in the table is 3, which is the same number that are allocated to the table when it is created, and limit the maximum number of extents to 100.

```
iSQL> CREATE TABLE local_tbl ( i1 INTEGER, i2 VARCHAR(32) )
   TABLESPACE usertbs
   STORAGE ( INITEXTENTS 3 MINEXTENTS 3 MAXEXTENTS 100 );
Create success.
```

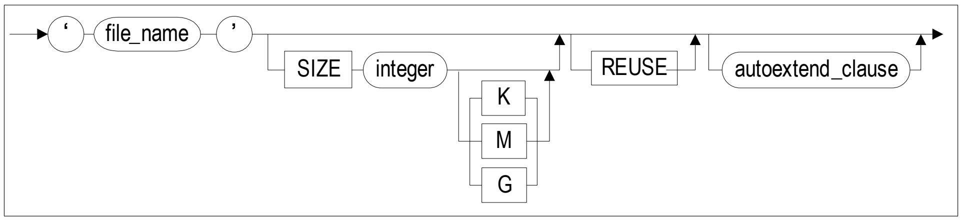
2.22 CREATE DISK TABLESPACE

2.22.1 Syntax

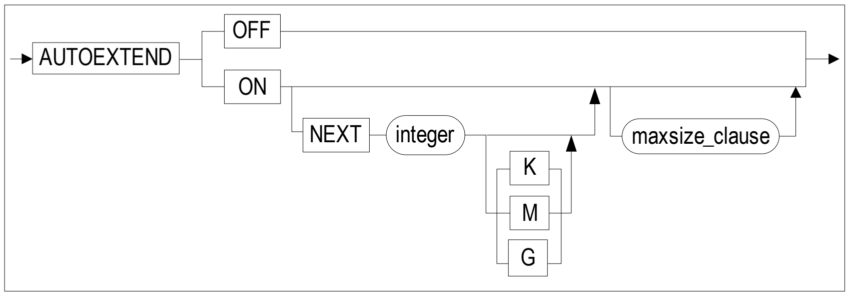
2.22.1.1 create_disk_tablespace ::=



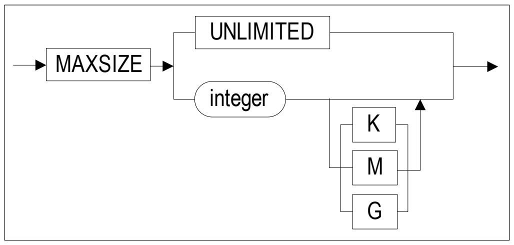
2.22.1.2 datafile_spec ::=



2.22.1.3 autoextend_clause ::=



2.22.1.4 maxsize_clause ::=



2.22.2 Prerequisites

Only the SYS user and users having the CREATE TABLESPACE system privilege can create tablespaces.

2.22.3 Description

The CREATE DISK TABLESPACE statement is used to create a disk tablespace, in which database objects can be permanently stored within the database. Tablespaces created using this command can be used to hold tables and indexes.

2.22.3.1 DISK

This keyword is used to specify that the tablespace to be created will be a disk tablespace. A disk tablespace is created even when the CREATE TABLESPACE statement is executed without the DISK keyword.

2.22.3.2 DATA

This keyword is used to specify that the tablespace to be created will be used to store user data. A data tablespace is created even when the CREATE TABLESPACE statement is executed without the DATA keyword.

2.22.3.3 tblspace_name

This is used to specify the name of the tablespace to be created.

2.22.3.4 DATAFILE datafile_spec

This is used to specify the attributes of the data file constituting the tablespace.

2.22.3.5 EXTENTSIZE clause

This is used to specify the size of an extent, which is a collection of pages. This cannot be changed after the tablespace is created. The default unit used to specify the size of an extent is kB (kilobytes, expressed as “K“), but it is also permissible to use MB (megabytes, expressed as “M“) or GB (gigabytes, expressed as “G“).

If size of the extents in the tablespace is not specified, the extents will be the default size, which is 64 times the size of a single page. When specifying the size of the extents in the tablespace, the extent size must be set to a multiple of the size of a single page. If the extent size is set to a value other than a multiple of the page size, the extent size will be rounded to the closest value internally so that it is a multiple of the page size.

Additionally, the size of an extent must be at least five times the size of a single page. In other words, because the size of a single page is 8kB, the extent size must be set to at least 40kB.

2.22.3.6 SEGMENT MANAGEMENT Clause

This is used to specify how segments are to be managed in the disk tablespace to be created. This clause is optional. If this option is not specified, segments in the newly created disk space will be managed according to the setting of the DEFAULT_SEGMENT_MANAGEMENT_TYPE property in the altibase.properties file. (The default value for this property is AUTO.)

- MANUAL: This specifies that segments are created on the basis of a so-called “free list“ method of managing available space in the user tablespace.
- AUTO: This specifies that segments are created on the basis of a so-called “bitmap index“ method of managing available space in the user tablespace.

2.22.3.7 file_name

This is used to specify the absolute path and name of the data file to be created.

2.22.3.8 SIZE clause

This is used to specify the size of the data file. If this clause is omitted, the data file will be the default size, which is 100 megabytes. This default file size can be changed by setting the USER_DATA_FILE_INIT_SIZE property as desired.

The size is specified by providing an integer followed by one of the following units: kB (kilobytes, expressed as “K”), MB (megabytes, expressed as “M”) or GB (gigabytes, expressed as “G”). If no units are specified, the default unit is kilobytes.

2.22.3.9 REUSE

This is used to specify whether or not to reuse an existing data file. If a file with the name specified in *file_name* exists, the REUSE option must be specified. Note however that if an existing file is reused, the original contents of the file will be lost, so care must be taken in order to prevent data loss.

If the REUSE option is specified but no file with the name specified in *file_name* exists, this option will be ignored, and a new file will be created.

2.22.3.10 autoextend_clause

This is used to specify whether to automatically increase the size of the data file when it fills up, and the maximum size to which it can increase. If this clause is omitted, AUTOEXTEND is disabled by default.

- ON

This enables the AUTOEXTEND option for the file.

- OFF

  This disables the AUTOEXTEND option for the file.

- NEXT

  This is used to specify the amount by which the size of the file will increase when it is automatically increased in size.

  If AUTOEXTEND is enabled but this value is not set, the default NEXT value is the value set in the USER_DATA_FILE_NEXT_SIZE property in the altibase.properties file.

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

- maxsize_clause

  This is used to specify the maximum size to which the data file can increase. If AUTOEXTEND is enabled but this value is not set, the default is the value set using the USER_DATA_FILE_MAX_SIZE property in the altibase.properties file.

  It can be expressed in kB (kilobytes, expressed as "K"), MB (megabytes, expressed as "M") or GB (gigabytes, expressed as "G"). If no units are specified, the default unit is kilobytes.

- UNLIMITED

  This is used to indicate that there is no upper limit to the size to which the file can increase. If this option is used, the actual maximum size of the file will be determined by the operating system or by the amount of available space in the file system.

2.22.4 Examples

<Query>Create the *user_data* tablespace, comprising three data files. Specify that segments are to be managed using the "free list" method.

```
iSQL> CREATE TABLESPACE user_data
  DATAFILE '/tmp/tbs1.user' SIZE 10M,
  '/tmp/tbs2.user' SIZE 10M,
  '/tmp/tbs3.user' SIZE 10M
  SEGMENT MANAGEMENT MANUAL;
Create success.
```

<Query> Create the *user_data* tablespace, which has an initial size of 10MB, comprises the *tbs.user* data file (in which the tables and indexes in this tablespace will be stored), and extends automatically.

```
iSQL> CREATE TABLESPACE user_data DATAFILE '/tmp/tbs.user' SIZE 10M AUTOEX-
TEND ON;
Create success.
```

<Query> Create the *user_data* tablespace, which can increase in size up to 100MB in 500kB increments.

```
iSQL> CREATE TABLESPACE user_data
  DATAFILE '/tmp/tbs.user' SIZE 500K REUSE
  AUTOEXTEND ON NEXT 500K MAXSIZE 100M
  EXTENTSIZE 250K;
Create success.
```

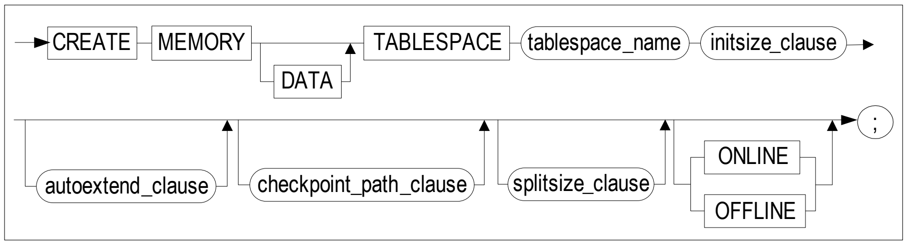
<Query> Create the *user_data* tablespace comprising the *tbs.user* data file, which does not automatically increase in size.

```
iSQL> CREATE TABLESPACE user_data
  DATAFILE '/tmp/tbs.user' AUTOEXTEND OFF;
Create success.
```

2.23 CREATE MEMORY TABLESPACE

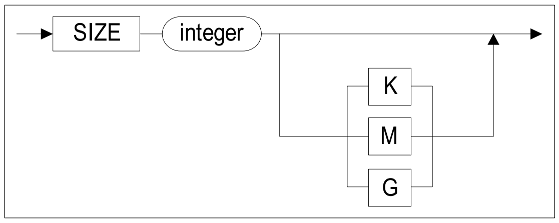
2.23.1 Syntax

2.23.1.1 create_memory_tablespace ::=

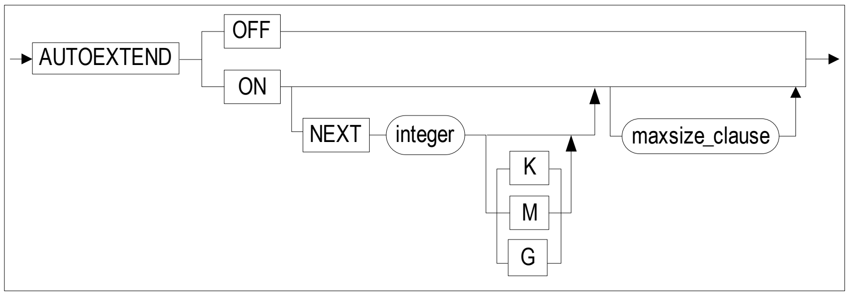


initsize_clause ::=, *autoextend_clause ::=*, *checkpoint_path_clause ::=*, *splitsize_clause ::=*

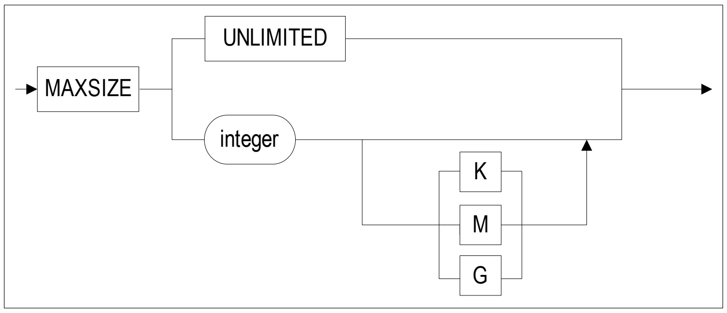
2.23.1.2 initsize_clause ::=



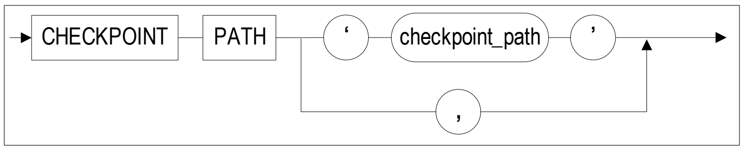
2.23.1.3 autoextend_clause ::=



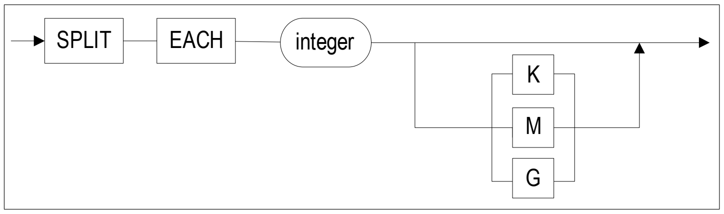
2.23.1.4 maxsize_clause ::=



2.23.1.5 checkpoint_path_clause ::=



2.23.1.6 splitsize_clause ::=



2.23.2 Prerequisites

Only the SYS user and users having the CREATE TABLESPACE system privilege can create tablespaces.

2.23.3 Description

The CREATE MEMORY TABLESPACE statement is used to create a memory data tablespace, in which database objects can be stored within the database. Tablespaces created using this command can be used to hold memory tables.

2.23.3.1 MEMORY

This keyword is used to specify that the tablespace to be created will be a memory tablespace.

2.23.3.2 DATA

This keyword is used to specify that the tablespace to be created will be used to store user data. A data tablespace is created even when the CREATE TABLESPACE statement is executed without the

DATA keyword.

2.23.3.3 tablespace_name

This is used to specify the name of the tablespace to create.

2.23.3.4 initsize_clause

This is used to specify the initial size of the tablespace to create.

SIZE

This is used to specify the initial size of the tablespace.

The initial size of a memory tablespace must be a multiple of the default allocation size. (i.e. the number of pages specified in the EXPAND_CHUNK_PAGE_COUNT property * the size of one page in memory tablespace (32kB)).

For example, if the EXPAND_CHUNK_PAGE_COUNT property is set to 128, the default allocation size would be 128 * 32 = 4MB. Therefore, the initial size must be set to a multiple of 4MB.

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

2.23.3.5 autoextend_clause

This is used to specify whether the tablespace automatically increases in size when it fills up, and the maximum size to which it can increase. If this clause is omitted, AUTOEXTEND is disabled by default.

ON

This enables the AUTOEXTEND option.

OFF

This disables the AUTOEXTEND option. This is the default.

NEXT

This is used to specify the increment by which the tablespace increases in size when it is automatically increased in size.

Note that this size must be a multiple of the default allocation size (the number of pages specified in the EXPAND_CHUNK_PAGE_COUNT property * the size of one page in memory tablespace (32kB)).

If AUTOEXTEND is enabled but this value is not set, the default is the value set using the EXPAND_CHUNK_PAGE_COUNT property in the altibase.properties file.

If AUTOEXTEND is OFF, this value is irrelevant.

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

maxsize_clause

This is used to specify the maximum size to which a tablespace can increase when it automatically increases in size. If AUTOEXTEND is enabled but this value is not set, the default value is UNLIMITED.

If AUTOEXTEND is OFF, this value is meaningless. The size can be specified in kB (kilobytes, expressed as "K"), MB (megabytes, expressed as "M") or GB (gigabytes, expressed as "G"). If no units are specified, the default unit is kilobytes.

UNLIMITED

This is used to indicate that there is no upper limit to the size to which the tablespace can increase.

If this option is used, the tablespace will automatically increase in size up to the point at which the total size of all memory tablespaces and all volatile tablespaces in the system reaches the size specified in the MEM_MAX_DB_SIZE property in the altibase.properties file.

2.23.3.6 checkpoint_path_clause

To ensure the durability of the data in memory tablespaces, the data must be saved in files. These memory tablespace data storage files are known as "checkpoint images".

The *checkpoint_path* clause is used to specify the checkpoint path, that is, the path and directory where these image files are stored.

If no checkpoint path is provided, the path specified in the MEM_DB_DIR property is used as the default path.

checkpoint_path

This is used to specify the location at which a checkpoint image is stored when checkpointing is performed for the memory tablespace. It is permissible to specify more than one path, which is helpful in distributing the disk I/O costs incurred when checkpointing is performed and when the contents of tablespaces are read from disk at startup.

2.23.3.7 split_each_clause

This clause is used to split checkpoint files into smaller files. This is useful when the size of the memory tablespace exceeds the maximum file size supported by the operating system, or in order to distribute I/O costs. The size of the resulting files can be specified by the user. If this size is not specified, the default split size specified in the DEFAULT_MEM_DB_FILE_SIZE property is used.

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

2.23.4 Examples

<Query 1> Create a user-defined memory data tablespace that is initially 512MB in size and does not automatically increase in size. (The checkpoint image is stored in the path specified in the MEM_DB_DIR property. If the checkpoint image file is split into multiple files, the size of those files will be the same as the value set in the DEFAULT_MEM_DB_FILE_SIZE property.)

```
iSQL> CREATE MEMORY DATA TABLESPACE user_data SIZE 512M;
Create success.
```

<Query 2> Create a user-defined memory data tablespace that is initially 512MB in size and increases in size in 128MB increments[1]. (The checkpoint image is stored in the path specified in the MEM_DB_DIR property. If the checkpoint image file is split into multiple files, the size of those files will be the same as the value set in the DEFAULT_MEM_DB_FILE_SIZE property.)

```
iSQL> CREATE MEMORY DATA TABLESPACE user_data
  SIZE 512M
  AUTOEXTEND ON NEXT 128M;
Create success.
```

<Query 3> Create a user-defined memory data tablespace that is initially 512MB in size and increases in size in 128MB increments up to a maximum size of 1GB. (The checkpoint image is stored in 3 directories, and the size of each of the checkpoint image files is 256MB.)

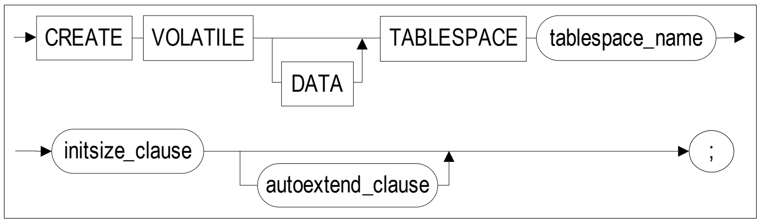
```
iSQL> CREATE MEMORY DATA TABLESPACE user_data
  SIZE 512M AUTOEXTEND ON NEXT 128M MAXSIZE 1G
  CHECKPOINT PATH '/dbs/path1', '/dbs/path2', '/dbs/path3'
  SPLIT EACH 256M;
Create success.
```

1. If the maximum size of the tablespace is not specified using the MAXSIZE clause, it defaults to UNLIMITED. In this case, the tablespace can increase in size as long as the combined size of all of the memory tablespaces and volatile tablespaces that exist in the system does not exceed the amount of memory specified in the MEM_MAX_DB_SIZE property.

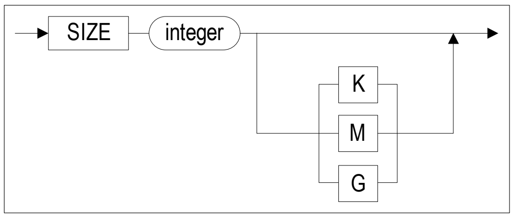
2.24 CREATE VOLATILE TABLESPACE

2.24.1 Syntax

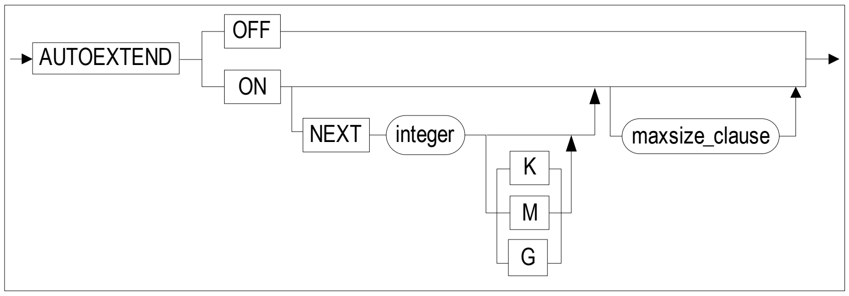
2.24.1.1 create_tablespace ::=



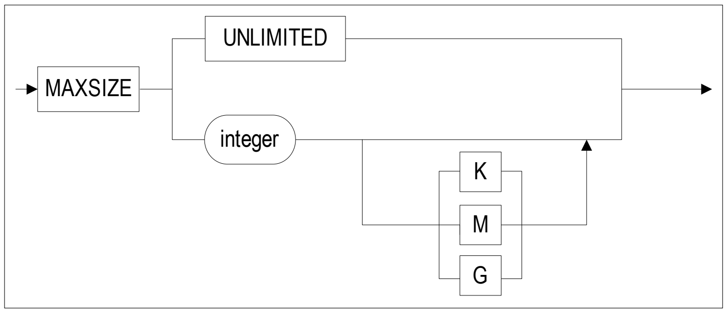
2.24.1.2 initsize_clause ::=



2.24.1.3 autoextend_clause ::=



2.24.1.4 maxsize_clause ::=



2.24.2 Prerequisites

Only the SYS user and users having the CREATE TABLESPACE system privilege can create tablespaces.

2.24.3 Description

The CREATE VOLATILE TABLESPACE statement is used to create a volatile tablespace for storing database objects in the database. Tablespaces created using this command are used to hold volatile tables.

2.24.3.1 VOLATILE

This keyword is used to specify that the tablespace to be created will be a volatile tablespace.

2.24.3.2 DATA

This keyword is used to specify that the tablespace to be created will be used to store user data. A data tablespace is created even when the CREATE TABLESPACE statement is executed without the DATA keyword.

2.24.3.3 tablespace_name

This is used to specify the name of the tablespace to create.

2.24.3.4 initsize_clause

This is used to specify the initial size of the tablespace to create.

SIZE

This is used to specify the initial size of a tablespace. The size can be specified in kB (kilobytes, expressed as “K”), MB (megabytes, expressed as “M”) or GB (gigabytes, expressed as “G”). If no units are specified, the default unit is kilobytes.

The initial size of a memory tablespace must be a multiple of the default allocation size. (i.e. the number of pages specified in the EXPAND_CHUNK_PAGE_COUNT property * the size of one page in memory tablespace (32kB)).

For example, if the EXPAND_CHUNK_PAGE_COUNT property is set to 128, the default allocation size would be 128 * 32 = 4MB. Therefore, the initial size must be set to a multiple of 4MB.

2.24.3.5 autoextend_clause

This is used to specify whether the tablespace automatically increases in size when necessary, and the maximum size to which it can increase. If this clause is omitted, AUTOEXTEND is disabled by default.

ON

This enables the AUTOEXTEND option.

OFF

This disables the AUTOEXTEND option. The is the default.

NEXT

This is used to specify the increment by which the tablespace increases in size when it is automatically increased in size.

Note that the initial size of the memory tablespace must be a multiple of this increment.

If AUTOEXTEND is enabled but this value is not set, the default is the value set using the EXPAND_CHUNK_PAGE_COUNT property in the altibase.properties file. If AUTOEXTEND is OFF, this value is meaningless.

The size can be specified in kB (kilobytes, expressed as “K”), MB (megabytes, expressed as “M”) or GB (gigabytes, expressed as “G”). If no units are specified, the default unit is kilobytes.

maxsize_clause

This is used to specify the maximum size to which a tablespace can increase when it automatically increases in size.

If AUTOEXTEND is enabled but this value is not set, the default value is UNLIMITED. If AUTOEXTEND is OFF, this value is meaningless.

The size can be specified in kB (kilobytes, expressed as “K”), MB (megabytes, expressed as “M”) or GB (gigabytes, expressed as “G”). If no units are specified, the default unit is kilobytes.

UNLIMITED

This is used to indicate that there is no upper limit to the size to which the tablespace can increase.

If this option is used, the tablespace will automatically increase in size up to the point at which the total size of all memory tablespaces and all volatile tablespaces in the system reaches the size specified in the MEM_MAX_DB_SIZE property in the altibase.properties file.

2.24.4 Examples

<Query 1> Create a user-defined volatile data tablespace that is initially 512MB in size and does not automatically increase in size.

```
iSQL> CREATE VOLATILE DATA TABLESPACE user_data SIZE 512M;
Create success.
```

<Query 2> Create a user-defined volatile data tablespace that is initially 512MB in size and increases in size in 128MB increments.

```
iSQL> CREATE VOLATILE DATA TABLESPACE user_data
  SIZE 512M
  AUTOEXTEND ON NEXT 128M;
Create success.
```

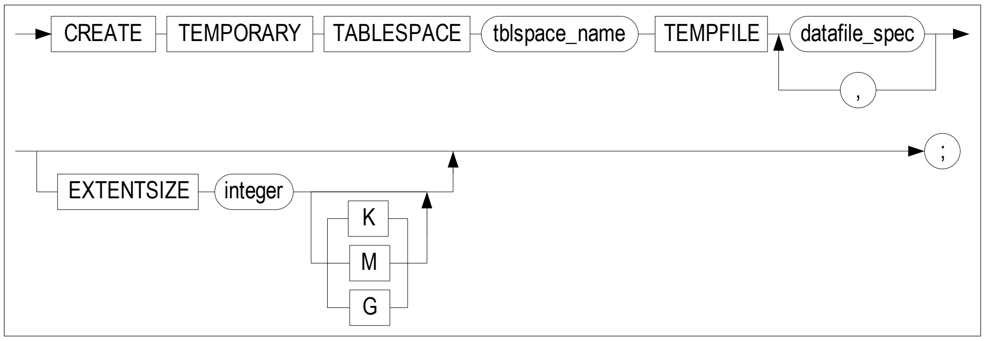
2.24.5 Restriction

Volatile tablespaces cannot be used to hold LOB type data.

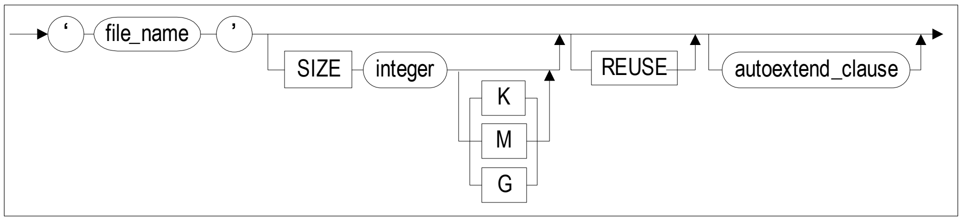
2.25 CREATE TEMPORARY TABLESPACE

2.25.1 Syntax

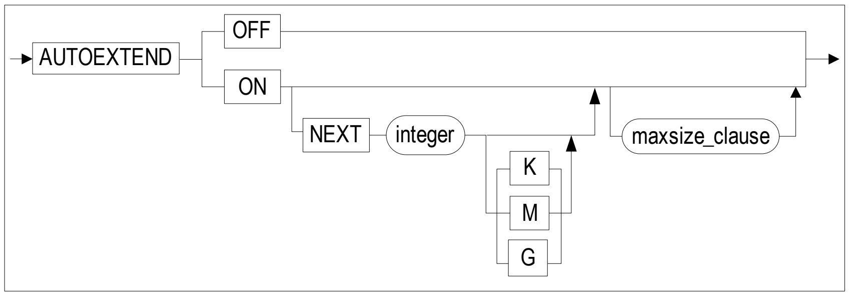
2.25.1.1 create_temporary_tablespace ::=



2.25.1.2 datafile_spec ::=



2.25.1.3 autoexetend_clause ::=



2.25.2 Prerequisites

Only the SYS user and users having the CREATE TABLESPACE system privilege can create temporary tablespaces.

2.25.3 Description

This command is used to create a temporary tablespace for storing temporary results that are used only for the duration of a session. The temporary tablespace will be created in disk space. Data in temporary tablespaces are stored in data files.

To create a tablespace in which to store database objects permanently, use the CREATE DISK TABLESPACE statement.

2.25.3.1 tblspace_name

This is used to specify the name of the temporary tablespace to be created.

2.25.3.2 TEMPFILE datafile_space

This clause is used to specify the temporary file(s) constituting the temporary tablespace.

2.25.4 Examples

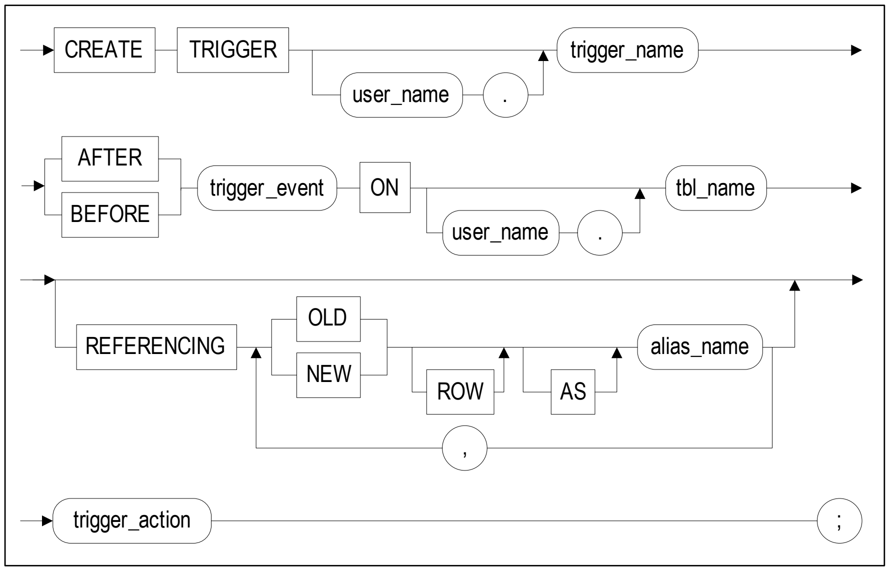
<Query> Create the temporary tablespace *temp_data*, which is 5 MB in size and is constituted by the *tbs.temp* data file.

```
iSQL> CREATE TEMPORARY TABLESPACE temp_data
  TEMPFILE '/tmp/tbs.temp' SIZE 5M
  AUTOEXTEND ON;
Create success.
```

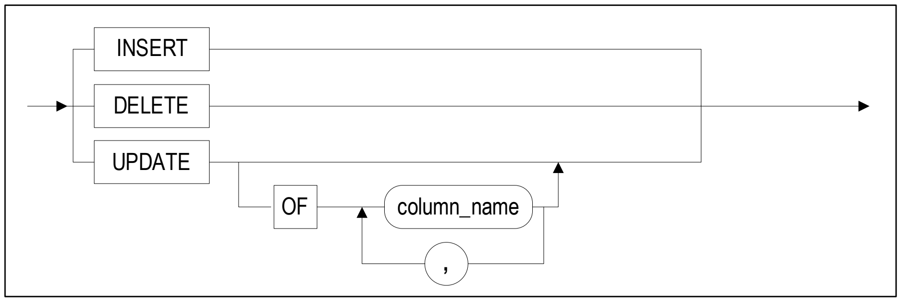
2.26 CREATE TRIGGER

2.26.1 Syntax

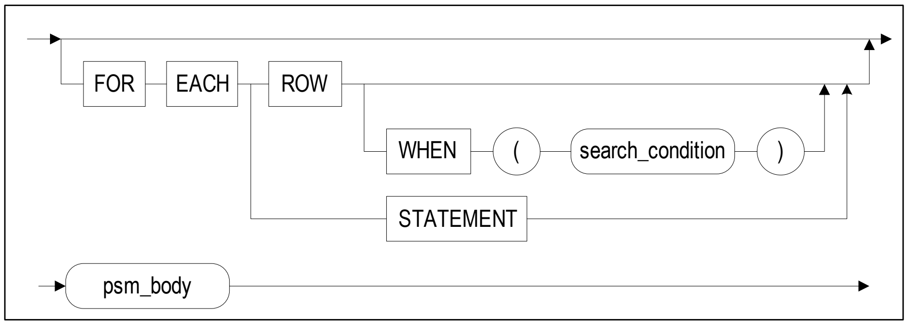
2.26.1.1 create_trigger ::=



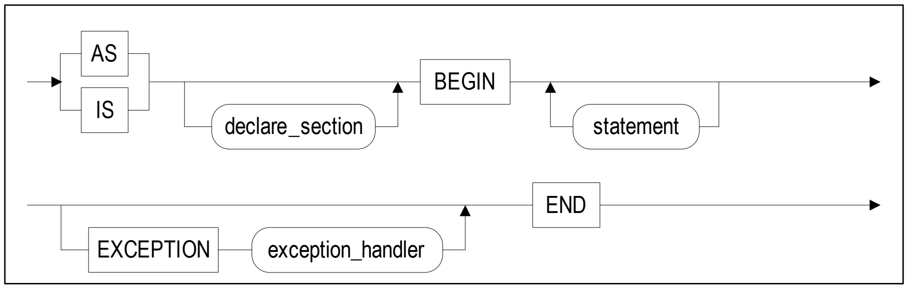
2.26.1.2 trigger_event ::=



2.26.1.3 trigger_action::=



2.26.1.4 psm_body::=



2.26.2 Prerequisites

Only the SYS user and users having the CREATE TRIGGER system privilege can create triggers. Additionally, it is necessary to be the SYS user or have the CREATE ANY TRIGGER system privilege in order to create a trigger in another user's schema.

2.26.3 Description

This command is used to create a trigger having the specified name.

2.26.3.1 user_name

This is used to specify the name of the owner of the trigger to be created. If this value is omitted, ALTIBASE HDB will assume that the trigger is to be created in the current user's schema.

2.26.3.2 trigger_name

This is used to specify the name of the trigger to be created.

2.26.3.3 AFTER

Choose the AFTER option to fire the trigger after the execution of *trigger_event*, that is, the event that caused the trigger to fire.

2.26.3.4 BEFORE

Choose the BEFORE option to fire the trigger before the execution of *trigger_event*, that is, the event that caused the trigger to fire.

2.26.3.5 trigger_event

This is an event that changes the data in a table and causes the trigger to fire. Note that in order to preserve database integrity, DML operations that change the data and are initiated by the replication Receiver thread in a replication target table (i.e. a table that is cited in an active replication object) are not treated as a trigger event (i.e. do not fire the trigger on the table).

The following three types of DML statements can be designated as *trigger_events*:

DELETE

The DELETE option specifies that the trigger will fire whenever a DELETE statement removes a row or rows from the table.

INSERT

The INSERT option specifies that the trigger will fire whenever an INSERT statement inserts data into the table.

UPDATE

The UPDATE option specifies that the trigger will fire whenever an UPDATE statement changes data in the table. The optional OF clause further specifies that the trigger will fire only when an UPDATE statement changes one of the columns specified therein.

Note that the UPDATE option is not presently supported for use together with the BEFORE option and the FOR EACH ROW trigger action.

2.26.3.6 ON table_name

This is used to specify the table that is referenced when determining whether the trigger will fire. The trigger will fire in response to a change made to the table specified in *table_name*.

Triggers can only reference regular tables. They cannot be created on the basis of objects such as views, sequences, or stored procedures.

Triggers cannot be created on the basis of tables that are referenced in replications. Likewise, any attempt to create a replication object that references a table that is already referenced by a trigger will fail.

If the *user_name* is omitted, ALTIBASE HDB will assume that the trigger is to be created for a table in the current user's schema.

2.26.3.7 REFERENCING syntax

One characteristic of triggers is the concept of old and new rows. When the data in the table referenced by a trigger are changed, an individual row that was changed will consequently have both old and new values. The REFERENCING clause makes it possible to refer to either the old value or the new value as desired.

The use of the REFERENCING clause is subject to the following restrictions:

- The REFERENCING clause must be used together with the FOR EACH ROW option.
- The REFERENCING clause must have the following structures so that it can be referred to in the *trigger_action* clause.
    - {OLD|OLD ROW|OLD ROW AS|OLD AS} alias_name
      This indicates the data contained in a record before it is modified. Old values can be referenced in the WHEN clause or in *psm_body* in *trigger_action*.
      It is of course impossible to reference an old value when the trigger event is an INSERT trigger event, because there is no old value.
    - {NEW|NEW ROW|NEW ROW AS|NEW AS} alias_name
      This indicates the data contained in a record after it is modified. Note that when the trigger is a BEFORE trigger, it is possible to change these data in the body of the trigger.
      It is impossible to reference a new value when the trigger event is a DELETE trigger event, because there is no new value.

2.26.3.8 trigger_action

The trigger action clause comprises the following three parts:

- Action granularity: Determines the unit (row or statement) by which the trigger operates.
- Action WHEN condition: Optionally used to set an additional condition to determine whether the trigger will fire.
- Action body: Determines what the trigger actually does.

FOR EACH {ROW|STATEMENT}

This is used to specify the unit of operation of the trigger. The changes to the data in the table take place according to this unit. The default is FOR EACH STATEMENT.

- FOR EACH ROW: The operations specified in action body are conducted once for each row that is affected by *trigger_event* and satisfies the WHEN clause. The FOR EACH ROW clause must be used when either the REFERENCING clause or the WHEN clause is used.
- FOR EACH STATEMENT: The trigger will be fired only once, either after or before execution of the DML statement that caused the trigger to fire.

WHEN search_condition

This is used to specify the conditions that are used to determine whether to fire the trigger. The action body of the trigger is executed only if the *search_condition* in the WHEN clause evaluates to TRUE. If the *search_condition* in the WHEN clause evaluates to FALSE, the action body of the trigger will not be executed. If no WHEN clause is specified, the action body of the trigger will be executed every time the trigger event occurs.

The following restrictions govern the use of conditions in the WHEN clause:

- The WHEN clause can only be used with the FOR EACH ROW clause.

- Only the *alias_name* defined in the REFERENCING clause can be used in the WHEN *search_condition*.
- Subqueries cannot be used in the WHEN *search_condition*.
- Stored procedures cannot be used in the WHEN *search_condition*.

psm_body

This is the so-called "action body" of the trigger, and is used to specify the actual operations that are to be carried out by the trigger. *psm_body* can be specified in the same way as a block statement in a stored procedure.

The following restrictions govern the implementation of *psm_body*:

- Transaction-related statements such as COMMIT and ROLLBACK cannot be used.
- Session-related statements such as CONNECT cannot be used.
- Schema-related statements such as CREATE TABLE cannot be used.
- Stored procedures cannot be called.
- Recursive triggers, that is, triggers that perform the operation specified in *trigger_event*, cannot be created.

For more information about the use of block statements in stored procedures, please refer to the *Stored Procedures Manual*.

2.26.4 Considerations

- Order of Trigger Execution

  It is possible to create multiple triggers that reference the same table. In such cases, the order in which the triggers fire is not fixed. If it is important to control the order in which the actions of multiple triggers occur, rewrite the triggers as a single trigger.

- Trigger Execution Failure

  If an error occurs while a trigger is executing, the DML statement that caused the trigger to fire will also fail.

- Execution of DDL on Tables Referenced in Triggers

  When a table that causes triggers to fire is deleted using the DROP TABLE statement, the triggers that fire in response to changes made to the table are also deleted.

  However, when a table that is referred to by the action body of a trigger is altered or dropped, the trigger is not dropped. In the case where the table is dropped, when it becomes impossible to perform the operations in the action body of such a trigger, any DML statements that cause the trigger to fire will fail. In the case where the table is altered, the trigger will be internally recompiled and executed at the time it is fired.

- Triggers and Replication

DML statements that are executed in the course of replication do not cause triggers to fire.

- Trigger and LOB

  The table that causes a trigger to fire must not contain any LOB columns. Any attempt to create a trigger that references a table containing a LOB column in *trigger_event* will fail and raise an error. The values contained in LOB columns cannot be used inside *psm_body*. However, it is acceptable to reference tables containing LOB columns inside *psm_body*.

2.26.5 Examples

< Query > The following example shows how to use a trigger to track the deletion of rows. In this example, the trigger operates on a FOR EACH ROW basis, and references the original values in the *ono*, *cno*, *qty* and *arrival_date* columns of the *orders* table when data about completed (processing='D') delivery orders are deleted from the table. The trigger creates a record of the deleted rows by inserting rows into the *log_tbl* table.

```
iSQL> CREATE TABLE log_tbl(
  ono BITINT,
  cno BITINT,
  qty INTEGER,
  arrival_date DATE,
  sysdate DATE);
Create success.

iSQL> 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;
/
Create success.

iSQL> DELETE FROM orders WHERE processing = 'D';
2 rows deleted.

iSQL> SELECT * FROM log_tbl;
ONO                  CNO                  QTY         ARRIVAL_DATE
-----------------------------------------------------------------------
SYSDATE
---------------
11290011             17                   1000        05-DEC-2011
25-APR-2012
11290100             11                   500         07-DEC-2011
25-APR-2012
2 rows selected.
```

<Query>In the following example, when a record is inserted into the *SCORES* table, a value of 0 is set for the *SCORE* column if NULL is specified. This is accomplished using a BEFORE INSERT trigger that fires FOR EACH ROW.

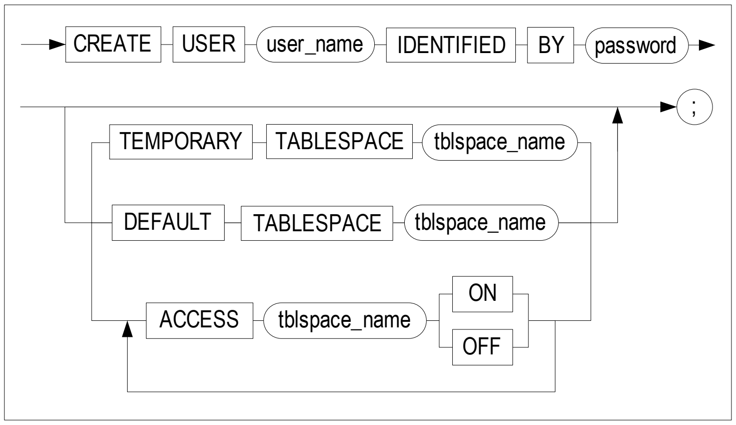
```
iSQL> CREATE TABLE SCORES( ID INTEGER, SCORE INTEGER );
Create success.
iSQL> CREATE TRIGGER SCORES_TRIGGER
BEFORE INSERT ON SCORES
REFERENCING NEW ROW NEW_ROW
```

```
FOR EACH ROW
AS BEGIN
  IF NEW_ROW.SCORE IS NULL THEN
    NEW_ROW.SCORE := 0;
  END IF;
END;
/
Create success.
iSQL> INSERT INTO SCORES VALUES( 1, 20 );
1 row inserted.
iSQL> INSERT INTO SCORES VALUES( 5, NULL );
1 row inserted.
iSQL> INSERT INTO SCORES VALUES( 17, 75 );
1 row inserted.
iSQL> SELECT * FROM SCORES;
ID          SCORE
---------------------------
1           20
5           0
17          75
3 rows selected.
```

2.27 CREATE USER

2.27.1 Syntax

2.27.1.1 create_user ::=



2.27.2 Prerequisites

Only the SYS user and users to whom the CREATE USER system privilege has been granted can create users.

2.27.3 Description

This statement is used to create a database user and specify the user's name, password, and tablespace access privileges.

2.27.3.1 user_name

This is used to specify the name of the user to create. The user name must be unique in the database.

2.27.3.2 IDENTIFIED BY password

ALTIBASE HDB uses passwords to authenticate users, meaning that users must enter their passwords in order to access the database.

The maximum password length is 8 bytes or 11 bytes, depending on the operating system. In Solaris x86 2.8 and above and Windows, the maximum password length is 11 bytes. In other operating systems, the maximum password length is 8 bytes. If an attempt is made to specify a password that is longer than the relevant limit when creating a user, ALTIBASE HDB will truncate the password to the relevant limit.

2.27.3.3 TEMPORARY TABLESPACE Clause

This clause is used to specify the default temporary tablespace for the user, and will be used to store intermediate results when the user performs operations on tables.

If no tablespace is specified, the system temporary tablespace[1] will be used as the temporary tablespace for the user.

A temporary tablespace is normally used to store intermediate results when the user performs operations on disk-based tables.

In the case where all of the tables being accessed by a query are memory tables, all query operations would take place in memory space, and thus no temporary tablespace would be used unless coerced using a query hint.

Only one temporary tablespace can be assigned to a user.

2.27.3.4 DEFAULT TABLESPACE clause

This is used to specify the default tablespace in which to store user-created objects. If this clause is omitted, the default tablespace for the user is the system memory default tablespace.

Only one default tablespace can be specified for a user.

2.27.3.5 ACCESS clause

This is used to specify whether or not the user has access to the specified tablespace. If this clause takes the form ACCESS *tablespace_name* ON, the user is permitted to access the specified tablespace, whereas if it takes the form ACCESS *tablespace_name* OFF, the user is not authorized to access the specified tablespace.

Users are also able to access tablespaces if the ALTER TABLESPACE system privilege has been granted to them.

2.27.4 Restrictions

A single user can use multiple data tablespaces. However, a single user can use only one temporary tablespace.

It is not possible for a user to expressly access the system undo tablespace or create tables, indexes, or other objects therein. Additionally, because there is one, and only one, system undo tablespace in the system, users cannot delete the system undo tablespace or create other system undo tablespaces.

