IBM Netezza 7.0.x

IBM Netezza Database User's Guide

Revised: October 5, 2012

20284-15 Rev. 1

Note: Before using this information and the product that it supports, read the information in “Notices and Trademarks” on page E-1.

Contents

Appendix C: Join Overview

Appendix D: nzsql Command Line Options

Appendix E: Notices and Trademarks

Index

Tables

Preface

The IBM® Netezza® data warehouse appliance includes a highly optimized SQL language called IBM® Netezza® Structured Query Language (SQL). You can use the SQL commands to create and manage your IBM Netezza databases, user access and permissions for the databases, as well as to query the contents of the databases.

Audience for This Guide

The IBM Netezza Database User's Guide is written for database administrators, database programmers, and data analysts.

Purpose of This Guide

This guide assists you in understanding how to manage and maintain databases using Netezza SQL, as well as how to construct queries for the data. It describes the commands, their syntax, and how to use them, and provides examples of most commands and their output.

Symbols and Conventions

This guide uses the following typographical conventions:

- Italics for terms, and user-defined variables such as file names
- Upper case for SQL commands; for example INSERT, DELETE
- Bold for command line input; for example, **nzsystem stop**

If You Need Help

If you are having trouble using the Netezza appliance, you should:

1. Retry the action, carefully following the instructions given for that task in the documentation.
2. Go to the IBM Support Portal at: http://www.ibm.com/support. Log in using your IBM ID and password. You can search the Support Portal for solutions. To submit a support request, click the **Service Requests & PMRs** tab.
3. If you have an active service contract maintenance agreement with IBM, you can contact customer support teams via telephone. For individual countries, visit the Technical Support section of the IBM Directory of worldwide contacts (http://www14.software.ibm.com/webapp/set2/sas/f/handbook/contacts.html#phone).

Comments on the Documentation

We welcome any questions, comments, or suggestions that you have for the IBM Netezza documentation. Please send us an e-mail message at netezza-doc@wwpdl.vnet.ibm.com and include the following information:

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

We appreciate your comments on the documentation.

CHAPTER 1

Netezza SQL Introduction

What's in this chapter

- Accessing Netezza SQL Using nzsql
- Using nzsql Commands
- nzsql Exit Codes

Netezza SQL is the Netezza Structured Query Language (SQL), which runs on the Netezza data warehouse appliance. Throughout this document, the term SQL refers to Netezza's SQL implementation. Several standards relate to the definition of Netezza SQL:

Several standards relate to the definition of Netezza SQL. SQL-92 (also called SQL/2), is the operative ANSI/ISO standard for relational databases today. While no vendor supports the complete SQL-92 standard, Netezza SQL conforms to all the commonly supported components of SQL-92. In addition, Netezza includes some SQL:1999 extensions and some SQL:2003 extensions. This document describes the Netezza SQL language support.

If you have direct access to the Netezza appliance from a command shell, or if you have UNIX clients with the Netezza CLI tools, you can run SQL commands using the **nzsql** command line interpreter. You can also run Netezza SQL commands using common SQL tools and applications that support ODBC, JDBC, and OLE DB data connectivity APIs.

The Netezza system can support multiple concurrent connections from clients. Within a connection, Netezza supports only one active SQL activity at a time.

This document uses the nzsql command to show query and command examples.

Accessing Netezza SQL Using nzsql

You can use the **nzsql** command on the Netezza system or from a UNIX client system that can access the Netezza host. The command uses a client/server model, which includes:

- A server that manages database files, accepts connections to the database from client applications, and performs actions on the database on behalf of the client.
- A client application that can perform a variety of database operations. The client could be one of many tools, and is often created by the user.

Logging On

When you invoke the **nzsql** command, you must supply a database account user name, password, and the name of the database to which you are connecting. You can enter this information on the nzsql command line, or you can specify the information in environment variables before you begin your nzsql session. For example, you can enter the following from a command window prompt:

```
nzsql -d sales -u mlee -pw blue
Welcome to nzsql, the Netezza SQL interactive terminal.

Type:  \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit
SALES(MLEE)=>
```

Or, you can set the variables in your command shell using variables such as the following, and then use the nzsql command without any arguments:

```
export NZ_USER=john
export NZ_PASSWORD=red
export NZ_DATABASE=sales
nzsql
Welcome to nzsql, the Netezza SQL interactive terminal.

Type:  \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit
SALES(JOHN)=>
```

Note: Throughout the remainder of this guide, the nzsql command output will be abbreviated to omit the "welcome" text for brevity in the examples.