2.27.5 Examples

<Query> Create a user whose name is *uare1* and password is *rose1*.

1. The system temporary tablespace is used for the temporary storage of data that are generated while a query is being executed. It is not logged, and thus the data stored therein cannot be recovered in the event of a media error.

```
iSQL> CREATE USER uare1 IDENTIFIED BY rose1;
Create success.
```

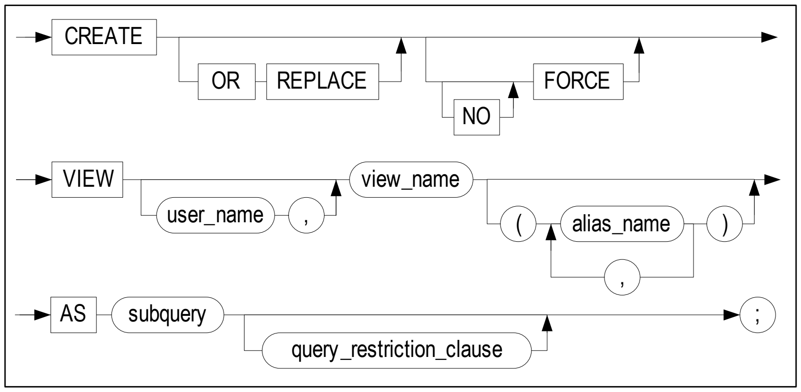
<Query> Create a user named *uare4* with the password *rose4*. Specify *user_data* as the default tablespace and *temp_data* as the temporary tablespace for the user, and grant the user privileges to access the SYS_TBS_MEMORY tablespace.

```
iSQL> CREATE USER uare4
  IDENTIFIED BY rose4
  DEFAULT TABLESPACE user_data
  TEMPORARY TABLESPACE temp_data
  ACCESS SYS_TBS_MEMORY ON;
Create success.
```

2.28 CREATE VIEW

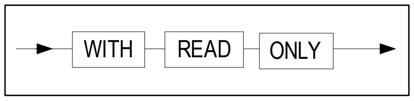
2.28.1 Syntax

2.28.1.1 create_view ::=



subquery ::=

2.28.1.2 query_restriction_clause ::=



2.28.2 Prerequisites

Only the SYS user and users to whom the CREATE VIEW system privilege has been granted can create views.

Additionally, it is necessary to be the SYS user or have the CREATE ANY VIEW system privilege in order to create a view in another user's schema.

The owner of the schema containing the view must have sufficient privileges to execute a SELECT statement on the tables or views on which the view is based.

2.28.3 Description

This statement is used to create a view having the specified name. A view is a logical table based on one or more tables or views. A view contains no data itself. The tables upon which a view is based are called base tables.

2.28.3.1 OR REPLACE

Use the OR REPLACE clause to replace a view having the same name if such a view already exists. This clause is used to change the definition of an existing view without having to drop the old view, create the new view, and grant previously granted privileges for the view.

2.28.3.2 FORCE

The FORCE clause is used to specify that the view is to be created even if the objects on which the view is based don't exist, and even if the owner of the schema containing the view does not have sufficient privileges to access the view.

This means that it is possible to use the FORCE option to create views that are semantically erroneous and thus invalid. In such cases, because an error will occur when a SELECT statement is executed on the view, it is advisable to test the view after creating it by executing a SELECT statement on it immediately, or to query the SYS_VIEWS_ meta table to verify that no errors were raised, which can be inferred to mean that the view is error-free.

2.28.3.3 NO FORCE

Use the NO FORCE clause to specify that the view is to be created only if the underlying objects exist and the owner of the schema in which the view is to be created has access privileges for them. This is the default view creation behavior.

2.28.3.4 user_name

This is used to specify the name of the owner of the schema in which the view is to be created. If this value is omitted, ALTIBASE HDB will create the view in the schema of the user who is connected via the current session.

2.28.3.5 view_name

This is used to specify the name of the view to create.

2.28.3.6 alias_name

If the query on which a view is based contains an expression without an alias, an alias for the expression must be specified. This alias will becomes the name of the corresponding expression in the view. The number of aliases must be same as the number of expressions and columns in the query.

2.28.3.7 subquery

This is used to specify the text of a query that identifies rows and columns in the base table(s) to display in the view.

2.28.3.8 WITH READ ONLY

This is used to specify that the view will be a read-only view. Because views cannot be used to change data, that is, because INSERT, UPDATE, and DELETE operations cannot be performed on views, a read-only view is always created, even if this clause is not used.

2.28.4 Limitations on the Use of Queries in Views

A maximum of 1024 expressions can be specified in the SELECT statement on which a view is based.

The CURRVAL and NEXTVAL pseudocolumns cannot be used in the SELECT statement on which a view is based.

2.28.5 Examples

- Creating a view

  <Query> In the following example, a view called *avg_sal* will be created on the basis of the *employees* table. The purpose of the view is to display the average salary for each department.

```
iSQL> CREATE VIEW avg_sal AS
 SELECT DNO, AVG(salary) emp_avg_sal
 FROM employees
 GROUP BY dno;
Create success.
iSQL> SELECT * FROM avg_sal;
DNO         EMP_AVG_SAL
---------------------------
1001        2150
1002        1340
1003        2438.25
2001        1400
3001        1800
3002        2500
4001        1550
4002        1396.66667
            1500
9 rows selected.
```

  Because *emp_avg_sal* was provided as an alias for the expression in the query on which the view is based, it is not necessary to specify an alias for this column.

- Creating a Join View[1]

  <Query> The following view shows the customer name and the employee name and number for each ordered product.

```
iSQL> CREATE VIEW emp_cus AS
 SELECT DISTINCT e.e_firstname, e.e_lastname,
  c.c_firstname, c.c_lastname
 FROM employees e, customers c, orders o
 WHERE e.eno = o.eno AND o.cno = c.cno;
Create success.
iSQL> select * from emp_cus;
E_FIRSTNAME           E_LASTNAME            C_FIRSTNAME      C_LASTNAME
-----------------------------------------------------------------------
Alvar                Marquez               Estevan               Sanchez
Sandra               Hammond               Pierre                Martin
William              Blake                 Pierre                Martin
Sandra               Hammond               Gabriel               Morris
Alvar                Marquez               Soo-jung              Park
Alvar                Marquez               James                 Stone
Sandra               Hammond               James                 Stone
```

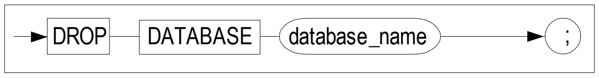
1. A join view is a view in which the underlying query contains a join.

```
William            Blake              Phil               Dureault
Sandra             Hammond            Yasmin             Lalani
Sandra             Hammond            Anh Dung           Nguyen
Alvar              Marquez            Naoki              Sato
William            Blake              Naoki              Sato
Sandra             Hammond            Aida               Rodriguez
William            Blake              Crystal            White
William            Blake              Cheol-soo          Kim
William            Blake              Fyodor             Fedorov
Alvar              Marquez            Fyodor             Fedorov
Alvar              Marquez            Daniel             Lefebvre
Sandra             Hammond            Daichi             Yoshida
William            Blake              Bao                Zhang
William            Blake              Saeed              Pahlavi
Sandra             Hammond            Saeed              Pahlavi
22 rows selected.
```

2.29 DROP DATABASE

2.29.1 Syntax

2.29.1.1 drop_database ::=



2.29.2 Prerequisites

This SQL statement can only be executed by the SYS user in -sysdba administrator mode, and can only be executed during the PROCESS phase.

2.29.3 Description

This statement is used to delete a database from the system.

2.29.3.1 database_name

This is used to specify the name of the database to delete.

When this command is executed, all of the data, log files and log anchor files that were used by the database are also deleted.

2.29.4 Example

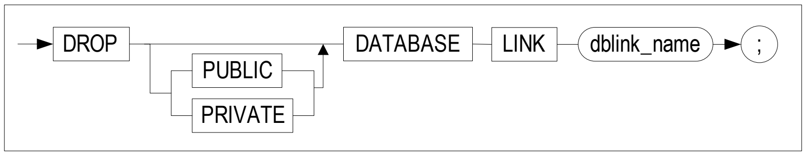
<Query> Delete a database named *mydb*.

```
iSQL(sysdba)> DROP DATABASE mydb;
Checking Log Anchor files
[Ok] /home /altibase_home/logs/loganchor0 Exist.
[Ok] /home /altibase_home/logs/loganchor1 Exist.
[Ok] /home /altibase_home/logs/loganchor2 Exist.
Removing DB files
Removing Log files
Removing Log Anchor files
Drop success.
```

2.30 DROP DATABASE LINK

2.30.1 Syntax

2.30.1.1 drop_database_link ::=



2.30.2 Prerequisites

Only the user who created the database link object can use the DROP DATABASE LINK statement to drop a database link object.

2.30.3 Description

This statement is used to remove the specified Database Link object.

2.30.3.1 dblink_name

This is used to specify the name of the Database Link object to remove.

2.30.4 Consideration

If the Database Link object to be removed is currently in use, it cannot be removed. In order to remove a Database Link object, there must not be any currently executing queries associated with the Database Link object. If there are any active queries associated with the Database Link object, an error will occur.

2.30.5 Examples

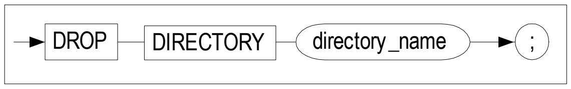
<Query> Remove a private Database Link object named *dblink1* that was created by the user *user1*.

```
iSQL> connect user1/user1;
iSQL> DROP DATABASE LINK dblink1;
```

2.31 DROP DIRECTORY

2.31.1 Syntax

2.31.1.1 drop_directory ::=



2.31.2 Prerequisites

Only the SYS user and users to whom the DROP ANY DIRECTORY system privilege has been granted can execute this statement.

2.31.3 Description

This statement is used to remove a directory. Note that only the reference to the directory in the database is removed; the actual directory is not removed from the file system.

2.31.3.1 directory_name

This is used to specify the name of the directory to drop.

2.31.4 Examples

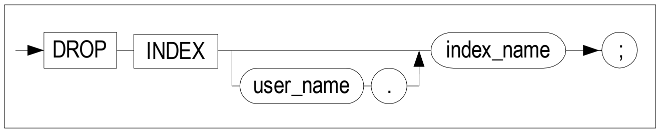
<Query> Drop the directory named *alti_dir1*.

```
iSQL> DROP DIRECTORY alti_dir1;
Drop success.
```

2.32 DROP INDEX

2.32.1 Syntax

2.32.1.1 drop_index ::=



2.32.2 Prerequisites

Only the SYS user, the owner of the schema containing the index, and users having the DROP ANY INDEX system privilege can execute the DROP INDEX statement.

2.32.3 Description

This statement is used to remove an index from the database.

2.32.3.1 user_name

This is used to specify the name of the owner of the index to be dropped. If omitted, ALTIBASE HDB will assume that the index belongs to the schema of the user connected via the current session.

2.32.3.2 index_name

This is used to specify the name of the index to drop.

2.32.4 Examples

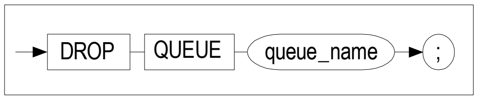
<Query> Drop the index *emp_idx1*.

```
iSQL> DROP INDEX emp_idx1;
Drop success.
```

2.33 DROP QUEUE

2.33.1 Syntax

2.33.1.1 drop_queue ::=



2.33.2 Description

This statement is used to delete the specified queue. The queue table, the index for the queue table and the sequence used to generate MSGID values in the queue table are deleted along with the queue.

2.33.3 Example

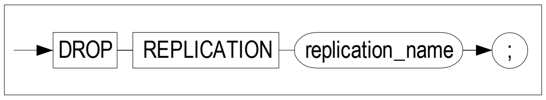
<Query> Delete the message queue *Q1* and its associated objects.

```
DROP QUEUE Q1;
```

2.34 DROP REPLICATION

2.34.1 Syntax

2.34.1.1 drop_replication ::=



2.34.2 Prerequisites

Only the SYS user can execute replication-related statements.

2.34.3 Description

This statement is used to drop a replication.

2.34.3.1 replication_name

This is used to specify the name of the replication to drop.

2.34.4 Limitation

A replication that is currently active cannot be dropped. That is, a replication cannot be dropped if the ALTER REPLICATION START command has been executed for the replication; the ALTER REPLICATION STOP command must first be executed, after which it will be possible to drop the replication.

2.34.5 Examples

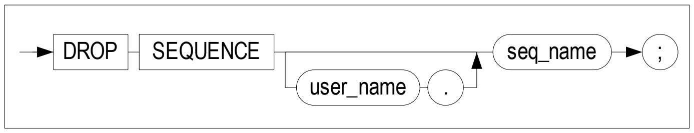
<Query> Drop the replication *rep1*.

```
iSQL> DROP REPLICATION rep1;
Drop success.
```

2.35 DROP SEQUENCE

2.35.1 Syntax

2.35.1.1 drop_sequence ::=



2.35.2 Prerequisites

Only the SYS user, the owner of the schema containing the sequence, and users having the DROP ANY SEQUENCE system privilege can execute the DROP SEQUENCE statement.

2.35.3 Description

This statement is used to remove a sequence from the database.

2.35.3.1 user_name

This is used to specify the name of the owner of the sequence to be dropped. If omitted, ALTIBASE HDB will assume that the sequence belongs to the schema of the user connected via the current session.

2.35.3.2 seq_name

This is used to specify the name of the sequence to drop.

2.35.4 Examples

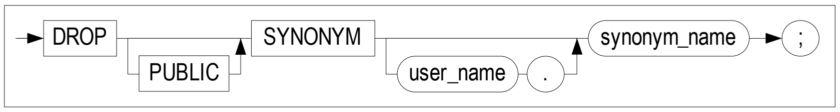
<Query> Delete the sequence *seq1*.

```
iSQL> DROP SEQUENCE seq1;
Drop success.
```

2.36 DROP SYNONYM

2.36.1 Syntax

2.36.1.1 drop_synonym ::=



2.36.2 Prerequisites

Only the SYS user, the owner of the schema containing the synonym, and users to whom the DROP ANY SYNONYM system privilege has been granted can execute the DROP SYNONYM statement.

Additionally, only the SYS user and users to whom the DROP PUBLIC SYNONYM system privilege has been granted can drop public synonyms.

2.36.3 Description

This statement is used to remove a synonym from the database.

2.36.3.1 PUBLIC

To remove a public synonym, use the PUBLIC keyword. If the PUBLIC keyword is not used, a PRIVATE synonym having the specified name will be removed.

When the PUBLIC keyword is used, *user_name* cannot be specified.

2.36.3.2 user_name

This is used to specify the name of the owner of the synonym to drop. If omitted, ALTIBASE HDB will assume that the synonym belongs to the schema of the user connected via the current session.

2.36.3.3 synonym_name

This is used to specify the name of the synonym to drop.

2.36.4 Examples

<Query> Drop the synonym *my_dept*.

```
iSQL> DROP SYNONYM my_dept;
Drop success.
```

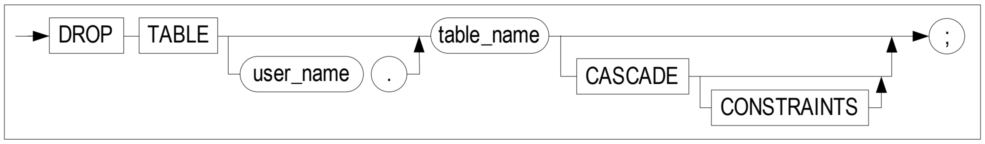
<Query> Drop the PUBLIC synonym *dept*.

```
iSQL> DROP PUBLIC SYNONYM dept;
Drop success.
```

2.37 DROP TABLE

2.37.1 Syntax

2.37.1.1 drop_table ::=



2.37.2 Prerequisites

Only the SYS user, the owner of the schema containing the table, and users to whom the DROP ANY TABLE system privilege has been granted can execute the DROP TABLE statement.

2.37.3 Description

This statement is used to remove a table and all of its data from a database.

2.37.3.1 user_name

This is used to specify the name of the owner of the table to be dropped. If omitted, ALTIBASE HDB will assume that the table belongs to the schema of the user connected via the current session.

2.37.3.2 table_name

2.37.3.3 This is used to specify the name of the table to be dropped.

2.37.3.4 CASCADE / CASCADE CONSTRAINTS

These options are used to delete referential integrity constraints in other tables that reference the primary key or unique keys in the table being dropped.

2.37.4 Examples

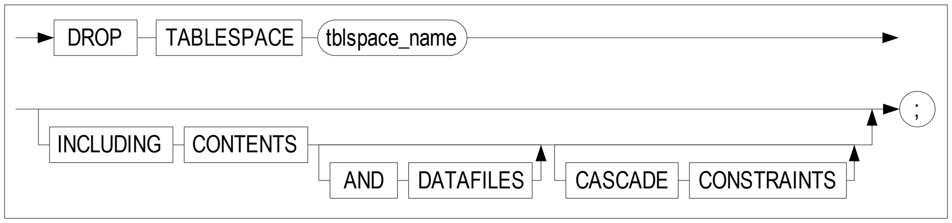
<Query> Drop the *employees* table.

```
iSQL> DROP TABLE employees;
Drop success.
```

2.38 DROP TABLESPACE

2.38.1 Syntax

2.38.1.1 drop_tablespace ::=



2.38.2 Prerequisites

Only the SYS user and users to whom the DROP TABLESPACE system privilege has been granted can execute the DROP TABLESPACE statement.

2.38.3 Description

This statement is used to remove a tablespace from the database.

2.38.3.1 tblspace_name

This is used to specify the name of the tablespace to drop.

2.38.3.2 INCLUDING CONTENTS

This is used to specify that all of the contents of the tablespace are to be deleted. If one or more objects exist in the tablespace, this clause must be specified in order to remove the tablespace. If this clause is not specified, ALTIBASE HDB will return an error and the DROP TABLESPACE statement will fail.

2.38.3.3 AND DATAFILES

If the AND DATAFILES clause is additionally specified along with the INCLUDING CONTENTS clause, all of the files related to the tablespace are deleted from the file system.

When a disk tablespace is dropped, all of the data files in the disk tablespace are deleted from the file system.

When a memory tablespace is dropped, all of the checkpoint image files for the memory tablespace are deleted from the file system. However, the checkpoint paths are not deleted.

The AND DATAFILES clause cannot be used when dropping a volatile tablespace.

2.38.3.4 CASCADE CONSTRAINTS

The CASCADE CONSTRAINTS clause must be specified in order to drop all referential integrity constraints in tables that are stored in tablespaces other than the tablespace being dropped but that refer to primary and unique keys in tables in the tablespace being dropped. If this clause is omitted when such referential integrity constraints exist, an error will be returned and the attempt to drop the tablespace will fail.

2.38.4 Limitation

The following tablespaces are system tablespaces, and thus cannot be removed using the DROP TABLESPACE statement:

- SYS_TBS_MEM_DIC
- SYS_TBS_MEM_DATA
- SYS_TBS_DISK_DATA
- SYS_TBS_DISK_UNDO
- SYS_TBS_DISK_TEMP

2.38.5 Examples

<Query> Drop the tablespace *user_data*.

```
iSQL> DROP TABLESPACE user_data;
Drop success.
```

<Query 2> Delete the disk tablespace *user_data* along with all associated objects and data files.

```
iSQL> DROP TABLESPACE user_data INCLUDING CONTENTS AND DATAFILES;
Drop success.
```

<Query 3> Delete the memory tablespace *user_data* along with all associated objects and data files.

```
iSQL> DROP TABLESPACE user_memory_tbs INCLUDING CONTENTS AND DATAFILES;
Drop success.
```

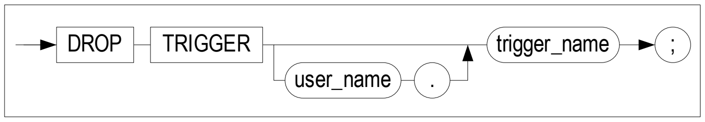
<Query 4>Delete the tablespace *user_data* along with all objects stored therein and all referential integrity constraints that refer to primary and unique keys in all tables in the tablespace.

```
iSQL> DROP TABLESPACE user_data INCLUDING CONTENTS CASCADE CONSTRAINTS;
Drop success.
```

2.39 DROP TRIGGER

2.39.1 Syntax

2.39.1.1 drop_trigger ::=



2.39.2 Prerequisites

Only the SYS user, the owner of the schema containing the trigger, and users to whom the DROP ANY TRIGGER system privilege has been granted can execute the DROP TRIGGER statement.

2.39.3 Description

This statement is used to drop the specified trigger from the database.

2.39.3.1 user_name

This is used to specify the name of the owner of the trigger to be dropped. If omitted, ALTIBASE HDB will assume that the trigger belongs to the schema of the user connected via the current session.

2.39.3.2 trigger_name

This is used to specify the name of the trigger to be dropped.

2.39.4 Examples

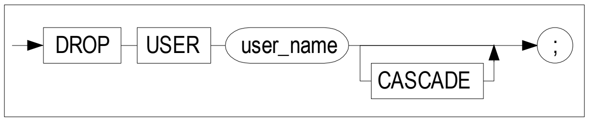
<Query> Drop a trigger from a table.

```
iSQL> DROP TRIGGER del_trigger;
Drop success.
```

2.40 DROP USER

2.40.1 Syntax

2.40.1.1 drop_user ::=



2.40.2 Prerequisites

Only the SYS user and users to whom the DROP USER system privilege has been granted can execute the DROP USER statement.

2.40.3 Description

This statement is used to drop the specified user from the database.

2.40.3.1 user_name

This is used to specify the name of the user to drop.

2.40.3.2 CASCADE

This is used to specify that not only the database user but also all objects in the user's schema will be dropped. Additionally, any referential integrity constraints that refer to primary and unique keys in tables belonging to the user's schema will also be dropped.

When there are any objects in the user’s schema, if the CASCADE keyword is omitted, an error will be returned and the attempt to drop the user will fail.

2.40.4 Examples

<Query> Drop the user *uare1*.

```
iSQL> DROP USER uare1;
Drop success.
```

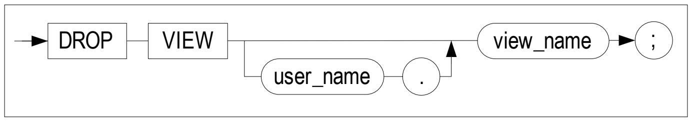
<Query> Drop the user *uare4* and all of the user’s objects.

```
iSQL> DROP USER uare4 CASCADE;
Drop success.
```

2.41 DROP VIEW

2.41.1 Syntax

2.41.1.1 drop_view ::=



2.41.2 Prerequisites

Only the SYS user, the owner of the schema containing the view, and users to whom the DROP ANY VIEW system privilege has been granted can execute the DROP VIEW statement.

2.41.3 Description

This statement is used to drop the specified view from the database.

2.41.3.1 user_name

This is used to specify the name of the owner of the view to be dropped. If omitted, ALTIBASE HDB will assume that the view belongs to the schema of the user connected via the current session.

2.41.3.2 view_name

This is used to specify the name of the view to be dropped.

2.41.4 Examples

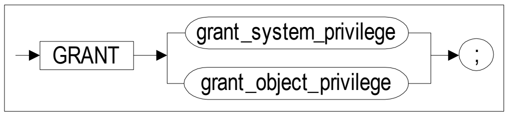
<Query> Drop the view *avg_sal*.

```
iSQL> DROP VIEW avg_sal;
Drop success.
```

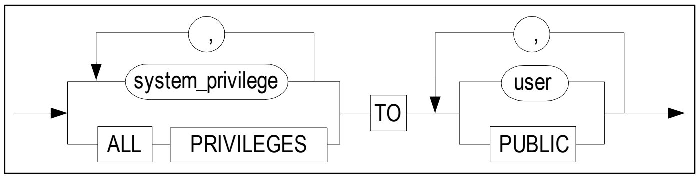
2.42 GRANT

2.42.1 Syntax

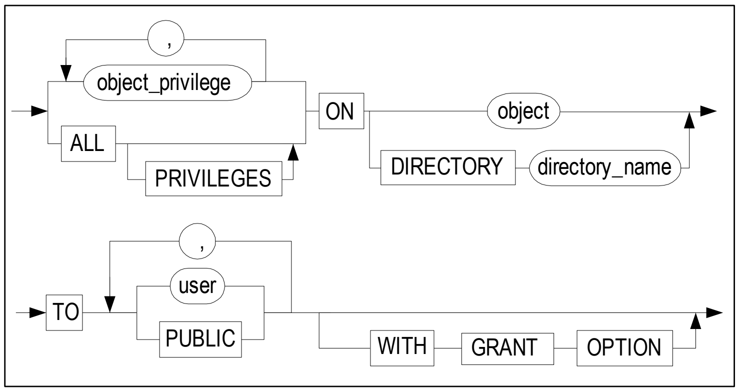
2.42.1.1 grant ::=



2.42.1.2 grant_system_privilege ::=



2.42.1.3 grant_object_privilege ::=



2.42.2 Prerequisites

Only the SYS user and users to whom the GRANT ANY PRIVILEGES system privilege has been granted can grant system privileges.

Only the SYS user, the owner of the object, and users to whom object privileges have been granted using the GRANT OPTION can grant object privileges.

2.42.3 Description

This statement is used to grant privileges to access the database or specified objects to one or more specified users.

Access privileges are classified as either system privileges or object privileges.

2.42.3.1 grant_system_privilege

System privileges are usually managed by the DBA. Limited system privileges can be granted in order to allow users to perform specific database tasks; alternatively, comprehensive privileges to control all objects in all schemas can be granted.

System privileges are required in order to execute DDL statements and DCL statements.

2.42.3.2 grant_object_privilege

Once a user has been granted privileges for a particular object, the user can access and/or manipulate the object. Object access privileges are typically managed by the owner of the object.

If system privileges have not been granted, object access privileges are required in order to execute DML statements.

2.42.3.3 System Privileges

system_privilege

This is used to specify the name of the system access privilege that will be granted.

ALL PRIVILEGES

This is used to grant all system privileges to the specified user or users.

TO user

This is used to specify name of the user or users to whom the system privilege(s) will be granted.

TO PUBLIC

This is used to specify that the system privilege(s) is/are to be granted to all users.

Notes:

1. Just like the SYS user, any user to whom the GRANT ANY PRIVILEGES system is granted can grant all system access privileges to other users.
2. The SYS user has all system access privileges.
3. The presence of the ANY keyword in the name of a system privilege indicates that the privilege pertains to all schema. For example, the SELECT ANY TABLE privilege will allow the user to whom it is granted to run a SELECT statement on any table in the database.
4. The CREATE privilege is granted to allow users to create objects, and includes permission to DROP (i.e. remove) the objects they have created.
5. The CREATE TABLE object privilege allows users to create indexes as well as tables. The authority to create indexes is an object privilege, not a system privilege.

6. When a new user is created, the following privileges are typically granted to the user:
   - CREATE SESSION
   - CREATE TABLE
   - CREATE SEQUENCE
   - CREATE SYNONYM
   - CREATE PROCEDURE
   - CREATE VIEW
   - CREATE TRIGGER
   - CREATE DATABASE LINK

The following query can be used to display the list of system privileges supported in ALTIBASE HDB:

```
iSQL> SELECT * FROM SYSTEM_.SYS_PRIVILEGES_ where PRIV_TYPE = 2;
```

ALTIBASE HDB supports a total of 51 system privileges.

PrivID	Related Object	Name	Purpose
1		ALL	Even if this privilege is granted to a user, the user will still not have the ALTER DATABASE, DROP DATABASE, or MANAGE TABLESPACE privileges.
201	DATABASE	ALTER SYSTEM	For changing ALTIBASE HDB property settings using ALTER SYSTEM statements when ALTIBASE HDB is online
233		ALTER DATABASE	Cannot be granted to any users other than the SYS user
234		DROP DATABASE	Cannot be granted to any users other than the SYS user
250	DIRECTORY	CREATE ANY DIRECTORY	For creating directory objects to be used for controlling files in stored procedures
251		DROP ANY DIRECTORY	For removing directory objects used for controlling files in stored procedures
202	INDEXES	CREATE ANY INDEX	For creating indexes not only in one's own schema but also in other users' schemas
203		ALTER ANY INDEX	For altering the definition of any index in the database
204		DROP ANY INDEX	For dropping any index from the database

PrivID	Related Object	Name	Purpose
205	PROCEDURES	CREATE PROCEDURE	For creating stored procedures and stored functions in one's own schema
206		CREATE ANY PROCE-DURE	For creating stored procedures and stored functions not only in one's own schema but also in other users' schemas
207		ALTER ANY PROCEDURE	For recompiling any stored procedure or function in the database
208		DROP ANY PROCEDURE	For dropping any stored proce-dure or function in the data-base
209		EXECUTE ANY PROCE-DURE	For executing any stored pro-cedure or function in the data-base
210	SEQUENCES	CREATE SEQUENCE	For creating sequences in one's own schema
211		CREATE ANY SEQUENCE	For creating sequences not only in one's own schema but also in other users' schemas
212		ALTER ANY SEQUENCE	For changing the definition of any sequence in the database
213		DROP ANY SEQUENCE	For deleting any sequence in the database
214		SELECT ANY SEQUENCE	For querying any sequence in the database
215	SESSIONS	CREATE SESSION	For connecting to the database
216		ALTER SESSION	Granted automatically to every user; reserved for future use.
245	SYNONYM	CREATE SYNONYM	For creating a private synonym
246		CREATE PUBLIC SYN-ONYM	For creating a public synonym
247		CREATE ANY SYNONYM	For creating private synonyms not only in one's own schema but also in other users' sche-mas
248		DROP ANY SYNONYM	For dropping any private syn-onym

PrivID	Related Object	Name	Purpose
249		DROP PUBLIC SYN-ONYM	For dropping public synonyms
217	TABLES	CREATE TABLE	For creating tables in one's own schema
218		CREATE ANY TABLE	For creating tables not only in one's own schema but also in other users' schemas
219		ALTER ANY TABLE	For truncating all records from any table or changing the definition of any table in the database
220		DELETE ANY TABLE	For deleting any table from the database
221		DROP ANY TABLE	For dropping any table in the database
222		INSERT ANY TABLE	For inserting new records into any table in the database
223		LOCK ANY TABLE	For locking any table in the database
224		SELECT ANY TABLE	For querying any table in the database
225		UPDATE ANY TABLE	For changing the data in any table in the database
226	USERS	CREATE USER	For creating new users
227		ALTER USER	For changing the definition of any user in the database
228		DROP USER	For dropping users
229	VIEWS	CREATE VIEW	For creating views in one's own schema
230		CREATE ANY VIEW	For creating views not only in one's own schema but also in other users' schemas
231		DROP ANY VIEW	For deleting any view in the database
232	MISCELLANEOUS	GRANT ANY PRIVILEGES	For granting any system privilege to other users
235	TABLESPACES	CREATE TABLESPACE	For creating tablespaces
236		ALTER TABLESPACE	For changing the definition of a tablespace

PrivID	Related Object	Name	Purpose
237		DROP TABLESPACE	For deleting tablespaces
238		MANAGE TABLESPACE	Cannot be granted to any users other than the SYS user
240		SYSDBA	Cannot be granted to any users other than the SYS user
241	TRIGGER	CREATE TRIGGER	For creating new triggers
242		CREATE ANY TRIGGER	For creating triggers not only in one's own schema but also in other users' schemas
243		ALTER ANY TRIGGER	For changing the definition of any trigger in the database
244		DROP ANY TRIGGER	For dropping any trigger in the database

2.42.3.4 Object Privileges

object_privilege

This clause is used when it is desired to grant only particular privileges for the object (the table later in this section shows which privileges are supported for which objects).

ALL [PRIVILEGES]

This clause is used to grant all possible privileges for the object.

ON object

This is used to specify the object, such as a table, sequence, or stored procedure, for which to grant privileges.

ON DIRECTORY directory_name

This clause is used to specify the name of the directory object, which is used in stored procedures to manipulate directories and files in the file system, for which to grant privileges.

TO user

This is used to specify the name of the user or users to whom the object privilege(s) will be granted.

TO PUBLIC

This is used to specify that the object privilege(s) is/are to be granted to all users.

WITH GRANT OPTION

The WITH GRANT OPTION is used to enable the grantee to grant the object privileges to other users.

2.42.4 Summary

- The term “object owner” refers to the user who created an object.
- In order to grant object access privileges, it is necessary to be the SYS user, the owner of the object, or a user to whom the relevant object access privileges have been granted with the WITH GRANT OPTION.
- The owner of an object automatically has all privileges for the object.

The following query can be used to display all of the object privileges supported in ALTIBASE HDB:

```
SELECT * FROM SYSTEM_.SYS_PRIVILEGES_ where PRIV_TYPE = 1;
```

ALTIBASE HDB supports the following object privileges:

PrivID	Object privileges	Table	Sequence	PSM	View	directory
101	ALTER	O	O			
102	DELETE	O				
103	EXECUTE			O		
104	INDEX	O				
105	INSERT	O				
106	REFERENCES	O				
107	SELECT	O	O		O	
108	UPDATE	O				
109	READ					O
110	WRITE					O

All users automatically have SELECT privileges for meta tables.

2.42.5 Examples

2.42.5.1 System Privileges

<Query> In the following example, the EXECUTE ANY PROCEDURE, SELECT ANY TABLE, ALTER ANY SEQUENCE, INSERT ANY TABLE, and SELECT ANY SEQUENCE system privileges are granted to the user *user5*.

```
iSQL> CREATE TABLE seqtbl(i1 INTEGER);
Create success.
iSQL> CREATE OR REPLACE PROCEDURE proc1
```

```
AS
BEGIN
  FOR i IN 1 .. 10 LOOP
    INSERT INTO seqtbl VALUES(i);
  END LOOP;
END;
/
Create success.
iSQL> CREATE USER uare5 IDENTIFIED BY rose5;
Create success.
iSQL> GRANT EXECUTE ANY PROCEDURE, SELECT ANY TABLE TO uare5;
Grant success.
iSQL> CONNECT uare5/rose5;
Connect success.
iSQL> EXEC sys.proc1;
Execute success.
iSQL> SELECT * FROM sys.seqtbl;
SEQTBL.I1
--------------
1
2
3
4
5
6
7
8
9
10
10 rows selected.
iSQL> CONNECT sys/manager;
Connect success.
iSQL> CREATE SEQUENCE seq1
  START WITH 13
  INCREMENT BY 3
  MINVALUE 0 NOMAXVALUE;
Create success.
iSQL> INSERT INTO seqtbl VALUES(seq1.NEXTVAL);
1 row inserted.
iSQL> INSERT INTO seqtbl VALUES(seq1.NEXTVAL);
1 row inserted.
iSQL> SELECT * FROM seqtbl;
SEQTBL.I1
--------------
1
2
3
4
5
6
7
8
9
10
13
16
12 rows selected.
iSQL> GRANT ALTER ANY SEQUENCE, INSERT ANY TABLE, SELECT ANY SEQUENCE TO
uare5;
Grant success.
iSQL> CONNECT uare5/rose5;
Connect success.
iSQL> ALTER SEQUENCE sys.seq1
  INCREMENT BY 50
  MAXVALUE 100
  CYCLE;
Alter success.
iSQL> INSERT INTO sys.seqtbl VALUES(sys.seq1.NEXTVAL);
1 row inserted.
iSQL> INSERT INTO sys.seqtbl VALUES(sys.seq1.NEXTVAL);
1 row inserted.
iSQL> INSERT INTO sys.seqtbl VALUES(sys.seq1.NEXTVAL);
1 row inserted.
iSQL> INSERT INTO sys.seqtbl VALUES(sys.seq1.NEXTVAL);
1 row inserted.
iSQL> SELECT * FROM sys.seqtbl;
SEQTBL.I1
--------------
1
2
3
4
5
6
7
8
9
10
13
16
66
0
50
100
16 rows selected.
```

2.42.5.2 Object Privileges

<Query> In the following example, the SELECT and DELETE object privileges on the table *employees* are granted to the user *uare6* with the WITH GRANT OPTION. This user then passes these privileges on to the *uare7* and *uare8* users.

```
iSQL> CREATE USER uare6 IDENTIFIED BY rose6;
Create success.
iSQL> GRANT CREATE USER TO uare6;
Grant success.
iSQL> @schema.sql
iSQL> GRANT SELECT, DELETE ON employees TO uare6 WITH GRANT OPTION;
Grant success.
iSQL> CONNECT uare6/rose6;
Connect success.
iSQL> CREATE USER uare7 IDENTIFIED BY rose7;
Create success.
iSQL> GRANT SELECT, DELETE ON sys.employees TO uare7;
Grant success.
iSQL> CONNECT uare7/rose7;
Connect success.
iSQL> DELETE FROM SYS.employees WHERE eno = 12;
1 row deleted.
iSQL> SELECT eno, e_firstname, e_lastname FROM sys.employees WHERE eno = 12;
ENO E_FIRSTNAME        E_LASTNAME
-----------------------------------------------
No rows selected.
iSQL> CONNECT sys/manager;
Connect success.
iSQL> CREATE USER uare8 IDENTIFIED BY rose8;
Create success.
iSQL> CONNECT uare6/rose6;
```

```
Connect success.
iSQL> GRANT SELECT, DELETE ON sys.employees TO uare8;
Grant success.
```

Because the *uare6* user was granted object access privileges using the WITH GRANT OPTION, this user can grant these privileges not only to the user *uare7*, who was created by *uare6*, but also to the user *uare8*, who was created by the original grantor (the SYS user).

```
iSQL> CONNECT uare8/rose8;
Connect success.
iSQL> DELETE FROM sys.employees WHERE eno = 13;
1 row deleted.
iSQL> SELECT eno, e_firstname, e_lastname FROM sys.employees WHERE eno = 13;
ENO E_FIRSTNAME         E_LASTNAME
------------------------------------------------
No rows selected.
```

<Query> In the following example, system and object privileges are first granted and then revoked.

```
1.   iSQL> CONNECT sys/manager;
     Connect success.
     iSQL> CREATE TABLE book(
       isbn CHAR(10) PRIMARY KEY,
       title VARCHAR(50),
       author VARCHAR(30),
       edition INTEGER DEFAULT 1,
       publishingyear INTEGER,
       price NUMBER(10,2),
       pubcode CHAR(4));
     Create success.
     iSQL> CREATE TABLE inventory(
       subscriptionid CHAR(10) PRIMARY KEY,
       storecode CHAR(4),
       purchasedate DATE,
       quantity INTEGER,
       paid CHAR(1));
     Create success.
     iSQL> CREATE USER uare9 IDENTIFIED BY rose9;
     Create success.
     iSQL> GRANT ALL PRIVILEGES TO uare9;
     Grant success.
```

The SYS user has granted all system privileges to *uare9*.

```
2.   iSQL> GRANT REFERENCES ON book TO uare9 WITH GRANT OPTION;
     Grant success.
```

The user *uare9* receives the REFERENCES object privilege for the *book* object from the SYS user with the WITH GRANT OPTION, and thus *uare9* is able to grant another user (*uare10*) the REFERENCES object privilege for the *book* object.

```
3.   iSQL> CONNECT uare9/rose9;
     Connect success.
     iSQL> INSERT INTO sys.book VALUES ('0070521824', 'Software Engineering',
     'Roger S. Pressman', 4, 1982, 100000, 'CHAU');
     1 row inserted.
     iSQL> INSERT INTO sys.book VALUES ('0137378424', 'Database Processing',
     'David M. Kroenke', 6, 1972, 80000, 'PREN');
     1 row inserted.
```

The user *uare9* inputs data into the *book* table, which is owned by the SYS user.

```
     iSQL> INSERT INTO sys.inventory VALUES('BORD000002', 'BORD', '12-Jun-
2003', 6, 'N');
1 row inserted.
iSQL> INSERT INTO sys.inventory VALUES('MICR000001', 'MICR', '07-Jun-
2003', 7, 'N');
1 row inserted.
```

The user *uare9* inputs data into the *inventory* table, which is owned by the SYS user.

4.
```
iSQL> SELECT * FROM sys.book;
BOOK.ISBN BOOK.TITLE
------------------------------------------------
BOOK.AUTHOR BOOK.EDITION BOOK.PUBLISHINGYEAR BOOK.PRICE
------------------------------------------------
BOOK.PUBCODE
----------------
0070521824 Software Engineering
Roger S. Pressman 4 1982 100000
CHAU
0137378424 Database Processing
David M. Kroenke 6 1972 80000
PREN
2 rows selected.
```

The user *uare9* queries the *book* table, which is owned by the SYS user.

```
iSQL> SELECT * FROM sys.inventory;
INVENTORY.SUBSCRIPTIONID INVENTORY.STORECODE INVENTORY.PURCHASEDATE
------------------------------------------------
INVENTORY.QUANTITY INVENTORY.PAID
----------------------------------------
BORD000002 BORD 2003/06/12 00:00:00
6 N
MICR000001 MICR 2003/06/07 00:00:00
7 N
2 rows selected.
```

The user *uare9* queries the *inventory* table, which is owned by the SYS user.

5.
```
iSQL> CREATE TABLE book(
  isbn CHAR(10) PRIMARY KEY,
  title VARCHAR(50),
  author VARCHAR(30),
  edition INTEGER DEFAULT 1,
  publishingyear INTEGER,
  price NUMBER(10,2),
  pubcode CHAR(4));
Create success.
iSQL> CREATE TABLE inventory(
  subscriptionid CHAR(10) PRIMARY KEY,
  isbn CHAR(10) CONSTRAINT fk_isbn REFERENCES book(isbn),
  storecode CHAR(4),
  purchasedate DATE,
  quantity INTEGER,
  paid CHAR(1));
Create success.
iSQL> CREATE USER uare10 IDENTIFIED BY rose10;
Create success.
```

Because the SYS user granted ALL PRIVILEGES to the user *uare9*, *uare9* can create other users.

6.
```
iSQL> GRANT REFERENCES ON sys.book TO uare10;
Grant success.
```

Because the SYS user granted the REFERENCES privilege to the user *uare9* with the WITH

GRANT OPTION, *uare9* can pass this privilege on to other users.

```
7.  iSQL> GRANT ALTER ANY TABLE, INSERT ANY TABLE, SELECT ANY TABLE, DELETE
    ANY TABLE TO uare10;
    Grant success.
```

Because the SYS user granted the GRANT ANY PRIVILEGES privilege to the user *uare9*, *uare9* can grant system privileges to other users.

```
8.  iSQL> CONNECT uare10/rose10;
    Connect success.
    iSQL> ALTER TABLE sys.inventory
      ADD COLUMN (isbn CHAR(10) CONSTRAINT fk_isbn REFERENCES
    sys.book(isbn));
    Alter success.
```

Because the user *uare10* has the ALTER ANY TABLE and REFERENCES privileges, *uare10* can create a constraint in a table belonging to another user.

```
9.  iSQL> INSERT INTO uare9.book VALUES('0471316156', 'JAVA and CORBA', 'Rob-
    ert Orfali', 2, 1998, 50000, 'PREN');
    1 row inserted.
    iSQL> INSERT INTO uare9.inventory VALUES('TOWE000001', '0471316156',
    'TOWE', '01-Jun-2003', 5, 'N');
    1 row inserted.
```

Because the user *uare10* has the INSERT ANY TABLE privilege, *uare10* can enter data into a table belonging to *uare9*.

```
iSQL> INSERT INTO sys.book VALUES('053494566X', 'Working Classes', 'Rob-
ert Orfali', 1, 1999, 80000, 'WILE');
1 row inserted.
iSQL> INSERT INTO sys.inventory VALUES('MICR000005', 'WILE', '28-JUN-
1999', 8, 'N', '053494566X');
1 row inserted.
```

Because the user *uare10* has the INSERT ANY TABLE privilege, *uare10* can enter data into a table belonging to the SYS user.

```
10. iSQL> SELECT * FROM uare9.book;
    BOOK.ISBN         BOOK.TITLE
    -----------------------------------------------
    BOOK.AUTHOR       BOOK.EDITION       BOOK.PUBLISHINGYEAR     BOOK.PRICE
    -----------------------------------------------
    BOOK.PUBCODE
    ----------------
    0471316156        JAVA and CORBA
    Robert Orfali     2                  1998                    50000
    PREN
    1 row selected.
    iSQL> SELECT * FROM uare9.inventory;
    INVENTORY.SUBSCRIPTIONID    INVENTORY.ISBN       INVENTORY.STORECODE
    -----------------------------------------------
    INVENTORY.PURCHASEDATE      INVENTORY.QUANTITY   INVENTORY.PAID
    -----------------------------------------------
    TOWE000001                  0471316156            TOWE
    2003/06/01 00:00:00         5                     N
    1 row selected.
```

Because the user *uare10* has the SELECT ANY TABLE privilege, *uare10* can query a table belonging to *uare9*.

```
iSQL> SELECT * FROM sys.book;
BOOK.ISBN          BOOK.TITLE
------------------------------------------------
BOOK.AUTHOR        BOOK.EDITION       BOOK.PUBLISHINGYEAR     BOOK.PRICE
------------------------------------------------
BOOK.PUBCODE
----------------
0070521824         Software Engineering
Roger S.Pressman   4                  1982                    100000
CHAU
0137378424         Database Processing
David M. Kroenke   6                  1972                    80000
PREN
053494566X         Working Classes
Robert Orfali      1                  1999                    80000
WILE
3 rows selected.
iSQL> SELECT * FROM sys.inventory;
INVENTORY.SUBSCRIPTIONID     INVENTORY.STORECODE     INVENTORY.PURCHASE-
DATE
------------------------------------------------
INVENTORY.QUANTITY           INVENTORY.PAID          INVENTORY.ISBN
------------------------------------------------
BORD000002                  BORD                    2003/06/12 00:00:00
6                           N
MICR000001                  MICR                    2003/06/07 00:00:00
7                           N
MICR000005                  WILE                    1999/06/28 00:00:00
8                           N                        053494566X
3 rows selected.
```

Because the user *uare10* has the SELECT ANY TABLE privilege, *uare10* can query a table belonging to the SYS user.

11.
```
iSQL> DELETE FROM uare9.inventory WHERE subscriptionid = 'TOWE000001';
1 row deleted.
iSQL> SELECT * FROM uare9.inventory;
INVENTORY.SUBSCRIPTIONID    INVENTORY.ISBN       INVENTORY.STORECODE
------------------------------------------------
INVENTORY.PURCHASEDATE      INVENTORY.QUANTITY   INVENTORY.PAID
------------------------------------------------
No rows selected.
iSQL> DELETE FROM sys.inventory WHERE subscriptionid = 'MICR000005';
1 row deleted.
iSQL> SELECT * FROM sys.inventory;
INVENTORY.SUBSCRIPTIONID     INVENTORY.STORECODE     INVENTORY.PURCHASE-
DATE
------------------------------------------------
INVENTORY.QUANTITY           INVENTORY.PAID          INVENTORY.ISBN
------------------------------------------------
BORD000002                  BORD                    2003/06/12 00:00:00
6                           N
MICR000001                  MICR                    2003/06/07 00:00:00
7 N
2 rows selected.
```

Because the user *uare10* has the DELETE ANY TABLE privilege, *uare10* can delete data from a table belonging to the SYS user.

12.
```
iSQL> CONNECT uare9/rose9;
Connect success.
iSQL> REVOKE ALTER ANY TABLE, INSERT ANY TABLE, SELECT ANY TABLE, DELETE
ANY TABLE FROM uare10;
Revoke success.
```

The user *uare9* revokes all privileges that *uare9* granted to *uare10* without executing the REVOKE ALL statement.

```
13. iSQL> REVOKE REFERENCES ON sys.book FROM uare10 CASCADE CONSTRAINTS;
    Revoke success.
```

When *uare10*'s REFERENCES privilege is revoked, referential integrity constraints that refer to primary key or unique keys in the *sys.book* table, which belongs to *uare10*'s schema, are also dropped.

```
14. iSQL> CONNECT sys/manager;
    Connect success.
    iSQL> REVOKE ALL PRIVILEGES FROM uare9;
    Revoke success.
```

All of *uare9*'s system privileges are revoked.

```
15. iSQL> REVOKE GRANT ANY PRIVILEGES FROM uare9;
    Revoke success.
```

The GRANT ANY PRIVILEGES privilege is revoked from uare9.

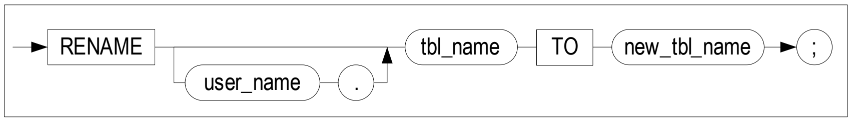
```
16. iSQL> REVOKE REFERENCES ON book FROM uare9;
    Revoke success.
```

The REFERENCES privilege on the *book* table is revoked from uare9.

2.43 RENAME TABLE

2.43.1 Syntax

2.43.1.1 rename ::=



2.43.2 Prerequisites

Only the SYS user, the owner of the schema containing the table, and users having the ALTER ANY TABLE system privilege can execute the RENAME TABLE statement.

2.43.3 Description

This statement is used to change the name of the specified table. Only the table name is altered; none of the data stored therein are changed.

2.43.3.1 user_name

This is used to specify the name of the owner of the table to be renamed. If this is omitted, ALTIBASE HDB will assume that the table belongs to the schema of the user connected via the current session.

2.43.3.2 old_name

This is used to specify the current name of the table.

2.43.3.3 new_name

This is used to specify the new name for the table.

2.43.4 Considerations

The name of a replication target table cannot be changed.

2.43.5 Examples

<Query> Rename a table.

```
iSQL> RENAME employees TO emp1;
Rename success.
```

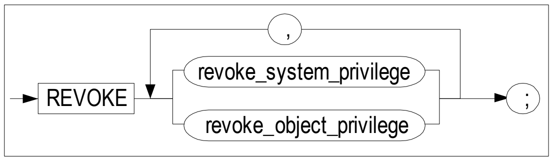
or

```
iSQL> ALTER TABLE employees
 RENAME TO emp1;
Alter success.
```

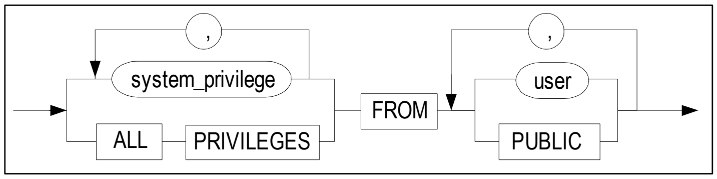
2.44 REVOKE

2.44.1 Syntax

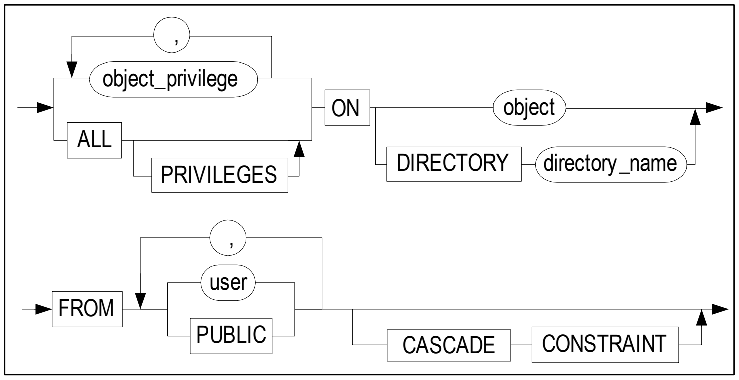
2.44.1.1 revoke ::=



2.44.1.2 revoke_system_privilege ::=



2.44.1.3 revoke_object_privilege ::=



2.44.2 Prerequisites

Only the SYS user and the user who originally granted the privilege to be revoked can revoke privileges.

2.44.3 Description

This statement is used to revoke system privileges or privileges for particular objects from users.

2.44.3.1 System Privileges

system_privilege

This is used to specify the system privilege(s) to be revoked. Please refer to the description of the GRANT statement for the complete list of system privileges.

ALL PRIVILEGES

This is used to specify that all system privileges that have been granted by the user executing this revoke statement are to be revoked.

System privileges that were granted using the ALL PRIVILEGES clause can be revoked using the All PRIVILEGES clause. However, using a statement such as:

```
REVOKE ALL PRIVILEGES FROM user_name;
```

is not the only way to revoke system privileges granted in this way; additionally, a statement such as:

```
REVOKE SELECT ANY TABLE FROM user_name;
```

can be used to revoke individual privileges.

FROM user

This is used to identify the user from whom the privilege(s) will be revoked.

FROM PUBLIC

Use the PUBLIC keyword to revoke the privilege(s) from all users.

Note: System privileges granted using the PUBLIC keyword can be revoked using the PUBLIC keyword.

2.44.3.2 Object Privileges

object_privilege

This is used to specify the object privilege that is to be revoked. Please refer to the table in the description of the GRANT statement for more information about object privileges.

ALL [PRIVILEGES]

The ALL PRIVILEGES (or merely ALL) clause is used to revoke all object privileges that have been granted to the user by the user executing this revoke statement.

When revoking privileges using the ALL [PRIVILEGES] clause, all object access privileges granted to the user are revoked. This even includes object privileges that were not granted using the ALL [PRIVILEGES] clause. For example, an object privilege granted to a user in this way:

```
GRANT SELECT ON table_name TO user_name;
```

can of course be explicitly revoked in this way:

```
REVOKE SELECT ON table_name FROM user_name;
```

It can also be revoked together with all other privileges in this way:

```
REVOKE ALL ON table_name FROM user_name;
```

ON object

This is used to specify the object (table, sequence, stored procedure, etc.) for which the permissions are to be revoked.

ON DIRECTORY directory_name

This clause is used to revoke privileges from the specified directory object.

FROM user

This clause is used to identify the user(s) from whom the privilege(s) will be revoked.

FROM PUBLIC

The PUBLIC keyword is used to revoke the privilege(s) from all users.

CASCADE CONSTRAINTS

This clause is relevant only when revoking the REFERENCES privilege or using the ALL [PRIVILEGES] clause. It is used to specify that any related referential integrity constraints are also to be dropped. These were granted either explicitly or implicitly using the ALL [PRIVILEGES] clause.

2.44.4 Examples

<Query> Revoke object privileges.

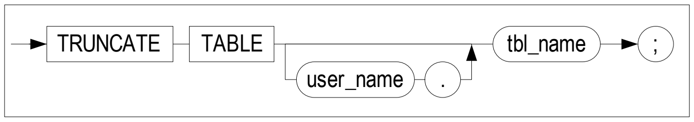
```
iSQL> CONNECT uare6/rose6;
Connect success.
iSQL> REVOKE SELECT, DELETE ON sys.employees
 FROM uare7, uare8;
Revoke success.
iSQL> CONNECT uare7/rose7;
Connect success.
iSQL> SELECT eno, e_lastname FROM sys.employees WHERE eno = 15;
[ERR-311B1 : The user must have the SELECT_ANY_TABLE privilege(s) to execute
this statement.]
```

After the SELECT and DELETE privileges for the *employees* table have been revoked, an error message is displayed when an attempt is made to execute a SELECT statement on that table.

2.45 TRUNCATE TABLE

2.45.1 Syntax

2.45.1.1 truncate ::=



2.45.2 Prerequisites

Only the SYS user, the owner of the schema containing the table, and users having the ALTER ANY TABLE system privilege can execute the TRUNCATE TABLE statement.

2.45.3 Description

The TRUNCATE TABLE statement is used to remove all records from the specified table.

2.45.3.1 user_name

This is used to specify the name of the owner of the table to be truncated. If omitted, ALTIBASE HDB will assume that the table belongs to the schema of the user connected via the current session.

2.45.3.2 tbl_name

This is used to specify the name of the table to be truncated.

If the name of a queue table is specified in *tbl_name*, all ENQUEUE messages are also deleted at the same time.

2.45.3.3 TRUNCATE vs. DELETE

When the TRUNCATE statement is executed, all of the pages in the table are returned to the database as free pages. Therefore, these pages are available for use by other tables. In contrast, when the DELETE statement is used to remove all of the rows from a table, any pages that are emptied are not returned to the database, but remain in a state in which they are reserved for future use by the same table, meaning that memory usage is not reduced.

Because the TRUNCATE statement is a DDL statement, it cannot be rolled back once it has executed successfully.

2.45.4 Considerations

Once the records have been successfully deleted, they cannot be recovered. However, if an error occurs before the completion of execution of the statement, or in the event of a server error, the statement can be rolled back.

2.45.5 Examples

<Query> Use the TRUNCATE statement to remove all data from the *employees* table.

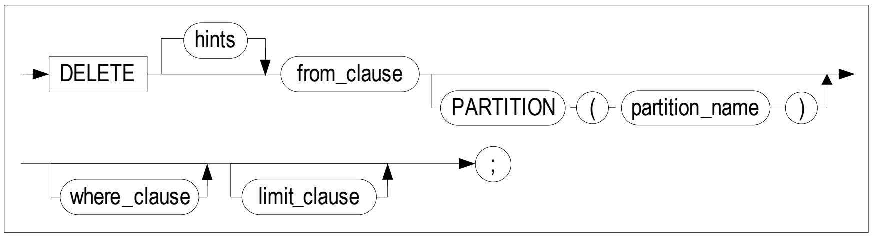
```
iSQL> TRUNCATE TABLE employees;
Truncate success.
```

3 Data Manipulation Statements

3.1 DELETE

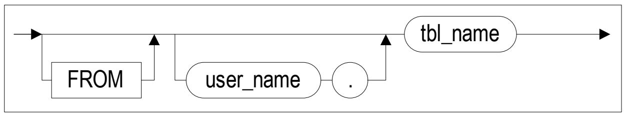
3.1.1 Syntax

3.1.1.1 delete ::=

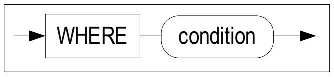


from_clause ::=, *where_clause ::=*, *limit_clause ::=*

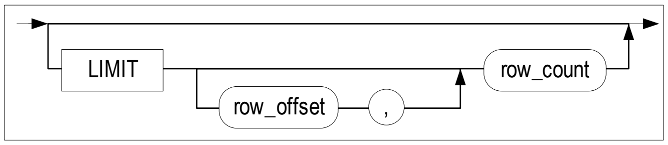
3.1.1.2 from_clause ::=



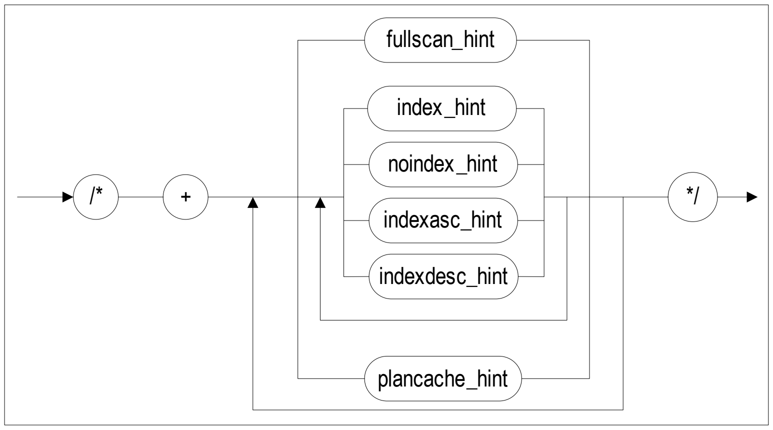
3.1.1.3 where_clause ::=



3.1.1.4 limit_clause ::=

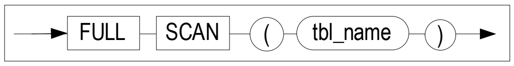


3.1.1.5 hints ::=

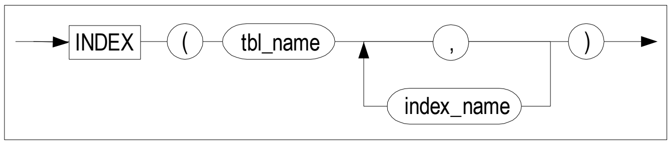


fullscan_hint ::=, *index_hint ::=*, *noindex_hint ::=*, *indexasc_hint ::=*, *indexdesc_hint ::=*, *plancache_hint ::=*

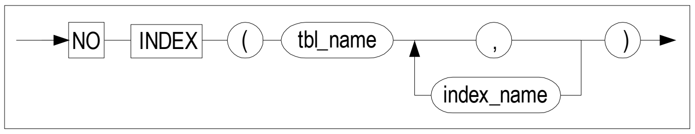
3.1.1.6 fullscan_hint ::=



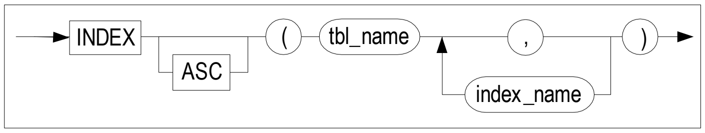
3.1.1.7 index_hint ::=



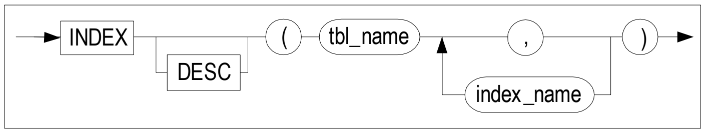
3.1.1.8 noindex_hint ::=



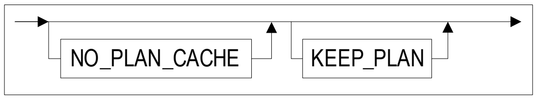
3.1.1.9 indexasc_hint ::=



3.1.1.10 indexdesc_hint ::=



3.1.1.11 plancache_hint ::=



3.1.2 Prerequisites

Only the SYS user, the owner of the schema containing the table, users having the DELETE ANY TABLE system privilege, and users having the DELETE privilege for the specified table can delete rows from tables using this statement.

3.1.3 Description

This statement is used to remove records that meet the specified conditions, if any, from the specified table. It can also be used to delete data from a specified partition.

The WHERE clause is identical to the WHERE clause of a SELECT statement. If it is omitted, all of the data in the table are deleted.

3.1.3.1 user_name

This is used to specify the name of the owner of the table from which to delete records. If omitted, ALTIBASE HDB will assume that the table belongs to the schema of the user connected via the current session.

3.1.3.2 tbl_name

This is used to specify the name of the table containing the records to delete.

3.1.4 HINTS Options

In ALTIBASE HDB, instructions to the query optimizer, known as "hints", can be included within a comment in an SQL statement. The query optimizer will obey these hints, if possible, when choosing an execution plan for the statement.

Using the plus sign ("+") within a comment signifies to ALTIBASE HDB that the comment is a hint. The plus sign must be located immediately after the opening comment delimiter, with no space between them, like this:

```
/*+ FULL SCAN */
```

When a hint contains a syntax error, the query itself will still execute correctly, but the hint will be ignored.

FULL SCAN

This hint stipulates that the entire table is to be searched without using an index, even if there is an index that can be used for the search.

INDEX

This hint stipulates that one of the specified indexes is to be used to perform an index scan of the table.

INDEX ASC

This hint stipulates that one of the specified indexes is to be used to perform an index scan of the table, and that the index scan is to be performed in ascending order.

INDEX DESC

This hint stipulates that one of the specified indexes is to be used to perform an index scan of the table, and that the index scan is to be performed in descending order.

NO INDEX

This hint stipulates that none of the specified indexes are to be used to perform an index scan of the table.

plancache_hint

The hints that pertain to the plan cache are NO_PLAN_CACHE and KEEP_PLAN.

The NO_PLAN_CACHE hint stipulates that the plan that is created is not to be saved in the plan cache.

The KEEP_PLAN hint stipulates that once a plan has been created, it is not to be regenerated, but is to be reused, even if the table statistics have changed since the plan was created.

The KEEP_PLAN hint can be used not only when queries are executed directly ("direct execute") but also when queries are prepared in advance and then executed ("prepare/execute").

For more information on hint syntax and a description of individual hints, please refer to "Chapter 11: SQL Tuning" in the *Administrator's Manual*.

3.1.5 Examples

<Query> Delete all data from a table.

```
iSQL> DELETE FROM orders;
30 rows deleted.
```

<Query> Delete partition *P2* from table *T1*.

```
iSQL> DELETE FROM T1 PARTITION (P2);
```

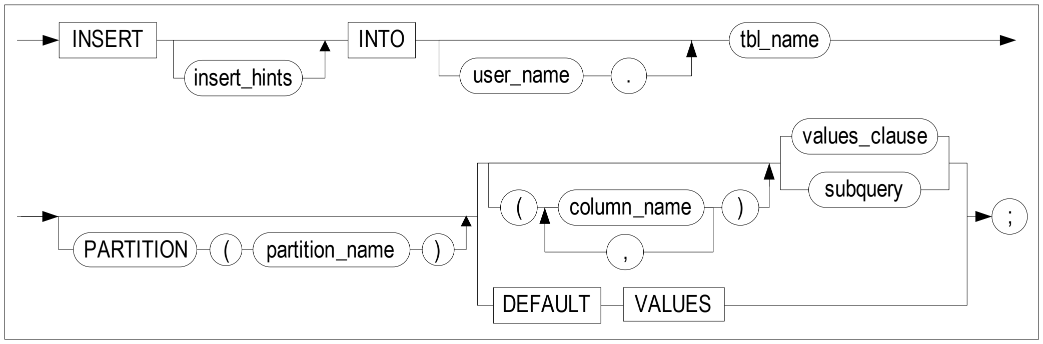
<Query> Delete selected data from a table.

```
iSQL> DELETE
FROM orders
 WHERE eno = (SELECT eno FROM employees
 WHERE e_firstname = 'William');
9 rows deleted.
```

3.2 INSERT

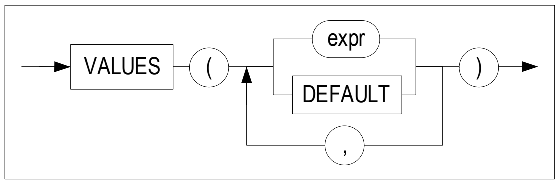
3.2.1 Syntax

3.2.1.1 insert ::=

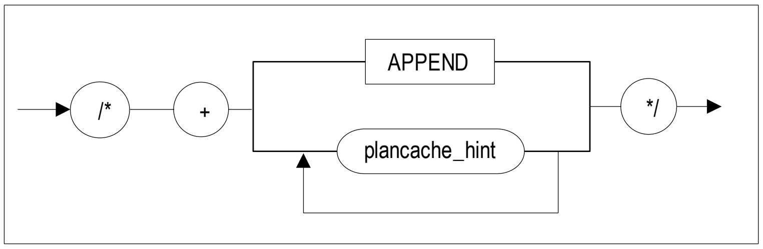


values_clause ::=, *subquery ::=*

3.2.1.2 values_clause ::=



3.2.1.3 insert_hints ::=



3.2.2 Prerequisites

Only the SYS user, the owner of the schema containing the table, users having the INSERT ANY TABLE system privilege, and users having the INSERT privilege for the specified table can insert rows into tables using this statement.

3.2.3 Description

The INSERT statement is used to insert a new record into the specified table or partition. If an index has been defined for the table, the index data will also be modified.

3.2.3.1 user_name

This is used to specify the name of the owner of the table into which the record(s) are to be inserted. If omitted, ALTIBASE HDB will assume that the table belongs to the schema of the user connected via the current session.

3.2.3.2 tbl_name

This is used to specify the name of the table into which the record(s) are to be inserted.

3.2.3.3 NULL

When the values to be inserted are provided only for some columns, but not for others, NULL is inserted into every column for which an insert value is not provided and which does not have a set DEFAULT value. (The default value for a TIMESTAMP column is the system time at the time that the INSERT operation occurred. Therefore, if no insert value is provided for a TIMESTAMP column, the system time, rather than NULL, is inserted into that column.) NULL values can also be inserted by explicitly specifying `NULL` in the VALUES clause.

3.2.3.4 DEFAULT

If the DEFAULT keyword is specified in the VALUES clause, the previously set default value will be inserted into the corresponding column. To insert the default values for all columns, use the DEFAULT VALUES clause. If the DEFAULT keyword is specified for a timestamp column, the system time will be inserted.

3.2.3.5 INSERT ~ SELECT

This type of query is used to insert the results of execution of a SELECT query into a table. The table from which the results were retrieved can be the same table as the table into which the records are to be inserted. The number of columns to be inserted must be the same as the number of columns in the SELECT statement, and corresponding columns must have compatible data types.

3.2.4 HINTS Options

- In ALTIBASE HDB, instructions (“hints“) to the query optimizer can be included within a comment in an SQL statement. The query optimizer will obey these hints, if possible, when choosing an execution plan for the statement. Using the plus sign (“+“) within a comment signifies to ALTIBASE HDB that the comment is a hint. The plus sign must be located immediately after the opening comment delimiter, with no space between them.
- APPEND: This hint is used to stipulate that Direct-Path INSERT is to be performed. Direct-Path INSERT means that when data are inserted, one or more new pages are created, and the data are inserted therein, rather than searching for blank spaces within pages and inserting data in those blank spaces.

The V$DIRECT_PATH_INSERT performance view can be queried to view statistics pertaining to Direct-Path INSERT. For more information about the V$DIRECT_PATH_INSERT performance view, please refer to the *General Reference*.

3.2.5 Precautions

When inserting data using an INSERT statement, please be mindful of the following:

The number of columns that are specified must be the same as the number of values to be inserted, and the data types must be compatible.

When a partition is specified, it is impossible to insert values that do not match the partition conditions.

It is permissible to insert NULL values when no default value has been specified for a column, as long as the column does not have the NOT NULL constraint.

The use of Direct-Path INSERT is governed by the following restrictions:

- The destination table must be a disk table, and it cannot have any LOB columns or indexes.
- The destination table can't be a replication target table.
- The destination table can't have any triggers or referential integrity constraints defined for it.

3.2.6 Examples

- Simple data INSERT statement

  <Query>Insert a record containing the information about the customer *Louise Leroux* into the *customers* table without specifying column names.

```
iSQL> INSERT INTO customers VALUES ( '25', 'Leroux', 'Louise', 'student',
'025282222', 'F', '0101', 150763, '#3 825 - 17th Ave SW Calgary Canada');
1 row inserted.
```

  <Query> Insert a record containing only the employee “Rosalia Jung”’s name and gender and the corresponding employee number into the *employees* table.

```
iSQL> INSERT INTO employees(eno, e_firstname, e_lastname, sex) VAL-
UES(21, 'Rosalia', 'Jung', 'F');
1 row inserted.
```

- Complicated data INSERT statement

  <Query>Copy the customer number and order date for all delayed orders from the *orders* table to the *delayed_processing* table.

```
iSQL> CREATE TABLE delayed_processing(
 cno CHAR(14), order_date DATE);
Create success.
iSQL> INSERT INTO delayed_processing
 SELECT cno, order_date
 FROM orders
 WHERE PROCESSING = 'D';
1 row inserted.
```

- Partitioned data INSERT statement

```
iSQL> CREATE TABLE T1 ( I1 INTEGER, I2 INTEGER )
PARTITION BY RANGE ( I1 )
(
PARTITION P1 VALUES LESS THAN ( 300 ),
PARTITION P2 VALUES LESS THAN ( 400 ),
PARTITION P3 VALUES DEFAULT
) TABLESPACE SYS_TBS_DISK_DATA;
iSQL> INSERT INTO T1 PARTITION ( P1 ) VALUES ( 123, 456 );
1 row inserted.
```

- Using the Direct-Path INSERT hint with an INSERT statement

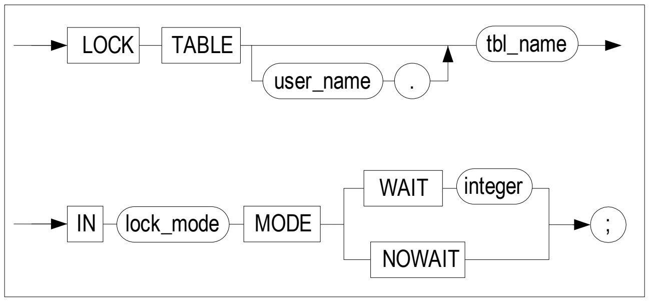
  <Query> Copy all data from table T1 to table T2 using the Direct-Path INSERT hint.

```
iSQL> INSERT /*+ APPEND */ INTO T2 SELECT * FROM T1;
```

3.3 LOCK TABLE

3.3.1 Syntax

3.3.1.1 lock_table ::=



3.3.2 Prerequisites

Only the SYS user, the owner of the schema containing the table, and users having the LOCK ANY TABLE system privilege can execute this statement.

3.3.3 Description

This statement is used to lock a table in a particular mode. A locked table remains locked until the associated transaction is committed or rolled back.

3.3.3.1 *user_name*

This is used to specify the name of the owner of the table to be locked. If omitted, ALTIBASE HDB will assume that the table belongs to the schema of the user connected via the current session.

3.3.3.2 *tbl_name*

This is used to specify the name of the table to lock.

3.3.3.3 *lock_mode*

One of the following lock modes must be specified when locking a table:

ROW SHARE

In ROW SHARE mode, other transactions are permitted to access the table while it is locked, but other users are prevented from locking the table in EXCLUSIVE mode.

SHARE UPDATE

This mode is the same as ROW SHARE mode.

ROW EXCLUSIVE

ROW EXCLUSIVE mode is the same as ROW SHARE mode, with the exception that other transactions are prevented from locking the table in SHARE mode. A ROW EXCLUSIVE lock is automatically obtained in the course of performing an update, insert, or delete operation.

SHARE ROW EXCLUSIVE

SHARE ROW EXCLUSIVE mode is used to view an entire table while permitting others to view the table. However, other users are prohibited from locking the table in SHARE mode and from updating rows.

SHARE

In SHARE mode, other transactions are permitted to view the table while it is locked, but are prevented from updating the table.

EXCLUSIVE

In EXCLUSIVE mode, the current transaction can read or update the locked table, but no other users are allowed to access the table.

3.3.3.4 WAIT | NOWAIT

This clause is used to specify whether to wait until a lock has been obtained. It can be omitted, in which case the wait to obtain a lock on an individual row will continue for an unlimited time.

WAIT n

This option specifies that a transaction is to wait for n seconds to obtain a lock on the row. If the row cannot be locked during this time, an error will be raised.

NOWAIT

The NOWAIT option specifies that a transaction is not to wait to obtain a lock if it is not immediately possible. In this case, ALTIBASE HDB returns an error indicating that the specified table has already been locked by another user.

The following table shows which kind of lock must be obtained in order to execute each kind of DML statement, and additionally shows whether it is possible (“Y“ or ”N“) to obtain that kind of lock when each kind of lock has already been obtained by another transaction. The lock mode shown in parentheses in each cell indicates the lock mode to which the existing lock is escalated when the lock conflict occurs.

Table 3-1 Summary of Table Lock Conflicts

SQL Statement	Table Lock Mode	Permitted if Lock Exists? (Escalated Lock Mode)				
		IS	IX	S	SIX	X
SELECT … FROM tbl_name …	IS	Y (IS)	Y (IX)	Y (S)	Y (SIX)	N (X)

SQL Statement	Table Lock Mode	Permitted if Lock Exists? (Escalated Lock Mode)				
		IS	IX	S	SIX	X
INSERT INTO tbl_name …	IX	Y (IX)	Y (IX)	N (SIX)	N (SIX)	N (X)
UPDATE tbl_name …	IX	Y* (IX)	Y* (IX)	N (SIX)	N (SIX)	N (X)
DELETE FROM tbl_name …	IX	Y* (IX)	Y* (IX)	N (SIX)	N (SIX)	N (X)
SELECT … FROM tbl_name FOR UPDATE …	IS	Y* (IX)	Y* (IX)	Y* (S)	Y* (SIX)	N (X)
LOCK TABLE tbl_name IN ROW SHARE MODE	IS	Y (IS)	Y (IX)	Y (S)	Y (SIX)	N (X)
LOCK TABLE tbl_name IN ROW EXCLUSIVE MODE	IX	Y (IX)	Y (IX)	N (SIX)	N (SIX)	N (X)
LOCK TABLE tbl_name IN SHARE MODE	S	Y (S)	N (SIX)	Y (S)	N (SIX)	N (X)
LOCK TABLE tbl_name IN SHARE ROW EXCLUSIVE MODE	SIX	Y (SIX)	N (SIX)	N (SIX)	N (SIX)	N (X)
LOCK TABLE tbl_name IN EXCLUSIVE MODE	X	N (X)	N (X)	N (X)	N (X)	N (X)

IS: row share (Intent share lock)
IX: row exclusive (Intent exclusive lock)
S: share
SIX: share row exclusive (Shared with intent exclusive lock)
X: exclusive
* Y: The transaction will be able to obtain a lock as long as there is no row-level conflict with the previous transaction, but will enter a waiting state in the event of a row-level conflict.

Regarding the lock type shown in parentheses:

- When it is possible to change the mode of the lock currently being held by the other transaction ("Y"), the currently held lock is changed to the type indicated in parentheses.
- When it is not possible for any transaction other than the transaction holding the lock to change the mode of the lock ("N"), the type of the lock is changed to the type indicated in parentheses when the transaction acquires a new lock.

3.3.4 Examples

The following example shows how ALTIBASE HDB manages data concurrency, integrity, and consistency when using LOCK TABLE and SELECT statements.

Transaction A	Time Point	Transaction B

`iSQL> AUTOCOMMIT OFF;`
`Set autocommit off success.`
		`iSQL> AUTOCOMMIT OFF;`
		`Set autocommit off success.`

	1	(request X lock on *employees*)
		`iSQL> LOCK TABLE employees IN EXCLUSIVE MODE;`
		`Command execute success.`
		(acquire X lock on *employees*)

`iSQL> DROP TABLE employees;`
`[ERR-11170 : The transaction has exceeded the lock timeout specified by the user.]`
	2	

	3	`iSQL> UPDATE employees`
		`SET salary = 2500`
		`WHERE eno = 15;`
		`1 row updated.`

(request S lock on *employees*) `iSQL> LOCK TABLE employees IN SHARE MODE;` (the request conflicts with the X lock already held by transaction B) wait wait wait	4	

	5	`iSQL> COMMIT;`
		`Commit success.`
		(release X lock on *employees*)

(resume)
Lock success.
(acquire S lock on *employees*)
`iSQL> SELECT salary`
`FROM employees`
`WHERE eno = 15;`
`SALARY`
`--------------`
`2500`
`1 row selected.`
(It can be seen that the data have been committed.)
	6	

`iSQL> ROLLBACK;`
`Rollback success.`
(release S lock on *employees*)
	7	

`iSQL> LOCK TABLE employees IN EXCLUSIVE MODE;`
`Lock success.`
(acquire X lock on *employees*)
	8	

Transaction A	Time Point	Transaction B

		`iSQL> SELECT SALARY`
		`FROM employees`
		`WHERE eno = 15;`
		wait
		wait
		wait

`iSQL> UPDATE employees`
`SET eno = 30`
`WHERE eno = 15;`
`1 row updated.`
	10	

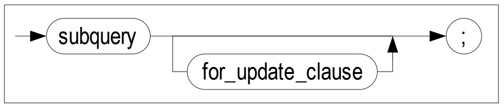
`iSQL> COMMIT;`
`Commit success.`
(release X lock on *employees*)
	11	

	12	(resume)
		`SALARY`
		`--------------`
		`2500`
		`1 row selected.`

3.4 SELECT

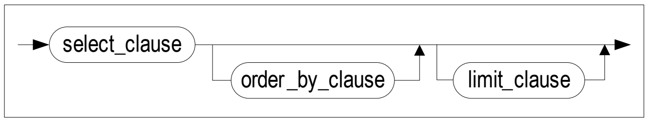
3.4.1 Syntax

3.4.1.1 select ::=



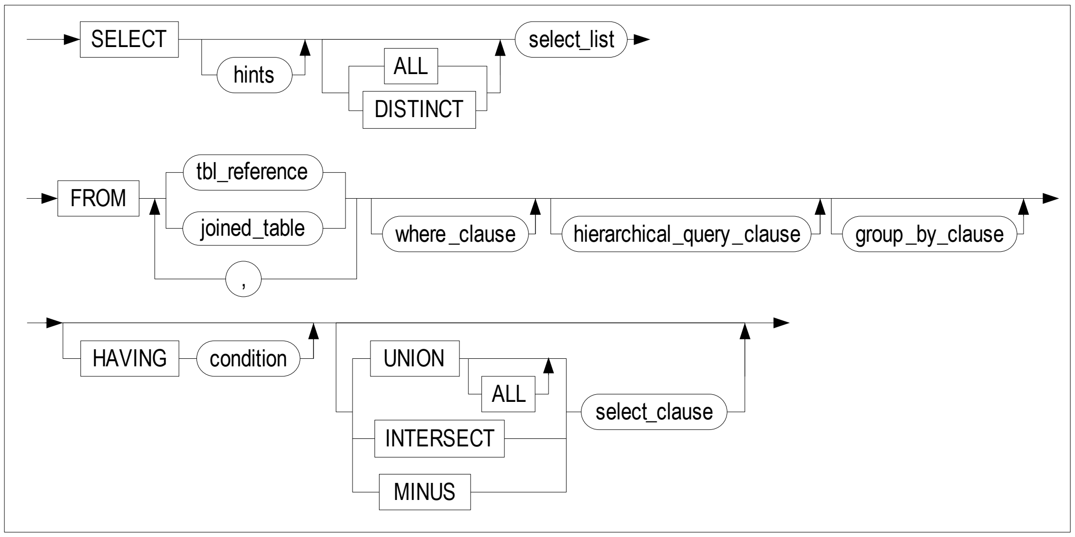
subquery ::=, *for_update_clause ::=*

3.4.1.2 subquery ::=



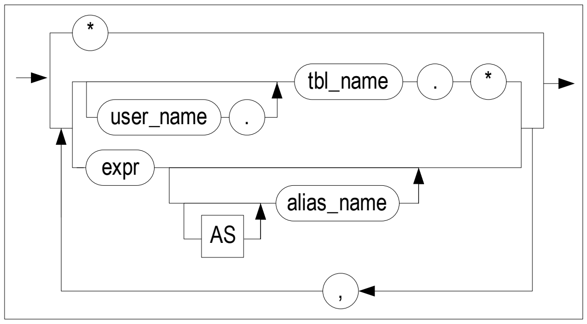
select_clause ::=, *order_by_clause ::=*, *limit_clause ::=*

3.4.1.3 select_clause ::=

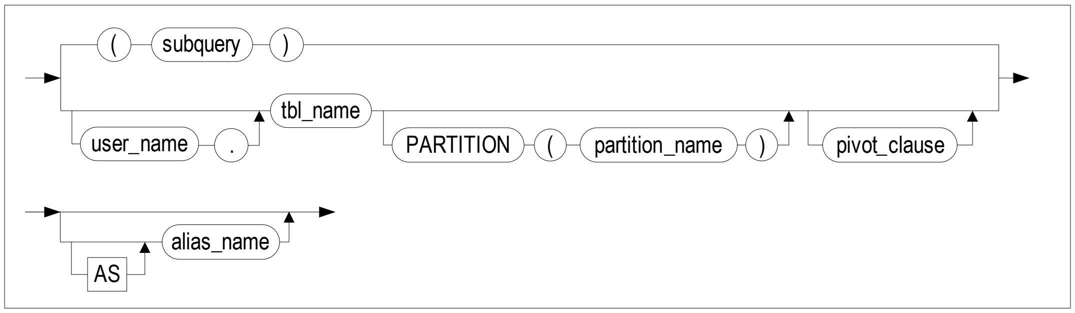


select_list ::=, *tbl_reference ::=*, *joined_table ::=*, *where_clause ::=*, *hierarchical_query_clause ::=*, *group_by_clause ::=*

3.4.1.4 select_list ::=

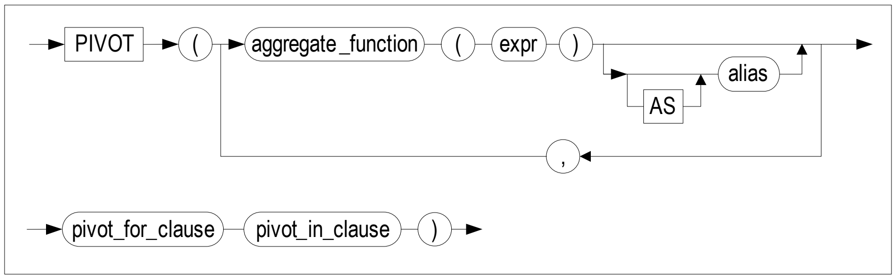


3.4.1.5 tbl_reference ::=

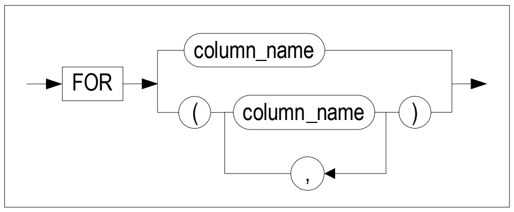


subquery ::=

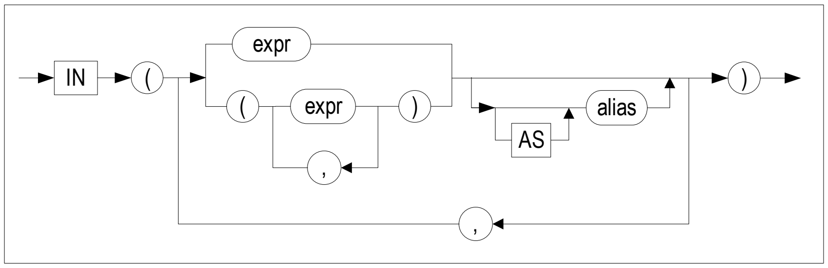
3.4.1.6 pivot_clause ::=



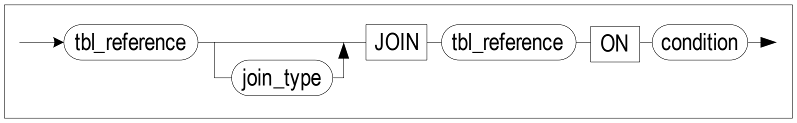
3.4.1.7 pivot_for_clause ::=



3.4.1.8 pivot_in_clause ::=

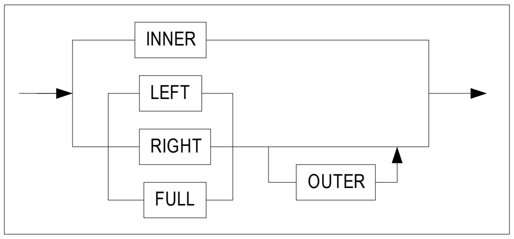


3.4.1.9 joined_table ::=

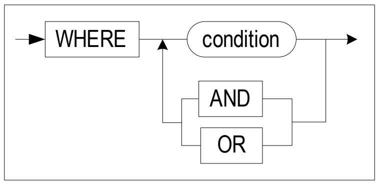


tbl_reference ::=, join_type ::=

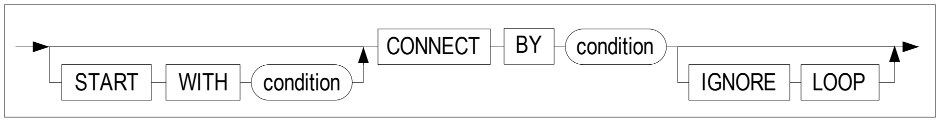
3.4.1.10 join_type ::=



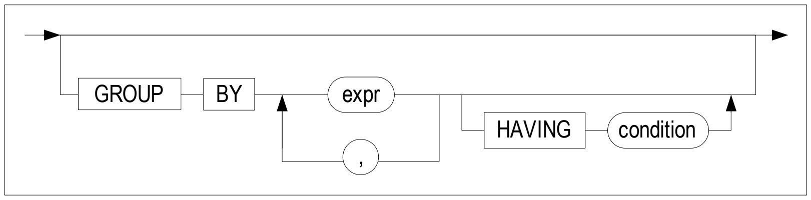
3.4.1.11 where_clause ::=



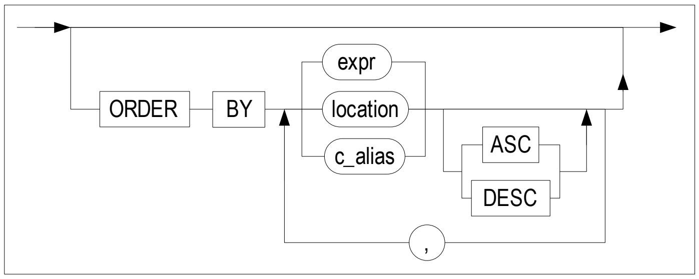
3.4.1.12 hierarchical_query_clause ::=



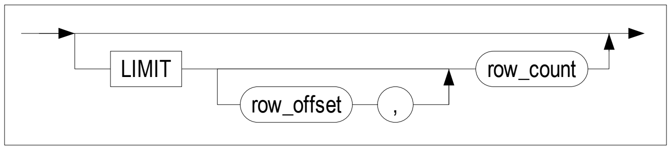
3.4.1.13 group_by_clause ::=



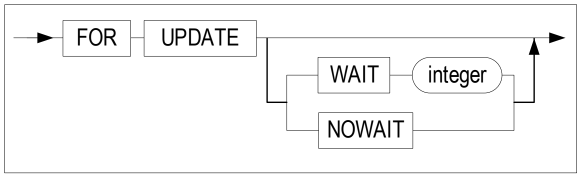
3.4.1.14 order_by_clause ::=



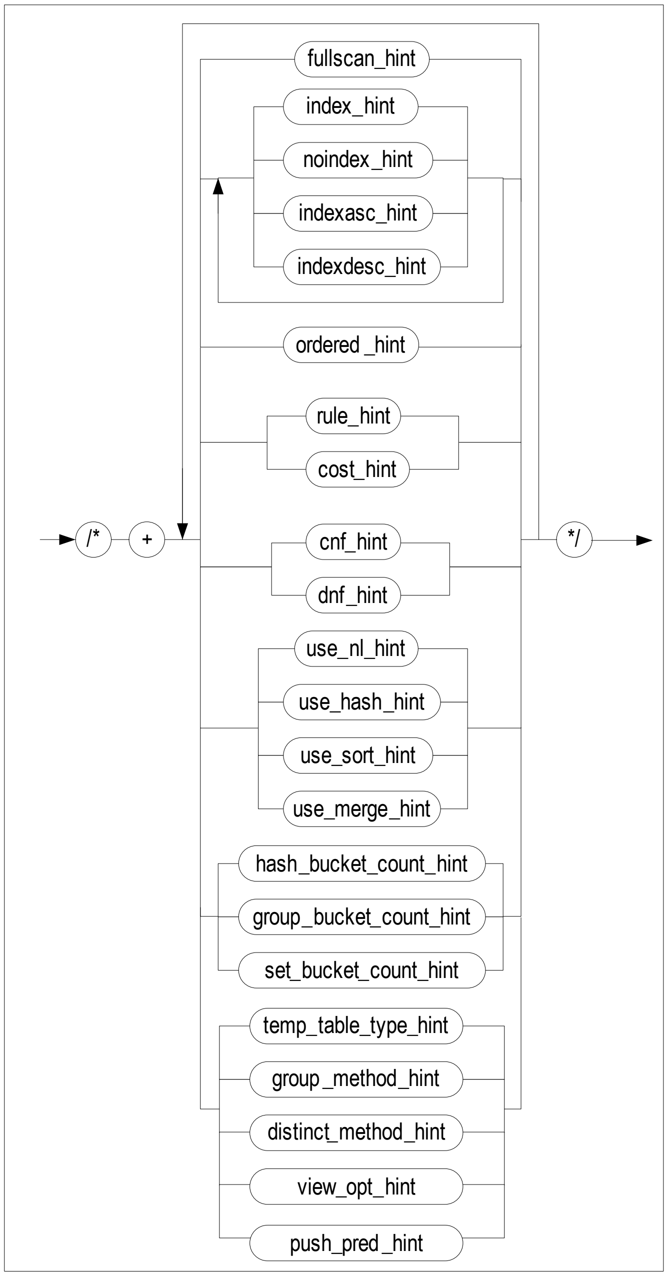
3.4.1.15 limit_clause ::=



3.4.1.16 for_update_clause ::=

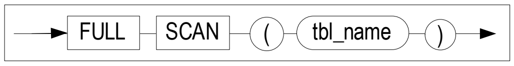


3.4.1.17 hints ::=

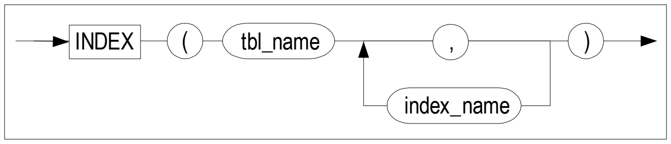


fullscan_hint ::=, *index_hint ::=*, *noindex_hint ::=*, *indexasc_hint ::=*, *indexdesc_hint ::=*, *ordered_hint ::=*, *rule_hint ::=*, *cost_hint ::=*, *cnf_hint ::=*, *dnf_hint ::=*, *use_nl_hint ::=*, *use_hash_hint ::=*, *use_sort_hint ::=*, *use_merge_hint ::=*, *hash_bucket_count_hint ::=*, *group_bucket_count_hint ::=*, *set_bucket_count_hint ::=*, *temp_table_type_hint ::=*, *group_method_hint ::=*, *distinct_method_hint ::=*, *view_opt_hint ::=*, *push_pred_hint ::=*