The Netezza administrator creates and manages the database user accounts using SQL commands or the Netezza NzAdmin and Web Admin administration interfaces. For a complete description of how to manage user accounts, see the *IBM Netezza System Administrator's Guide*.

The Netezza system has a default "admin" database user account who is the superuser of the Netezza databases. The admin user can connect to any database; load data; create, alter and drop any objects; create and manage new database users; and so on. Typically the admin user creates new accounts so that other users can access one or more databases and run queries. The admin user can also create accounts with administrative permissions so that other users can be allowed to perform tasks such as manage databases and user setup, backups, and other administrative tasks.

Session Management

Each client user who connects to the Netezza system opens a session. Users can view information about their sessions, as well as manage them to do such tasks as alter or drop their sessions. The admin account or any permitted user can also show, drop, and manage sessions (that is, change the priority and/or rollback a transaction) for a session. For a description of the SQL commands to manage sessions (ALTER SESSION, DROP SESSION, and SHOW SESSION), refer to Appendix B, "Netezza SQL Command Reference."

SSL Support for Clients

Starting in Release 4.5, the Netezza system supports secure sockets layer (SSL) encryption and authentication for connections to the Netezza system. When you run the **nzsql** command, you can use the following two options to specify the security options for the connection:

- -securityLevel specifies the security level that you want to use for the session. The argument has four values:
  - preferredUnsecured — This is the default value. Specify this option when you would prefer an unsecured connection, but you will accept a secured connection if the Netezza system requires one.
  - preferredSecured — Specify this option when you want a secured connection to the Netezza system, but you will accept an unsecured connection if the Netezza system is configured to use only unsecured connections.
  - onlyUnsecured — Specify this option when you want an unsecured connection to the Netezza system. If the Netezza system requires a secured connection, the connection will be rejected.
  - onlySecured — Specify this option when you want a secured connection to the Netezza system. If the Netezza system accepts only unsecured connections, or if you are attempting to connect to a Netezza system that is running a release prior to 4.5, the connection will be rejected.

  Table 1-1 on page 1-4 describes some best practices for selecting the -securityLevel setting based on the Netezza system release and SSL configuration.
- -caCertFile specifies the pathname of the root certification authority (CA) file. The CA file must be obtained from the Netezza system administrator and installed on the client system. The CA file authenticates the server (the Netezza host) to the client. The default value is NULL, which indicates that no peer authentication will occur.

When you invoke the **nzsql** command, you can specify these arguments on the command line or you can specify the information in environment variables before you begin your nzsql session. The environment variables follow:

- `export NZ_SECURITY_LEVEL=`*`level`*
- `export NZ_CA_CERT_FILE=`*`pathname`*

These SSL security arguments are also used with the nzsql \c switch when a user attempts to connect to a different Netezza database. If you do not specify values for these fields, the Netezza system uses the values specified for the existing connection.

Table 1-1 describes some best practices for the -securityLevel setting when a Release 4.5 client connects to Netezza systems that are running 4.5 or later. Release 4.5 clients can also connect to Netezza hosts running releases prior to 4.5, but those Netezza hosts do not have SSL support.

Table 1-1: Security Settings and Netezza Host Configurations

Netezza Host Release	Netezza Security Configuration	Connections Allowed	-securityLevel Settings
Release 4.5 and later	host	Secured and Unsecured	All 4 settings accepted (onlyUnsecured,preferredUnsecured, onlySecured, preferredSecured)
	hostssl	Secured only	onlySecured, preferredSecured; preferredUnsecured will be accepted but result in a secured connection.
	hostnossl	Unsecured Only	onlyUnsecured, preferredUnsecured; preferredSecured will be accepted but result in an unsecured connection.
Releases prior to 4.5	N/A	Unsecured Only	onlyUnsecured, preferredUnsecured; preferredSecured will be accepted but result in an unsecured connection.

For details about SSL communication from the Netezza clients to the Netezza system, refer to the *IBM Netezza ODBC, JDBC and OLE DB Configuration and Installation Guide*. For a description of how to configure the Netezza host for SSL support, refer to the *IBM Netezza System Administrator's Guide*.