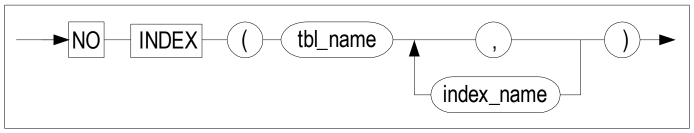
3.4.1.18 fullscan_hint ::=



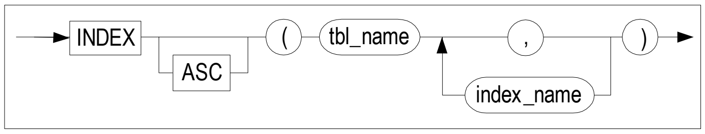
3.4.1.19 index_hint ::=



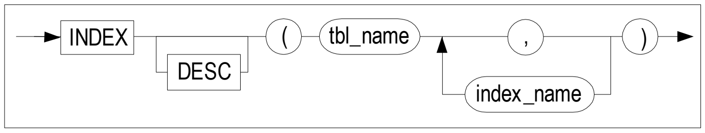
3.4.1.20 noindex_hint ::=



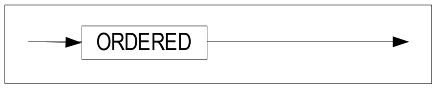
3.4.1.21 indexasc_hint ::=



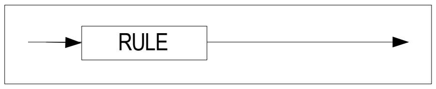
3.4.1.22 indexdesc_hint ::=



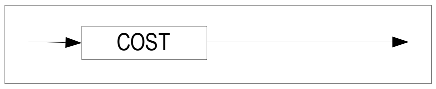
3.4.1.23 ordered_hint ::=



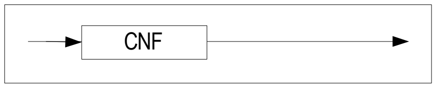
3.4.1.24 rule_hint ::=



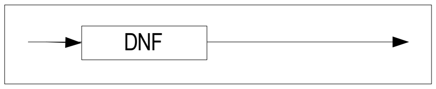
3.4.1.25 cost_hint ::=



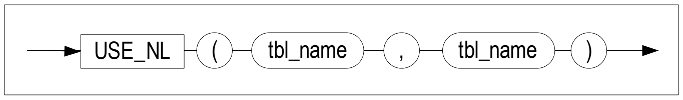
3.4.1.26 cnf_hint ::=



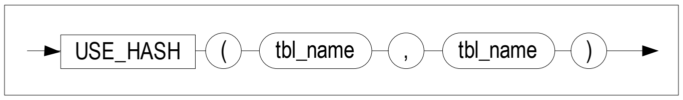
3.4.1.27 dnf_hint ::=



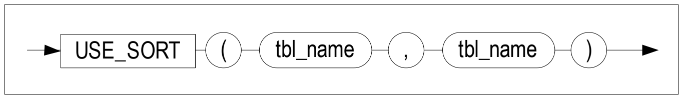
3.4.1.28 use_nl_hint ::=



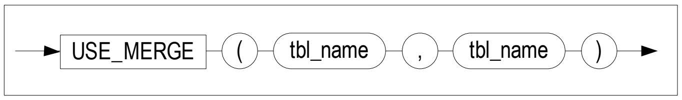
3.4.1.29 use_hash_hint ::=



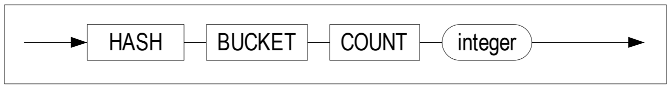
3.4.1.30 use_sort_hint ::=



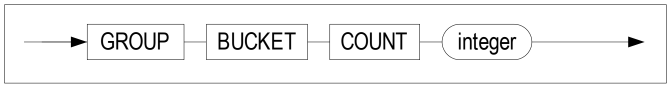
3.4.1.31 use_merge_hint ::=



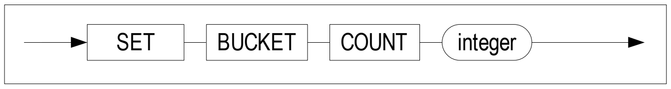
3.4.1.32 hash_bucket_count_hint ::=



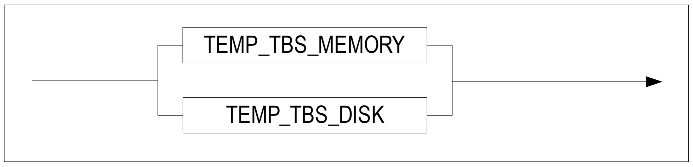
3.4.1.33 group_bucket_count_hint ::=



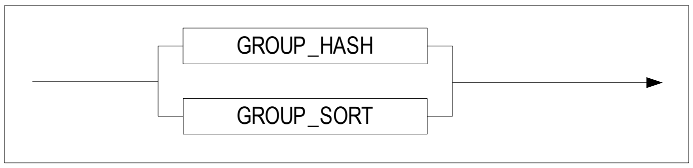
3.4.1.34 set_bucket_count_hint ::=



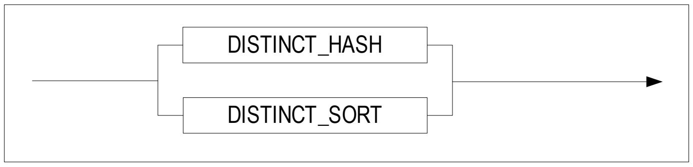
3.4.1.35 temp_table_type_hint ::=



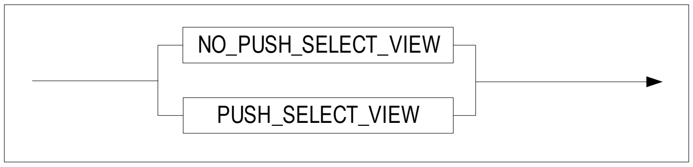
3.4.1.36 group_method_hint ::=



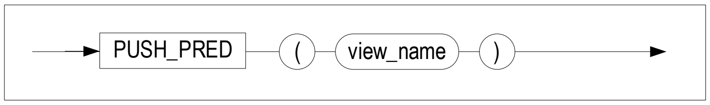
3.4.1.37 distinct_method_hint ::=



3.4.1.38 view_opt_hint ::=



3.4.1.39 push_pred_hint ::=



3.4.2 Prerequisites

Only the SYS user, the owner of the schema containing the table, users having the SELECT ANY TABLE system privilege, and users having the SELECT privilege for the specified table can select data from tables using this statement.

3.4.3 Description

A SELECT statement or subquery is used to retrieve data from one or more tables or views.

3.4.3.1 select_list clause

The DISTINCT keyword is used to specify that duplicate records are to be removed from the result set before it is returned.

If a SELECT statement contains a GROUP BY clause, then only constants, aggregate functions, the expressions in the GROUP BY clause, and expressions that are combinations of the foregoing can be specified in the SELECT list.

Hints can be specified immediately after the SELECT keyword. Hints are delimited like this:

```
/*+ your_hint */
```

For more information about hints, please refer to Chapter 11: SQL Tuning in the *Administrator's Manual*.

3.4.3.2 FROM clause

An alias cannot be used more than once in a FROM clause. When the same table is used more than once in a FROM clause, different aliases must be specified.

A maximum of 32 different tables or views can be cited in a FROM clause.

OUTER JOIN

This is an extended SQL form of JOIN for processing data that do not satisfy the join condition. Unlike an (INNER) JOIN, which only returns records having corresponding key values from two tables, an OUTER JOIN returns all of the data from one of the two tables. When a record in the returned result set comprises a row from one table that does not have a matching row from the other table, empty columns are filled with NULL values.

In-line Views

A subquery in a FROM clause is called an “inline view”.

pivot_clause

pivot_clause simultaneously performs a data aggregation operation and rearranges the data in separate rows into columns. This presents the data in a format that is easier to read than when using two columns in a GROUP BY clause.

pivot_clause performs the following steps:

1. *pivot_clause* first performs a grouping operation, just like a GROUP BY clause. The results are grouped according to all of the columns that are not referred to in *pivot_clause*, and according to the values specified in *pivot_in_clause*.

2. *pivot_clause* then arranges the resulting grouping columns and aggregate values in cross-tabular form.

For convenience, *pivot_clause* is typically used with an inline view, in order to prevent the output of large numbers of columns or the difficulty of specifying the names of specific columns resulting from the transformation operation.

pivot_for_clause

pivot_for_clause specifies one or more source columns whose values (specified in *pivot_in_clause*) are to be transposed to form columns.

pivot_in_clause

pivot_in_clause is used to specify values found in the columns specified in *pivot_for_clause*. These values will be used as column names in the pivot operation.

3.4.3.3 WHERE condition clause

For information about the use of conditions in the WHERE clause, please refer to Chapter8: SQL Conditions.

3.4.3.4 Hierarchical Query clause

Hierarchical queries are used to query tables containing hierarchical data. They return so-called "root records", meaning records that correspond to hierarchical roots, that satisfy given search conditions, along with all records that are subordinate to the root records.

When querying hierarchical data, do not use either the ORDER BY or the GROUP BY clause, because doing so would destroy the hierarchical order established using the CONNECT BY clause.

START WITH clause

This clause is for specifying the conditions that are used to identify which record(s) are to be used as the root record(s) for a hierarchical query. The database uses all record(s) that satisfy this condition as root record(s). If this clause is omitted, then ALTIBASE HDB treats all records in the table as root records.

This clause cannot be used without the CONNECT BY clause. However, it can be omitted when using the CONNECT BY clause.

The ROWNUM pseudocolum cannot be used in a START WITH clause.

CONNECT BY Clause

This clause is used to specify the conditions that identify the relationships between parent rows and child rows in the hierarchy.

To distinguish previously retrieved records from current records, the PRIOR operator is used. That is, the PRIOR operator must be used in order to reference parent records.

In queries that have a CONNECT BY clause, the PRIOR operator can be used only in a SELECT list, WHERE clause, or CONNECT BY clause.

A CONNECT BY clause cannot contain a subquery, and cannot be used with a join.

The CONNECT BY clause must be used after the WHERE clause and before any ORDER BY, GROUP BY, and HAVING clauses.

The PRIOR operator cannot be used in an ORDER BY clause when set operators (UNION, INTERSECT, etc.) are used in a query.

The LEVEL Pseudocolumn

SELECT statements that contain hierarchical queries can contain the LEVEL pseudocolumn in *select_list*. The LEVEL pseudocolumn indicates the hierarchical distance between the root record and subordinate records that have parent-child relationships between them. In other words, LEVEL is 1 for a root record, 2 for a child record, 3 for a grandchild record, and so on.

In addition to *select_list*, the LEVEL pseudocolumn can also be used in the WHERE, ORDER BY, GROUP BY, and HAVING clauses. Additionally, the LEVEL pseudocolumn can be used in *select_list* even in a query that does not have a CONNECT BY clause, for example:

```
select level from t1;
```

IGNORE LOOP

When the hierarchical relationships between records form a loop, ALTIBASE HDB returns an error. (In this context, the term "loop", in its simplest form, indicates the situation where one row is both the parent and child of another row.) However, if the IGNORE LOOP option is used, the formation of a loop during query execution does not raise an error; instead, the records that form the loop are removed from the query result set.

3.4.3.5 GROUP BY clause

The GROUP BY clause is used to group records that have the same value for one or more given expression(s) and return a single row of aggregate information for each group.

The groups that are returned cannot be limited using a WHERE condition. Instead, the HAVING clause is used to restrict the groups of returned rows to those groups for which the specified condition is TRUE. If the HAVING clause is omitted, one record will be returned for every group.

Locate the GROUP BY and HAVING clauses after the WHERE clause and *hierarchical_clause*. The ORDER BY clause, if present, must appear at the very end of an SQL statement.

3.4.3.6 HAVING condition clause

The HAVING clause can only be used when the GROUP BY clause is also present.

This clause is used to restrict the returned rows to those pertaining to groups for which the specified condition is TRUE. The HAVING clause can only comprise constants, aggregate functions, the expressions in the GROUP BY clause, and expressions that are combinations of the foregoing. If the HAVING clause is omitted, one record will be returned for every group.

For more information about the use of conditions in the HAVING clause, please refer to Chapter8: SQL Conditions.

3.4.3.7 UNION (ALL), INTERSECT, MINUS

The set operators combine the rows returned by two SELECT statements into a single result. The number and data types of the columns returned by each of the queries must be the same, but the column lengths can be different. The names of the columns in the result set will be the names of the expressions in the *select_list* preceding the set operator.

When set operators are used to combine more than two queries, the queries are evaluated and result sets joined from left to right. Parentheses can be used to specify a different order of evaluation.

For more about set operators, please refer to Chapter5: Set Operators in this manual.

3.4.3.8 ORDER BY clause

The ORDER BY clause is used to set the order in which the records returned by the statement are presented. The result set can be sorted in ascending or descending order. The default order is ascending order.

Without an ORDER BY clause, there is no guarantee that the resultant records will be returned in a consistent order when the same query is repeatedly executed.

An ORDER BY clause can be used only once in a SELECT statement. It cannot be used in a subquery.

If the elements in the ORDER BY are specified as expressions, the records are sorted according to the result of evaluation of the expressions. The expressions are based on columns in *select_list*, or in tables or views in the FROM clause. If the elements in the ORDER BY clause are specified according to their position in *select_list*, the search results are sorted according to the returned values. The positions must be indicated using integers.

When set operators (UNION, INTERSECT, etc.) are used, only the position or the alias of the search target can be used in the ORDER BY clause.

Multiple expressions can be specified in the ORDER BY clause. The result set is first sorted based on the values for the first expression. Records having the same values for the first expression are then sorted based on their values for the second expression, and so on.

If an ascending index exists for the column, the database places NULL values after all others when sorting in ascending order and preceding all others when sorting in descending order. If a descending index exists for the column, the database places NULL values preceding all others when sorting in ascending order and after all others when sorting in descending order. If no index exists for the column, the database places NULL values after all others, regardless of the sort order.

If the DISTINCT keyword is used in the SELECT statement, then only expressions that appear in the SELECT list or combinations of those expressions can be used in the ORDER BY clause.

If a GROUP BY clause is present, then the following expressions can be used in the ORDER BY clause:

- constants
- aggregate functions
- the expressions in the GROUP BY clause
- expressions that are combinations of the foregoing

3.4.3.9 LIMIT clause

The LIMIT clause can be used to limit the number of rows returned by a query.

- row_offset: This is used to specify the first record to return. If omitted, the first record in the result set will be returned.
- row_count: This is used to specify the number of records to retrieve.

The LIMIT clause can also be used in subqueries.

3.4.3.10 FOR UPDATE clause

This is used to lock the records in the result set so that other users cannot lock or edit the records until the transaction has been completed.

The WAIT keyword and subsequent value tells ALTIBASE HDB how long to wait to obtain the lock on the table. Alternatively, the NOWAIT keyword can be used, which instructs ALTIBASE HDB not to wait to obtain a lock if the table has already been locked by another user.

The FOR UPDATE clause can only be used with the main query of an SQL statement; it cannot be used with subqueries. Therefore, the following usage is invalid:

```
select eno from employees where (select eno from departments for update);
```

The FOR UPDATE clause cannot be used together with the DISTINCT or GROUP BY clauses, aggregate functions, or set operators (UNION, INTERSECT, etc).

3.4.3.11 HINTS Clause

For detailed information on hint syntax and a description of individual hints, please refer to "Chapter 11: SQL Tuning" in the *Administrator's Manual*.

3.4.4 Restrictions

The execution of SQL statements and stored procedures in ALTIBASE HDB is governed by the following restrictions:

- A maximum of 65536 internal tuples[1] can be used to process one query.
- A maximum of 32 tables or views can be used in a FROM clause.
- A maximum of 32 tables or views can be used in operations within clauses such as the WHERE, GROUP BY, and ORDER BY clauses.

If the above restrictions are violated, one of the following errors will be returned.

qpERR_ABORT_QTC_TUPLE_SHORTAGE

There are too many DML statements in the stored procedure, or the SQL query is too long.

qpERR_ABORT_QTC_TOO_MANY_TABLES

Too many tables are referenced in a phrase.

3.4.5 Examples

- Simple search command

  <Query> Retrieve the names, hiring dates, and salaries of all employees.

```
iSQL> SELECT e_firstname, e_lastname, join_date, salary
 FROM employees;
E_FIRSTNAME           E_LASTNAME            JOIN_DATE    SALARY
-----------------------------------------------------------------------
Chan-seung            Moon
Susan                 Davenport             18-NOV-2009  1500
Ken                   Kobain                11-JAN-2010  2000
.
.
.
20 rows selected.
```

- Searching a partitioned table.

```
CREATE TABLE T1 (I1 INTEGER)
```

1. Internal tuples are the units of memory that ALTIBASE HDB assigns internally for query processing.

```
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;

INSERT INTO T1 VALUES (55);

INSERT INTO T1 VALUES (123);

SELECT * FROM T1 PARTITION (P1);
 I1
----------
 55

SELECT * FROM T1 PARTITION (P2);
 I1
----------
 123

SELECT * FROM T1 PARTITION (P3);
No rows selected.
```

- Using search conditions

  <Query> Display the name, title, 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.
```

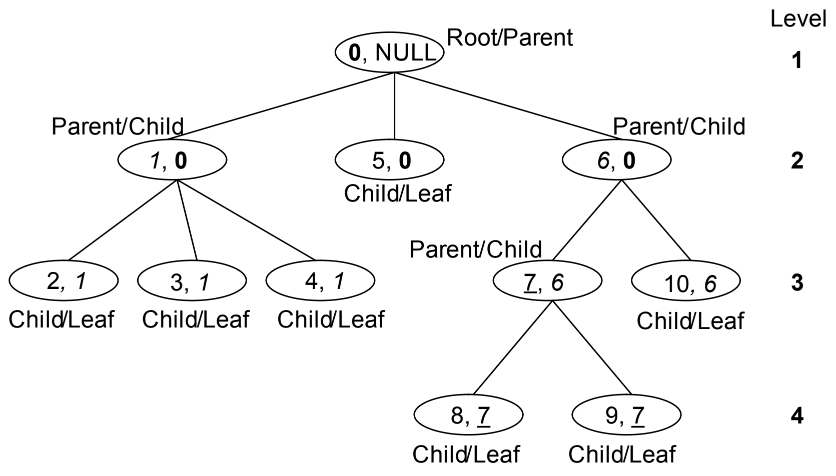
- Searching using a hierarchical query

  <Query> The following query uses a CONNECT BY clause to define a hierarchical relationship in which the value of *id* in the parent record is equal to the value of *parent_id* in the child record, starting with records for which the value in the *id* column is 0 as the root of the hierarchy.

```
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> 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.
```

Figure 3-1 Hierarchically Structured Data



<Query> The START WITH clause is omitted from the following query, meaning that all rows in the table are used as root rows. This query also returns all records that satisfy the condition (PRIOR id = parent).

```
iSQL> SELECT id, parent, level
FROM hier_order CONNECT BY PRIOR id = parent ORDER BY id;
ID       PARENT    LEVEL
-----------------------------------------------
0                  1
1        0         1
1        0         2
2        1         1
```

```
2        1        3
2        1        2
3        1        2
3        1        1
3        1        3
4        1        1
4        1        2
4        1        3
5        0        1
5        0        2
6        0        2
6        0        1
7        6        1
7        6        2
7        6        3
8        7        3
8        7        1
8        7        2
8        7        4
9        7        2
9        7        3
9        7        4
9        7        1
10       6        1
10       6        2
10       6        3
30 rows selected.
```

<Query>The following hierarchical query uses the IGNORE LOOP clause to remove records that formed loops during query execution from the result set and return the rest of the result set, rather than returning an error.

```
iSQL> CREATE TABLE triple(
  num INTEGER,
  tri INTEGER,
  PRIMARY KEY(num, tri));
Create success.
iSQL> CREATE OR REPLACE PROCEDURE proc_tri
AS
  v1 INTEGER;
BEGIN
  FOR v1 IN 1 .. 1000 LOOP
    INSERT INTO triple VALUES(v1, v1 * 3);
  END LOOP;
  INSERT INTO triple VALUES(1, 1);
END;
/
Create success.
iSQL> EXEC proc_tri;
Execute success.
iSQL> SELECT num, tri, level
  FROM triple
  WHERE num < 3001
    START WITH num = 1
    CONNECT BY PRIOR tri = num
    IGNORE LOOP;
NUM    TRI    LEVEL
-----------------------------------------------
1      1      1
1      3      2
3      9      3
9      27     4
27     81     5
81     243    6
```

```
243    729    7
729    2187   8
1      3      1
3      9      2
9      27     3
27     81     4
81     243    5
243    729    6
729    2187   7
15 rows selected.
```

- Searching Using the GROUP BY Clause

  <Query> Calculate the average salary for each department:

```
iSQL> SELECT dno, AVG(salary) AS avg_sal
 FROM employees
 GROUP BY dno;
DNO         AVG_SAL
---------------------------
1001        2150
1002        1340
1003        2438.25
2001        1400
3001        1800
3002        2500
4001        1550
4002        1396.66667
            1500
9 rows selected.
```

— All of the columns in *select_list* that are not operated on by an aggregate function must be present in the GROUP BY clause.

— To assign an alias to a column, or when it is desired to use an alias other than the name of the column, use the AS keyword after the name as shown above, and the desired column name will be output. The AS keyword can be omitted.

— Two hyphens ("--") indicates that the remainder of the line is to be handled as a comment.

<Query> Use a GROUP BY clause that references multiple columns to display the total wages paid to each position within each department.

```
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
--------------------------------------------------------------------
3002        CEO              1
            designer         1                    1500
1001        engineer         1                    2000
3001        PL               1                    1800
3002        PL               1                    2500
1002        programmer       1                    1700
4002        manager          1                    500
4001        manager          1
4001        planner          2                    3100
1003        programmer       1                    4000
1003        webmaster        2                    3750
4002        sales rep        3                    3690
1002        PM               1                    980
1003        PM               1                    2003
1001        manager          1                    2300
```

```
2001        PM               1                    1400
16 rows selected.
```

<Query> Display the average wage in departments for which the average wage is higher than $1500 USD.

This kind of query is often erroneously written as shown below:

```
iSQL> SELECT dno, AVG(salary)
  FROM employees
  WHERE AVG(salary) > 1500
  GROUP BY dno;
[ERR-31061 : An aggregate function is not allowed here.
0003 :   WHERE AVG(SALARY) > 1500000
              ^                    ^
]
```

To correct the above error, use the HAVING clause.

```
iSQL> SELECT dno, AVG(salary)
 FROM employees
 GROUP BY dno
 HAVING AVG(salary) > 1500;
DNO         AVG(SALARY)
---------------------------
1001        2150
1003        2438.25
3001        1800
3002        2500
4001        1550
5 rows selected.
```

<Query> Display the product number and the total number of products ordered, but only for orders for which the total number of items ordered is more than two:

```
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.
```

<Query> For all orders placed during the month of December for which the total number of items ordered was more than one, display the product number and the average number of products ordered. Order the results according to the number of items ordered.

```
iSQL> SELECT gno, AVG(qty) month_avg
  FROM orders
  WHERE order_date BETWEEN '01-Dec-2000' AND '31-Dec-2000'
  GROUP BY gno
  HAVING COUNT(*) > 1
  ORDER BY AVG(qty);
GNO          MONTH_AVG
---------------------------
A111100002   35
D111100003   300
D111100004   750
C111100001   1637.5
```

```
D111100010   1750
D111100002   1750
E111100012   4233.33333
D111100008   5500
8 rows selected.
```

- Searching using ORDER BY

  <Query> Display the names, department numbers, and wages of all employees. Sort them according to department number and then according to wage in descending order.

```
iSQL> SELECT e_firstname, e_lastname, dno, salary
 FROM employees
 ORDER BY dno, salary DESC;
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
.
.
.
20 rows selected.
```

  <Query> Display the names and wages of all employees, sorted first by department number and then by wage in descending order. (Note that it is possible to sort the results on the basis of columns that do not appear in *select_list*.)

```
iSQL> SELECT e_firstname, e_lastname, salary
 FROM employees
 ORDER BY dno, salary DESC;
E_FIRSTNAME           E_LASTNAME            SALARY
------------------------------------------------------------
Wei-Wei               Chen                  2300
Ken                   Kobain                2000
Ryu                   Momoi                 1700
Mitch                 Jones                 980
Elizabeth             Bae                   4000
.
.
.
20 rows selected.
```

- Searching using an operator

  <Query> Display the name and the total value of all inventory for each inventory item.

```
iSQL> SELECT gname, (stock*price) inventory_value
 FROM goods;
GNAME      INVENTORY_VALUE
-------------------------------------
IM-300     78000000
IM-310     9800000
NT-H5000   27924000
.
.
.
30 rows selected.
```

- Searching using an alias

<Query> Specify an alias for the *dep_location* column.

```
iSQL> SELECT dname, 'District Name', dep_location location
 FROM departments;
DNAME                        'District Name'       LOCATION
-----------------------------------------------
Applied Technology Team       District Name        Mapo
Engine Development Team       District Name        Yeoido
Marketing Team                District Name        Gangnam
Planning & Management Team    District Name        Gangnam
Sales Team                    District Name        Shinchon
5 rows selected.
```

- Searching using a LIMIT clause

<Query> Display the names of only five employees from the *employees* table, starting with the 3rd record.

```
iSQL> SELECT e_firstname first_name, e_lastname last_name
 FROM employees
 LIMIT 3, 5;
FIRST_NAME            LAST_NAME
-----------------------------------------------
Ken                   Kobain
Aaron                 Foster
Farhad                Ghorbani
Ryu                   Momoi
Gottlieb              Fleischer
5 rows selected.
```

<Query> Display the name and the wage of the first employee in the *managers* table.

```
iSQL> CREATE TABLE managers(
 mgr_no INTEGER PRIMARY KEY,
 m_lastname VARCHAR(20),
 m_firstname VARCHAR(20),
 address VARCHAR(60));
Create success.
iSQL> INSERT INTO managers VALUES(7, 'Fleischer', 'Gottlieb', '44-25
YouIDo-dong Youngdungpo-gu Seoul Korea');
1 row inserted.
iSQL> INSERT INTO managers VALUES(8, 'Wang', 'Xiong', '3101 N Wabash Ave
Brooklyn NY');
1 row inserted.
iSQL> INSERT INTO managers VALUES(12, 'Hammond', 'Sandra', '130 Gongpy-
eongno Jung-gu Daegu Korea');
1 row inserted.
iSQL> SELECT e_firstname, e_lastname, salary FROM employees WHERE eno =
(SELECT mgr_no FROM managers LIMIT 1);
E_FIRSTNAME           E_LASTNAME            SALARY
------------------------------------------------------------
Gottlieb              Fleischer             500
1 row selected.
iSQL>
```

- Searching using FOR UPDATE

The following example shows how to use the FOR UPDATE clause.

Transaction A	Time Point	Transaction B

`iSQL> AUTOCOMMIT OFF;`
`Set autocommit off success.`
		`iSQL> AUTOCOMMIT OFF;`
		`Set autocommit off success.`

(request X lock on *employees*)
`iSQL> LOCK TABLE employees IN EXCLUSIVE MODE;`
`Lock success.`
(acquire X lock on *employees*)
`iSQL> SELECT e_lastname`
` FROM employees`
` WHERE eno = 15;`
`E_LASTNAME`
`------------------------`
`Davenport`
`1 row selected.`
	1	

	2	`iSQL> SELECT e_lastname`
		` FROM employees`
		` WHERE eno = 15`
		` FOR UPDATE;`
		(the request conflicts with the X lock already held by transaction A)
		wait
		wait
		wait

`iSQL> UPDATE employees`
` SET ENO = 30`
` WHERE eno = 15;`
`1 row updated.`
`iSQL> SELECT e_lastname`
` FROM employees`
` WHERE eno = 30;`
`E_LASTNAME`
`------------------------`
`Davenport`
`1 row selected.`
	3	

`iSQL> COMMIT;`
`Commit success.`
	4	

	5	(resume)
		`E_LASTNAME`
		`------------------------`
		`No rows selected.`

- Search using Hints

1. Using Table Access Method Hints

   - full scan, index scan, index ascending order scan, index descending order scan, no index scan

   The following query retrieves the employee number, name, and position of all female employees.

```
SELECT eno, e_firstname, e_lastname, emp_job FROM employees WHERE
sex = 'F';
```

   For example, assume that an index has been defined for the *sex* column of the *employees* table, which contains many records, and that the value of the column can be 'M' or 'F'.

If the number of male employees is similar to the number of female employees, querying the entire table using a full scan will be much faster than using an index scan. However, if the number of female employees is substantially lower than the number of male employees, using an index scan will be faster than scanning the entire table.

In other words, when a column contains only two different values, the optimizer assumes that each value accounts for 50% of the rows in a table, and therefore, when using a cost-based approach to find records that match one of the two values for that column, opts to perform a full table scan rather than an index scan.

In the following queries, it can be seen that the table is accessed 20 times and 4 times in order to perform a full scan and an index scan, respectively.

<Query> Display the employee number, name, and position of all female employees (by performing a full scan).

```
iSQL> SELECT /*+ FULL SCAN(employees) */ eno, e_firstname,
e_lastname, emp_job
 FROM employees
 WHERE sex = 'F';
ENO E_FIRSTNAME E_LASTNAME EMP_JOB
-----------------------------------------------
.
.
.
-----------------------------------------------
PROJECT ( COLUMN_COUNT: 4, TUPLE_SIZE: 65 )
 SCAN ( TABLE: EMPLOYEES, FULL SCAN, ACCESS: 20, SELF_ID: 2 )
-----------------------------------------------
```

<Query> Display the employee number, name, and position of all female employees (by performing an index scan).

```
iSQL> CREATE INDEX gender_index ON employees(sex);
Create success.
iSQL> SELECT /*+ INDEX(employees, gender_INDEX) use gender_index
because there are few female employees */ eno, e_firstname,
e_lastname, emp_job
 FROM employees
 WHERE sex = 'F';
ENO E_FIRSTNAME E_LASTNAME EMP_JOB
-----------------------------------------------
.
.
.
-----------------------------------------------
PROJECT ( COLUMN_COUNT: 4, TUPLE_SIZE: 65 )
 SCAN ( TABLE: EMPLOYEES, INDEX: GENDER_INDEX, ACCESS: 4, SELF_ID: 2
)
-----------------------------------------------
```

<Query> Display the order number, product number, and quantity for all orders placed during the first quarter (by performing an index scan). Assume that the name of the order table for each month is *orders_##*.

```
create view orders as
select ono, order_date, eno, cno, gno, qty from orders_01
union all
select ono, order_date, eno, cno, gno, qty from orders_02
union all
select ono, order_date, eno, cno, gno, qty from orders_03;
create index order1_gno on orders_01(gno);
create index order2_gno on orders_02(gno);
create index order3_gno on orders_03(gno);

iSQL> select /*+ index( orders,
           orders1_gno, orders2_gno,orders3_gno ) */
           ONO, GNO, QTY
      from orders;
ONO                  GNO         QTY
-------------------------------------------------
.
.
.
------------------------------------------------
PROJECT ( COLUMN_COUNT: 3, TUPLE_SIZE: 24 )
 VIEW ( ORDERS, ACCESS: 14, SELF_ID: 6 )
  PROJECT ( COLUMN_COUNT: 6, TUPLE_SIZE: 48 )
   VIEW ( ACCESS: 14, SELF_ID: 5 )
    BAG-UNION
     PROJECT ( COLUMN_COUNT: 6, TUPLE_SIZE: 48 )
     SCAN ( TABLE: ORDERS_01, INDEX: ORDERS1_GNO, ACCESS: , SELF_ID:
0 )
     PROJECT ( COLUMN_COUNT: 6, TUPLE_SIZE: 48 )
      SCAN ( TABLE: ORDERS_02, INDEX: ORDERS2_GNO, ACCESS: 4,
SELF_ID: 1 )
     PROJECT ( COLUMN_COUNT: 6, TUPLE_SIZE: 48 )
      SCAN ( TABLE: ORDERS_03, INDEX: ORDERS3_GNO, ACCESS: 7,
SELF_ID: 4 )
------------------------------------------------
```

2. Join Order Hints (ordered, optimized)

<Query> Retrieve the employee number and last name and the customer's last name for every ordered product. (Use the ORDERED hint to join the *employees* table with the *customers* table and then join the result set with the *orders* table.)

```
iSQL> SELECT /*+ ORDERED */ DISTINCT o.eno, e.e_lastname,
c.c_lastname
FROM employees e, customers c, orders o
WHERE e.eno = o.eno AND o.cno = c.cno;
ENO E_LASTNAME C_LASTNAME
------------------------------------------------
.
.
.
------------------------------------------------
PROJECT ( COLUMN_COUNT: 3, TUPLE_SIZE: 48 )
 DISTINCT ( ITEM_SIZE: 40, ITEM_COUNT: 21, BUCKET_COUNT: 1024,
ACCESS: 21, SELF_ID: 4, REF_ID: 3 )
 JOIN
 JOIN
 SCAN ( TABLE: EMPLOYEES E, FULL SCAN, ACCESS: 20, SELF_ID: 1 )
 SCAN ( TABLE: CUSTOMERS C, FULL SCAN, ACCESS: 400, SELF_ID: 2 )
 SCAN ( TABLE: ORDERS O, FULL SCAN, ACCESS: 12000, SELF_ID: 3 )
------------------------------------------------
```

<Query>Retrieve the employee number and last name and the customer's last name for every ordered product. (Allow the optimizer to set the order in which tables are joined without considering the order in which the tables appear in the FROM clause.)

```
iSQL> SELECT DISTINCT o.eno, e.e_lastname, c.c_lastname
FROM employees e, customers c, orders o
WHERE e.eno = o.eno AND o.cno = c.cno;
ENO E_LASTNAME C_LASTNAME
------------------------------------------------
.
.
.
```

```
------------------------------------------------
PROJECT ( COLUMN_COUNT: 3, TUPLE_SIZE: 48 )
 DISTINCT ( ITEM_SIZE: 40, ITEM_COUNT: 21, BUCKET_COUNT: 1024,
ACCESS: 21, SELF_ID: 4, REF_ID: 1 )
 JOIN
 JOIN
 SCAN ( TABLE: CUSTOMERS C, FULL SCAN, ACCESS: 20, SELF_ID: 2 )
 SCAN ( TABLE: ORDERS O, INDEX: ODR_IDX2, ACCESS: 30, SELF_ID: 3 )
 SCAN ( TABLE: EMPLOYEES E, INDEX: __SYS_IDX_ID_366, ACCESS: 30,
SELF_ID: 1 )
------------------------------------------------
```

3. Optimizer Mode Hints (RULE, COST)

```
iSQL> SELECT /*+ RULE */ * FROM t1, t2 WHERE t1.i1 = t2.i1;
iSQL> SELECT /*+ COST */ * FROM t1, t2 WHERE t1.i1 = t2.i1;
```

4. Normal Form Hints (CNF, DNF)

```
iSQL> SELECT /*+ CNF */ * FROM t1 WHERE i1 = 1 OR i1 = 2;
iSQL> SELECT /*+ DNF */ * FROM t1 WHERE i1 = 1 OR i1 = 2;
```

5. Join Method Hints (nested loop, hash, sort, sort merge)

```
iSQL> SELECT /*+ USE_NL (t1,t2) */ * FROM t1, t2 WHERE t1.i1 = t2.i1;
iSQL> SELECT /*+ USE_HASH (t1,t2) */ * FROM t1, t2 WHERE t1.i1 =
t2.i1;
iSQL> SELECT /*+ USE_SORT (t1,t2) */ * FROM t1, t2 WHERE t1.i1 =
t2.i1;
iSQL> SELECT /*+ USE_MERGE (t1,t2) */ * FROM t1, t2 WHERE t1.i1 =
t2.i1;
```

6. Hash Bucket Size Hints (hash bucket count, group bucket count)

```
iSQL> SELECT /*+ HASH BUCKET COUNT (20) */ DISTINCT * FROM t1;
iSQL> SELECT * FROM t1 GROUP BY i1, i2;
iSQL> SELECT /*+ GROUP BUCKET COUNT (20) */ * FROM t1 GROUP BY i1,
i2;
iSQL> SELECT * FROM t1 INTERSECT SELECT * FROM t2;
iSQL> SELECT /*+ SET BUCKET COUNT (20) */ * FROM t1 INTERSECT SELECT
* FROM t2;
```

7. Push Predicate Hint

<Query> For all orders placed during the first quarter for which the number of items ordered at one time was more than 10000, display the customer name and the product number. (Use the Push Predicate hint to join the customers table with the orders table.)

```
create view orders as
select ono, order_date, eno, cno, gno, qty from orders_01
union all
select ono, order_date, eno, cno, gno, qty from orders_02
union all
select ono, order_date, eno, cno, gno, qty from orders_03;
```

```
iSQL> select /*+ PUSH_PRED(orders) */ c_lastname, gno
    2   from customers, orders
    3  where customers.cno = orders.cno
    4    and orders.qty >= 10000;
C_LASTNAME            GNO
------------------------------------
.
.
.
------------------------------------------------
PROJECT ( COLUMN_COUNT: 2, TUPLE_SIZE: 34 )
 JOIN
  SCAN ( TABLE: CUSTOMERS, FULL SCAN, ACCESS: 20, SELF_ID: 2 )
  FILTER
   [ FILTER ]
   AND
    OR
```

```
        ORDERS.QTY >= 10000
      VIEW ( ORDERS, ACCESS: 1, SELF_ID: 8 )
       PROJECT ( COLUMN_COUNT: 6, TUPLE_SIZE: 48 )
        VIEW ( ACCESS: 1, SELF_ID: 7 )
         BAG-UNION
         PROJECT ( COLUMN_COUNT: 6, TUPLE_SIZE: 48 )
           SCAN ( TABLE: ORDERS_01, INDEX: ODR1_IDX2, ACCESS: 3,
SELF_ID: 3 )
            [ VARIABLE KEY ]
            OR
             AND
            [ FILTER ]
            AND
             OR
          PROJECT ( COLUMN_COUNT: 6, TUPLE_SIZE: 48 )
           SCAN ( TABLE: ORDERS_02, INDEX: ODR2_IDX2, ACCESS: 4,
SELF_ID: 4 )
            [ VARIABLE KEY ]
            OR
             AND
            [ FILTER ]
            AND
             OR
          PROJECT ( COLUMN_COUNT: 6, TUPLE_SIZE: 48 )
           SCAN ( TABLE: ORDERS_03, INDEX: ODR3_IDX2, ACCESS: 7,
SELF_ID: 6 )
            [ VARIABLE KEY ]
            OR
             AND
            [ FILTER ]
            AND
             OR
------------------------------------------------
```

- Searching using OUTER JOIN

<Query> Retrieve the department numbers and employee names for all departments. (Note that the department number 5001 is output even though there are no employees in that department.)

```
iSQL> INSERT INTO departments VALUES('5001', 'Quality Assurance', 'Jon-
glo', 22);
1 row inserted.
iSQL> SELECT d.dno, e.e_lastname
FROM departments d LEFT OUTER JOIN employees e ON d.dno = e.dno
ORDER BY d.dno;
DNO E_LASTNAME
-------------------------------
.
5001
.
```

<Query> Retrieve the department numbers and employee names for all departments. (Note that “Davenport” is returned even though she does not belong to any department.)

```
iSQL> SELECT d.dno, e.e_lastname
FROM departments d RIGHT OUTER JOIN employees e ON d.dno = e.dno
ORDER BY d.dno;
DNO E_LASTNAME
-------------------------------
.
            Davenport
.
```

<Query>Retrieve the name and numbers of all departments and the numbers of all products. Where possible, indicate the department at which the products are located.