Understanding the nzsql Prompt

After you invoke the **nzsql** command, the prompt contains the name of the database and your user name. In the following example, the database is system and the user is admin:

```
SYSTEM(ADMIN)=>
```

By default, the Netezza system uses uppercase letters to display SQL output. The system case can be configured to use lowercase instead, which was the default in earlier Netezza releases.

To connect to another database without exiting the **nzsql** command, use the \c option:

```
\c[onnect] [dbname [user] [password]]
```

For example, the follow command connects to the database named sales as the user mlee with the password blue:

```
SYSTEM(ADMIN)=> \c sales mlee blue
You are now connected to database sales as user mlee.
SALES(MLEE)=>
```

Getting Command Feedback

When you issue a Netezza SQL command, you either succeed or receive an error. In either case, the system provides feedback that you can use in a script.

The system feedback for inserts, updates, and deletes shows you the number of rows acted upon. The feedback for inserts includes an extra zero before the actual number (due to a historical artifact). Sample commands (shown in bold) and the command feedback follows:

```
nzsql
CREATE TABLE test1 (col1 INTEGER, col2 INTEGER, col3 CHARACTER(40));
CREATE TABLE

INSERT INTO test1 VALUES (100, 200, 'This is a test');
INSERT 0 1

INSERT INTO test1 VALUES (101, 201, 'Another test');
INSERT 0 1

UPDATE test1 SET col2 = 999 WHERE col1 < 1000;
UPDATE 2

INSERT INTO test1 SELECT * FROM test1;
INSERT 0 2

delete from test1 where col1 > 0;
DELETE 4

TRUNCATE TABLE test1;
TRUNCATE TABLE

DROP TABLE test1;
DROP TABLE
```

Displaying SQL User Session Variables

You can display the current user-defined session variables using the \set command with no arguments. For example:

```
SALES(MLEE)=> \set
VERSION = 'Netezza SQL Version 1.1'
PROMPT1 = '%/%(%n%)%R%# '
PROMPT2 = '%/%(%n%)%R%# '
PROMPT3 = '>> '
HISTSIZE = '500'
DBNAME = 'SALES'
USER = 'MLEE'
HOST = '127.0.0.1'
PORT = '5480'
ENCODING = 'LATIN9'
NZ_ENCODING = 'UTF8'
LASTOID = '0'
```

Using nzsql Commands

The **nzsql** command provides many command line options.

Using Command Inputs

Using the **nzsql** command, you can specify different input options:

- To run a single query from the command line, enter:

```
NZSQL -c "SELECT * FROM test_table"
```

- To read input from the current source, for example a script, enter:

```
NZSQL <<eof
SELECT * FROM test_table;
eof
```

- To read input (standard in), enter:

```
NZSQL < foo.sql
```

- To execute queries from a file (command line argument), enter:

```
NZSQL -f foo.sql
```

- To execute queries from a file (nzsql option), enter:

```
NZSQL
\i foo.sql
```

Using Command Outputs

Using the **nzsql** command, you can specify different output options:

- To save the resulting count value into a variable for later use, enter:

```
VAR1=`NZSQL -A -t -c "SELECT COUNT(*) FROM test_table"`
```

- To pipe the output to a printer, enter:

```
NZSQL | lpr
```

- To send the output to a file (command line argument), enter:

```
NZSQL -o foo.out
```

- To send the output to a file (nzsql option), enter:

```
NZSQL
\o foo.out
```

Note: The **nzsql** command pipes interactive output (to a screen) through the **more** command so it can be paginated and scrolled for viewing. To change to another command, set the PAGER environment variable. For example, export PAGER=cat.

Using the nzsql Command Line Options

When you type **nzsql -h**, the system displays the command line options that you can use. For a list and descriptions of the all the command line options, see Table D-1 on page D-1.

The following describes some useful command line options:

- -A — Unaligned table output mode

  Normally output is well formatted with white space added so that columns align. If you use the -A option, the **nzsql** command removes extra white space. Use this command when you want to compare results between two systems or when you want to transfer data.

- -c — Run a single query and exit

  This option lets you run a single query. When you combine it with options -A, and -t (print rows only) you can create useful scripts. For example, to find out the number of records in a table, enter:

```
#!/bin/bash
EXPORT CNT=`nzsql -A -t -c "SELECT COUNT (*) FROM $1"`
echo "The number of records in table $1 is $CNT"
```

- -E — Display queries that internal commands generate

  This option lets you see how the SQL is being generated. For example, \l displays the list of databases and when you add -E to the command line, the system shows you the actual SQL used to generate the list.

```
NZSQL -E
\l
********* QUERY *********
SELECT Database, Owner FROM _v_database
*************************
  List of databases
   database    | owner
---------------+-------
 database_one | admin
 database_two | admin
    system     | admin
(3 rows)
```

- -f — Execute queries from a file
- -F — Set the field separator

  The default delimiter between output columns is a pipe (|) symbol. When used with the -A option (unaligned output), you can specify a different delimiter string, such as a space, tab, comma, colon, and so on. Note that you can use the -R <string> to change the record separator. The default is newline.

- -H — HTML Table Output Mode

  You can format the nzsql output to use HTML tags.

- -t — Print rows only

  Normally the **nzsql** command includes column headings and a summary row for all SQL queries. Use the -t option to eliminate the column headings and summary row. Use this option with the -A option to produce data in a transportable format.

- -x — Expand table output

  You can use this option to display the query results vertically instead of in the default table/2D grid.

Using Miscellaneous Command Options

The **nzsql** command has the following command line options that you use when running queries:

- -- — Two dashes denote the beginning of a single-line comment.
- /* — Forward slash and an asterisk denote the beginning of a multiline comment.
- */ — Asterisk and forward slash denote the end of a multiline comment.

- 'literal' — Use single quotes around literals. For example, 'May 10, 2000', 'Netezza'. 'US'. Use a pair of single quotes when you want to embed a single quote. For example, 'Mark''s Test'.
- "label" — Use double quotes around labels. For example, **SELECT lst_name AS "Employee Last Name" FROM emp_table;**
- Identifiers — The system automatically converts identifiers, such as database, table, and column names, to the default system case, which is Upper on new systems. If you want to use mixed case and/or spaces, you must use double quotes around the identifier. For example:

```
CREATE TABLE "Emp Table" (emp_id integer, emp_name char(20));
SELECT emp_id FROM "Emp Table";
```

Using the nzsql Internal Slash Options

When you use the **nzsql** command in interactive mode, there are many options that you can use. These options, known as internal slash options, are called with a backslash (\) . Many of these options are the same as those available on the command line.

The following are some useful internal slash options. For a list and description of all the internal slash options, see Table D-2 on page D-2.

- \d — Describe a table or view.

  Displays the DDL for a specific table.
- \dt and \dv — List tables or views.

  Lists the tables or views in the current database.
- \dSt and \dSv — List system tables or views.

  Lists the Netezza internal tables or views if you are the admin user. If you are another user, you must have the appropriate privileges. Note that internal tables begin with _t_ and internal views begin with _v_.

  Note: Do not modify these tables. Doing so could impact the integrity of your system.
- \du and \dU — List users and users' groups.

  Displays a list of all users or a list of users and the groups in which they are members.
- \dg and \dG — List groups and groups of users.

  Displays a list of all groups or a list of all the groups and their members.
- \dGr — List resource sharing groups.

  Displays a list of the groups with Guaranteed Resource Allocation (GRA) percentages.
- \echo <text> — Write text to standard output.

  Allows you to include descriptive text between SQL statements. This is especially useful when writing scripts, as in the following example:

```
NZSQL <<eof
 \echo Rowcount before the truncate
SELECT COUNT(*) FROM customer;
 \echo Rowcount after the truncate
 TRUNCATE TABLE customer;
 SELECT COUNT(*) FROM customer;
 eof
```

When you run this script, the system displays the messages "Rowcount before (or after) the truncate count" before the two select statements.

- \h [cmd] — Display help on SQL syntax.

  Use this option to display help for SQL syntax for a specific command. The help displays a description of the command and the command syntax. For a list of all the SQL commands and their syntax, see Appendix B, "Netezza SQL Command Reference."

- \l — List all databases.

  Use this option to list all the databases and their owners.

- \![cmd] — Issue shell command.