```
iSQL> INSERT INTO departments VALUES('6002', 'headquarters', 'CE0002',
100);
1 row inserted.
iSQL> SELECT d.dno, d.dname, g.gno
FROM departments d FULL OUTER JOIN goods g
 ON d.dep_location = g.goods_location;
DNO         DNAME                           GNO
------------------------------------------------------------
.
6002        headquarters                    E111100005
.
```

- Search using In-line View

<Query> Retrieve the name, wage, and department of every employee who earns a higher wage than the average wage in his or her department, along with the average wage in that department.

```
iSQL> SELECT 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;
ENAME   SALARY   DNO   SALAVG
------------------------------------------------
.
.
.
```

- Search using a PIVOT clause

Retrieve the number of men and women who work in each department.

```
iSQL> SELECT * FROM
 (SELECT d.dname, e.sex
   FROM departments d, employees e
   WHERE d.dno = e.dno)
 PIVOT (COUNT(*) FOR sex in ('M', 'F'))
 ORDER BY dname;
DNAME                           'M'                  'F'
-------------------------------------------------------------------------
BUSINESS DEPT                   3                    1
CUSTOMERS SUPPORT DEPT          1                    0
MARKETING DEPT                  3                    0
PRESALES DEPT                   2                    0
QUALITY ASSURANCE DEPT          1                    0
RESEARCH DEVELOPMENT DEPT 1     1                    1
RESEARCH DEVELOPMENT DEPT 2     2                    0
SOLUTION DEVELOPMENT DEPT       3                    1
8 rows selected.
```

For comparison, note that the same information can be output using only GROUP BY and ORDER BY clauses, but that is it much harder to read:

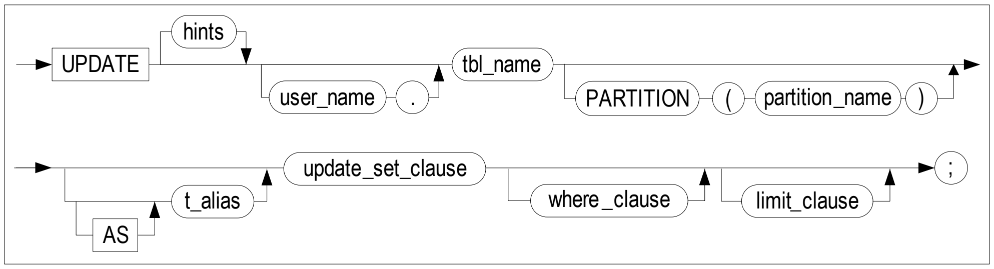
```
iSQL> SELECT d.dname, e.sex, count(*) FROM departments d, employees e
WHERE d.dno = e.dno GROUP BY d.dname, e.sex ORDER BY d.dname, e.sex DESC;
DNAME                           SEX  COUNT
-------------------------------------------------------------
BUSINESS DEPT                   M  3
BUSINESS DEPT                   F  1
```

```
CUSTOMERS SUPPORT DEPT          M  1
MARKETING DEPT                  M  3
PRESALES DEPT                   M  2
QUALITY ASSURANCE DEPT          M  1
RESEARCH DEVELOPMENT DEPT 1     M  1
RESEARCH DEVELOPMENT DEPT 1     F  1
RESEARCH DEVELOPMENT DEPT 2     M  2
SOLUTION DEVELOPMENT DEPT       M  3
SOLUTION DEVELOPMENT DEPT       F  1
11 rows selected.
```

3.5 UPDATE

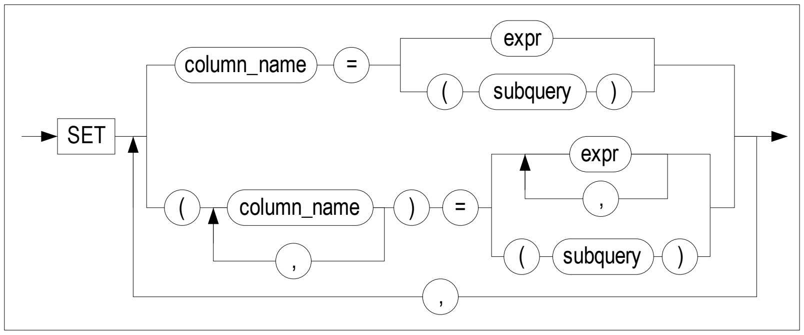
3.5.1 Syntax

3.5.1.1 update ::=



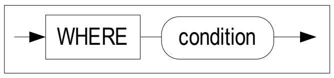
update_set_clause ::=, *where_clause ::=*, *limit_clause ::=*

3.5.1.2 update_set_clause ::=

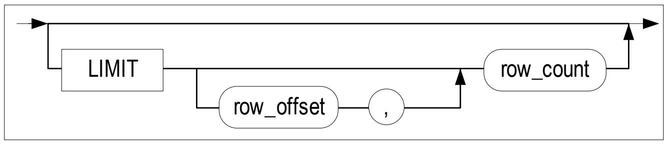


subquery ::=

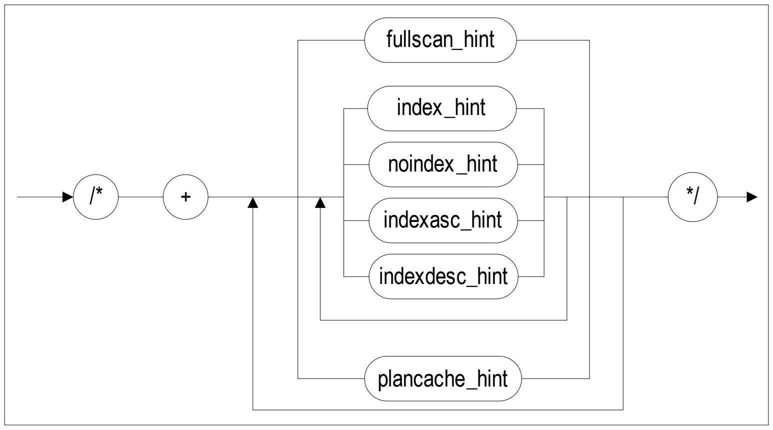
3.5.1.3 where_clause ::=



3.5.1.4 limit_clause ::=

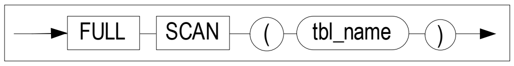


3.5.1.5 hints ::=

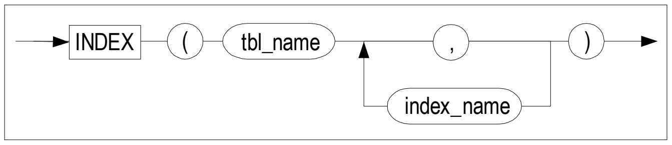


fullscan_hint ::=, *index_hint ::=*, *noindex_hint ::=*, *indexasc_hint ::=*

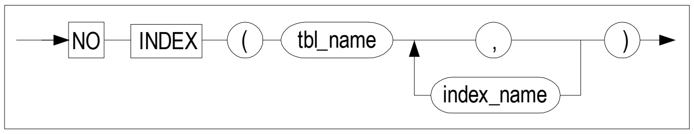
3.5.1.6 fullscan_hint ::=



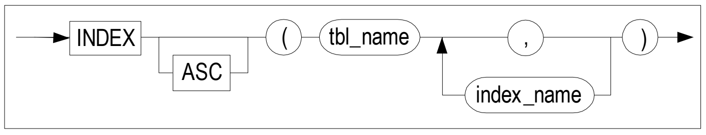
3.5.1.7 index_hint ::=



3.5.1.8 noindex_hint ::=



3.5.1.9 indexasc_hint ::=



3.5.2 Prerequisites

Only the SYS user, the owner of the schema containing the table, users having the UPDATE ANY TABLE system privilege, and users having the UPDATE privilege for the specified table can update values in tables using this statement.

3.5.3 Description

The UPDATE statement is used to find records that satisfy specific conditions and change the values in specified columns.

When a partition is specified, the column values are changed for the records that satisfy the conditions and are located in the specified partition.

3.5.3.1 user_name

This is used to specify the name of the owner of the table containing the record(s) to be changed. If omitted, ALTIBASE HDB will assume that the table belongs to the schema of the user connected via the current session.

3.5.3.2 tbl_name

This is used to specify the name of the table containing the record(s) to be changed.

3.5.3.3 subquery

A subquery within a SET clause can be used to update data.

1. subquery must return one row for each row updated.
2. if only one column is specified in a SET clause, the subquery must return only one value.
3. if multiple columns are specified in a SET clause, the number of values returned by the subquery must be the same as the number of columns specified.
4. If no rows are returned by a subquery within a SET clause, the specified column(s) will be populated with NULL values.

3.5.3.4 Modifying the data in a timestamp column

When an UPDATE statement is executed on a TIMESTAMP column, the default behavior is to update the column with the system time. Therefore, if no value is specified when a TIMESTAMP column is updated, it will be updated with the system time, rather than NULL.

Another way to update a TIMESTAMP column with the system time is to use the DEFAULT keyword in the UPDATE statement.

3.5.4 Hint Options

For detailed information about hint options, please refer to 3.1.4 HINTS Options in the DELETE statement.

3.5.5 Considerations

The same column cannot be used more than once in a SET clause.

When data corresponding to a partition key are changed such that the records containing the data need to be moved from one partition to another, it will be possible to move the data if the table attributes are set such that records can be moved between partitions, that is, if the table was created with the ENABLE ROW MOVEMENT option, or if the ALTER TABLE ENABLE ROW MOVEMENT command has been executed on the table. However, if the table attributes have not been set appropriately, an error will be raised.

It is impossible to insert a NULL value or change a value to NULL in a column having the NOT NULL constraint.

3.5.6 Examples

- Updating a column

  <Query> Change the wage of any employees whose last name is “Davenport” to 2500.

```
iSQL> UPDATE employees
SET salary = 2500
WHERE e_lastname = 'Davenport';
1 row updated.
```

  <Query> Increase all employees’ wages by 7%.

```
iSQL> UPDATE employees
SET salary = salary * 1.07;
20 rows updated.
```

- Updating records using a subquery in the WHERE clause

  <Query> Subtract 50 from the quantity of all orders taken by any employee whose last name is “Hammond”.

```
iSQL> UPDATE orders
SET qty = qty - 50
WHERE eno IN(
 SELECT eno
 FROM employees
 WHERE e_lastname ='Hammond');
9 rows updated.
```

- Update data in a partitioned table.

```
iSQL> UPDATE T1 PARTITION(P1) SET I1 = 200;
```

- Updating records using a subquery in the SET clause

  <Query> The following example shows the structure of an UPDATE statement containing two nested SELECT subqueries.

```
iSQL> CREATE TABLE bonuses
 (eno INTEGER, bonus NUMBER(10, 2) DEFAULT 100, commission NUMBER(10, 2)
DEFAULT 50);
Create success.
iSQL> INSERT INTO bonuses(eno)
 (SELECT e.eno FROM employees e, orders o
 WHERE e.eno = o.eno
 GROUP BY e.eno);
3 rows inserted.
iSQL> SELECT * FROM bonuses;
BONUSES.ENO BONUSES.BONUS BONUSES.COMMISSION
------------------------------------------------
12          100           50
19          100           50
20          100           50
3 rows selected.
iSQL> UPDATE bonuses
SET eno = eno + 100, (bonus, commission) =
 (SELECT 1.1 * AVG(bonus), 1.5 * AVG(commission) FROM bonuses)
 WHERE eno IN
 (SELECT eno
 FROM orders
 WHERE qty >= 10000);
1 row updated.
iSQL> SELECT * FROM bonuses;
BONUSES.ENO BONUSES.BONUS BONUSES.COMMISSION
------------------------------------------------
12          100           50
20          100           50
119         110           75
3 rows selected.
```

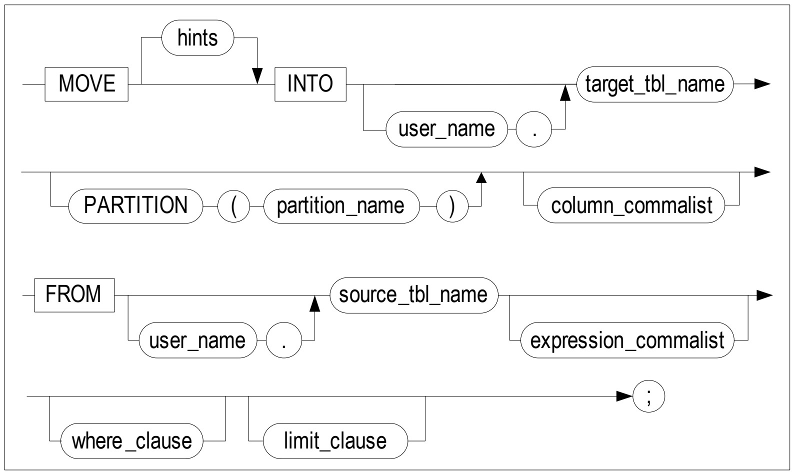
Note: *If a subquery in a WHERE clause does not return any records, no records will be affected, whereas if a subquery in a SET clause does not return any records, the corresponding columns will be updated with NULL values.*

```
iSQL> UPDATE orders
SET qty = qty - 50
WHERE eno IN(
 SELECT eno
 FROM employees
 WHERE e_lastname ='Frederick');
No rows updated.
iSQL> UPDATE employees
SET dno =
 (SELECT dno
 FROM departments
 WHERE dep_location = 'Timbuktu');
20 rows updated.
iSQL> SELECT e_lastname, dno
 FROM employees
 WHERE eno = 12;
E_LASTNAME DNO
-------------------------------
Hammond
1 row selected.
```

3.6 MOVE

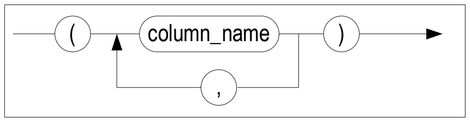
3.6.1 Syntax

3.6.1.1 move ::=

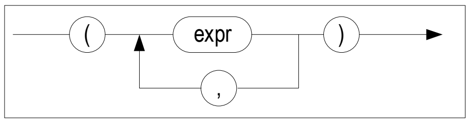


column_commalist ::=, *expression_commalist ::=*, *where_clause ::=*, *limit_clause ::=*

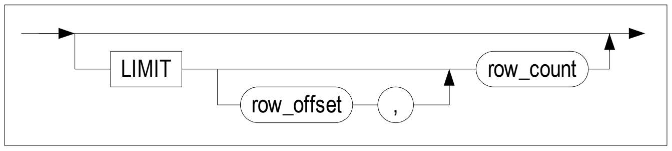
3.6.1.2 column_commalist ::=



3.6.1.3 expression_commalist ::=



3.6.1.4 limit_clause ::=



3.6.2 Prerequisites

In order to move table records, it is necessary to have both DELETE privileges for the table from which the records are to be moved and INSERT privileges for the table to which the records are to be moved. This is because moving data involves both inserting and deleting data.

In order to insert records into the table specified in the INTO clause, it is necessary to be the SYS user or the owner of the schema containing the table, to have the INSERT ANY TABLE system privilege, or to have the INSERT privilege for the table.

In order to delete records from the table specified in the FROM clause, it is necessary to be the SYS user or the owner of the schema containing the table, to have the DELETE ANY TABLE system privilege, or to have the DELETE privilege for the table.

3.6.3 Description

The MOVE statement is used to move data that satisfy certain conditions from one table to another. It is also possible to move data to a specified partition.

3.6.3.1 hints

The use of hints is supported with the FROM clause, and is the same as when using hints in a SELECT statement.

3.6.3.2 source_tbl_name, target_tbl_name

These are used to specify the names of the tables from and to which the data will be moved. They must not be views or meta tables.

3.6.3.3 column_commalist

This is a list of actual columns belonging to the target table.

3.6.3.4 expression_commalist

This is a comma-delimited list of expressions. Each expression can be a column belonging to the FROM table, a constant or an expression.

3.6.3.5 where_clause

The structure of the WHERE clause is the same as that in a SELECT statement.

3.6.3.6 limit_clause

The structure of the LIMIT clause is the same as that in a SELECT statement.

3.6.4 Considerations

Data cannot be moved from one table to the same table.

When a partition is specified, a value that is inappropriate for the partition cannot be entered.

3.6.5 Examples

<Query>Move all records that satisfy the condition (T2.I2 = 4) from columns *I1* and *I2* of table *T2* into the corresponding columns in table *T1* and delete the original records from table *T2*.

```
iSQL> MOVE INTO T1(I1, I2) FROM T2(I1, I2) WHERE T2.I2 = 4;
```

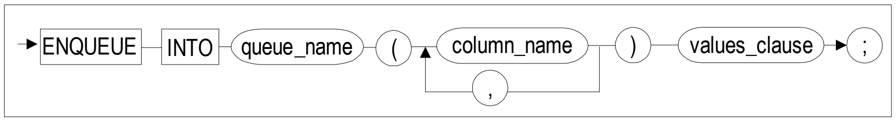
<Query> Insert records comprising columns *I1*, *I2*, and *I3* from table *T2* into table *T1* and delete them from table *T2*. (Table *T1* must have columns corresponding to columns *I1*, *I2*, and *I3* in table *T2*, and must have the same number of columns, that is, three columns.)

```
iSQL> MOVE INTO T1 FROM T2 (I1, I2, I3);
```

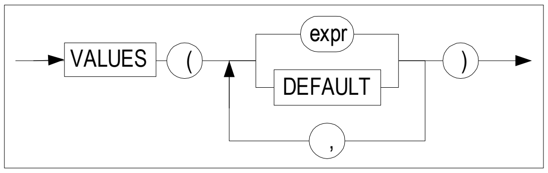
3.7 ENQUEUE

3.7.1 Syntax

3.7.1.1 enqueue ::=



3.7.1.2 values_clause ::=



3.7.2 Description

This statement is used to insert a message into a queue. The structure of the ENQUEUE statement is similar to that of the INSERT statement. The names of one or more queue columns must be specified after the INTO clause.

In most cases, a user simply enters a message to store. However, in cases where it is necessary to sort or classify messages, it is possible to additionally specify a user-defined Correlation ID for the message to be entered.

3.7.3 Examples

<Query>Enter a message consisting of the text “This is a message” to message queue *Q1*.

```
ENQUEUE INTO Q1(message) VALUES ('This is a message');
```

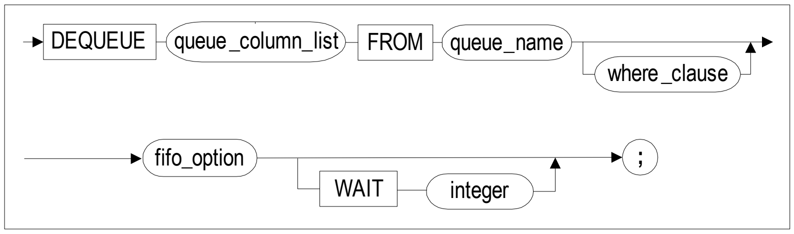
<Query> Enter a message consisting of the text “This is a message” to message queue Q1 with Correlation ID 237.

```
ENQUEUE INTO Q1(message,corrid) VALUES ('This is a message', 237);
```

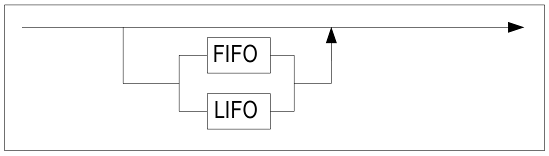
3.8 DEQUEUE

3.8.1 Syntax

3.8.1.1 dequeue ::=



3.8.1.2 fifo_option ::=



3.8.2 Description

The DEQUEUE statement retrieves a message that satisfies the condition in *where_clause* and then deletes it.

FIFO_option

If the FIFO option is enabled, or if neither FIFO nor LIFO has been specified, the oldest message that satisfies the condition is retrieved. If the LIFO option is enabled, the newest message is retrieved.

WAIT integer

When there are no messages in the queue, the DEQUEUE statement waits until a message has been added to the queue using ENQUEUE. The amount of time to wait is specified in the WAIT clause. If no wait time is specified, the DEQUEUE statement will wait indefinitely.

3.8.3 Considerations

The following should be taken into consideration when using the DEQUEUE statement:

Only the names of columns in a queue table can be specified in *queue_column_list*.

The DEQUEUE statement has some of the characteristics of the SELECT statement. However, the name of only one queue table can be specified in the FROM clause of a DEQUEUE statement. If more than one queue table is specified, an error will be raised.

A subquery cannot be used in the WHERE clause of a DEQUEUE statement.

3.8.4 Examples

<Query> Read all messages having the Correlation ID 237 from message queue *Q1*.

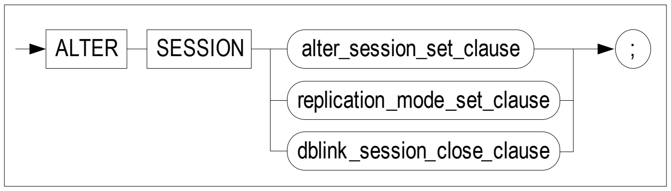
```
DEQUEUE MESSAGE, CORRID FROM Q1 WHERE CORRID=237;
```

4 Data Control Statements

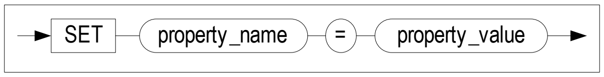
4.1 ALTER SESSION

4.1.1 Syntax

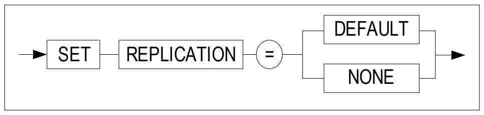
4.1.1.1 alter_session::=



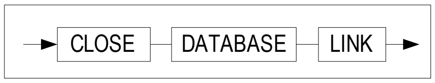
4.1.1.2 alter_session_set_clause::=



4.1.1.3 replication_mode_set_clause::=



4.1.1.4 dblink_session_close_clause::=



4.1.2 Description

This statement is used to change the attributes of the current session.

4.1.2.1 alter_session_set_clause

For more information on *property_name* and *property_value* in *alter_session_set_clause*, please refer to Chapter 2 of the *General Reference*, which explains all ALTIBASE HDB properties.

4.1.2.2 replication_mode_set_clause

This clause is used to set the replication mode for transactions that are executed in the current session.

If DEFAULT is specified, replication will be performed in the mode that was selected as the default mode when the replication object was created. However, if NONE is specified, none of the DDL, DML, or DCL statements executed in the session will be replicated.

For more information on the replication mode, please refer to the *Replication Manual*.

4.1.2.3 dblink_session_close_clause

When a user connects to a server, a session is created in the server. If Database Link is used in that session, a Database Link session, for use in performing Database Link-related tasks, is created in association with the session allocated to the user. When the user's session is subsequently terminated, the Database Link session is also terminated. However, if the user's session is not terminated after the Database Link operations have been performed, the Database Link session will also needlessly remain open.

In such cases, *dblink_session_close_clause* can be used to explicitly end the Database Link session.

4.1.3 Example

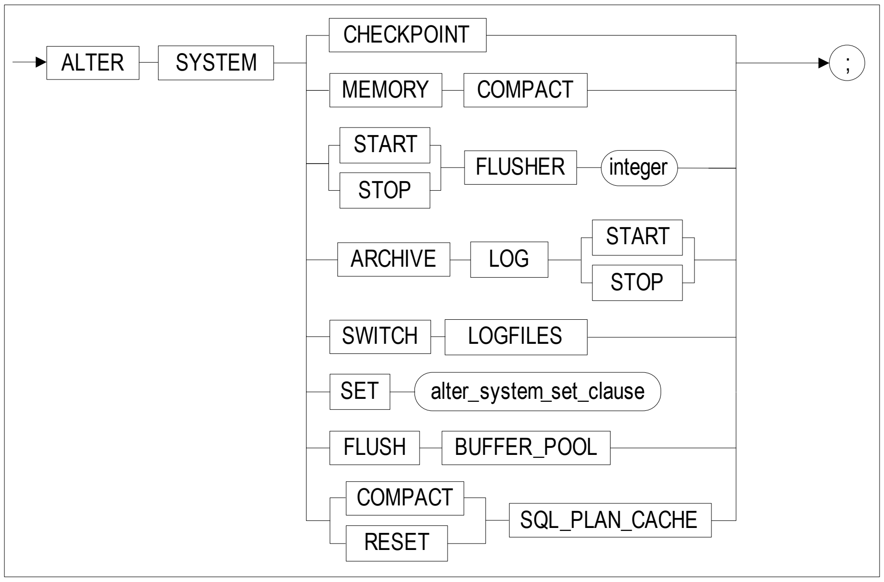
<Query>Close the database link session only, keeping the current session alive.

```
iSQL> ALTER SESSION CLOSE DATABASE LINK;
```

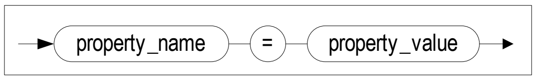
4.2 ALTER SYSTEM

4.2.1 Syntax

4.2.1.1 alter_system ::=



4.2.1.2 alter_system_set_clause ::=



4.2.2 Description

The ALTER SYSTEM statement is used to change ALTIBASE HDB system properties. Only the SYS user or a user to whom the ALTER SYSTEM privilege has been granted can access the complete functionality of the ALTER SYSTEM statement.

4.2.2.1 CHECKPOINT

This is used to execute checkpointing.

4.2.2.2 COMPACT

This is used to perform memory compaction. It is only useful on the IBM AIX platform.

4.2.2.3 START/STOP FLUSHER

This is used to start up or shut down the flusher.

4.2.2.4 ARCHIVE LOG START/STOP

Executing ALTER SYSTEM ARCHIVE LOG START starts the Archivelog thread, whereas ALTER SYSTEM ARCHIVE LOG STOP stops the thread. This statement can be executed only when the system is running in Archivelog mode.

To determine whether the system is running in Archivelog mode, query the V$LOG or V$ARCHIVE performance view. for more details about Archivelog mode, please refer to Chapter 10: Backup and Recovery in the *Administrator's Manual*.

4.2.2.5 SWITCH LOGFILES

This statement forcibly archives log files. Even if the current log file is not full, this command instructs the database to close it and resume logging in the next log file.

Only the sysdba can execute this statement.

4.2.2.6 SET alter_system_set_clause

This statement is used to change the values of database properties. For more information about these properties, please refer to the *General Reference*.

4.2.2.7 FLUSH BUFFER_POOL

This statement flushes all pages that are in the buffer to disk, thereby emptying the buffer.

Only the sysdba can execute this statement. It should only be executed after careful consideration. Because execution of this statement deletes all of the pages in the buffer, “buffer miss”, that is, failure to find records that are being sought in the buffer, can occur upon subsequent query execution.

4.2.3 Example

<Query> Stop flusher 1.

```
iSQL> ALTER SYSTEM STOP FLUSHER 1;
```

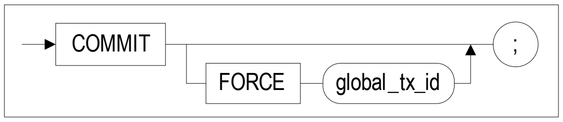
<Query> Start the Archivelog thread in archive mode.

```
iSQL> ALTER SYSTEM ARCHIVE LOG START;
```

4.3 COMMIT

4.3.1 Syntax

4.3.1.1 commit ::=



4.3.2 Description

The COMMIT statement explicitly commits the current transaction to the database.

This statement is useful when AUTOCOMMIT mode has been set to FALSE.

4.3.2.1 FORCE global_tx_id

In an XA environment, a transaction can be forcefully committed even when it is in an “in-doubt“ state.

global_tx_id is a character string that comprises the format identifier, the global transaction ID, and the branch qualifier of the global transaction.

4.3.3 Restriction

This statement cannot be executed in AUTOCOMMIT mode.

4.3.4 Examples

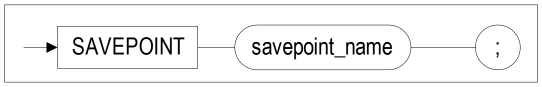
The following statement applies the result of all of the transaction's previously executed commands to the database.

```
iSQL> COMMIT;
Commit success.
```

4.4 SAVEPOINT

4.4.1 Syntax

4.4.1.1 savepoint::=



4.4.2 Description

This statement is used to create a savepoint, which means to temporarily save the result of transaction processing up to the current point in time. In other words, SAVEPOINT is used to explicitly define a point within a transaction to which the transaction can be rolled back. This statement is useful in non-autocommit mode, that is, when AUTOCOMMIT mode has been set to FALSE.

4.4.3 Example

```
iSQL> AUTOCOMMIT OFF;
Set autocommit off success.
iSQL> CREATE TABLE savept(num INTEGER);
Create success.
iSQL> INSERT INTO savept VALUES(1);
1 row inserted.
iSQL> SAVEPOINT sp1;
Savepoint success.
iSQL> INSERT INTO savept VALUES(2);
1 row inserted.
iSQL> SELECT * FROM savept;
SAVEPT.NUM
--------------
1
2
2 rows selected.
```

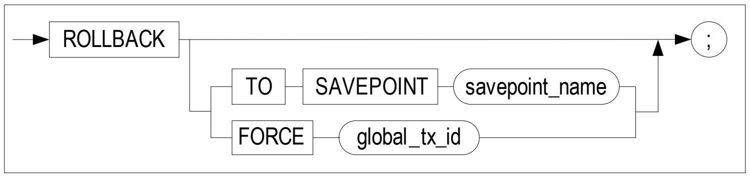
The transaction is rolled back to the time point at which the savepoint *sp1* was defined.

```
iSQL> ROLLBACK TO SAVEPOINT sp1;
Rollback success.
iSQL> SELECT * FROM savept;
SAVEPT.NUM
--------------
1
1 row selected.
iSQL> COMMIT;
Commit success.
```

4.5 ROLLBACK

4.5.1 Syntax

4.5.1.1 rollback ::=



4.5.2 Description

4.5.2.1 ROLLBACK (TO SAVEPOINT)

This statement is used to roll back the current transaction, either partially (to a previously defined savepoint) or completely.

4.5.2.2 FORCE global_tx_id

In an XA environment, this option is used to forcefully roll back a transaction that is in an “in-doubt“ state.

global_tx_id is a character string that comprises the format identifier, the global transaction ID, and the branch qualifier of the global transaction.

4.5.3 Precaution

This statement cannot be used in AUTOCOMMIT mode.

4.5.4 Example

```
iSQL> AUTOCOMMIT OFF;
Set autocommit off success.
iSQL> UPDATE employees SET salary = 2300 WHERE eno = 3;
1 row updated.
iSQL> SAVEPOINT emp3_sal;
Savepoint success.
iSQL> DELETE FROM employees WHERE eno = 19;
1 row deleted.
iSQL> SAVEPOINT emp19_ret;
Savepoint success.
iSQL> INSERT INTO employees(eno, e_lastname, e_firstname, salary, sex) VAL-
UES(21, 'Templeton', 'Kimmie', 3000, 'F');
1 row inserted.
iSQL> SAVEPOINT emp21_join;
```

```
Savepoint success.
iSQL> UPDATE employees SET salary = 2200 WHERE eno=18;
1 row updated.
iSQL> SELECT eno, e_lastname, e_firstname, salary FROM employees WHERE eno in
(3, 18, 19, 21);
ENO         E_LASTNAME            E_FIRSTNAME           SALARY
-------------------------------------------------------------------------
3           Kobain                Ken                   2300
18          Huxley                John                  2200
21          Templeton             Kimmie                3000
3 rows selected.
```

The transaction is rolled back to the time point at which the savepoint *emp21_join* was defined.

```
iSQL> ROLLBACK TO SAVEPOINT emp21_join;
Rollback success.
iSQL> SELECT eno, e_lastname, e_firstname, salary FROM employees WHERE eno in
(3, 18, 19, 21);
ENO         E_LASTNAME            E_FIRSTNAME           SALARY
-------------------------------------------------------------------------
3           Kobain                Ken                   2300
18          Huxley                John                  1900
21          Templeton             Kimmie                3000
3 rows selected.
```

The transaction is rolled back to the time point at which the savepoint *emp19_ret* was defined.

```
iSQL> ROLLBACK TO SAVEPOINT emp19_ret;
Rollback success.
iSQL> SELECT eno, e_lastname, e_firstname, salary FROM employees WHERE eno in
(3, 18, 19, 21);
ENO         E_LASTNAME            E_FIRSTNAME           SALARY
-------------------------------------------------------------------------
3           Kobain                Ken                   2300
18          Huxley                John                  1900
2 rows selected.
```

All of the changes made by the first UPDATE statement, the first DELETE statement and the last DML statement (the second INSERT statement) are committed. All of the other SQL DML statements were rolled back before the COMMIT statement was executed, and are thus lost. Additionally, the *emp21_join* savepoint is no longer available.

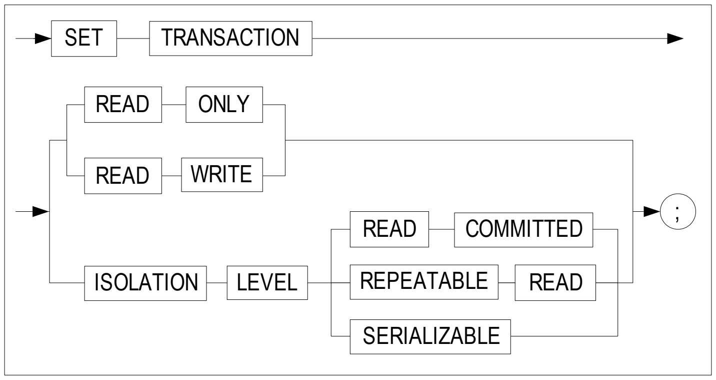
```
iSQL> ROLLBACK TO SAVEPOINT emp21_join;
[ERR-11016 : Savepoint not found]
iSQL> INSERT INTO employees(eno, e_lastname, e_firstname, sex, join_date)
VALUES(22, 'Chow', 'May', 'F', TO_DATE('2011-11-19 00:00:00', 'YYYY-MM-DD
HH:MI:SS'));
1 row inserted.
iSQL> COMMIT;
Commit success.
iSQL> SELECT eno, e_lastname, e_firstname, salary FROM employees;
ENO         E_LASTNAME            E_FIRSTNAME           SALARY
-------------------------------------------------------------------------
1           Moon                  Chan-seung
2           Davenport             Susan                 1500
4           Foster                Aaron                 1800
5           Ghorbani              Farhad                2500
6           Momoi                 Ryu                   1700
7           Fleischer             Gottlieb              500
8           Wang                  Xiong
```

```
9           Diaz                 Curtis               1200
10          Bae                  Elizabeth            4000
11          Liu                  Zhen                 2750
12          Hammond              Sandra               1890
13          Jones                Mitch                980
14          Miura                Yuu                  2003
15          Davenport            Jason                1000
16          Chen                 Wei-Wei              2300
17          Fubuki               Takahiro             1400
18          Huxley               John                 1900
20          Blake                William
3           Kobain               Ken                  2300
22          Chow                 May                  0
20 rows selected.
iSQL> COMMIT;
Commit success.
```

4.6 SET TRANSACTION

4.6.1 Syntax

4.6.1.1 set_transaction ::=



4.6.2 Description

The SET TRANSACTION statement is used to set the current transaction as read-only or read/write, or to set its isolation level.

Setting the isolation level to READ COMMITTED or SERIALIZABLE combines a row-level access method with a method of maintaining multiple versions of records, thereby realizing excellent data consistency, concurrency, and performance.

The changes made using the SET TRANSACTION statement affect only the current transaction, not other users or other transactions.

The isolation level can be set to any of the three levels described below.

4.6.2.1 READ COMMITTED

This isolation level allows data within a table that have been changed by a committed transaction to be read, while also allowing other transactions to read the previous version of data that will be changed by a transaction that has not been committed. READ COMMITED is the default ALTIBASE HDB transaction isolation level.

4.6.2.2 REPEATABLE READ

Because a read transaction maintains a shared lock on the data it retrieves until the transaction is complete, other transactions are prevented from changing these data. Locking records in this way guarantees that when a value is repeatedly retrieved, it will always be the same as the first time it was read. However, it is possible that other transactions will generate new records that satisfy the search conditions while such a lock is held. These records will be found on subsequent searches by the read transaction holding the lock, even though they were not found on the original search. This phenomenon is known as “Phantom Reads“.

4.6.2.3 SERIALIZABLE

This is the highest isolation level. This isolation level avoids locking all data when some records are retrieved using a SELECT statement with a ranged WHERE clause, but rather locks all data having key values that fall within the range of the data being read. This has the effect of preventing the “phantom reads” phenomenon and guaranteeing transaction isolation.

4.6.3 Considerations

This statement cannot be used when the current mode is AUTOCOMMIT mode.

This statement cannot be used if there are any active transactions.

4.6.4 Examples

```
iSQL> AUTOCOMMIT OFF;
Set autocommit off success.

iSQL> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Command execute success.

iSQL> SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
Command execute success.
```

Transaction A	Time Point	Transaction B

`iSQL> AUTOCOMMIT OFF;`
`Set autocommit off success.`
		`iSQL> AUTOCOMMIT OFF;`
		`Set autocommit off success.`

`iSQL> SET TRANSACTION READ ONLY;`
`Command execute success.`
	1	

`iSQL> SELECT e_lastname`
` FROM employees`
` WHERE eno = 20;`
`E_LASTNAME`
`------------------------`
`Blake`
`1 row selected.`
	2	

	3	`iSQL> UPDATE EMPLOYEES`
		` SET e_lastname = 'Blair'`
		` WHERE eno = 20;`
		`1 row updated.`

`iSQL> SELECT e_lastname`
` FROM employees`
` WHERE eno = 20;`
`ENAME`
`------------------------`
`Blake`
`1 row selected.`
	4	

Transaction A	Time Point	Transaction B

	5	`iSQL> commit;`
		`Commit success.`

`iSQL> SELECT e_lastname`
` FROM employees`
` WHERE eno = 20;`
`ENAME`
`------------------------`
`Blair`
`1 row selected.`
	6	

5 Set Operators

5.1 UNION

5.1.1 Syntax

```
SELECT statement1 UNION SELECT statement2
```

5.1.2 Description

This operator is used to output all of the results of two query statements.

Note that overlapping results, that is, results common to both queries, as well as duplicate results within each query, will be output only once.

5.1.3 Examples

<Query> Display the employee number of every employee who was born in 1980 or later and/or took an order for 100 or fewer items. Remove duplicate employee numbers.

```
iSQL> SELECT eno
 FROM employees
 WHERE birth > '800101'
 UNION
 SELECT eno
 FROM orders
 WHERE qty < 100;
ENO
--------------
4
7
8
12
13
15
20
7 rows selected.
```

5.2 UNION ALL

5.2.1 Syntax

```
SELECT statement1 UNION ALL SELECT statement2
```

5.2.2 Description

This operator is used to output all of the results of two query statements. Note that overlapping results, that is, results common to both queries, are output without removing any duplicates.

5.2.3 Examples

<Query> Display the employee number of all employees born in 1980 or later, as well as the employee number associated with all orders for 100 or fewer items. Do not remove any duplicate employee numbers.

```
iSQL> SELECT eno
 FROM employees
 WHERE birth > BYTE'800101'
 UNION ALL
 SELECT eno
 FROM orders
 WHERE qty < 100;
ENO
--------------
4
7
8
12
13
15
12
20
20
9 rows selected.
```

5.3 INTERSECT

5.3.1 Syntax

```
SELECT statement1 INTERSECT SELECT statement2
```

5.3.2 Description

The INTERSECT operator is used to output only records that are common to two queries, that is, records that are retrieved by both queries.

5.3.3 Examples

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
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 )
------------------------------------------------------------
```

5.4 MINUS

5.4.1 Syntax

```
SELECT statement1 MINUS SELECT statement2
```

5.4.2 Description

The MINUS operator is used to output the result set of one query after the result set of another query is removed therefrom.

5.4.3 Examples

<Query> Display the product number of all products that have never been ordered.

```
iSQL> SELECT gno FROM goods
MINUS
SELECT gno FROM orders;
GNO
--------------
A111100001
B111100001
C111100002
.
.
.
14 rows selected.
```

5.5 Order of Operations

The Order of Operations, also known as Operator Precedence, is the order in which the database evaluates the different operators in an expression. When an expression containing multiple operators is evaluated, operators having higher precedence are evaluated before those having lower precedence. Operators having equal precedence are evaluated in the order in which they appear in an expression, i.e. from left to right.

5.5.1 Description

The SQL operators are listed in the following table in decreasing order of operator precedence. Parentheses can be used within an expression to override operator precedence.

Precedence Level	Operator
1	All comparison operators
2	NOT
3	AND
4	OR

5.5.2 Examples

<Query> Display the name, position, and wage of engineers who earn more than 1850 dollars per month, as well as the name, position, and wage of all salespersons, regardless of their salary.

```
iSQL> SELECT e_firstname, e_lastname, emp_job, salary
 FROM employees
 WHERE emp_job = 'sales rep'
 OR emp_job = 'engineer'
 AND salary >= 1850;
E_FIRSTNAME           E_LASTNAME            EMP_JOB          SALARY
---------------------------------------------------------------------------
Ken                   Kobain                engineer         2000
Sandra                Hammond               sales rep        1890
Alvar                 Marquez               sales rep        1800
William               Blake                 sales rep
4 rows selected.
```

<Query> Display the name, position, and wage of all salespersons who earn more than 1850 dollars per month and all engineers who earn more than 1850 dollars per month.

```
iSQL> SELECT e_firstname, e_lastname, emp_job, salary
 FROM employees
 WHERE (emp_job = 'sales rep' OR emp_job = 'engineer')
 AND salary >= 1850;
E_FIRSTNAME           E_LASTNAME            EMP_JOB          SALARY
---------------------------------------------------------------------------
Ken                   Kobain                engineer         2000
Sandra                Hammond               sales rep        1890
2 rows selected.
```

6 SQL Functions

6.1 Introduction to SQL Functions

SQL functions are built into ALTIBASE HDB, and are available for use in various SQL statements. In addition to the built-in SQL functions, it is also possible for the user to define stored functions. For more information, please refer to the *Stored Procedures Manual*.

If an SQL function is called and an argument having a data type other than the data type expected by the SQL function is provided, then ALTIBASE HDB attempts to convert the argument to the expected data type before executing the SQL function. If an argument having a NULL value is provided when calling an SQL function, then the SQL function returns NULL.

This section lists the SQL functions that are supported in ALTIBASE HDB and explains how they are classified.

6.1.1 Classifying SQL Functions

SQL functions can be broadly classified into the categories set forth in the following table.

Category	Description and Functions
Aggregate Functions	These functions return a single value as a result of processing multiple rows. They can be used in select_list, ORDER BY, and HAVING clauses. These functions comprise AVG, COUNT, MAX, MIN, STDDEV, SUM and VARIANCE.
Analytic Functions	ALTIBASE HDB provides two kinds of analytic functions: analytic aggregate functions and analytic ranking functions. All of the aggregate functions listed above can also be used as analytic functions. They can take the form of either analytic aggregate functions or analytic ranking functions. ALTIBASE HDB additionally supports three other analytic functions: RANK, DENSE_RANK, and ROW_NUMBER. These can only take the form of analytic ranking functions.
Numeric Functions	These functions perform a task on an input numerical value and return a numerical value. ABS, ACOS, ASIN, ATAN, ATAN2, CEIL, COS, COSH, EXP, FLOOR, LN, LOG, MOD, POWER, RANDOM ROUND, SIGN, SIN, SINH, SQRT, TAN, TANH, TRUNC, BITAND, BITOR, BITXOR, BITNOT

Category	Description and Functions
Character Functions	These functions perform a task on an input string and return either a string or a numeric value. Function names in parentheses are aliases of preceding functions. **Functions that Return Strings** LOWER, LPAD, LTRIM, NCHR, REPLICATE, REPLACE2, REVERSE_STR, RPAD, RTRIM, STUFF, SUBSTRB (SUBSTRING), TRANSLATE, TRIM, UPPER **Functions that Return Numeric Values** CHAR_LENGTH (CHARACTER_LENGTH, LENGTH), DIGEST, INSTR (POSITION, INSTRB), OCTET_LENGTH (LENGTHB), SIZEOF
Date Functions	These functions perform a task on an input date/time value and return a character string, a numerical value, or a date/time value. Function names in parentheses are aliases of preceding functions. ADD_MONTHS, DATEADD, DATEDIFF, DATENAME, EXTRACT (DATE-PART), LAST_DAY, MONTHS_BETWEEN, NEXT_DAY, SYSDATE, SYS-TIMESTAMP
Conversion Functions	These functions convert an input character, numeric or date/time value and return a character, numeric or date/time value. The information in parentheses indicates the input type of the function. ASCIISTR, BIN_TO_NUM, CONVERT, HEX_TO_NUM, OCT_TO_NUM, TO_BIN, TO_CHAR(datetime), TO_CHAR(number), TO_DATE, TO_HEX, TO_NCHAR(character), TO_NCHAR(datetime),TO_NCHAR(number), TO_NUMBER, TO_OCT, UNISTR
Encryption Functions	These functions are used to perform DES encryption and decryption on strings. DESENCRYPT, DESDECRYPT
Other Functions	BINARY_LENGTH, CASE2, DECODE, DUMP, GREATEST, LEAST, NVL, NVL2

6.2 Aggregate Functions

Aggregate functions process multiple rows and return a single resultant value. Aggregate functions can be used in the SELECT list and in the ORDER BY and HAVING clauses.