  Use this option to run a shell command without terminating your nzsql session. You can use this option to issue shell commands between SQL statements, which is especially useful in scripts.

```
NZSQL <<eof
 \!  date
 SELECT COUNT(*) FROM customer;
 \!  date
 eof
```

The example produces the following output:

```
Wed Jun 27 11:23:50 EDT 2007
 count
-------
 12399
(1 row)
Wed Jun 27 11:23:50 EDT 2007
```

You can use the \set command to store an often-used expression or SQL statement in a variable. This variable is visible for the length of your connected session.

a. Set the variable:

```
\set my_sql 'SELECT * FROM sales_tbl WHERE amt > '
```

b. Use the variable in a query:

```
:my_sql 4000;
    city    | state |   region   | quarter | amt
------------+-------+------------+---------+------
 New York   | NY    | Northeast  |       3 | 4300
 New York   | NY    | Northeast  |       4 | 5000
```

Using the Query Buffer

Because the **nzsql** command is line oriented, it is difficult to edit a complex, multiline SQL statement. To make it easier, use the query buffer.

- \e — Edit the current query buffer or file with an external editor.

  When you exit the editor, the system automatically runs your query. Note that the query buffer stores only the last SQL statement. The default editor is vi. To change to another editor, set the EDITOR environment variable. For example, export EDITOR=emacs.

- \p — Show the contents of the query buffer.

- \r — Reset (clear) the query buffer.

- \w <file> — Write the query buffer to a file.

nzsql Exit Codes

When running queries interactively at the **nzsql** prompt or using a script (**nzsql -f**), the command returns one of the following exit codes:

- 0 — All queries were successful.
- 1 — At least one query was successful.
- 2 — All queries failed.

For example, the mytest.sql file contains a query that selects from an invalid table (T11):

```
[user@nzhost nz]$ nzsql -f mytest.sql
nzsql:mytest.sql:1: ERROR:  Relation 'T11' does not exist
[user@nzhost nz]$ echo $?
2
```

When running a query using the **nzsql -c** command, the exit codes are:

- 0 — Query succeeded
- -1 or 255 — SQL failed (issues with syntax, failed authentication, permissions)
- -2 or 254 — Connection failed
- -3 or 253 — User cancellation request
- -4 or 252 — Session was terminated

On many platforms, return codes that are outside the range of 0 to 255 are not supported. On Linux and UNIX platforms, -1 through -4 are converted modulo 256, therefore the values -1 and 255 are equivalent.

```
[user@nzhost nz]$ nzsql -c "select * from t11"
ERROR:  Relation 'T11' does not exist
[user@nzhost nz]$ echo $?
255
```

When using the **nzsql -f** command line option, the **nzsql** command executes all statements in the file. If you want the **nzsql** command to stop and exit when it encounters an error with a query, include '-v ON_ERROR_STOP=1' on the command line. The exit code is 1 if there were successful queries before the failed query, or 2 if the first query failed.

If you create an ON_ERROR_STOP block inside a query file, as in this example:

```
<initial SQL queries>
\set ON_ERROR_STOP ON
     <queries inside block>
\unset ON_ERROR_STOP
<trailing SQL queries>
```

The success or failure of queries in the initial SQL command section or the trailing SQL-command section are ignored when there is an ON_ERROR_STOP block. The success or failure of the commands **inside** the block determine the exit value. The exit codes are 0 (all queries in the block succeeded), 1 (some queries were successful), or 2 (all queries inside the block failed). If you do not include the command to unset the ON_ERROR_STOP block, the "block" ends at the end of the file.

CHAPTER 2

Using the SQL Grammar

What's in this chapter

You can use SQL commands to create database objects, run queries, and manage the database. This chapter describes how to use the commands to perform common tasks. Appendix B provides more detailed syntax and usage for the commands

Table 2-1 lists the SQL commands that are available. While most of the commands are described in this guide, some commands may be documented in other feature-specific guides. The *IBM Netezza User-Defined Functions Developer's Guide* is available only to members of the Netezza Developer Network; contact your Netezza Sales representative for more information.