If a SELECT statement contains a GROUP BY clause, then only constants, aggregate functions, the expressions in the GROUP BY clause, and expressions that are based on the foregoing can be specified in the SELECT list.

6.2.1 AVG

6.2.1.1 Syntax

```
AVG ([ALL | DISTINCT] expr)
```

6.2.1.2 Description

This function calculates the average of the input expressions. NULL values are excluded from the calculation. The function returns a FLOAT type value.

6.2.1.3 Example

<Query> Calculate and display the average price in the *goods* table.

```
iSQL> SELECT AVG(price) FROM goods;
AVG(PRICE)
--------------
30406.173
1 row selected.
```

6.2.2 COUNT

6.2.2.1 Syntax

```
COUNT ([ * | [ALL | DISTINCT] expr ])
```

6.2.2.2 Description

COUNT returns the number of rows returned by the query. NULL values are not counted.

6.2.2.3 Examples

<Query> Display the number of records in the *employees* table.

```
iSQL> SELECT COUNT(*) Rec_count FROM employees;
REC_COUNT
-----------------------
20
1 row selected.
```

<Query> Display the number of non-NULL birthdays in the *employees* table.

```
iSQL> SELECT COUNT(birth) Rec_count
 FROM employees;
REC_COUNT
-----------------------
13
1 row selected.
```

6.2.3 MAX

6.2.3.1 Syntax

```
MAX ([ALL | DISTINCT] expr)
```

6.2.3.2 Description

MAX returns the greatest value of *expr* from among the input values.

6.2.3.3 Example

<Query> Display the highest price in the *goods* table.

```
iSQL> SELECT MAX(price) FROM goods;
MAX(PRICE)
--------------
100000
1 row selected.
```

6.2.4 MIN

6.2.4.1 Syntax

```
MIN ([ALL | DISTINCT] expr)
```

6.2.4.2 Description

MIN returns the lowest value of *expr* from among the input values.

6.2.4.3 Example

<Query> Display the lowest price in the *goods* table.

```
iSQL> SELECT MIN(price) FROM goods;
MIN(PRICE)
--------------
966.99
1 row selected.
```

6.2.5 STDDEV

6.2.5.1 Syntax

```
STDDEV ([ALL | DISTINCT] expr)
```

6.2.5.2 Description

STDDEV returns the standard deviation of the input expressions.

Because error can accumulate when using system calls to perform repeated operations on real numbers, the return value may have some amount of inherent tolerance (error).

6.2.5.3 Example

<Query> Calculate the standard deviation of the wages in the *employees* table.

```
iSQL> SELECT STDDEV(salary) standard_deviation
 FROM employees;
STANDARD_DEVIATION
---------------------
797.706787
1 row selected.
```

6.2.6 SUM

6.2.6.1 Syntax

```
SUM ([ALL | DISTINCT] expr)
```

6.2.6.2 Description

SUM returns the result of addition of the input expressions.

6.2.6.3 Example

<Query> Calculate the total number of items stored using the *goods* table.

```
iSQL> SELECT SUM(stock) FROM goods;
SUM(STOCK)
-----------------------
379420
1 row selected.
```

6.2.7 VARIANCE

6.2.7.1 Syntax

```
VARIANCE ([ALL | DISTINCT] expression)
```

6.2.7.2 Description

VARIANCE returns the variance of the input expressions.

Because error can accumulate when using system calls to perform repeated operations on real numbers, the return value may have some amount of inherent tolerance (error).

6.2.7.3 Example

<Query> Calculate the variance of the wages in the *employees* table.

```
iSQL> SELECT VARIANCE(salary) variance
 FROM employees;
VARIANCE
--------------
636336.117649059
```

6.3 Analytic Functions

Analytic functions differ from aggregate functions in that they have an OVER expression (which may be nothing more than empty parentheses). The OVER expression may or may not contain a PARTITION BY expression.

Analytic functions are classified as either analytic aggregate functions or analytic ranking functions, depending on whether the OVER expression contains an ORDER BY expression.

All aggregate functions (AVG, COUNT, MAX, MIN, STDDEV, SUM and VARIANCE) can be used as analytic functions. They can take the form of either analytic aggregate functions or analytic ranking functions.

ALTIBASE HDB additionally supports three other analytic functions: RANK, DENSE_RANK, and ROW_NUMBER. These can only take the form of analytic ranking functions.

Analytic functions can only appear in a SELECT list or in an ORDER BY clause.

The steps taken when processing queries containing analytic functions are as follows.

- Step 1: The query is processed, with the exception of any analytic functions and the ORDER BY clause (if present).
- Step 2: If the analytic functions contain partitioning expressions, the results are divided and classified into the partitions on the basis of which the analytic functions are to be executed.
- Step 3: If an ORDER BY expression is present, the results are ordered for each of the partitions.
- Step 4: The analytic functions are executed.
- Step 5: If an ORDER BY clause is present, it is processed.

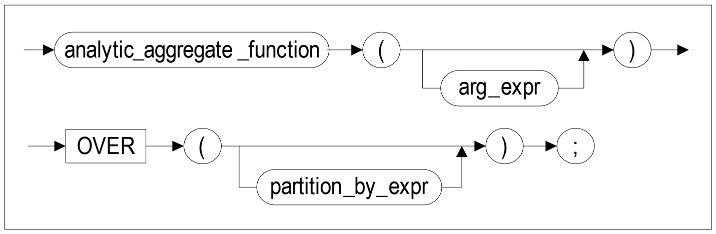
6.3.1 Analytic Aggregate Functions

When the OVER expression in an analytic function does not contain an ORDER BY expression, the aggregate function is considered an analytic aggregate function.

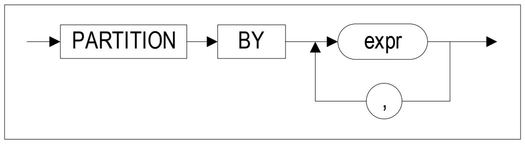
When an aggregate function (such as AVG, MAX, MIN, etc.) is used in the form of an analytic aggregate function, the resultant value is returned once for every row. This is different from the typical use of an aggregate function, in which case the resultant value is returned only once.

6.3.1.1 Syntax

analytic_aggregate_function::=



partition_by_expr::=



6.3.1.2 Description

- *analytic_aggregate_function*

  This is used to indicate the name of the aggregate function that is being used as an analytic aggregate function. As noted above, in ALTIBASE HDB, all aggregate functions are supported for use as analytic aggregate functions.

- *arg_expr*

  This used to specify the expression to use as a argument for the analytic function.

- OVER

  This indicates to the Query Processor that the function is an analytic aggregate function.

- *partition_by_expr*

  The PARTITION BY expression is used to partition the results of a query, that is, return them in groups.

6.3.1.3 Example

<Query> Output each employee's salary as a percentage of the maximum salary in the employee's department.

```
iSQL(sysdba)> SELECT e_lastname, dno, salary,
 ROUND(salary/MAX(salary) OVER (PARTITION BY dno)*100) rel_sal
 FROM employees;
E_LASTNAME            DNO         SALARY      REL_SAL
---------------------------------------------------------------
Kobain                1001        2000        87
Chen                  1001        2300        100
Jones                 1002        980         58
Momoi                 1002        1700        100
Davenport             1003        1000        25
Liu                   1003        2750        69
Miura                 1003        2003        50
Bae                   1003        4000        100
Fubuki                2001        1400        100
Foster                3001        1800        100
Ghorbani              3002        2500        100
Moon                  3002
Huxley                4001        1900        100
Wang                  4001
Diaz                  4001        1200        63
Fleischer             4002        500         26
Marquez               4002        1800        95
Blake                 4002
Hammond               4002        1890        100
```

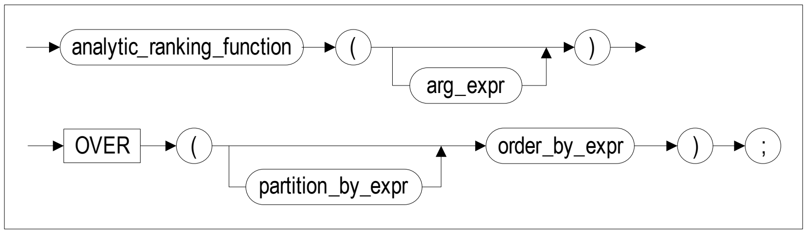
```
Davenport                          1500        100
20 rows selected.
```

6.3.2 Analytic Ranking Functions

When the OVER expression in an analytic function contains an ORDER BY expression, the aggregate function is considered an analytic ranking function.

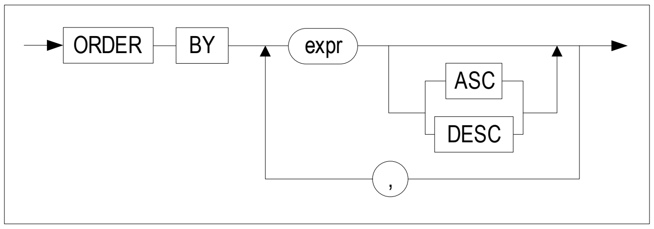
6.3.2.1 Syntax

analytic_ranking_function::=



partition_by_expr::=

order_by_expr::=



6.3.2.2 Description

- *analytic_ranking_function*

  This is used to indicate the name of the analytic ranking function. In ALTIBASE HDB, all aggregate functions are supported for use as analytic ranking functions.

  Additionally, ALTIBASE HDB provides the following analytic ranking functions. These are described in greater detail below.

  — RANK
  — DENSE_RANK
  — ROW_NUMBER

- *arg_expr*

  This is the same as for analytic aggregate functions.

- OVER

This is the same as for analytic aggregate functions.

- *partition_by_expr*

  This is the same as for analytic aggregate functions.

- *order_by_expr*

  The analytic ranking function will be processed in the order specified by each *expr* in *order_by_expr* for each of the partitions defined based on *partition_by_expr*.

6.3.2.3 RANK

The RANK function is used to assign rank values to the members of the result set (or partition). RANK leaves a gap in the sequence after duplicate values ("tie" values).

6.3.2.4 DENSE_RANK

Like the RANK function, the DENSE_RANK function is used to assign rank values to the members of a result set or partition. However, DENSE_RANK does not leave a gap in the sequence after the occurrence of duplicate values.

6.3.2.5 ROW_NUMBER

The ROW_NUMBER function is used to sequentially assign unique numbers to the members of a result set or partition. The numbers are assigned starting from 1 in the order determined by the ORDER BY expression.

If duplicate values exist in the result set, the order in which the row numbers are assigned to the duplicate values cannot be guaranteed. If you need to guarantee the order in which the row numbers are assigned, use an ORDER BY clause that references some other column.

6.3.2.6 Examples

<Query> Rank the employees in each department by salary in decreasing order.

```
iSQL(sysdba)> SELECT e_lastname, dno, salary, RANK() OVER (PARTITION BY dno
ORDER BY salary DESC) FROM employees;
E_LASTNAME             DNO         SALARY      RANK
------------------------------------------------------------------------
Chen                   1001        2300        1
Kobain                 1001        2000        2
Momoi                  1002        1700        1
Jones                  1002        980         2
Bae                    1003        4000        1
Liu                    1003        2750        2
Miura                  1003        2003        3
Davenport              1003        1000        4
Fubuki                 2001        1400        1
Foster                 3001        1800        1
Ghorbani               3002        2500        1
Moon                   3002                    2
Huxley                 4001        1900        1
Diaz                   4001        1200        2
Wang                   4001                    3
Hammond                4002        1890        1
```

```
Marquez              4002        1800        2
Fleischer            4002        500         3
Blake                4002                    4
Davenport                        1500        1
20 rows selected.
```

<Query> This example illustrates the differences in the output of the RANK, DENSE_RANK, and ROW_NUMBER functions.

```
iSQL(sysdba)> SELECT salary,
 RANK() OVER (ORDER BY salary DESC),
 DENSE_RANK() OVER (ORDER BY salary DESC),
 ROW_NUMBER() OVER (ORDER BY salary DESC)
 FROM employees;
SALARY      RANK                DENSE_RANK          ROW_NUMBER
----------------------------------------------------------------------------
4000        1                   1                   1
2750        2                   2                   2
2500        3                   3                   3
2300        4                   4                   4
2003        5                   5                   5
2000        6                   6                   6
1900        7                   7                   7
1890        8                   8                   8
1800        9                   9                   9
1800        9                   9                   10
1700        11                  10                  11
1500        12                  11                  12
1400        13                  12                  13
1200        14                  13                  14
1000        15                  14                  15
980         16                  15                  16
500         17                  16                  17
            18                  17                  18
            18                  17                  19
            18                  17                  20
20 rows selected.
```

6.4 Numeric Functions

Numeric functions accept numeric input values as parameters, perform operations on them, and return numeric values.

6.4.1 ABS

6.4.1.1 Syntax

```
ABS (number)
```

6.4.1.2 Description

ABS returns the absolute value of the input number.

6.4.1.3 Examples

<Query> Display the absolute values for three numeric values.

```
iSQL> SELECT ABS(-1), ABS(0.0), ABS(1) FROM dual;
ABS(-1) ABS(0.0) ABS(1)
----------------------------------------
1       0        1
1 row selected.
```

<Query> Calculate the difference between the most expensive item and the most inexpensive item in the *goods* table.

```
iSQL> SELECT ABS(MIN(price) - MAX(price)) absolute_value FROM goods;
ABSOLUTE_VALUE
-----------------
99033.01
1 row selected.
```

6.4.2 ACOS

6.4.2.1 Syntax

```
ACOS (n)
```

6.4.2.2 Description

ACOS returns the arccosine of the input argument, which must be within the range from -1 to 1 inclusive. If the input value does not fall within this range, 0.000000 is returned. The function returns a DOUBLE type value in radians within the range from 0 to π(pi).

```
1 radian = 180°/pi
```

6.4.2.3 Example

<Query>

```
iSQL> SELECT ACOS(.3) Arc_Cosine FROM dual;
ARC_COSINE
--------------
1.266104
1 row selected.
```

6.4.3 ASIN

6.4.3.1 Syntax

```
ASIN (n)
```

6.4.3.2 Description

ASIN returns the arcsine of the input argument, which must be within the range from -1 to 1 inclusive. If the input value does not fall within this range, 0.000000 is returned. The function returns a DOUBLE type numeric value in radians within the range from -π/2 to π/2 inclusive.

6.4.3.3 Example

<Query>

```
iSQL> SELECT ASIN(.3) Arc_Sine FROM dual;
ARC_SINE
--------------
0.304693
1 row selected.
```

6.4.4 ATAN

6.4.4.1 Syntax

```
ATAN (n)
```

6.4.4.2 Description

ATAN returns the arc tangent of the input argument, which can be any real number. The function returns a DOUBLE type numeric value in radians within the range from -π/2 to π/2 inclusive.

6.4.4.3 Example

<Query>

```
iSQL> SELECT ATAN(.3) Arc_Tangent FROM dual;
ARC_TANGENT
--------------
```

```
0.291457
1 row selected.
```

6.4.5 ATAN2

6.4.5.1 Syntax

```
ATAN2 (n, m)
```

6.4.5.2 Description

ATAN2 returns the arc tangent of two input arguments, which can be any real number. The function returns a DOUBLE type numeric value in radians within the range from -π (exclusive) to π (inclusive).

6.4.5.3 Example

<Query>

```
iSQL> SELECT ATAN2(.3, .2) Arc_Tangent2 FROM dual;
ARC_TANGENT2
---------------
0.982794
1 row selected.
```

6.4.6 CEIL

6.4.6.1 Syntax

```
CEIL (n)
```

6.4.6.2 Description

CEIL returns the smallest integer that is greater than or equal to *n*.

6.4.6.3 Examples

<Query> Return the smallest integer that is greater than or equal to each of the input numbers.

```
iSQL> SELECT CEIL(99.9), CEIL(-99.9) FROM dual;
CEIL(99.9) CEIL(-99.9)
---------------------------
100        -99
1 row selected.
```

<Query> Calculate the difference between the most expensive item and the most inexpensive item in the *goods* table, and return the smallest integer that is greater than or equal to this difference.

```
iSQL> SELECT CEIL(ABS (MIN(price) - MAX(price))) Smallest_int FROM goods;
```

```
SMALLEST_INT
---------------
99034
1 row selected.
```

6.4.7 COS

6.4.7.1 Syntax

```
COS (n)
```

6.4.7.2 Description

COS is a trigonometric function that returns the cosine, expressed in radians, of an input floating-point number, also expressed in radians. The return type is DOUBLE.

6.4.7.3 Example

<Query>

```
iSQL> SELECT COS(180 * 3.14159265359/180) Cos_of_180_degrees FROM dual;
COS_OF_180_DEGREES
---------------------
-1
1 row selected.
```

6.4.8 COSH

6.4.8.1 Syntax

```
COSH (n)
```

6.4.8.2 Description

COSH returns the hyperbolic cosine of the input number. The return type is DOUBLE.

COSH(n) = (e^{n} + e^{-n})/2

6.4.8.3 Example

<Query>

```
iSQL> SELECT COSH(0) FROM dual;
COSH(0)
--------------
1
1 row selected.
```

6.4.9 EXP

6.4.9.1 Syntax

```
EXP (n)
```

6.4.9.2 Description

EXP returns the mathematical constant *e* to the power of the input number. (*e* = 2.71828183…) The return type is DOUBLE.

6.4.9.3 Example

<Query>

```
iSQL> SELECT EXP(2.4) FROM dual;
EXP(2.4)
--------------
11.023176
1 row selected.
```

6.4.10 FLOOR

6.4.10.1 Syntax

```
FLOOR (n)
```

6.4.10.2 Description

FLOOR returns the greatest integer that is less than or equal to *n*.

6.4.10.3 Examples

<Query> Return the greatest integer that is less than or equal to each of the input numbers.

```
iSQL> SELECT FLOOR(99.9), FLOOR(-99.9) FROM dual;
FLOOR(99.9) FLOOR(-99.9)
----------------------------
99          -100
1 row selected.
```

<Query> Calculate the difference between the most expensive item and the most inexpensive item in the *goods* table, and return the greatest integer that is less than or equal to this difference.

```
iSQL> SELECT FLOOR(ABS(MIN(price) - MAX(price))) Largest_int FROM goods;
LARGEST_INT
--------------
99033
1 row selected.
```

6.4.11 LN

6.4.11.1 Syntax

```
LN (n)
```

6.4.11.2 Description

LN returns the natural logarithm of *n*, which must be greater than 0.

6.4.11.3 Example

<Query>

```
iSQL> SELECT LN(2.4) FROM dual;
LN(2.4)
--------------
0.875469
1 row selected.
```

6.4.12 LOG

6.4.12.1 Syntax

```
LOG (m, n)
```

6.4.12.2 Description

LOG returns the logarithm of *n* to base *m*. The base *m* can be any positive value other than 0 or 1, and *n* can be any positive value.

6.4.12.3 Example

<Query>

```
iSQL> SELECT LOG(10, 100) FROM dual;
LOG(10, 100)
---------------
2
1 row selected.
```

6.4.13 MOD

6.4.13.1 Syntax

```
MOD (m, n)
```

6.4.13.2 Description

MOD returns the remainder of division of *m* by *n*. If *n* is 0, a division by zero error is raised.

6.4.13.3 Examples

<Query> Find the remainder of division of 10 by 3.

```
iSQL> SELECT MOD(10, 3) FROM dual;
MOD(10, 3)
--------------
1
1 row selected.
```

<Query> Add up all salaries, divide the total by the lowest salary, and output the remainder.

```
iSQL> SELECT MOD(SUM(salary), MIN(salary)) Remainder FROM employees;
REMAINDER
--------------
223
1 row selected.
```

6.4.14 POWER

6.4.14.1 Syntax

```
POWER (m, n)
```

6.4.14.2 Description

POWER returns *m*, the base, raised to the power of *n*, the exponent. The base *m* and the exponent *n* can be any real numbers, but if the base is negative, then the exponent must be an integer.

6.4.14.3 Example

<Query>

```
iSQL> SELECT POWER(3, 2) FROM dual;
POWER(3, 2)
--------------
9
1 row selected.
```

6.4.15 RANDOM

6.4.15.1 Syntax

```
RANDOM (n)
```

6.4.15.2 Description

RANDOM returns a pseudorandom integer value. The range of possible return values is from 0 to the maximum value of the INTEGER type, that is, 2,147,483,647.

If *n* is set to a nonzero number, that number will be used as the random seed value, on the basis of which a pseudorandom integer will be returned. If the RANDOM function is repeatedly called with the same nonzero seed value, the same value will be repeatedly returned. Set *n* to zero to prevent the same value from being repeatedly returned.

6.4.15.3 Examples

<Query> Retrieve a random number without specifying a seed value.

```
iSQL> SELECT RANDOM(0) FROM dual;
RANDOM(0)
--------------
16838
1 row selected.
```

<Query> Retrieve a random number using a seed value.

```
iSQL> SELECT RANDOM(100) FROM dual;
RANDOM(100)
--------------
12662
1 row selected.
```

6.4.16 ROUND

6.4.16.1 Syntax

```
ROUND (n1 [, n2])
```

6.4.16.2 Description

ROUND returns *n1* rounded to *n2* places to the right of the decimal point. If *n2* is omitted, then *n1* is rounded to 0 (zero) decimal places. *n2* can also be set to a negative number, in which case *n1* will be rounded off *n2* places to the left of the decimal point.

6.4.16.3 Examples

<Query> Return the results of the ROUND function on each of the two following expressions.

```
iSQL> SELECT ROUND(123.9994, 3), ROUND(123.9995, 3) FROM dual;
ROUND(123.9994, 3) ROUND(123.9995, 3)
-----------------------------------------
123.999            124
1 row selected.
```

<Query> Round the price of the most inexpensive product up to the nearest integer and output it.

```
iSQL> SELECT ROUND( MIN(price) ) FROM goods;
ROUND( MIN(PRICE) )
```

```
----------------------
967
1 row selected.
```

Example	Result
ROUND(748.58, -1)	750
ROUND(748.58, -2)	700
ROUND(748.58, -3)	1000

ROUND always returns a value. If *integer* is a negative number that is greater than the number of digits to the left of the decimal point, ROUND will return 0.

Example	Result
ROUND(748.58, -4)	0

6.4.17 SIGN

6.4.17.1 Syntax

```
SIGN (number)
```

6.4.17.2 Description

SIGN returns the sign of *n*, which can be any numeric data type. If *n* is positive, SIGN returns 1, whereas if *n* is negative, SIGN returns -1. if *n* is 0, SIGN returns 0.

6.4.17.3 Examples

<Query> Return the results of the SIGN function on each of the following expressions.

```
iSQL> SELECT SIGN(15), SIGN(0), SIGN(-15) FROM dual;
SIGN(15) SIGN(0) SIGN(-15)
----------------------------------------
1        0       -1
1 row selected.
```

<Query> Output 1 if the wage is higher than 1000 dollars, -1 if it is lower than 1000 dollars, and 0 if it is exactly 1000 dollars.

```
iSQL> SELECT e_firstname, e_lastname, SIGN(salary-1000) As Wage_class
 FROM employees;
E_FIRSTNAME           E_LASTNAME            WAGE_CLASS
------------------------------------------------------------
Chan-seung            Moon
Susan                 Davenport             1
Ken                   Kobain                1
Aaron                 Foster                1
```

```
Farhad                  Ghorbani                1
Ryu                     Momoi                   1
Gottlieb                Fleischer               -1
Xiong                   Wang
Curtis                  Diaz                    1
Elizabeth               Bae                     1
Zhen                    Liu                     1
Sandra                  Hammond                 1
Mitch                   Jones                   -1
Yuu                     Miura                   1
Jason                   Davenport               0
Wei-Wei                 Chen                    1
Takahiro                Fubuki                  1
John                    Huxley                  1
Alvar                   Marquez                 1
William                 Blake
20 rows selected.
```

6.4.18 SIN

6.4.18.1 Syntax

```
SIN (n)
```

6.4.18.2 Description

SIN is a trigonometric function that returns the trigonometric sine of *n*, expressed in radians. The return type is DOUBLE.

6.4.18.3 Example

<Query> Retrieve the sine of a 30-degree angle.

```
iSQL> SELECT SIN (30 * 3.14159265359/180) Sine_of_30_degrees FROM dual;
SINE_OF_30_DEGREES
----------------------
0.5
1 row selected.
```

6.4.19 SINH

6.4.19.1 Syntax

```
SINH (n)
```

6.4.19.2 Description

SINH returns the hyperbolic sine of *n*.

SINH(n) = $(e^{n} - e^{-n})/2$

6.4.19.3 Example

<Query> Return the hyperbolic sine of 1:

```
iSQL> SELECT SINH(1) Hyperbolic_sine_of_1 FROM dual;
HYPERBOLIC_SINE_OF_1
-----------------------
1.175201
1 row selected.
```

6.4.20 SQRT

6.4.20.1 Syntax

```
SQRT (n)
```

6.4.20.2 Description

SQRT returns the square root of *n*. *n* must not be a negative value.

6.4.20.3 Example

<Query> Return the square root of 10.

```
iSQL> SELECT SQRT(10) FROM dual;
SQRT(10)
--------------
3.162278
1 row selected.
```

6.4.21 TAN

6.4.21.1 Syntax

```
TAN (n)
```

6.4.21.2 Description

TAN is a trigonometric function that returns the tangent of n, an angle expressed in radians. The return type is DOUBLE.

6.4.21.3 Example

<Query> Retrieve the tangent of a 135-degree angle:

```
iSQL> SELECT TAN (135 * 3.14159265359/180) Tangent_of_135_degrees FROM dual;
TANGENT_OF_135_DEGREES
-------------------------
-1
1 row selected.
```

6.4.22 TANH

6.4.22.1 Syntax

```
TANH (n)
```

6.4.22.2 Description

TANH returns the hyperbolic tangent of n.

6.4.22.3 Example

<Query>Return the hyperbolic tangent of 0.5:

```
iSQL> SELECT TANH(.5) Hyperbolic_tangent_of_ FROM dual;
HYPERBOLIC_TANGENT_OF_
-------------------------
0.462117
1 row selected.
```

6.4.23 TRUNC

6.4.23.1 Syntax

```
TRUNC (n1 [, n2])
```

6.4.23.2 Description

TRUNC returns *n1* truncated to *n2* decimal places. The return type is FLOAT.

If *n2* is omitted, then *n1* is truncated to an integer. *n2* can be set to a negative number, in which case *n1* will be truncated *n2* places to the left of the decimal point.

6.4.23.3 Examples

<Query> Return the results of the TRUNC function on each of the following expressions.

```
iSQL> SELECT TRUNC(15.79, 1), TRUNC(15.79, -1) FROM dual;
TRUNC(15.79, 1) TRUNC(15.79, -1)
------------------------------------
15.7            10
1 row selected.
```

<Query> Retrieve the integer portion of the price of the most inexpensive product.

```
iSQL> SELECT TRUNC(MIN(price)) FROM goods;
TRUNC(MIN(PRICE))
--------------------
966
1 row selected.
```

6.4.24 BITAND

6.4.24.1 Syntax

```
BITAND (bit_a, bit_b)
```

6.4.24.2 Description

BITAND returns the result of a bitwise AND operation on *bit_a* and *bit_b*.

6.4.24.3 Example

```
iSQL> SELECT TO_CHAR( BITAND( BIT'01010101', BIT'10101010' ) ) FROM DUAL;
TO_CHAR( BITAND( BIT'01010101', BIT'1010
--------------------------------------------
00000000
1 row selected.
```

6.4.25 BITOR

6.4.25.1 Syntax

```
BITOR (bit_a, bit_b)
```

6.4.25.2 Description

BITOR returns the result of a bitwise OR operation on *bit_a* and *bit_b*.

6.4.25.3 Example

```
iSQL> SELECT TO_CHAR( BITOR( BIT'01010101', BIT'10101010' ) ) FROM DUAL;
TO_CHAR( BITOR( BIT'01010101', BIT'10101
--------------------------------------------
11111111
```

6.4.26 BITXOR

6.4.26.1 Syntax

```
BITXOR (bit_a, bit_b)
```

6.4.26.2 Description

BITXOR returns the result of a bitwise XOR (exclusive OR) operation on *bit_a* and *bit_b*.

6.4.26.3 Example

```
iSQL> SELECT TO_CHAR( BITXOR( BIT'01010101', BIT'10101010' ) ) FROM DUAL;
TO_CHAR( BITXOR( BIT'01010101', BIT'1010
--------------------------------------------
11111111
1 row selected.
```

6.4.27 BITNOT

6.4.27.1 Syntax

```
BITNOT (bit_a)
```

6.4.27.2 Description

BITNOT returns the result of a bitwise NOT operation on *bit_a*.

6.4.27.3 Example

```
iSQL> SELECT TO_CHAR( BITNOT( BIT'01010101' ) ) FROM DUAL;
TO_CHAR( BITNOT( BIT'01010101' ) )
--------------------------------------
10101010
1 row selected.
```

6.5 Character Functions

Character functions return either character or numeric values. They can be classified into two types depending on which kind of data they return.

Character functions that return character data

CHR, CONCAT, DIGITS, INITCAP, LOWER, LPAD, LTRIM, NCHR, REPLICATE, REPLACE2, REVERSE_STR, RPAD, RTRIM, STUFF, SUBSTRB (SUBSTRING), TRANSLATE, TRIM, UPPER

Character functions that return numeric data

ASCII, INSTR (POSITION), CHAR_LENGTH (CHARACTER_LENGTH, LENGTH), INSTRB, OCTET_LENGTH (LENGTHB), SIZEOF

6.5.1 ASCII

6.5.1.1 Syntax

```
ASCII (expr)
```

6.5.1.2 Description

ASCII returns the ASCII code value for the first (i.e. leftmost) character in *expr*.

6.5.1.3 Example

<Query> Output the ASCII code for the letter 'A'.

```
iSQL> SELECT ASCII('A') FROM dual;
ASCII('A')
--------------
65
```

6.5.2 CHAR_LENGTH, CHARACTER_LENGTH, LENGTH

6.5.2.1 Syntax

```
CHAR_LENGTH (expr)
CHARACTER_LENGTH (expr)
LENGTH (expr)
```

6.5.2.2 Description

CHAR_LENGTH returns the length of the input character string.

CHARACTER_LENGTH and LENGTH have the same function as CHAR_LENGTH.

6.5.2.3 Examples

<Query> Output the length of the street addresses in the *managers* table.

Note that the character set is KO16KSC5601.

```
CREATE TABLE managers(
mgr_no INTEGER PRIMARY KEY,
m_lastname VARCHAR(20),
m_firstname VARCHAR(20),
address VARCHAR(60));

INSERT INTO managers VALUES(1, 'Jones', 'Davey', '3101 N. Wabash Ave. Brook-
lyn, NY');
INSERT INTO managers VALUES(15, 'Min', 'Sujin', '서울 마포구 아현 1');

iSQL> SELECT CHAR_LENGTH(address) FROM managers;
CHAR_LENGTH (ADDRESS)
------------------------
32
11
2 rows selected.
```

6.5.3 CHR

6.5.3.1 Syntax

```
CHR (n)
```

6.5.3.2 Description

This function converts an input integer ASCII code value into the corresponding character. Multiple characters can be concatenated using double vertical bars.

6.5.3.3 Examples

<Query> Output the word “ALTIBASE” using ASCII code values.

```
iSQL> SELECT CHR(65) || CHR(76) || CHR(84) || CHR(73) || CHR(66) || CHR(65)
|| CHR(83) || CHR(69) mmdbms
FROM dual;
MMDBMS
------------------------------------
ALTIBASE
1 row selected.
```

<Query>Use the line feed character, which has an ASCII value of 10, to format SELECT query results to make them suitable for printing.

```
iSQL> SELECT RTRIM(c_firstname) || ' ' || c_lastname || CHR(10) || sex || ' '
|| cus_job || CHR(10) || address cus_info
 FROM customers
 WHERE cno = 10;
CUS_INFO
----------------------------------------------------------------------------
-----------------------------------------------------
```

```
Anh Dung Nguyen
M
8A Ton Duc Thang Street District 1 HCMC Vietnam
1 row selected.
```

Note:

Control Character	ASCII Value
Tab	9
New line	10
Carriage Return	13

6.5.4 CONCAT

6.5.4.1 Syntax

```
CONCAT (expr1, expr2)
```

6.5.4.2 Description

CONCAT returns *expr1* concatenated with *expr2*. This function is the same as using the double-vertical-bars concatenation operator ("||").

6.5.4.3 Example

<Query> Concatenate the results of a SELECT query into an English sentence.

```
iSQL> SELECT CONCAT(CONCAT(CONCAT(CONCAT(CONCAT(RTRIM(e_firstname), ' '),
RTRIM(e_lastname)), ' is a ' ), emp_job ), '.') Job
 FROM employees
 WHERE eno = 10;
JOB
-------------------------------------------------------------------
Elizabeth Bae is a programmer.
1 row selected.
```

6.5.5 DIGITS

6.5.5.1 Syntax

```
DIGITS (n)
```

6.5.5.2 Description

Returns an integer as a character string.

The length of the character string depends on the data type (or size) of *n*. A string comprising 5 digits is returned for a SMALLINT, 10 digits for an INTEGER, and 19 digits for a BIGINT. If the number of digits in the input value is less than the number of digits that can be stored in the numeric data type, the leading spaces are populated with 0's (zeroes).

6.5.5.3 Example

<Query> Output a character string for each of three input numbers having different numeric data types. The length of the string is set differently for each data type.

```
CREATE TABLE T1 (I1 SMALLINT, I2 INTEGER, I3 BIGINT);
INSERT INTO T1 VALUES (357, 12, 5000);

iSQL> SELECT DIGITS(I1), DIGITS(I2), DIGITS(I3) FROM T1;
DIGITS(I1)    DIGITS(I2)    DIGITS(I3)
------------------------------------------------
00357         0000000012    0000000000000005000
1 row selected.
```

6.5.6 INITCAP

6.5.6.1 Syntax

```
INITCAP (expr)
```

6.5.6.2 Description

INITCAP converts the first character in each word in the input character expression to uppercase and returns the result. Words are delimited by spaces or by characters that are not letters or numbers.

6.5.6.3 Example

<Query> Output the character string “the soap” after converting the first letter in each word to uppercase.

```
iSQL> SELECT INITCAP ('the soap') Capital FROM dual;
CAPITAL
------------
The Soap
1 row selected.
```

6.5.7 INSTR, INSTRB, POSITION

6.5.7.1 Syntax

```
INSTR (expr, substring [, start [, occurrence]])

INSTRB (expr, substring [, start [, occurrence]])

POSITION (expr, substring [, start [, occurrence]])
```

6.5.7.2 Description

The INSTR function looks for *substring* in *expr* and returns the location of the first character in *substring* if *substring* is found. It returns 0 (zero) if *substring* is not found. The INSTRB function returns the position of the specified string in bytes rather than in characters.

Start specifies the position in *expr* at which the search begins. The default start value is 1, i.e. the first character. If a negative start value is specified, the search begins the specified number of characters from the end of *expr*. If a start value of 0 (zero) is specified, the function simply returns 0 (zero). If the value of *start* is greater than the length of *expr*, an error will occur.

occurrence indicates the instance of *substring* in *expr* for which to return the position. The default value is 1. If *occurrence* is set to 1, the position of the first instance of *substring* that is found will be returned; if it is set to 2, the position of the second instance of *substring* will be returned; etc. If it is set to 0 (zero) or to a value that is greater than the number of times that *substring* is found in *expr*, 0 (zero) will be returned. If it is set to a negative value, an error will occur.

POSITION has the same function as INSTR.

This function is case-sensitive.

6.5.7.3 Examples

<Query> Return the position of the second occurrence of "OR" in the string "CORPORATE FLOOR", beginning the search at the third character.

```
iSQL> SELECT INSTR ('CORPORATE FLOOR','OR', 3, 2) Instring FROM dual;
INSTRING
--------------
14
1 row selected.
```

<Query> Return the position of the second occurrence of " 베이 " in the string " 알티베이스5 데이터베이스 ", beginning the search at the third character. (Note that the character set is KO16KSC5601.)

```
iSQL> SELECT INSTR ('알티베이스5 데이터베이스','베이', 3, 2) Instring FROM dual;
INSTRING
--------------
11
1 row selected.
```

6.5.8 LOWER

6.5.8.1 Syntax

```
LOWER (expr)
```

6.5.8.2 Description

LOWER returns the input string with all letters converted to lowercase.

6.5.8.3 Example

<Query> Convert the input string into lowercase letters and output it.

```
iSQL> SELECT LOWER('ONE PAGE PROPOSAL') Lowercase FROM dual;
LOWERCASE
---------------------
one page proposal
1 row selected.
```

6.5.9 LPAD

6.5.9.1 Syntax

```
LPAD (expr1, n [, expr2])
```

6.5.9.2 Description

LPAD pads the left of *expr1* with the sequence of characters in *expr2*, repeatedly if necessary, until the resultant string is *n* characters long. If *expr2* is not specified, then *expr1* is padded with blank spaces. If *expr1* is longer than *n*, this function simply returns the leftmost *n* characters of *expr1*.

Note that *n* is the number of characters, not bytes, and thus the number of bytes in the string may vary depending on the respective characters set that are in use on the server and the client (NLS_USE).

This function is useful for formatting the output of a query.

6.5.9.3 Example

<Query> Pad the left of the character string “abc” with “xyz” and return a total of 10 characters.

```
iSQL> SELECT LPAD('abc', 10, 'xyz') Lpad_ex FROM dual;
LPAD_EX
------------------------------------------------
xyzxyzxabc
1 row selected.
```

6.5.10 LTRIM

6.5.10.1 Syntax

```
LTRIM (expr1 [, expr2])
```

6.5.10.2 Description

LTRIM compares each of the characters in *expr1* with each of the characters in *expr2*, starting from the leftmost character in *expr1*. If any of the characters in *expr2* is the same as the current character in *expr1*, that character is deleted from *expr1*. This process occurs repeatedly until none of the characters in *expr2* match the current character in *expr1*, at which point the current character in *expr1* (i.e the first character without a match in *expr2*) is output along with all subsequent characters.

The default value for *expr2* is a single blank. Therefore, if *expr2* is omitted, blank spaces are trimmed from the left of *expr1*.

This function is case-sensitive.

6.5.10.3 Examples

<Query> Trim all occurrences of the letters “a” and “b” from the beginning of the string “abaAabLEFT TRIM” and output the result.

```
iSQL> SELECT LTRIM ('abaAabLEFT TRIM', 'ab') Ltrim_ex
 FROM dual;
LTRIM_EX
-------------------
AabLEFT TRIM
1 row selected.
```

<Query> Retrieve the month that each employee joined the company by removing the leading day information and the hyphen between the day and month.

```
iSQL> SELECT e_lastname, LTRIM(LTRIM(join_date, '1234567890'), '-')
Join_Month
 FROM employees;
E_LASTNAME            JOIN_MONTH
-----------------------------------------------------------
.
.
.
Ghorbani              DEC-2009
Momoi                 SEP-2010
Fleischer             JAN-2004
Wang                  NOV-2009
.
.
.
20 rows selected.
```

6.5.11 NCHR

6.5.11.1 Syntax

```
NCHR (n)
```

6.5.11.2 Description

NCHR returns the character corresponding to the value of *n* in the national character set. The return value is of type NVARCHAR.

6.5.11.3 Example

Output the 187th (U+00BB) character in the national character set.

```
SELECT NCHR(187) FROM DUAL;
```

```
NC
--
>>
```

6.5.12 OCTET_LENGTH, LENGTHB

6.5.12.1 Syntax

```
OCTET_LENGTH (expr)
```

6.5.12.2 Description

OCTET_LENGTH returns the size, in bytes, of the input character string.

The number of bytes in the input character string can vary depending on the database character set and the national character set of the database.

LENGTHB has the same function as OCTET_LENGTH.

6.5.12.3 Examples

<Query> Output the length, in bytes, assigned to the character string ' 우리나라 '. (Note that the database character set in this case has been set to K016KSC5601.)

```
iSQL> SELECT OCTET_LENGTH(' 우리나라 ') FROM dual;
OCTET_LENGTH(' 우리나라 ')
---------------------------
8
1 row selected.
```

<Query> Return the lengths, in bytes, of the addresses in the *managers* table.

```
iSQL> SELECT OCTET_LENGTH(address) FROM managers;
OCTET_LENGTH(ADDRESS)
------------------------
32
18
2 rows selected.
```

6.5.13 REPLACE2

6.5.13.1 Syntax

```
REPLACE2 (expr1, expr2, [expr3])
```

6.5.13.2 Description

REPLACE2 replaces every occurrence of *expr2* in *expr1* with *expr3* and returns the result. If *expr3* is omitted or NULL, then all occurrences of *expr2* are removed. If *expr2* is NULL, then *expr1* is returned unchanged.

Unlike the TRANSLATE function, which replaces individual characters, the REPLACE2 function replaces an entire character string with another character string.

This function is case-sensitive.

6.5.13.3 Examples

<Query> Query the *departments* table and replace all instances of the word "team" in the *dname* column with the word "division".

```
iSQL> SELECT REPLACE2(dname, 'Team', 'Division')
FROM departments;
REPLACE2(DNAME, 'Team', 'Division')
------------------------------------------------
Applied Technology Division
Engine Development Division
Marketing Division
Planning and Management Division
Sales Division
5 rows selected.
```

<Query> In the character string "abcdefghicde", replace all instances of "cde" with "xx".

```
iSQL> SELECT REPLACE2('abcdefghicde', 'cde', 'xx') FROM dual;
REPLACE2('abcdefghicde', 'cde', 'xx')
-----------------------------------------
abxxfghixx
1 row selected.
```

6.5.14 RPAD

6.5.14.1 Syntax

```
RPAD (expr1, n [, expr2])
```

6.5.14.2 Description

RPAD pads the right end of *expr1* with the sequence of characters in *expr2*, repeatedly if necessary, until the resultant string is *n* characters long. If *expr2* is not specified, then *expr1* is padded with blank spaces. If *expr1* is longer than *n*, this function simply returns the leftmost *n* characters of *expr1*.

Note that n is the number of characters, not bytes, and thus the number of bytes in the string may vary depending on the respective character sets that are in use on the server and the client (NLS_USE).

6.5.14.3 Example

<Query> In the following example, the right side of the character string "123" is padded with "0" (zeroes) and then type-converted to return a 10-digit number.

```
iSQL> SELECT TO_NUMBER(RPAD('123', 10, '0')) rpad_ex FROM dual;
RPAD_EX
--------------
1230000000
```

```
1 row selected.
```

6.5.15 RTRIM

6.5.15.1 Syntax

```
RTRIM (expr1 [, expr2])
```

6.5.15.2 Description

RTRIM compares each of the characters in *expr1* with each of the characters in *expr2*, starting from the rightmost character in *expr1* and working toward the left. If any of the characters in *expr2* is the same as the current character in *expr1*, that character is deleted from *expr1*. This process occurs repeatedly until none of the characters in *expr2* match the current character in *expr1*, at which point the current character in *expr1* (i.e the rightmost character without a match in expr2) is output along with all preceding characters.

The default value for *expr2* is a single blank. Therefore, if *expr2* is omitted, blank spaces are trimmed from the right of *expr1*.

This function is case-sensitive.

6.5.15.3 Examples

<Query> Remove all lowercase “a” and “b” characters from the end of the character string “RightTrimbaAbab” and output the result.

```
iSQL> SELECT RTRIM ('RIGHTTRIMbaAbab', 'ab') rtrim_ex FROM dual;
RTRIM_EX
-------------------
RIGHTTRIMbaA
1 row selected.
```

<Query> Retrieve the day and month that each employee joined the company by removing the trailing year information and the hyphen between the month and year.

```
iSQL> SELECT e_lastname, RTRIM(RTRIM(join_date, '1234567890'), '-') Join_Date
 FROM employees;
E_LASTNAME            JOIN_DATE
---------------------------------------------------------------------------
.
.
.
Ghorbani              20-DEC
Momoi                 09-SEP
Fleischer             24-JAN
Wang                  29-NOV
.
.
.
20 rows selected.
```

6.5.16 SIZEOF

6.5.16.1 Syntax

```
SIZEOF (expr)
```

6.5.16.2 Description

SIZEOF returns the size of a character string or the size allocated thereto. The input character string can be CHAR, VARCHAR or any numeric data type. If the input value is of a numeric data type, it is converted to VARCHAR and the size allocated thereto is returned.

Unlike OCTET_LENGTH, which returns the actual size of the input character string, SIZEOF returns the length of the space allocated to the input character string, or the length of the column, which was specified when the table was created.

Note therefore that SIZEOF returns 20 for the INTEGER, BIGINT, and SMALLINT data types, 47 for the DECIMAL, FLOAT, NUMBER, and NUMERIC types, and 22 for the DOUBLE and REAL data types.

6.5.16.3 Example

<Query> Retrieve the length of the column dummy in the table *dual*.

```
iSQL> SELECT SIZEOF(dummy) FROM dual;
SIZEOF(DUMMY)
--------------
1
1 row selected.
```

6.5.17 SUBSTR, SUBSTRB, SUBSTRING

6.5.17.1 Syntax

```
SUBSTR (expr, start [, length])
```

6.5.17.2 Description

SUBSTR returns a portion of *expr* that is *length* characters long, beginning at character *start*.

If *start* is positive, which is typical, ALTIBASE HDB counts from the beginning of the input string to find the first character. *start* can also be set to a negative value, in which case ALTIBASE HDB counts backwards from the end of string to find the first character. If *start* is set to 0, then it is handled as though it were set to 1.

If *length* is omitted, then ALTIBASE HDB returns all characters to the end of the string.

The input character string can be CHAR, VARCHAR or any numeric data type. If the input value is of a numeric data type, it is converted to VARCHAR. The type of the return value is VARCHAR.

Unlike SUBSTR, which determines the position and length in units of the characters of the input character set, SUBSTRB determines the position and length in units of bytes rather than characters.

SUBSTRING has the same function as SUBSTR.

6.5.17.3 Examples

<Query> Return a substring of the character string "SALESMAN", 5 characters long and starting from the first character.

```
iSQL> SELECT SUBSTR('SALESMAN', 1, 5) Substring FROM dual;
SUBSTRING
-------------
SALES
1 row selected.
```

<Query> Return a substring of the input string "ABCDEFG".

```
iSQL> SELECT SUBSTR('ABCDEFG', -5, 4) Substring FROM dual;
SUBSTRING
-------------
CDEF
1 row selected.
```

<Query> Return a character string 2 bytes in length, starting from the 5th byte in the character string "ABCDEFG".

```
iSQL> SELECT SUBSTRB('ABCDEFG', 5, 2) Substring_with_bytes FROM dual;
SUBSTRING_WITH_BYTES
------------------------
EF
1 row selected.
```

6.5.18 TRANSLATE

6.5.18.1 Syntax

```
TRANSLATE (expr1, expr2, expr3)
```

6.5.18.2 Description

TRANSLATE checks each character in *expr1* to determine whether it is found in *expr2*. If the character is not found in *expr2*, it is left as it is. If, however, the character is found in *expr2*, it is replaced by the character at the corresponding position in *expr3*. This function returns the character string that results from modifying *expr1* in this way.

It is possible to specify *expr2* so that it has more characters than *expr3*. In this case, some characters at the end of *expr2* will have no corresponding characters in *expr3*. If any of these characters are found in *expr1*, they are simply removed from the character string to be returned. Setting *expr3* to an empty string causes this function to remove all characters in *expr2* from *expr1*.

If *expr3* is longer than *expr2*, the remaining characters in expr3 are ignored. If the same character is specified multiple times in *expr2*, the first match in *expr3* is used.

This function is case-sensitive.

6.5.18.3 Examples

<Query> For all items for which the quantity in inventory is more than 50000, replace the letter "M" in the product name with "L".

```
iSQL> SELECT TRANSLATE(gname, 'M', 'L')
FROM goods
WHERE stock > 50000;
TRANSLATE(GNAME, 'M', 'L')
------------------------------------------------
TL-U200
L-190G
2 rows selected.
```

<Query> Convert all alphabetic characters in a string to lowercase.

```
iSQL> SELECT
TRANSLATE('0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789',
'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
'abcdefghijlkmnopqrstuvwxyz')
FROM dual;
TRANSLATE('0123456789ABCDEFGHIJKLMNOPQRS
------------------------------------------------
0123456789abcdefghijlkmnopqrstuvwxyz0123456789
1 row selected.
```

<Query> Remove all alphabetic characters from a license number and return only the numbers.

```
iSQL> SELECT TRANSLATE('3PQR334',
'0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ',
'0123456789') License
FROM dual;
LICENSE
------------------
3334
1 row selected.
```

6.5.19 TRIM

6.5.19.1 Syntax

```
TRIM (expr1 [, expr2])
```

6.5.19.2 Description

TRIM returns a substring of *expr1* that starts with the first character that is not found in *expr2* and ends with the last character that is not found in *expr2*. The portion of *expr1* between these two characters is returned without change.

In other words, TRIM removes all characters found in *expr2* from the beginning and end of *expr1*, and returns the result.

TRIM is, therefore, functionally the same as executing both LTRIM and RTRIM with the same value of *expr2*.

The default value for *expr2* is a single space. This means that if *expr2* is not specified, all blank spaces are trimmed from both ends of *expr1*.

This function is case-sensitive.

6.5.19.3 Example

<Query> Remove all instances of lowercase "a" and "b" from both ends of the character string "abbAaBbAbba" and output the result.

```
iSQL> SELECT TRIM ('abbAaBbAbba', 'ab') trim_ex FROM dual;
TRIM_EX
---------------
AaBbA
1 row selected.
```

6.5.20 UPPER

6.5.20.1 Syntax

```
UPPER (char)
```

6.5.20.2 Description

UPPER returns the input string with all letters converted to uppercase.

6.5.20.3 Example

<Query> Convert the input string into uppercase letters and output it.

```
iSQL> SELECT UPPER('Capital') Uppercase FROM dual;
UPPERCASE
-------------
CAPITAL
1 row selected.
```

6.5.21 REPLICATE

6.5.21.1 Syntax

```
REPLICATE (expr, n)
```

6.5.21.2 Description

REPLICATE returns a character string in which *expr* is repeated *n* times. *expr* must be a string and *n* must be a positive number. If 0 (zero) or a negative number is entered, the REPLICATE function will return NULL.

6.5.21.3 Example

<Query> Output a string comprising three repetitions of "KSKIM".

```
iSQL> SELECT REPLICATE ('KSKIM', 3) FROM dual;
REPLICATE ('KSKIM', 3)
----------------------------------
KSKIMKSKIMKSKIM
1 row selected.
```

6.5.22 REVERSE_STR

6.5.22.1 Syntax

```
REVERSE_STR (expr)
```

6.5.22.2 Description

REVERSE_STR reverses the order of the characters in *expr* and returns the result.

6.5.22.3 Examples

<Query> Output the string "KSKIM" in reverse.

```
iSQL> SELECT REVERSE_STR ('KSKIM') FROM dual;
REVERSE_STR ('KSKIM')
-------------------------
MIKSK
1 row selected.
```

<Query> Reverse the characters in the string ' 알티베이스 5' and output the result. (Note that the database character set has been set to KO16KSC5601.)

```
iSQL> SELECT REVERSE_STR (' 알티베이스 5') FROM dual;
REVERSE_STR (' 알티베이스 5')
-------------------------
5 스이베티알
1 row selected.
```

6.5.23 STUFF

6.5.23.1 Syntax

```
STUFF (expr1, start, length, expr2)
```

6.5.23.2 Description

STUFF removes a substring, beginning at the position specified by *start* and having the length specified by *length*, from *expr1*, and inserts *expr2* in its place.

If *length* is specified as 0 (zero), *expr2* is simply inserted without first deleting anything from *expr1*. In this case, *expr2* is inserted to the left of the position specified using *start*. If *length* is greater than the number of characters to the right of the start position in *expr1*, all characters in *expr1* to the right of *start* are removed, and *expr2* is inserted.

To simply concatenate *expr2* to the end of *expr1*, set *start* to (1 + the length of *expr1*) and *length* to 0 (zero) or a positive number.

If the value of either *start* or *length* is negative, or if the value of *start* is greater than (1 + the length of *expr1*), an error will occur.

6.5.23.3 Examples

<Query> Use the STUFF function to convert "KDHONG" to "KILDONG HONG".

```
iSQL> SELECT STUFF ('KDHONG', 2, 1, 'ILDONG ') FROM dual;
STUFF ('KDHONG', 2, 1, 'ILDONG ')
--------------------------------------------------
KILDONG HONG
1 row selected.
```

<Query> Insert *expr2* before *expr1*.

```
iSQL> SELECT STUFF ('KDHONG', 1, 0, 'ILDONG ') FROM dual;
STUFF ('KDHONG', 1, 0, 'ILDONG ')
--------------------------------------------------
ILDONG KDHONG
1 row selected.
```

<Query> Insert *expr2* after *expr1*.

```
iSQL> SELECT STUFF ('KDHONG', 7, 0, 'ILDONG ') FROM dual;
STUFF ('KDHONG', 7, 0, 'ILDONG ')
--------------------------------------------------
KDHONGILDONG
1 row selected.
```

<Query> Set length to 0 to insert *expr2* to the left of the start position without deleting anything.

```
iSQL> SELECT STUFF ('KDHONG', 2, 0, 'ILDONG ') FROM dual;
STUFF ('KDHONG', 2, 0, 'ILDONG ')
--------------------------------------------------
KILDONG DHONG
1 row selected.
```

<Query> Use the STUFF function to change the contents of the input string when KO16KSC5601 is set as the database character set.

```
iSQL> SELECT STUFF ('알티베이스0', 5, 1, '데이터베이스') FROM dual;
STUFF ('알티베이스0', 5, 1, '데이터베이스')
-----------------------------------------------
알티베이데이터베이스0
1 row selected.
```

6.6 Datetime Functions

These functions perform tasks on entered date and time values and return character, numeric or date/time type values. If a value of an input argument is of the character data type, the value must be entered in the format set in the ALTIBASE_DATE_FORMAT environment variable or the DEFAULT_DATE_FORMAT property (the former takes precedence). Additionally, this format is also used to display date/time values that are returned when these functions are executed at the iSQL prompt.

For more information on the DATE data type and the datetime format model, which is used when returning date type data, please refer to the *General Reference*.

6.6.1 ADD_MONTHS

6.6.1.1 Syntax

```
ADD_MONTHS (date, n)
```

6.6.1.2 Description

ADD_MONTHS adds *n* months to *date* and returns the result. The *n* argument can be an integer or any value that can be implicitly converted to an integer.

6.6.1.3 Example

<Query> Display the date on which the employee whose employee number is 10 was hired and the date six month after the hiring date.

```
SELECT join_date,
 ADD_MONTHS(join_date, 6)
 FROM employees
 WHERE eno = 10;
JOIN_DATE    ADD_MONTHS(JOIN_DATE, 6)
-----------------------------------------
05-JAN-2010  05-JUL-2010
1 row selected.
```

6.6.2 DATEADD

6.6.2.1 Syntax

```
DATEADD (date, n, date_field_name)
```

6.6.2.2 Description

DATEADD increases the element of *date* that is specified using *date_field_name* by the amount specified using *n*. If *n* is a non-integer, the places after the decimal point are first discarded (i.e. the value is truncated), and then the operation is carried out.

If the value of *date_field_name* is 'SECOND', *n* (that is, the number of seconds) cannot exceed the equivalent of 68 years. If it is 'MICROSECOND', *n* cannot exceed the equivalent of 30 days.

The following table shows the *date_field_name* values that can be used with the DATEADD function, and the result in each case:

Date Field Name	Description
CENTURY	The year portion of *date* is increased by *n**100.
YEAR	The year portion of *date* is increased by *n*.
QUARTER	The month portion of *date* is increased by *n**3.
MONTH	The month portion of *date* is increased by *n*.
WEEK	The day portion of *date* is increased by *n**7.
DAY	The day portion of *date* is increased by *n*.
HOUR	The hour portion of *date* is increased by *n*.
MINUTE	The minute portion of *date* is increased by *n*.
SECOND	The second portion of *date* is increased by *n*.
MICROSECOND	The microsecond portion of *date* is increased by *n*.

6.6.2.3 Example

<Query> Get the number of employees who were hired less than 40 days ago.

```
iSQL> SELECT COUNT(*) FROM employees WHERE join_date > DATEADD (SYSDATE, -40,
‘DAY’);
COUNT
----------
5
1 row selected.
```

6.6.3 DATEDIFF

6.6.3.1 Syntax

```
DATEDIFF (startdate, enddate, date_field_name)
```

6.6.3.2 Description

DATEDIFF returns the difference between *enddate* and *startdate* (i.e. *enddate* - *startdate*) in the units specified using *date_field_name*. If *startdate* is greater than *enddate*, a negative number is returned.

This function first determines the values of *enddate* and *startdate* in the units specified using *date_field_name*, and then returns the difference. This function always returns an integer. Noninteger results are truncated, that is, rounded down.

The possible values for *date_field_name* with DATEDIFF, that is, the units in which the resultant value can be returned, are as follows:

- CENTURY
- YEAR
- QUARTER
- MONTH
- WEEK
- DAY
- HOUR
- MINUTE
- SECOND
- MICROSECOND

The range of values that the DATEDIFF function can return is limited for particular values of *date_field_name*. If the value of *date_field_name* is 'SECOND', the difference between *enddate* and *startdate* cannot exceed 68 years. If it is 'MICROSECOND', the difference between *enddate* and *start-date* cannot exceed 30 days.

The result returned by this function is in the BIG INTEGER data type.

6.6.3.3 Example

<Query> Get the difference between August 31, 2005 and November 30, 2005 in months.

```
iSQL> SELECT DATEDIFF ('31-AUG-2005', '30-NOV-2005', 'MONTH') FROM dual;
DATEDIFF ('31-AUG-2005', '30-NOV-2005',
-----------------------------------------
3
1 row selected.
```

6.6.4 DATENAME

6.6.4.1 Syntax

```
DATENAME (date, date_field_name)
```

6.6.4.2 Description

Depending on the input *date_field_name* value, DATENAME returns the name of the month or week-day for the specified date.

The following table shows the values of *date_field_name* that can be used with the DATENAME func-tion:

Date Field Name	Description
MONTH, Month, month	The month (unabbreviated)
MON, Mon, mon	The month (abbreviated)
DAY, Day, day	The day of the week (unabbreviated)
DY, Dy, dy	The day of the week (abbreviated)

The possible return values for each value of *date_field_name* are as shown below. *date_field_name* can be entered in uppercase, lowercase, or title case (i.e. first character capitalized). The case of the output will match that of *date_field_name*.

- MONTH

  JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER

- MON

  JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC

- DAY

  SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY

- DY

  SUN, MON, TUE, WED, THU, FRI, SAT

6.6.4.3 Example

<Query> Output the month of Dec. 28, 1980.

```
iSQL> SELECT DATENAME ('28-DEC-1980', 'Month') FROM dual;
DATENAME ('28-DEC-1980', 'Month')
-------------------------------------
December
1 row selected.
```

6.6.5 EXTRACT, DATEPART

6.6.5.1 Syntax

```
EXTRACT (date, date_field_name)

DATEPART (date, date_field_name)
```

6.6.5.2 Description

These functions, which are identical, return the value corresponding to *date_field_name* for the input *date*.

Date Field Name	Description
CENTURY	The current century (for example, returns “21” for “2011”, or “20” for “1999”)
YEAR	Year
QUARTER	Quarter
MONTH	Month
WEEK	Returns the week of the year. (The first Saturday of the year and any preceding days are in the first week of the year, meaning that the first “week” might be only one day long.)
WEEKOFMONTH	Returns the week of the month. (The first Saturday of the month and any preceding days are in the first week of the month, meaning that the first “week” might be only one day long.)
DAY	Day
DAYOFYEAR	Returns the day of the year.
DAYOFWEEK	Returns the day of the week. (Sunday = 1)
HOUR	Hour
MINUTE	Minute
SECOND	Second
MICROSECOND	Microsecond

6.6.5.3 Example

<Query> Find the quarter in which employee number 10 was hired.

```
iSQL> SELECT DATEPART (join_date,'QUARTER')
 FROM employees
 WHERE eno = 10;
DATEPART (JOIN_DATE,'QUARTER')
---------------------------------
1
1 row selected.
```

6.6.6 MONTHS_BETWEEN

6.6.6.1 Syntax

```
MONTHS_BETWEEN (date1, date2)
```

6.6.6.2 Description

MONTHS_BETWEEN subtracts *date2* from *date1* and returns the result in months. If *date1* is less than *date2*, a negative number will be returned.

If *date1* and *date2* are the same day of the same month, the same day of different months, or the respective last days of different months, an integer will be returned. In these cases, the hour, minute, second and microsecond portions of *date1* and *date2* are ignored.

In other cases, the return value is not an integer, and the fractional part is determined by converting the hour, minute, second and microsecond portions of the input dates to months, on the assumption that there are 31 days in every month.

The return type is DOUBLE.

6.6.6.3 Example

<Query> Get the difference between February 2, 1995 and January 1, 1995 in months.

```
iSQL> SELECT MONTHS_BETWEEN (TO_DATE('02-02-1995','MM-DD-YYYY'), TO_DATE('01-
01-1995','MM-DD-YYYY') ) Months FROM DUAL;
MONTHS
-------------------------
1.03225806451613
1 row selected.
```

6.6.7 ROUND

6.6.7.1 Syntax

```
ROUND (date [, date_field_name])
```

6.6.7.2 Description

ROUND returns a date that has been rounded to the nearest unit specified using *date_field_name*. The default value of *date_field_name* is 'DAY'.

The following table shows the values of *date_field_name* that can be used with the ROUND function:

Date Field Name	Description
CENTURY	Returns the first day in the nearest century. Dates in the year xx51 and beyond are rounded up to the next century. (Note that a century begins in the year xxx1, e.g. 2001.)
YEAR	Dates from July 1 and onward are rounded up to the next year.
QUARTER	Returns the first day in the nearest quarter. The 16th day of the second month and subsequent dates are rounded up to the next quarter.
MONTH	The 16th and subsequent dates are rounded up to the next month.
WEEK	Returns the nearest Sunday. Thursdays are rounded up; Wednesdays are rounded down.
DAY	Times after 12:00 PM are rounded to the following day.
HOUR	Times 30 minutes after the hour are rounded to the next hour.
MINUTE	Times 30 seconds after the minute are rounded to the next minute.

6.6.7.3 Example

<Query> Round December 27, 1980 to the nearest new year and output the result.

```
iSQL> SELECT ROUND ( TO_DATE('27-DEC-1980', 'DD-MON-YYYY'), 'YEAR') FROM 
dual;
ROUND ( TO_DATE('27-DEC-1980', 'DD-MON-Y
-------------------------------------------
1981/01/01 00:00:00
1 row selected.
```

6.6.8 LAST_DAY

6.6.8.1 Syntax

```
LAST_DAY (date)
```

6.6.8.2 Description

LAST_DAY returns the last day of the month that contains *date*. The return type is always DATE, regardless of whether the type of *date* is DATE, CHAR or VARCHAR.

6.6.8.3 Examples

<Query> Display the last day in December 2001.

```
iSQL> SELECT LAST_DAY(TO_DATE('15-DEC-2001')) FROM dual;
LAST_DAY(TO_DATE('15-DEC-2001'))
-----------------------------------
2001/12/31 00:00:00
1 row selected.
```

<Query> Display the last day of the month in which each employee was hired.

```
iSQL> SELECT LAST_DAY(join_date ) FROM employees;
LAST_DAY(JOIN_DATE )
-----------------------

2009/11/30 00:00:00
2010/01/31 00:00:00
.
.
.
20 rows selected.
```

6.6.9 NEXT_DAY

6.6.9.1 Syntax

```
NEXT_DAY (date, weekday)
```

6.6.9.2 Description

The return type is always DATE, regardless of whether the type of *date* is DATE, CHAR or VARCHAR. NEXT_DAY returns the date of the first weekday, specified using *weekday*, that is later than *date*. If the day of the week of *date* is the same as *weekday*, the same weekday the following week is returned.

The weekday argument must be one of the days of the week. It is not case-sensitive, and can be either the entire weekday or the first three characters of the weekday.

6.6.9.3 Example

<Query> For each employee, display the hiring date and the date of the following Sunday.

```
iSQL> SELECT join_date, NEXT_DAY(join_date, 'SUNDAY') First_sunday
 FROM employees;
JOIN_DATE    FIRST_SUNDAY
-----------------------------
.
.
.
24-JAN-2004  25-JAN-2004
29-NOV-2009  06-DEC-2009
14-JUN-2010  20-JUN-2010
05-JAN-2010  10-JAN-2010
.
.
.
20 rows selected.
```

6.6.10 SYSDATE

6.6.10.1 Syntax

```
SYSDATE
```

6.6.10.2 Description

SYSDATE returns the current date and time set for the operating system on which ALTIBASE HDB is installed.

6.6.10.3 Example

<Query> Display the system date.

```
iSQL> SELECT SYSDATE System_Date FROM dual;
SYSTEM_DATE
-----------------------
2005/01/20 09:49:33
1 row selected.
```

6.6.11 SYSTIMESTAMP

6.6.11.1 Syntax

```
SYSTIMESTAMP
```

6.6.11.2 Description

Output the date in the current system. It is an alias of the SYSDATE function. It does not support time zones.

6.6.11.3 Example

<Query> Output the system date (i.e. the current date).

```
iSQL> SELECT SYSTIMESTAMP FROM dual;
SYSTIMESTAMP
-----------------------
2005/01/20 09:49:33
1 row selected.
```

6.6.12 TRUNC (date)

6.6.12.1 Syntax

```
TRUNC (date [, fmt])
```

6.6.12.2 Description

TRUNC returns *date* truncated to the unit specified using *fmt*. In other words, it replaces the value of all units below the unit specified using *fmt* with zeroes (0's).

If *fmt* is omitted, then the time portion of the day is removed, that is, *date* is truncated to the beginning of *date*.

6.6.12.3 Valid string inputs for *fmt*

- YEAR
- MONTH
- DAY
- HOUR
- MINUTE
- SECOND
- MICROSECOND

6.6.12.4 Examples

<Query> Round off the time portion of the system time and return the resultant date.

```
iSQL> SELECT TRUNC(SYSDATE) FROM DUAL;
<Result>
TRUNC(SYSDATE)
-----------------------
2005/07/19 00:00:00
1 row selected.
```

<Query> Round off the date portion of the input date and return the resultant year.

```
iSQL> SELECT TRUNC(TO_DATE('2005-JUL-19','YYYY-MON-DD'), 'YEAR') New_Year
FROM DUAL;
<Result>
NEW_YEAR
-----------------------
2005/01/01 00:00:00
1 row selected.
```

6.7 Conversion Functions

Conversion functions convert an input value from one data type to another.

6.7.1 ASCIISTR

6.7.1.1 Syntax

```
ASCIISTR (expr)
```

6.7.1.2 Description

ASCIISTR converts a string in any character set to an ASCII string and returns it. Non-ASCII characters in *expr* are expressed as '\xxxx', that is, in UTF-16 code. The return type is VARCHAR.

To convert a string to the national character set, use UNISTR.

6.7.1.3 Example

<Query> Convert the input strings to ASCII strings.

```
iSQL> SELECT ASCIISTR('ABÄCDE') FROM DUAL;
ASCIISTR('
----------
AB\00C4CDE
1 row selected.
iSQL> select asciistr('abcåñö') from dual;
ASCIISTR('ABCÅÑÖ')
------------------
abc\00E5\00F1\00F6
1 row selected.
```

6.7.2 BIN_TO_NUM

6.7.2.1 Syntax

```
BIN_TO_NUM (expr)
```

6.7.2.2 Description

This function converts *expr* into a decimal number. *expr* can be a binary number or a string that consists exclusively of ones (1's) and zeroes (0's) and is a maximum of 32 characters long. The return type is INTEGER.

Note: If any arithmetic operations are performed in expr, the numbers are handled as decimal numbers. The resultant decimal number has to comprise only ones and zeroes in order to be implicitly converted to BIN before being converted back to a decimal number.

6.7.2.3 Example

<Query> Convert the given binary number to a decimal number.

```
iSQL> SELECT BIN_TO_NUM ('1010') FROM dual;
BIN_TO_NUM ('1010')
----------------------
10
1 row selected.
```

6.7.3 CONVERT

6.7.3.1 Syntax

```
CONVERT(expr, dest_char_set [,source_char_set])
```

6.7.3.2 Description

This function converts *expr* from *source_char_set* to *dest_char_set*. Any character sets that are supported by ALTIBASE HDB can be specified in *dest_char_set* and *source_char_set*. If *expr* contains any characters that are not supported by either *source_char_set* or *dest_char_set*, question marks ("?") will be returned. If *source_char_set* is not specified, the default database character set is taken as *source_char_set*.

6.7.3.3 Example

<Query> Convert the characters "ABC" from the UTF8 character set to the US7ASCII character set.

```
iSQL> select convert( 'ABC', 'US7ASCII', 'UTF8') from dual;
CONVER
------
ABC
```

6.7.4 HEX_TO_NUM

6.7.4.1 Syntax

```
HEX_TO_NUM (expr)
```

6.7.4.2 Description

HEX_TO_NUM converts *expr* into a decimal number. *expr* must be a string that consists exclusively of the numeric characters from 0 to 9 and the alphabet characters from A to F, and can be a maximum of 8 characters long.

The return type is INTEGER.

6.7.4.3 Example

<Query> Convert the given hexadecimal number to a decimal number.

```
iSQL> SELECT HEX_TO_NUM ('1A') FROM dual;
HEX_TO_NUM ('1A')
--------------------
26
1 row selected.
```

6.7.5 OCT_TO_NUM

6.7.5.1 Syntax

```
OCT_TO_NUM (expr)
```

6.7.5.2 Description

OCT_TO_NUM converts *expr* into a decimal number. *expr* can be an octal number or a string that consists exclusively of the numeric characters from 0 to 7 and is a maximum of 11 characters long. The return type is INTEGER.

Note: If any arithmetic operations are performed in expr, the numbers are handled as decimal numbers. The resultant decimal number can comprise only the digits 0 through 7 in order to be implicitly converted to OCT before being converted back to a decimal number.

6.7.5.3 Example

<Query> Convert the given octal number to a decimal number.

```
iSQL> SELECT OCT_TO_NUM ('71') FROM dual;
OCT_TO_NUM ('71')
--------------------
57
1 row selected.
```

6.7.6 TO_BIN

6.7.6.1 Syntax

```
TO_BIN (n)
```

6.7.6.2 Description

TO_BIN converts *n* into a binary number. *n* can be a decimal integer or a string consisting of the numeric characters from 0 to 9.

The range of possible input values is from -2147483647 to 2147483647, that is, from -(2^{31}-1) to (2^{31}-1). When a negative value is entered, all bits are flipped, and 1 is added to the output.

The output is signed. Because leading zeroes are not output, the sign value does not appear for positive numbers.

Note: The return type is VARCHAR.

6.7.6.3 Example

<Query> Convert the given value into a binary number.

```
iSQL> SELECT TO_BIN(1000) FROM dual;
TO_BIN(1000)
------------------------------------
1111101000
1 row selected.
```

6.7.7 TO_CHAR (datetime)

6.7.7.1 Syntax

```
TO_CHAR (date [, 'fmt'])
```

6.7.7.2 Description

TO_CHAR(datetime) converts *date*, which is a date type value, to a VARCHAR type string having the datetime format specified using *fmt*. If *fmt* is omitted, the output will be in the format specified using the DEFAULT_DATE_FORMAT property in the altibase.properties file. The default value for the DEFAULT_DATE_FORMAT property is DD-MON-RRRR. For more information about the datetime format, which is used when converting date type data, please refer to the *General Reference*.

6.7.7.3 Example

<Query>Retrieve the start date of all employees and output it in the form YYYY-MM-DD HH:MI:SS.

```
iSQL> SELECT e_firstname, e_lastname, TO_CHAR(join_date, 'YYYY-MM-DD
HH:MI:SS') Join_date
 FROM employees;
E_FIRSTNAME           E_LASTNAME            JOIN_DATE
---------------------------------------------------------------------------
.
.
.
Farhad                Ghorbani              2009-12-20 00:00:00
Ryu                   Momoi                 2010-09-09 00:00:00
Gottlieb              Fleischer             2004-01-24 00:00:00
Xiong                 Wang                  2009-11-29 00:00:00
.
.
.
20 rows selected.
```

6.7.8 TO_CHAR (number)

6.7.8.1 Syntax

```
TO_CHAR(n [, 'fmt'])
```

6.7.8.2 Description

TO_CHAR (number) converts input numeric type data to VARCHAR type data and outputs the result. It is possible to specify the format in which the result is output.

If a string is input, it is implicitly converted to a decimal number so that numeric operations can be performed on it before it is converted back into character type data.

6.7.8.3 Examples

<Query> Perform implicit conversion on the character string in order to interpret it as a number, perform the requested calculation, and return the result.

```
iSQL> SELECT TO_CHAR('01110' + 1) FROM dual;
TO_CHAR('01110' + 1)
------------------------------------------------
1111
1 row selected.
```

The following examples show various ways in which an SQL statement can be used to output numeric type data as character data.

<Query> Output the input numeric data as a string having the specified format.

```
iSQL> SELECT TO_CHAR (123, '99999') FROM dual;
TO_CHAR (123, '99999')
--------------------------
   123
1 row selected.

iSQL> SELECT TO_CHAR (123.4567, '999999') FROM dual;
TO_CHAR (123.4567, '999999')
--------------------------------
    123
1 row selected.

iSQL> SELECT TO_CHAR (1234.578, '9999.99') FROM dual;
TO_CHAR (1234.578, '9999.99')
---------------------------------
 1234.58
1 row selected.

iSQL> SELECT TO_CHAR (1234.578, '999.99999') FROM dual;
TO_CHAR (1234.578, '999.99999')
-----------------------------------
##########
1 row selected.
```

6.7.9 TO_DATE

6.7.9.1 Syntax

```
TO_DATE (expr[, 'fmt' ])
```

6.7.9.2 Description

TO_DATE converts CHAR or VARCHAR type data into a date type value. *fmt* is used to indicate the date format of *expr*. If *fmt* is omitted, *expr* must have the form set in the ALTIBASE_DATE_FORMAT environment variable or the DEFAULT_DATE_FORMAT property (the former takes precedence).

If the year or month of the input date is not specified in *expr*, the current year or month at the time of execution of TO_DATE will be returned as the year or month. For example, if TO_DATE (TO_CHAR (sysdate,'YYYY'),'YYYY') is executed at 17:32:34 on the date 2011/08/24, the result of execution would be '2011/08/01 00:00:00'.

If the date (i.e. day of the month) is not specified in *expr*, the current month at the time of execution of TO_DATE, together with the initial values for date and time, that is, 00 hours, 00 minutes, and 00 seconds on the first of the month, will be returned.

6.7.9.3 Examples

<Query>Add a record for a newly hired employee, including the employee number, name, and gender. Set the hiring date to November 19, 2011.

```
iSQL> INSERT INTO employees(eno, e_lastname, e_firstname, sex, join_date)
VALUES(22, 'Jones', 'Mary', 'F', TO_DATE('2011-11-19 00:00:00', 'YYYY-MM-DD
HH:MI:SS'));
1 row inserted.
```

<Query> Execute the TO_DATE function without outputting the month or the date.

```
iSQL>select to_char(to_date(to_char(sysdate,'YYYY'),'YYYY'),'YYYYMMDD HH24:M
I:SS') from dual;
TO_CHAR(TO_DATE(TO_CHAR(SYSDATE,'YYYY'),
-------------------------------------------
20050801 00:00:00
1 row selected.
(SYSDATE = 2005/08/24 17:32:34)
```

6.7.10 TO_HEX

6.7.10.1 Syntax

```
TO_HEX (n)
```

6.7.10.2 Description

TO_HEX converts *n* into a hexadecimal number. *n* can be a decimal integer or a string consisting of the numeric characters from 0 to 9.

Note: TO_HEX returns VARCHAR type data.

6.7.10.3 Example

<Query> Convert the input decimal number into a hexadecimal number.

```
iSQL> SELECT TO_HEX(1000) FROM dual;
TO_HEX(1000)
----------------
3E8
1 row selected.
```

6.7.11 TO_NCHAR (character)

6.7.11.1 Syntax

```
TO_NCHAR(expr)
```

6.7.11.2 Description

This function converts character type data from the database character set to the national character set. The return type is NVARCHAR.

This functions is similar to the CONVERT function.

6.7.11.3 Example

<Query> Convert the Korean syllable “ 안 ” to the national character set UTF-16 and output information about it. (Note that the code point corresponding to the syllable “ 안 ” is U+C548.)

```
iSQL> select dump( to_nchar('안'), 16 ) from dual;
DUMP( TO_NCHAR('안'), 16 )
--------------------------------------------------------------------------
-----
Type=NVARCHAR(UTF16) Length=4: 2,0,c5,48
```

6.7.12 TO_NCHAR (datetime)

6.7.12.1 Syntax

```
TO_NCHAR(date [, fmt])
```

6.7.12.2 Description

TO_NCHAR(datetime) converts date type data from the database character set to the national character set.

6.7.12.3 Example

<Query>Retrieve the date on which each employee was hired and convert it to the national character set.

```
iSQL> SELECT e_lastname, e_firstname, TO_NCHAR(join_date, 'YYYY-MM-DD 
HH:MI:SS') Join_date
 FROM employees;
E_LASTNAME            E_FIRSTNAME           JOIN_DATE
---------------------------------------------------------------------------
.
.
.
Momoi                 Ryu                   2010-09-09 00:00:00
Fleischer             Gottlieb              2004-01-24 00:00:00
Wang                  Xiong                 2009-11-29 00:00:00
Diaz                  Curtis                2010-06-14 00:00:00
.
.
.
20 rows selected.
```

6.7.13 TO_NCHAR (number)

6.7.13.1 Syntax

```
TO_NCHAR(n [,fmt])
```

6.7.13.2 Description

TO_NCHAR(number) converts numeric type data from the database character set to the national character set.

6.7.13.3 Example

<Query> Perform implicit conversion and an arithmetic operation on the input string and convert it to the national character set.

```
iSQL> SELECT TO_NCHAR('01110' + 1) FROM dual;
TO_NCHAR('01110' + 1)
------------------------------------------------
1111
1 row selected.
```

6.7.14 TO_NUMBER

6.7.14.1 Syntax

```
TO_NUMBER (char [, number_fmt] )
```

6.7.14.2 Description

TO_NUMBER converts a string into a numeric data type. The user can specify the desired numeric output format. For more information about the numeric output format, please refer to the *General Reference*.

The return type is FLOAT.

6.7.14.3 Examples

<Query> Convert the string “200.00” to FLOAT and then input the result into the database.

```
iSQL> UPDATE employees
SET salary = salary + TO_NUMBER( '200.00')
WHERE eno = 10;
1 row updated.
```

<Query> Convert the given strings into various numeric output formats.

```
iSQL> SELECT TO_NUMBER ( '0123.4500', '0990.9909' ) FROM dual;
TO_NUMBER ( '0123.4500', '0990.9909' )
-----------------------------------------
123.45
1 row selected.
iSQL> SELECT TO_NUMBER ( '$12,3.45-', '09,$0.00S' ) FROM dual;
TO_NUMBER ( '$12,3.45-', '09,$0.00S' )
-----------------------------------------
-123.45
1 row selected.
iSQL> SELECT TO_NUMBER ( '<$183.5>', '$9,000.0PR' ) FROM dual;
TO_NUMBER ( '<$183.5>', '$9,000.0PR' )
-----------------------------------------
-183.5
1 row selected
```

6.7.15 TO_OCT

6.7.15.1 Syntax

```
TO_OCT (n)
```

6.7.15.2 Description

TO_OCT converts *n* into an octal number. *n* can be a decimal integer or a string consisting of the numeric characters from 0 to 9.

Note: TO_OCT returns VARCHAR type data.

6.7.15.3 Example

<Query> Convert the given value into an octal number.

```
iSQL> SELECT TO_OCT(1000) FROM dual;
TO_OCT(1000)
----------------
```

```
1750
1 row selected.
```

6.7.16 UNISTR

6.7.16.1 Syntax

```
UNISTR(expr)
```

6.7.16.2 Description

UNISTR converts the input character string to the national character set.

expr can be a Unicode-encoded string. A Unicode-encoded string is entered in UTF16 code units, such as \xxxx. The return type is NVARCHAR.

This function is the opposite of ASCIISTR.

6.7.16.3 Example

<Query> Convert a string containing both ASCII- and Unicode-encoded characters to the national character set.

```
iSQL> SELECT UNISTR('abc\00e5\00f1\00f6') FROM DUAL;
UNISTR
------
abcåñö
1 row selected.
```

6.8 Other Functions

6.8.1 BINARY_LENGTH

6.8.1.1 Syntax

```
BINARY_LENGTH (expr)
```

6.8.1.2 Description

This function returns the length of a value in a BLOB, BYTE or NIBBLE type column.

6.8.1.3 Example

<Query> Output the length of three binary data type values.

```
iSQL> CREATE TABLE T1 (I1 BLOB, I2 Byte(10), I3 NIBBLE(10) );
Create success.
iSQL> INSERT INTO T1 VALUES ( BLOB'3FD', Byte'123FD', NIBBLE'90BCD');
1 row inserted.
iSQL> SELECT BINARY_LENGTH (I1), BINARY_LENGTH (I2), BINARY_LENGTH (I3) FROM
T1;
BINARY_LENGTH (I1) BINARY_LENGTH (I2) BINARY_LENGTH (I3)
-------------------------------------------------------------
2                  10                 5
1 row selected.
```

6.8.2 CASE2

6.8.2.1 Syntax

```
CASE2 (expr1, return_expr1
       [, expr2, return_expr2,..]
       [, default])
```

6.8.2.2 Description

CASE2 evaluates *expr1* and returns *return_expr1* if *expr1* evaluates to TRUE. If *expr1* evaluates to FALSE, *expr2* is evaluated, and if *expr2* evaluates to TRUE, *return_expr2* is returned. This process continues until a TRUE expression is found. If none of the expressions evaluate to TRUE, *default* is returned. If none of the expressions evaluate to TRUE and *default* is not specified, NULL is returned.

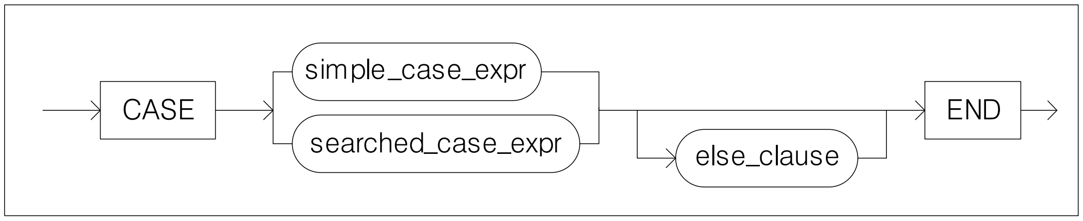
6.8.2.3 Example

<Query> Return employees' salaries. Substitute the word "HIGH" for every employee whose monthly pay is greater than 2000 dollars, the word "LOW" for every employee whose monthly pay is less than 1500, and "NULL" for employees without salary information. Return the actual salary for employees whose monthly pay is between 1500 and 2000 dollars.

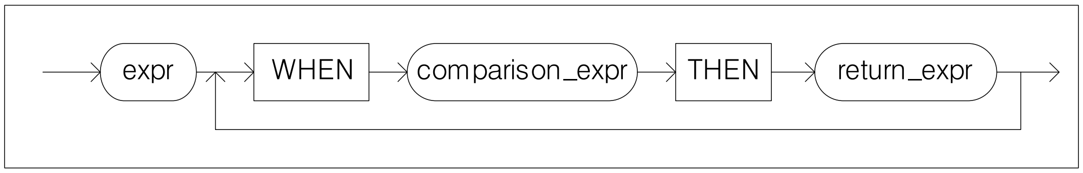
```
iSQL> SELECT e_lastname, e_firstname, emp_job, CASE2(salary > 2000, 'HIGH',
salary < 1500, 'LOW', salary IS NULL, 'NULL', TO_CHAR(salary)) Salary
 FROM employees;
E_LASTNAME            E_FIRSTNAME           EMP_JOB          SALARY
---------------------------------------------------------------------------
Moon                  Chan-seung            CEO              NULL
Davenport             Susan                 designer         1500
Kobain                Ken                   engineer         2000
Foster                Aaron                 PL               1800
Ghorbani              Farhad                PL               HIGH
Momoi                 Ryu                   programmer       1700
Fleischer             Gottlieb              manager          LOW
Wang                  Xiong                 manager          NULL
Diaz                  Curtis                planner          LOW
Bae                   Elizabeth             programmer       HIGH
Liu                   Zhen                  webmaster        HIGH
Hammond               Sandra                sales rep        1890
Jones                 Mitch                 PM               LOW
Miura                 Yuu                   PM               HIGH
Davenport             Jason                 webmaster        LOW
Chen                  Wei-Wei               manager          HIGH
Fubuki                Takahiro              PM               LOW
Huxley                John                  planner          1900
Marquez               Alvar                 sales rep        1800
Blake                 William               sales rep        NULL
20 rows selected.
```

6.8.3 CASE WHEN

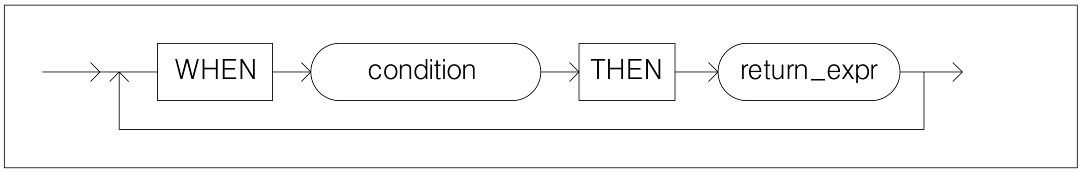
6.8.3.1 Syntax



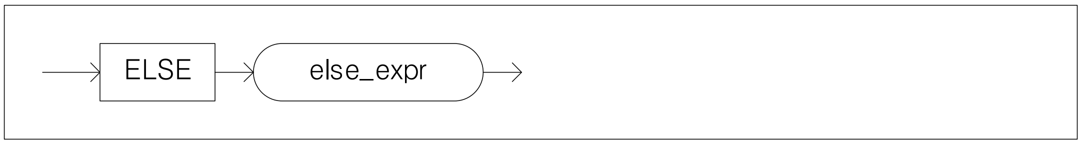
simple_case_expr



searched_case_expr



else_clause



6.8.3.2 Description

When CASE WHEN is used with *searched_case_expr*, it is the same as CASE2. That is, it returns *return_expr* for the first *condition* that evaluates to TRUE. If all of the specified *condition* evaluate to FALSE, CASE WHEN returns *else_expr* if specified, or NULL otherwise. When using this function in this way, a variety of comparison operators can be used.

When CASE WHEN is used with *simple_case_expr*, *expr* is compared with each of *comparison_expr* using the equality operator (“=”).

6.8.3.3 Example

<Query> Output ‘aaaaa’ if the value of the third character in column *c1* is a, output ‘bbbbb’ if it is b, output ‘ccccc’ if it is c, and output ‘zzzzz’ otherwise.

```
iSQL> create table test (c1 char(10));
iSQL> insert into test values('abcdefghi');
iSQL> select CASE substring(c1,3,1)
 WHEN 'a' THEN 'aaaaa'
 WHEN 'b' THEN 'bbbbb'
 WHEN 'c' THEN 'ccccc'
 ELSE 'zzzzz'
 END
 from test;
CASE SUBSTRING(C1,3,1)
-------------------------
ccccc
1 row selected.
```

6.8.4 DECODE

6.8.4.1 Syntax

```
DECODE (expr, comparison_expr1, return_expr1
[, comparison_expr2, return_expr2,..]
[, default])
```

6.8.4.2 Description

The functionality of DECODE is similar to that of CASE WHEN when used with *simple_case_expr*. That is, *expr* is sequentially compared with each of *comparison_expr* using the equality operator ("="), and the *return_expr* corresponding to the first *comparison_expr* that matches *expr* is returned. If none of *comparison_expr* match *expr*, *default* is returned. If none of *comparison_expr* match *expr* and *default* is not specified, NULL is returned.