Table 2-1: Netezza SQL Commands

Command	Description	More Information
ALTER AGGREGATE	Changes a user-defined aggregate (UDA)	See the *IBM Netezza User-Defined Functions Developer's Guide*.
ALTER DATABASE	Changes the database name, owner, or the default character set	See "ALTER DATABASE" on page B-5.
ALTER FUNCTION	Changes a user-defined function (UDF)	See the *IBM Netezza User-Defined Functions Developer's Guide*.
ALTER GROUP	Adds or removes users from a group	See "ALTER GROUP" on page B-6.

Table 2-1: Netezza SQL Commands (continued)

Command	Description	More Information
ALTER HISTORY CONFIGURATION	Modify the configuration of query history logging	See "ALTER HISTORY CONFIGURATION" on page B-10.
ALTER PROCEDURE	Changes a stored procedure	See the *IBM Netezza Stored Procedures Developer's Guide*.
ALTER SEQUENCE	Changes the sequence	See "ALTER SEQUENCE" on page B-15.
ALTER SESSION	Changes the priority of a session	See "ALTER SESSION" on page B-17.
ALTER SYNONYM	Changes the owner or name of a synonym	See "ALTER SYNONYM" on page B-19.
ALTER TABLE	Changes the definition of a table	See "ALTER TABLE" on page B-20.
ALTER USER	Changes a database user account	See "ALTER USER" on page B-24.
ALTER VIEW	Changes the owner or name of a view	See "ALTER VIEW" on page B-28.
BEGIN	Starts a transaction block	See "BEGIN" on page B-30.
CALL	Invokes a stored procedure on a Netezza host	See the *IBM Netezza Stored Procedures Developer's Guide*.
COMMENT	Defines or changes an object's comment	See "COMMENT" on page B-32.
COMMIT	Commits the current transaction	See "COMMIT" on page B-34.
COPY	Copies data between files and tables	See "COPY" on page B-35.
CREATE AGGREGATE	Creates a UDA	See the *IBM Netezza User-Defined Functions Developer's Guide*.
CREATE DATABASE	Creates a new database	See "CREATE DATABASE" on page B-38.
CREATE EXTERNAL TABLE	Creates an external table for metadata	See "CREATE EXTERNAL TABLE" on page B-40.
CREATE FUNCTION	Creates a UDF	See the *IBM Netezza User-Defined Functions Developer's Guide*.
CREATE GROUP	Defines a new user group	See "CREATE GROUP" on page B-40.

Table 2-1: Netezza SQL Commands (continued)

Command	Description	More Information
CREATE HISTORY CONFIGURATION	Create a configuration for query history logging	See "CREATE HISTORY CONFIGURATION" on page B-44.
CREATE MATERIALIZED VIEW	Defines a materialized view	See "CREATE MATERIALIZED VIEW" on page B-49.
CREATE PROCEDURE	Creates a stored procedure	See the *IBM Netezza Stored Procedures Developer's Guide*.
CREATE SEQUENCE	Creates a sequence	See "CREATE SEQUENCE" on page B-51.
CREATE SYNONYM	Creates a synonym	See "CREATE SYNONYM" on page B-54.
CREATE TABLE	Defines a new table	See "CREATE TABLE" on page B-55.
CREATE TABLE AS	Creates a new table based on query results	See "CREATE TABLE AS" on page B-61.
CREATE USER	Defines a new database user account	See "CREATE USER" on page B-65.
CREATE VIEW	Defines a new view	See "CREATE VIEW" on page B-69.
DELETE	Deletes rows of a table	See "DELETE" on page B-71.
DROP AGGREGATE	Removes a UDA	See the *IBM Netezza User-Defined Functions Developer's Guide*.
DROP CONNECTION	Drops a client access connection definition.	See "DROP CONNECTION" on page B-72.
DROP DATABASE	Removes a database	See "DROP DATABASE" on page B-73.
DROP FUNCTION	Removes a UDF	See the *IBM Netezza User-Defined Functions Developer's Guide*.
DROP GROUP	Removes a user group	See "DROP GROUP" on page B-75.
DROP HISTORY CONFIGURATION	Removes a configuration for query history logging	See "DROP HISTORY CONFIGURATION" on page B-76.
DROP PROCEDURE	Removes a stored procedure	See the *IBM Netezza Stored Procedures Developer's Guide*.
DROP SEQUENCE	Removes a sequence	See "DROP SEQUENCE" on page B-77.

Table 2-1: Netezza SQL Commands (continued)

Command	Description	More Information
DROP SYNONYM	Drops a synonym	See "DROP SYNONYM" on page B-80.
DROP TABLE	Removes a table	See "DROP TABLE" on page B-81.
DROP USER	Removes a database user account	See "DROP USER" on page B-82.
DROP VIEW	Removes a view	See "DROP VIEW" on page B-83.
EXECUTE [PROCEDURE]	Invokes a stored procedure on a Netezza host	See the *IBM Netezza Stored Procedures Developer's Guide*.
EXPLAIN	Shows the execution plan of a statement	See "EXPLAIN" on page B-84.
GENERATE EXPRESS STATISTICS	This command is deprecated starting in Release 4.6.	See "GENERATE EXPRESS STATISTICS" on page B-87.
GENERATE STATISTICS	Collects information on a database, table, or individual column	See "GENERATE STATISTICS" on page B-89.
GRANT	Defines access privileges	See "GRANT" on page B-91.
GROOM TABLE	Reclaims and reorganizes tables	See "GROOM TABLE" on page B-93.
INSERT	Creates new rows in a table	See "INSERT" on page B-95.
RESET	Restores the value of a runtime parameter to its default value	See "RESET" on page B-97.
REVOKE	Removes access privileges	See "REVOKE" on page B-98.
ROLLBACK	Aborts the current transaction	See "ROLLBACK" on page B-101.
SELECT	Retrieves rows from a table or view	See "SELECT" on page B-102.
SET	Changes a runtime parameter	See "SET" on page B-110.
SET AUTHENTICATION	Defines the user authentication method for Netezza access.	See "SET AUTHENTICATION" on page B-112.
SET CONNECTION	Defines a client connection record	See "SET CONNECTION" on page B-116.

Table 2-1: Netezza SQL Commands (continued)

Command	Description	More Information
SET HISTORY CONFIGURATION	Specify a configuration for query history logging to take effect after the next Netezza software restart.	See "SET HISTORY CONFIGURATION" on page B-117.
SET SESSION	Sets session characteristics including compatibility	See "SET SESSION" on page B-119.
SET SYSTEM DEFAULT	Sets the system defaults for session timeout, rowset limit, query timeout, and priority	See "SET SYSTEM DEFAULT" on page B-120.
SET TRANSACTION	Sets the isolation level of the current transaction	See "SET TRANSACTION" on page B-122.
SHOW	Shows the value of a runtime parameter	See "SHOW" on page B-123.
SHOW AGGREGATE	Displays information about one or more UDAs	See the *IBM Netezza User-Defined Functions Developer's Guide*.
SHOW AUTHENTICATION	Displays the user authentication method for the Netezza.	See "SHOW AUTHENTICATION" on page B-124.
SHOW CONNECTION	Displays the client connection records for the Netezza.	See "SHOW CONNECTION" on page B-126.
SHOW FUNCTION	Displays information about one or more UDAs	See the *IBM Netezza User-Defined Functions Developer's Guide*.
SHOW HISTORY CONFIGURATION	Display query history configuration settings	See "SHOW HISTORY CONFIGURATION" on page B-127.
SHOW PROCEDURE	Displays information about one or more stored procedures	See the *IBM Netezza Stored Procedures Developer's Guide*.
SHOW SYSTEM DEFAULT	Shows the system defaults	See "SHOW SYSTEM DEFAULT" on page B-132.
TRUNCATE	Empties a table	See "TRUNCATE" on page B-134.
UPDATE	Updates rows of a table	See "UPDATE" on page B-135.

Managing Databases

Using SQL commands, you can create, drop, rename, or change the owner of databases.

Creating a Database

To create a database, use the CREATE DATABASE command.

```
system(admin)=> CREATE DATABASE mydb;
CREATE DATABASE
```

If you do not have the privileges required to create a database, the system displays the following message:

```
ERROR:  CREATE DATABASE: permission denied.
```

Database names can have a maximum length of 128 bytes, otherwise the system displays an error message. Database names must be valid identifier names. For more information, see "Handling SQL Identifiers" on page 2-7.