6.8.4.3 Examples

<Query> If *i* is NULL, 1 or 2, return the string "NULL", "ONE" or "TWO", respectively.

```
iSQL> CREATE TABLE t2(i NUMBER);
Create success.
iSQL> INSERT INTO t2 VALUES(NULL);
1 row inserted.
iSQL> INSERT INTO t2 VALUES(1);
1 row inserted.
iSQL> INSERT INTO t2 VALUES(2);
1 row inserted.
iSQL> INSERT INTO t2 VALUES(3);
1 row inserted.
iSQL> SELECT DECODE(i, NULL, 'NULL', 1, 'ONE', 2, 'TWO') Revised_i FROM t2;
REVISED_I
-------------
NULL
ONE
TWO

4 rows selected.
```

<Query> Display the current wage of all employees. Increase the displayed wage 10% for engineers, 12% for sales representatives, and 20% for managers. Display the actual wage for all other employees.

```
iSQL> SELECT emp_job, salary,
 DECODE(RTRIM(emp_job, ' '),
 'engineer', salary*1.1,
 'sales rep', salary*1.12,
 'manager', salary*1.20,
 salary) Revised_salary
FROM employees;
EMP_JOB      SALARY      REVISED_SALARY
------------------------------------------------
CEO
designer     1500        1500
engineer     2000        2200
engineer     1800        1980
engineer     2500        2750
programmer   1700        1700
manager      500         600
.
.
.
20 rows selected.
```

6.8.5 DIGEST

6.8.5.1 Syntax

```
DIGEST(expr, algorithm_name)
```

6.8.5.2 Description

DIGEST uses a standard cryptographic hash algorithm to return the hash digest of *expr* as a VARCHAR type string. The only algorithm that is currently supported in ALTIBASE HDB is SHA-1.

6.8.5.3 Example

<Query > Use the SHA-1 algorithm to obtain the digest for the input string.

```
iSQL> SELECT DIGEST( 'I am a boy.' , 'SHA-1') FROM DUAL;
DIGEST( 'I am a boy.' , 'SHA-1')
--------------------------------------------
A817613E0B781BBF01816F36A8B0DC7C98B2C0CC
1 row selected.
```

6.8.6 DUMP

6.8.6.1 Syntax

```
DUMP (expr)
```

6.8.6.2 Description

DUMP analyzes *expr* and outputs information about its type, length, and contents.

6.8.6.3 Example

<Query> Display information about the employee number and name of three employees.

```
iSQL> SELECT DUMP(eno) Emp_Number, DUMP(e_lastname) Last_Name,
DUMP(e_firstname) First_Name FROM employees LIMIT 3;
EMP_NUMBER
-----------------------------------------------------------------------
LAST_NAME
-----------------------------------------------------------------------------
-------------------------------------------------------------------
FIRST_NAME
-----------------------------------------------------------------------------
-------------------------------------------------------------------
Type=INTEGER(ASCII) Length=4: 1,0,0,0
Type=CHAR(ASCII) Length=22:
20,0,77,111,111,110,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32
Type=CHAR(ASCII) Length=22:
20,0,67,104,97,110,45,115,101,117,110,103,32,32,32,32,32,32,32,32,32,32
Type=INTEGER(ASCII) Length=4: 2,0,0,0
Type=CHAR(ASCII) Length=22:
```

```
20,0,68,97,118,101,110,112,111,114,116,32,32,32,32,32,32,32,32,32,32,32
Type=CHAR(ASCII) Length=22:
20,0,83,117,115,97,110,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32
Type=INTEGER(ASCII) Length=4: 3,0,0,0
Type=CHAR(ASCII) Length=22:
20,0,75,111,98,97,105,110,32,32,32,32,32,32,32,32,32,32,32,32,32,32
Type=CHAR(ASCII) Length=22:
20,0,75,101,110,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32
3 rows selected.
```

6.8.7 GREATEST

6.8.7.1 Syntax

```
GREATEST (expr1 [, expr2, epr3...])
```

6.8.7.2 Description

GREATEST returns the input expression having the greatest value, that is, the one that would be last if the expressions were sorted in alphabetical order. The return type is VARCHAR.

6.8.7.3 Example

<Query> Return the expression that would be last if the input expressions were sorted in alphabetical order.

```
iSQL> SELECT GREATEST('HARRY', 'HARRIOT', 'HAROLD') Greatest FROM dual;
GREATEST
------------
HARRY
1 row selected.
```

6.8.8 LEAST

6.8.8.1 Syntax

```
LEAST (expr1 [,expr2, epr3...])
```

6.8.8.2 Description

LEAST returns the input expression having the lowest value, that is, the one that would come first if the expressions were sorted in alphabetical order. The return type is VARCHAR.

6.8.8.3 Example

<Query> Return the expression that would be first if the input expressions were sorted in alphabetical order.

```
iSQL> SELECT LEAST('HARRY','HARRIOT','HAROLD') Least FROM dual;
LEAST
```

```
-----------
HAROLD
1 row selected.
```

6.8.9 ROWNUM

6.8.9.1 Syntax

```
ROWNUM
```

6.8.9.2 Description

ROWNUM returns a pseudo record number (pseudo rownum) as a BIGINT. The range of possible return values is between 1 and the maximum possible value of the BIGINT numeric data type.

Record numbers are assigned in the order in which records appear in a table or view. However, they can be reordered using an ORDER BY, GROUP BY or HAVING clause.

ROWNUM can be used in SELECT statements, but cannot be used in UPDATE or DELETE statements.[1]

6.8.9.3 Example

<Query> Retrieve the first three employee records, sort them in order of employee name, and output the employee number, name, and phone number.

```
iSQL> SELECT eno, e_lastname, e_firstname, emp_tel FROM employees
 WHERE ROWNUM < 4
 ORDER BY e_lastname;
ENO         E_LASTNAME            E_FIRSTNAME           EMP_TEL
---------------------------------------------------------------------------
2           Davenport             Susan                 0113654540
3           Kobain                Ken                   0162581369
1           Moon                  Chan-seung            01195662365
3 rows selected.
```

6.8.10 NVL

6.8.10.1 Syntax

```
NVL (expr1, expr2)
```

6.8.10.2 Description

NVL replaces a NULL value with a number, date or string in the results of a query.

The data types that can be used with NVL are DATE, CHAR, and NUMBER. *expr2* must be the same data type as *expr1*.

1. When executing UPDATE or DELETE statements, the LIMIT clause can be used to achieve the same effect as ROWNUM, e.g. `iSQL> DELETE FROM employees LIMIT 1, 10;`

6.8.10.3 Example

<Query> Retrieve the name and the wage of all employees. Display the word 'Unknown' for employees without any wage data.

```
iSQL> SELECT e_firstname, e_lastname, NVL(TO_CHAR(salary), 'Unknown')
 FROM employees;
E_FIRSTNAME           E_LASTNAME            NVL(TO_CHAR(SALARY), 'Unknown')
----------------------------------------------------------------------------
Chan-seung            Moon                  Unknown
Susan                 Davenport             1500
Ken                   Kobain                2000
.
.
.
20 rows selected.
```

6.8.11 NVL2

6.8.11.1 Syntax

```
NVL2 (expr1, expr2, expr3)
```

6.8.11.2 Description

If *expr1* is not NULL, NVL2 returns expr2, but if NULL, it returns *expr3*.

6.8.11.3 Example

<Query> Display the name and the wage of all employees. For employees having wage data, display a value equal to 110% of the actual wage. For those without wage data, simply display 'Unknown'.

```
iSQL> SELECT e_firstname, e_lastname, salary, NVL2(TO_CHAR(salary),
TO_CHAR(salary * 1.1), 'Unknown') Nvl2_salary
 FROM employees;
E_FIRSTNAME           E_LASTNAME            SALARY      NVL2_SALARY
--------------------------------------------------------------------------
Chan-seung            Moon                              Unknown
Susan                 Davenport             1500        1650
Ken                   Kobain                2000        2200
Aaron                 Foster                1800        1980
.
.
.
20 rows selected.
```

6.8.12 SENDMSG

6.8.12.1 Syntax

```
SENDMSG ( VARCHAR ipaddr,
          INTEGER port,
          VARCHAR msg,
            INTEGER ttl )
```

6.8.12.2 Description

SENDMSG sends a user message to the specified IP address and port as a socket datagram. If a regular IP address is entered, a UDP datagram is sent. If a multicast IP address is entered, a multicast datagram is sent.

Multicast IP addresses are limited to the range from 225.0.0.0 - 238.0.0.255; that is, reserved multicast groups cannot be used.

The usable port range is from 1025 to 65535.

msg cannot exceed 2048 bytes.

ttl stands for time-to-live. This argument is useful when sending a message to a multicast IP address. It limits the transfer range during multicast transfer as shown below. The range of possible ttl values is from 0 to 255.

TTL	Range
0	Limited to the inside of the host, cannot be output past the network interface
1	Limited to the same subnet, not forwarded beyond router
< 32	Limited to the same site, organization or department
< 64	Limited to the same region
< 128	Limited to the same continent
< 255	Unlimited, worldwide

The return value is an INTEGER indicating the length of the message that was sent.

6.8.12.3 Examples

<Query> Send a user message to a regular IP address (in this case, the value of *ttl* is ignored).

```
iSQL> SELECT SENDMSG( '192.168.1.60', 12345, 'THIS IS A MESSAGE', 1 ) FROM
T1;
SENDMSG( '192.168.1.60', 12345, 'THIS IS
-------------------------------------------
17
1 row selected.
```

<Query> Send a user message to a multicast IP address (in this case, the value of *ttl* is used).

```
iSQL> SELECT SENDMSG( '226.0.0.37', 12345, 'THIS IS A MESSAGE', 0 ) FROM T1;
SENDMSG( '192.168.1.60', 12345, 'THIS IS
-------------------------------------------
17
1 row selected.
```

6.8.13 USER_ID

6.8.13.1 Syntax

```
USER_ID()
```

6.8.13.2 Description

USER_ID returns the ID of the connected user. The return type is INTEGER.

6.8.13.3 Example

<Query> View the information about all tables owned by the current user.

```
iSQL> SELECT table_name FROM system_.sys_tables_ WHERE user_id = USER_ID();
TABLE_NAME
-----------------------------------------
.
.
.
```

6.8.14 USER_NAME

6.8.14.1 Syntax

```
USER_NAME()
```

6.8.14.2 Description

USER_NAME returns the name of the connected user. The return type is VARCHAR.

6.8.14.3 Example

<Query> View the name of the current user.

```
 iSQL> SELECT user_name(), user_id() FROM dual;
 USER_NAME     USER_ID
 --------------------------------------
 SYS           2
 1 row selected.
```

6.8.15 SESSION_ID

6.8.15.1 Syntax

```
SESSION_ID()
```

6.8.15.2 Description

SESSION_ID returns the identifier of the session with which the current user is connected. The return type is INTEGER.

6.8.15.3 Example

<Query> Retrieve the character set in use on the currently connected client.

```
iSQL> SELECT client_nls FROM v$session WHERE id = SESSION_ID();
CLIENT_NLS
--------------------------------------------
US7ASCII
1 row selected.
```

6.8.16 Nested Functions

6.8.16.1 Description

Single-row functions can be nested several layers deep. Nested functions are evaluated starting with the innermost one and working outwards.

6.8.16.2 Example

<Query> display the date of the first Monday six months after each employee was hired by the company.

```
iSQL> SELECT TO_CHAR(NEXT_DAY(ADD_MONTHS(join_date, 6), 'MONDAY'), 'DD-Mon-
YYYY') Monday_six_months
 FROM employees
 ORDER BY join_date;
MONDAY_SIX_MONTHS
--------------------------------------
26-Jul-2004
21-May-2007
05-May-2008
24-May-2010
.
.
.
20 rows selected.
```

6.9 Encryption Functions

ALTIBASE HDB provides functions for encrypting strings and decrypting encrypted strings. The encryption algorithm that is implemented in ALTIBASE HDB is the Data Encryption Standard (DES).

The 8-byte block encryption algorithm that is implemented in ALTIBASE HDB is Cipher Block Chaining (CBC).

6.9.1 DESENCRYPT

6.9.1.1 Syntax

```
DESENCRYPT (expr, key_string)
```

6.9.1.2 Description

expr: This is the string to encrypt. The length of the string must be a multiple of 8.

key_string: This is the string that will be used as the encryption key. The minimum length of *key_string* is 8 characters. The 9th and subsequent characters are ignored.

6.9.1.3 Example

Please refer to the DESDECRYPT example below.

6.9.2 DESDECRYPT

6.9.2.1 Syntax

```
DESDECRYPT (encrypted_string, key_string)
```

6.9.2.2 Description

encrypted_string: This is the string to decrypt. The length of the string must be a multiple of 8.

key_string: This is the string that was used as the encryption key when encrypting *encrypted_string*. The minimum length of *key_string* is 8 characters. The 9th and subsequent characters are ignored.

Caution: Outputting an encrypted string to the screen may cause a terminal emulator error in some environments.

6.9.2.3 Example1

Save encrypted text in a table, and then retrieve, decrypt and output the text.

```
iSQL> create table t1( encrypted_string varchar(40) );
Create success.
```

1. Encrypt the text to be inserted and save it in the table.

```
iSQL> insert into t1 values( desencrypt( 'A4 ALTIBASE Corporation.', 
'altibase' ) );
1 row inserted.
```

2. The encrypted text that is saved in the table will be unreadable when retrieved.

```
iSQL> select * from t1;
T1.ENCRYPTED_STRING
--------------------------------------------
Z\uf900\u5b87\ub94c]\uffff\uffffu\uffffxE\uffffIXek\uffff
1 row selected.
```

3. Decrypt the text using the same encryption key, and then output it.

```
iSQL> select desdecrypt(encrypted_string, 'altibase') from t1;
DESDECRYPT(ENCRYPTED_STRING, ‘altibase’)
--------------------------------------------
A4 ALTIBASE Corporation.
1 row selected.
```

6.9.2.4 Example2

Attempt encryption and decryption without satisfying the requirements that the string to be encrypted must be a multiple of 8 characters and that the key must be at least 8 characters long.

```
iSQL> create table t1( encrypted_string varchar(40) );
Create success.
```

1. The length of the source text is not a multiple of 8 characters, therefore an error will occur.

```
iSQL> insert into t1 values( desencrypt( 'Altibase Client Query util-
ity.', 'altibase' ) );
[ERR-2100D : Invalid data type length]
```

2. The length of the source text is a multiple of 8 characters but the key length is shorter than 8 characters. Therefore, an error will occur.

```
iSQL> insert into t1 values( desencrypt( ‘Altibase Client Query util-
ity...’, ‘alti4’ ) );
[ERR-2100D : Invalid data type length]
```

3. Create a stored function that uses RPAD to increase the length of the source text so that it is a multiple of 8 characters and the key length so that it is 8 characters, and then encrypts the text.

```
create or replace function my_encrypt( input_string in varchar(100), 
key_string in varchar(40) )
return varchar(100)
as
  encrypted_string varchar(100);
  pieces_of_eight INTEGER;
begin
  pieces_of_eight := ((FLOOR(LENGTH(input_string)/8 + 0.9)) * 8);

  encrypted_string := desencrypt( RPAD( input_string, pieces_of_eight),
                                  RPAD( key_string, 8, '#' ) );
  return encrypted_string;
end;
/
```

4. Create a stored function that uses RPAD to increase the length of the key to 8 characters, decrypts the text, and trims the decrypted text.

```
create or replace function my_decrypt( input_string in varchar(100),
key_string in varchar(40) )
return varchar(100)
as
  decrypted_string varchar(100);
begin
  decrypted_string := desdecrypt( input_string,
                               RPAD( key_string, 8, '#') );

  return trim(decrypted_string);
end;
/
```

5. The user-defined stored function *my_encrypt()* can now be used to encrypt a string that is not a multiple of 8 characters long and save the encrypted data.

```
iSQL> insert into t1 values( my_encrypt( 'Altibase Client Query util-
ity.', 'altibase' ) );
1 row inserted.
```

6. The user-defined stored function *my_decrypt()* is used to retrieve and decrypt a string. The original string was not a multiple of 8 characters long.

```
iSQL> select my_decrypt( encrypted_string, 'altibase' ) from t1;
MY_DECRYPT( ENCRYPTED_STRING, 'altibase'
------------------------------------------------------------------------
-------------------
Altibase Client Query utility.
1 row selected.

iSQL> delete from t1;
1 row deleted.
```

7. The stored function *my_encrypt()* can also be used to encrypt and save a string using a key that is less than 8 characters long.

```
iSQL> insert into t1 values( my_encrypt( 'Altibase Client Query util-
ity...', 'alti4' ) );
1 row inserted.
```

8. The stored function *my_decrypt()* is used to retrieve and decrypt a string. The key that was originally used to encrypt the string is also used to decrypt it. It is OK even when this key is less than 8 characters long.

```
iSQL> select my_decrypt( encrypted_string, 'alti4' ) from t1;
MY_DECRYPT( ENCRYPTED_STRING, 'alti4' )
------------------------------------------------
Altibase Client Query utility...
1 row selected.
```

7 Arithmetic Operators

7.1 Arithmetic Operators

Arithmetic operators are used to sign, add, subtract, multiply, and divide numeric values. Some of these operators can also be used to perform arithmetic on date values. Arithmetic operations can only be performed on numeric data types or on data types that can be implicitly converted to numeric data types.

7.1.1 The Type of Arithmetic Operators

The following is a brief description of the kinds of arithmetic operators that are supported in ALTIBASE HDB.

Operator	Description
"+" Unary operator	Explicitly specifies a positive number
"-" Unary operator	Reverses the sign of the input number
Four basic arithmetic operators	Perform respective operations on two input numbers (arguments) to yield a result
Concatenation operator	Joins two character strings

7.2 Unary Operators

7.2.1 Positive Sign

7.2.1.1 Syntax

```
+ n
```

7.2.1.2 Description

This operator is used to explicitly indicate that *n* is a positive number.

7.2.2 Negative Sign

7.2.2.1 Syntax

```
- n
```

7.2.2.2 Description

This operator is used to change the sign of *n*.

7.3 Binary Operators

7.3.1 Addition

7.3.1.1 Syntax

```
n1 + n2
```

7.3.1.2 Description

This operator adds *n1* and *n2* and outputs the result.

7.3.2 Subtraction

7.3.2.1 Syntax

```
n1 - n2
```

7.3.2.2 Description

This operator subtracts *n2* from *n1* and outputs the result.

7.3.3 Multiplication

7.3.3.1 Syntax

```
n1 * n2
```

7.3.3.2 Description

This operator multiplies *n1* by *n2* and outputs the result.

7.3.4 Division

7.3.4.1 Syntax

```
n1 / n2
```

7.3.4.2 Description

This operator divides *n1* by *n2* and outputs the result.

7.3.5 Performing Arithmetic on DATE Type Values

When a number is added to or subtracted from a DATE type value, it is interpreted in units of days. Therefore, to add or subtract hours, minutes, or seconds from a DATE type value, the number of hours, minutes or seconds to be added or subtracted must be converted to days, as shown below:

```
date [ + | - ] n
date – date
date [ + | - ] days (plus/minus n days: n)
date [ + | - ] hours (plus/minus n hours: n/24) )
date [ + | - ] minutes (plus/minus n minutes: n/(24*60) )
date [ + | - ] seconds (plus/minus n seconds: n/(24*60*60))
```

It is not possible, or meaningful, to perform multiplication or division on DATE type values.

7.3.5.1 Examples

<Query> Display the name and the number of weeks worked for every employee in the department *4001*:

```
iSQL> SELECT e_firstname, e_lastname, (SYSDATE-join_date)/7 Weeks_worked
 FROM employees
 WHERE dno = 4001;
E_FIRSTNAME           E_LASTNAME            WEEKS_WORKED
------------------------------------------------------------------------
Xiong                 Wang                  115.778199044248
Curtis                Diaz                  87.6353419013905
John                  Huxley                224.492484758533
3 rows selected.
```

<Query> Display the time 10 minutes in the future, that is, 10 minutes from the current time:

```
iSQL> SELECT SYSDATE + (10/(24*60)) '10 MINUTES LATER' FROM dual;
10 MINUTES LATER
-----------------------
2005/01/20 09:59:34
1 row selected.
```

7.4 Concatenation Operator

7.4.1 Syntax

```
char1 || char2
```

7.4.2 Description

The concatenation operator is used to join two strings together.

7.4.3 Examples

<Query> Put the text ' is a ' between the employee name and the position to create a sentence, and display the sentence in a single column.

```
iSQL> SELECT RTRIM(e_firstname) || ' ' || RTRIM(e_lastname) || ' is a ' ||
emp_job || '.' Job_Description FROM employees;
JOB_DESCRIPTION
-------------------------------------------------------------------
.
.
.
Aaron Foster is a PL.
Farhad Ghorbani is a PL.
Ryu Momoi is a programmer.
Gottlieb Fleischer is a manager.
.
.
.
20 rows selected.
```

7.5 CAST Operator

7.5.1 Syntax

```
CAST (expr AS data_type)
```

7.5.2 Description

CAST converts *expr* to a value of the specified data_type. (All data types except BLOB and CLOB are supported.)

7.5.3 Example

<Query> Convert a string to a DOUBLE value.

```
iSQL> SELECT CAST('3.14159265359' AS DOUBLE) PI FROM dual;
PI
-------------------------
3.14159265359
1 row selected.
```

8 SQL Conditions

8.1 SQL Conditions Overview

An SQL condition comprises one or several logical operators and expressions. The return value of a condition is one of the three possible logical outcomes: TRUE, FALSE, or UNKNOWN.

Conditions can be used in any of these clauses of a SELECT statement:

- WHERE
- START WITH
- CONNECT BY
- HAVING

Additionally, conditions can be used in the WHERE clause of DELETE and UPDATE statements.

The sections that follow describe the various kinds of conditions in detail.

8.1.1 Logical Conditions

The following logical conditions are supported for use with ALTIBASE HDB. Each of them is described briefly below.

Local operators	Description
AND	Returns TRUE if both of the constituent conditions are TRUE. Returns FALSE if either or both of the constituent conditions are FALSE.
NOT	Returns the opposite of the condition to which it applies. That is, it returns TRUE if the condition is FALSE, and returns FALSE if the condition is TRUE.
OR	Returns TRUE if either or both of the constituent conditions are TRUE. Returns FALSE if both of the constituent conditions are FALSE.

8.1.2 Comparison Conditions

Comparison conditions can be categorized as either simple comparisons or group comparisons.

Simple comparison conditions are those in which one expression is compared with one expression.

Group comparison conditions are those in which one expression is compared with many expressions, or with multiple rows returned by a subquery.

8.1.3 Other Conditions

The other conditions that are supported for use with ALTIBASE HDB are listed and described briefly below.

Condition Type	Description
BETWEEN condition	This is a kind of comparison condition that is used to determine whether a value is within a given range.
EXISTS condition	An EXISTS condition is used to check whether a subquery returns at least one row.
IN condition	An IN condition is used to determine whether a value is the same as one or more in a list of values or results returned by a subquery. A NOT IN condition is used to determine whether a value is unlike all members in a list of values or results returned by a subquery.
IS NULL condition	A NULL condition is used to determine whether a value is a NULL value.
LIKE condition	LIKE is a pattern-matching condition that is used to determine whether a string contains a given sequence of characters ("pattern").
UNIQUE condition	A UNIQUE condition is used to check whether a subquery returns exactly one row.

8.2 Logical Conditions

8.2.1 AND

8.2.1.1 Syntax

```
condition1 AND condition2
```

8.2.1.2 Description

AND performs a logical AND evaluation on *condition1* and *condition2* and returns TRUE if both conditions are TRUE. It returns FALSE if either of them is FALSE. AND cannot return TRUE if either condition is UNKNOWN.

This is the AND Truth Table:

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

8.2.1.3 Example

<Query> Display the names, wages, and hiring dates of all employees who are engineers and earn two thousand dollars or more.

```
iSQL> SELECT e_firstname, e_lastname, salary, join_date
 FROM employees
 WHERE emp_job = 'engineer'
 AND salary >= 2000;
E_FIRSTNAME           E_LASTNAME            SALARY      JOIN_DATE
-----------------------------------------------------------------------
Ken                   Kobain                2000        11-JAN-2010
1 row selected.
```

8.2.2 NOT

8.2.2.1 Syntax

```
NOT condition
```

8.2.2.2 Description

NOT returns the opposite of the condition to which it applies; that is, it returns TRUE if the condition is FALSE and FALSE if it is TRUE. If the result of the condition is UNKNOWN, then NOT (condition) also returns UNKNOWN.

This is the NOT Truth Table:

Condition	TRUE	FALSE	UNKNOWN
NOT Result	`FALSE`	`TRUE`	`UNKNOWN`

8.2.2.3 Example

<Query> Display the names, departments, and birthdays of all employees except those born before 1980.

```
iSQL> SELECT e_lastname, e_firstname, dno, birth
 FROM employees
 WHERE NOT birth < BYTE'800101';
E_LASTNAME            E_FIRSTNAME           DNO         BIRTH
---------------------------------------------------------------------
Foster                Aaron                 3001        820730
Fleischer             Gottlieb              4002        840417
Wang                  Xiong                 4001        810726
Hammond               Sandra                4002        810211
Jones                 Mitch                 1002        801102
Davenport             Jason                 1003        901212
6 rows selected.
```

8.2.3 OR

8.2.3.1 Syntax

```
condition1 OR condition2
```

8.2.3.2 Description

OR performs a logical OR evaluation on *condition1* and *condition2* and returns TRUE if either or both conditions are TRUE. It returns FALSE only if both of them are FALSE. OR cannot return FALSE if either condition is UNKNOWN.

This is the OR Truth Table:

OR	TRUE	FALSE	UNKNOWN
TRUE	`TRUE`	`TRUE`	`TRUE`
FALSE	`TRUE`	`FALSE`	`UNKNOWN`
UNKNOWN	`TRUE`	`UNKNOWN`	`UNKNOWN`

8.2.3.3 Examples

<Query> Display the data for all inventory items numbering more than 20000 in stock or having a unit price of 100000 KRW or higher.

```
iSQL> SELECT *
FROM goods
WHERE stock > 20000 OR price >= 100000;
GNO         GNAME                GOODS_LOCATION  STOCK       PRICE
----------------------------------------------------------------------------
----
C111100001  IT-U950              FA0001     35000       7820.55
D111100008  TM-U200              AC0006     61000       10000
E111100004  M-190G               CE0001     88000       5638.76
E111100012  M-U420               CE0003     43200       3566.78
F111100001  AU-100               AC0010     10000       100000
5 rows selected.
```

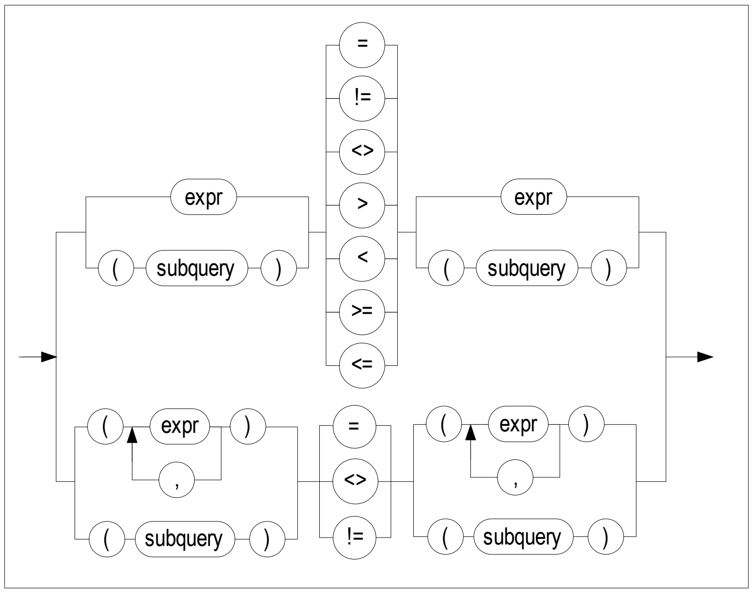
8.3 Comparison Conditions

Comparison conditions can be broadly classified into simple comparison conditions and group comparison conditions on the basis of whether a single expression is compared with one expression or with many expressions.

8.3.1 Simple Comparison Conditions

8.3.1.1 Syntax

simple_comparison_condition ::=



8.3.1.2 Description

Simple comparison conditions compare the expressions on the left and right on the basis of the specified operator and return TRUE, FALSE or UNKNOWN.

Simple comparison conditions can be classified into those that compare the size of the two expressions and those that simply determine whether the two expressions are equivalent.

When there are two or more expressions on each side of the operator (the lower path in the above diagram), only equality comparisons can be conducted. That is, size comparisons are not possible.

Additionally, the number of expressions on the left must be the same as the number of expressions on the right. This rule also applies when the expressions take the form of a subquery SELECT list.

Furthermore, when a subquery is used in a simple comparison, it must return only a single record.

8.3.1.3 Example

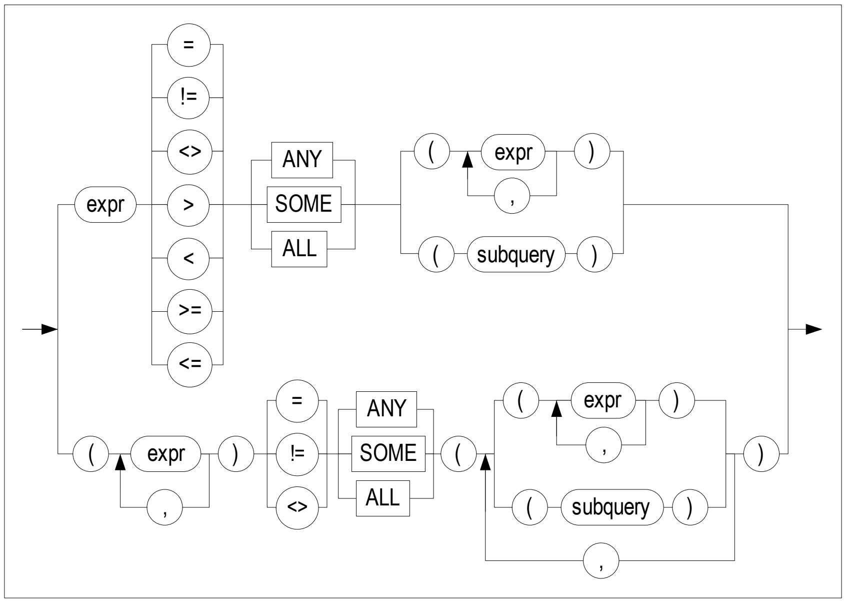
<Query> Display the name, quantity, unit price, and value of inventory for all products for which the value of inventory is more than 100 million KRW. (The value of inventory is the quantity multiplied by the unit price.)

```
iSQL> SELECT gname, stock, price, stock*price value_of_inv
FROM goods
WHERE stock*price > 100000000;
GNAME       STOCK     PRICE       VALUE_OF_INV
------------------------------------------------
IT-U950     35000     7820.55     273719250
TM-T88      10000     72000       720000000
TM-U950     8000      96200       769600000
.
.
.
11 rows selected.
```

8.3.2 Group Comparison Conditions

8.3.2.1 Syntax

group_comparison_condition ::=



8.3.2.2 Description

In a group comparison condition, the expression on the left is compared with each of the expressions or subquery results on the right. It is acceptable, and typical, for the subquery to return multiple rows.

When there are two or more expressions on the left side of the operator, only equality comparisons can be conducted, but size comparisons cannot. Additionally, the expressions on the right must be nested using parentheses to form groups. The number of elements in each group, or alternatively the number of columns returned by a subquery, must be the same as the number of expressions on the left.

- ANY / SOME

  The ANY and SOME keywords have the same meaning. Group comparison conditions containing either keyword return TRUE if the comparison of the expression on the left with at least one of the expressions or subquery results on the right returns TRUE.

- ALL

  Group comparison conditions containing the ALL keyword return TRUE only when the expression on the left is compared with all of the expressions or subquery results on the right and TRUE is returned in every case.

8.3.2.3 Example

<Query> Display information about all orders taken by any employee whose last name starts with "B".

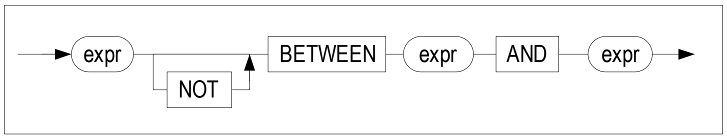
```
iSQL> SELECT ono, order_date, processing
 FROM orders
 WHERE eno = ANY
(SELECT eno FROM employees WHERE e_lastname LIKE 'B%');
ONO                  ORDER_DATE   PROCESSING
-------------------------------------------------
12300003             29-DEC-2011  P
12300004             30-DEC-2011  P
12300006             30-DEC-2011  P
12300008             30-DEC-2011  P
12300009             30-DEC-2011  P
12300011             30-DEC-2011  P
12300013             30-DEC-2011  P
12310001             31-DEC-2011  O
12310003             31-DEC-2011  O
12310005             31-DEC-2011  O
12310006             31-DEC-2011  O
12310010             31-DEC-2011  O
12 rows selected.
```

8.4 Other Conditions

8.4.1 BETWEEN

8.4.1.1 Syntax

between_condition ::=



8.4.1.2 Description

BETWEEN comparisons are used to check whether a value falls within a given range.

‘*column1* between x1 and x2’ is logically the same as ‘*column1* >= x1 and *column1* <= x2’.

8.4.1.3 Examples

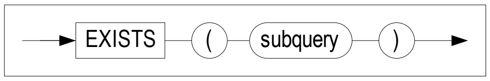
<Query> Display the name, quantity, unit price, and value of inventory for all products for which the value of inventory is between 1 million KRW and 10 million KRW. (The value of inventory is the quantity multiplied by the unit price.)

```
iSQL> SELECT gname, stock, price, stock*price value_of_inv
FROM goods
WHERE stock*price BETWEEN 1000000 AND 10000000;
GNAME       STOCK       PRICE     VALUE_OF_INV
------------------------------------------------
IM-310      100         98000     9800000
IT-U200     1000        9455.21   9455210
M-T245      900         2290.54   2061486
M-180       1000        2300.55   2300550
M-T153      900         2338.62   2104758
M-T102      7890        966.99    7629551.1
M-T500      5000        1000.54   5002700
7 rows selected.
```

8.4.2 EXISTS

8.4.2.1 Syntax

exists_condition ::=



8.4.2.2 Description

An EXISTS condition is used to check whether a subquery returns any rows. If at least one row is returned, the EXISTS condition returns TRUE.

8.4.2.3 Examples

<Query> Output the customer numbers of customers who ordered at least two kinds of products. (The *orders* table is first queried to find pairs of rows having the same customer number but different product numbers, which indicate customers who ordered more than one kind of product. If such a pair of rows exists, the EXISTS condition returns TRUE, so the customer number is output.

```
iSQL> SELECT DISTINCT cno
FROM orders a
WHERE EXISTS
 (SELECT *
 FROM orders b
 WHERE a.cno = b.cno
 AND NOT(a.gno = b.gno));
CNO
------------------
19
15
14
11
6
5
3
2
1
9 rows selected.
```

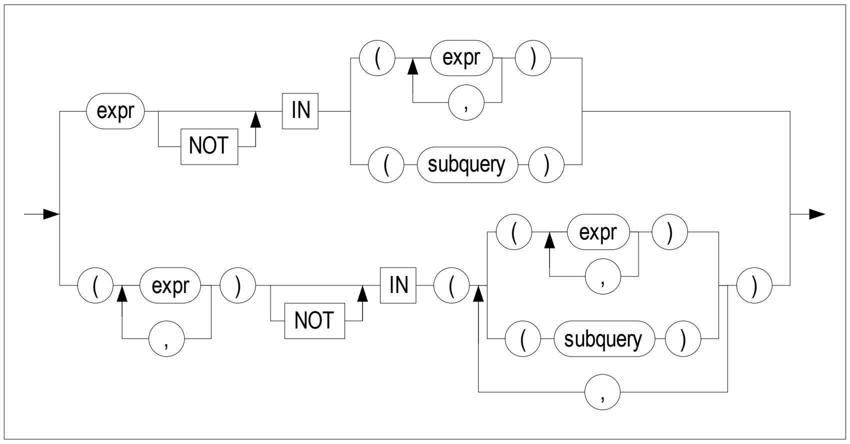
<Query> Retrieve the names of any customers who have ordered all available products. The innermost subquery, which is located at the end of the query string, finds the products ordered by each customer in the *orders* table. The next subquery, located in the middle of the query string, searches for products that have not been ordered by that customer. If there are no products that have not been ordered by that customer, the customer's name will be displayed.

```
iSQL> SELECT customer.cname
FROM customers
WHERE NOT EXISTS
 (SELECT *
 FROM goods
 WHERE NOT EXISTS
 (SELECT *
 FROM orders
 WHERE orders.cno = customers.cno
 AND orders.gno = goods.gno));
CNAME
------------------------
No rows selected.
```

8.4.3 IN

8.4.3.1 Syntax

in_condition ::=



8.4.3.2 Description

The IN condition is the same as a group comparison using the '=ANY' condition. This kind of condition returns TRUE if the expression on the left matches any of the expressions on the right.

The NOT IN condition is the same as a group comparison using the '!=ALL' condition. This kind of condition returns TRUE if none of the expressions on the right match the expression on the left.

8.4.3.3 Examples

<Query> Display the name, position, and telephone number of every employee who is on either the application development team or the marketing team.

```
iSQL> SELECT e_firstname, e_lastname, emp_job, emp_tel
 FROM employees
 WHERE dno IN (1003, 4001);
E_FIRSTNAME           E_LASTNAME            EMP_JOB          EMP_TEL
---------------------------------------------------------------------------
Elizabeth             Bae                   programmer       0167452000
Zhen                  Liu                   webmaster        0114553206
Yuu                   Miura                 PM               0197664120
Jason                 Davenport             webmaster        0119556884
Xiong                 Wang                  manager          0178829663
Curtis                Diaz                  planner          0165293668
John                  Huxley                planner          01755231044
7 rows selected.
```

The WHERE clause in the above SQL statement has the same meaning as:

```
WHERE DNO = 1003 or DNO = 4001.
```

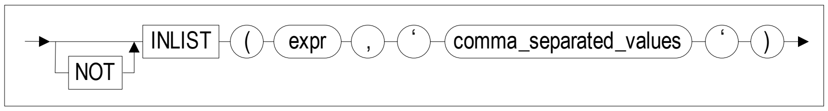
<Query> Retrieve the names of customers who ordered product number C111100001.

```
iSQL> SELECT DISTINCT customers.c_lastname, customers.c_firstname
 FROM customers
 WHERE customers.cno
 IN (SELECT orders.cno FROM orders WHERE orders.gno = 'C111100001');
C_LASTNAME            C_FIRSTNAME
-----------------------------------------------
Martin                Pierre
Fedorov               Fyodor
Dureault              Phil
Sanchez               Estevan
4 rows selected.
```

8.4.4 INLIST

8.4.4.1 Syntax

inlist_condition ::=



8.4.4.2 Description

The INLIST condition returns TRUE if any of the individual values in *comma_separated_values* match expr.

The NOT INLIST condition returns TRUE if none of the individual values in *comma_separated_values* match expr.

Each value in *comma_separated_values* must be a string containing only ASCII characters. The values in *comma_separated_values* are automatically converted to the type of *expr* in order to perform the comparison.

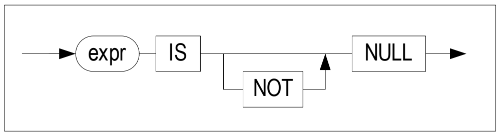
8.4.4.3 Examples

```
iSQL> SELECT dno, e_firstname, e_lastname
 FROM employees
 WHERE INLIST (dno, '1003,4001' );
DNO         E_FIRSTNAME           E_LASTNAME
------------------------------------------------------------
1003        Elizabeth             Bae
1003        Zhen                  Liu
1003        Yuu                   Miura
1003        Jason                 Davenport
4001        Xiong                 Wang
4001        Curtis                Diaz
4001        John                  Huxley
7 rows selected.
```

8.4.5 IS NULL

8.4.5.1 Syntax

null_condition ::=



8.4.5.2 Description

The IS NULL condition is used to check whether or not the expression is NULL.

8.4.5.3 Examples

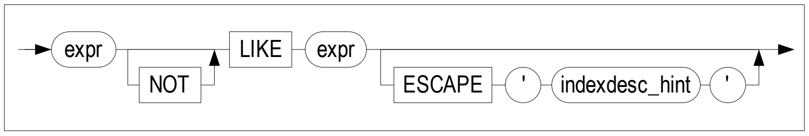
<Query> Display the employee number, name, and position of every employee whose birthday has not been input.

```
iSQL> SELECT eno, e_firstname, e_lastname, emp_job
 FROM employees
 WHERE salary IS NULL;
ENO         E_FIRSTNAME           E_LASTNAME            EMP_JOB
---------------------------------------------------------------------------
1           Chan-seung            Moon                  CEO
8           Xiong                 Wang                  manager
20          William               Blake                 sales rep
3 rows selected.
```

8.4.6 LIKE

8.4.6.1 Syntax

like_condition ::=



8.4.6.2 Description

LIKE is a pattern-matching condition that is used to determine whether a string contains a given sequence of characters ("pattern"). The percent ("%") and underscore ("_") characters are wildcard characters in LIKE conditions. "%" is used to represent a string, while "_" is used to represent a single character.

When it is desired to search for the actual characters “%” or “_”, rather than using them as wildcards, use the ESCAPE keyword at the end of the LIKE condition to define an escape character, and then use the escape character in front of “%” or “_” to indicate that it is not to be handled as a wildcard character.

8.4.6.3 Examples

<Query> Display the employee number, name, department number, and telephone number of every employee whose last name starts with “D”.

```
iSQL> SELECT eno, e_lastname, e_firstname, dno, emp_tel FROM employees WHERE
e_lastname LIKE 'D%';
ENO         E_LASTNAME           E_FIRSTNAME          DNO         EMP_TEL
-----------------------------------------------------------------------------
2           Davenport            Susan                            0113654540
9           Diaz                 Curtis               4001        0165293668
15          Davenport            Jason                1003        0119556884
3 rows selected.
```

<Query> Output the information about all departments that contain the underscore (“_”) character in their name.

```
iSQL> INSERT INTO departments VALUES(5002, 'USA_HQ', 'Palo Alto', 100);
1 row inserted.
iSQL> SELECT * FROM departments
WHERE dname LIKE '%\_%' ESCAPE '\';
DNO         DNAME                          DEP_LOCATION  MGR_NO
----------------------------------------------------------------------------
5002        USA_HQ                         Palo Alto     100
1 row selected.
```

In the above example, the backslash (“\”) is defined as an escape character using the ESCAPE option. Using this escape character before the underscore character indicates that the underscore character is not to be handled as a wildcard.

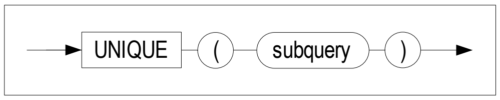
<Query> Display the first names of all employees who have the letter “h” in their first names.

```
iSQL> SELECT e_firstname
 FROM employees
 WHERE e_firstname LIKE '%h%';
E_FIRSTNAME
------------------------
Chan-seung
Farhad
Elizabeth
Zhen
Mitch
Takahiro
John
7 rows selected.
```

8.4.7 UNIQUE

8.4.7.1 Syntax

unique_condition ::=



8.4.7.2 Description

UNIQUE is used to determine whether a subquery returns only a single row.

8.4.7.3 Examples

<Query> If there is only one CEO, output the following message: "There is only one CEO".

```
iSQL> SELECT 'There is only one CEO.' message
 FROM dual
 WHERE UNIQUE
  (SELECT *
  FROM employees
  WHERE emp_job = 'CEO');
MESSAGE
--------------------------
There is only one CEO.
1 row selected.
```

<Query> If there is only one female customer in the *customers* table, output the following message: 'There is only one female customer.'

```
iSQL> SELECT 'There is only one female customer.' message
FROM dual
WHERE UNIQUE
 (SELECT *
 FROM customers
 WHERE SEX = 'F');
ENAME
------------------------
No rows selected.
```

Index