Dropping a Database

If you are logged in as the admin user or the owner of a database, you can drop the database using the DROP DATABASE command. Dropping the database removes the entries for an existing database and deletes the directory that contains the data. For example:

```
system(admin)=> DROP DATABASE mydb;
DROP DATABASE
```

Renaming a Database

If you are logged in as the admin user or the owner of a database, you can rename the database using the ALTER DATABASE command. The data remains of the same type and size. For example:

```
system(admin)=> ALTER DATABASE mydb RENAME TO newdb;
ALTER DATABASE
```

Note: After you rename a database, recompile any views that are associated with that database. Any materialized views in the database will be converted to regular (non-materialized) views. For more information, refer to "Replacing Views" on page 2-23.

Changing Database Ownership

If you are logged in as the admin user or the owner of a database, you can change the owner using the ALTER DATABASE command. The new owner must be a current user of the system. For example:

```
system(admin)=> ALTER DATABASE mydb OWNER TO jane;
ALTER DATABASE
```

Specifying International Character Sets

You can use Netezza SQL to specify international characters based on the syntax extensions to SQL:1999, which use Unicode and ISO standards. Using these extensions, you can specify Latin and other international characters sets, including Kanji.

For more information, see "Using National Character Sets" on page 6-1.

Understanding Database Maximums

Table 2-2 describes the Netezza SQL maximums on such items as the number of database columns, names, and connections.

Table 2-2: Netezza SQL Maximums

Parameter	Description
Columns	Maximum per table or view: 1600. Maximum per distribution: 4
Names	Maximum length of database and column names: 128 bytes
Characters	Maximum number of characters in a char/varchar field: 64,000
Connections	Maximum connections to the server: 2000. Default: 500
Row size	Maximum row size: 65,535 bytes. Limit also applies to the result set of a query.[a]

a. Within each row, there is a small amount of overhead for special columns and other factors such as padding. For more information, see "Calculating Row Size" on page 3-8.

Handling SQL Identifiers

A SQL identifier is the name of a database object such as a table, column, user, group, user-defined object, and database. Netezza supports the SQL 1999 definition for naming identifiers, and they can be up to 128 bytes in length. There are two types of identifiers — regular and delimited.

Note: Account passwords, the names of files, and other values are not identifiers, and thus may support a reduced set of characters, including only 7-bit ASCII characters. The file names for external tables must be in UTF-8.

A *regular identifier* is not case sensitive; that is, if you create a database named SALES, you can refer to it using any case combination of letters. For example, SALES, sales, SaLeS, and SALEs all match the database named SALES. The ANSI SQL standard specifies that systems should convert all regular SQL identifiers to the corresponding upper-case characters, so the Netezza system converts any regular identifier you specify into uppercase characters when it is saved in the database, and also when the regular identifiers are used in query processing.

Regular identifiers can contain only letters (in any alphabet, not just the Latin alphabet), syllables (as in the Japanese Hiragana syllabary), ideographs, decimal digits, underscores, and dollar sign ($). Regular identifiers must begin with a letter; they *cannot* begin with a digit, underscore, or dollar sign. Regular identifiers also cannot be a SQL reserved word (as described in Appendix A, "SQL Reserved Words and Keywords"). The encoding in the Netezza catalog is in UTF-8; the encoding for any display will depend on the client.

A *delimited identifier* is also a name of a database object, but it is enclosed in double-quotation marks and has special considerations. A delimited identifier is case-sensitive, so a database named "Sales" is not the same database as one named SALES, for example. The Netezza system does not convert delimited identifiers to the default system case, nor does it save the enclosing double-quotation marks in the database.

Note: The system automatically truncates leading and trailing spaces in a delimited identifier. If you use leading and trailing spaces to format output, for example, make sure that

you precede leading spaces and end trailing spaces with a non-space character to preserve the spacing in the identifier.

Within the double quotation marks, a delimited identifier can include the same letters, syllables, ideographs, decimal digits, and underscores as a regular identifier, but it can also include spaces, special characters such as hyphens and percent signs, and SQL reserved keywords. With the exception of underscores, which are not allowed, a delimited identifier can begin with any of these letters, digits, or symbols.

For example, the following query uses delimited identifiers for both column and table names:

```
SELECT "My Field" FROM "My Table" WHERE "My Field" LIKE 'A%';
```

Note that the string literal 'A%' is enclosed in single quotes.

You can change the system default lettercase behavior at system initialization by using the **nzinitsystem -lowercase** command. For more information about the **nzinitsystem** command, see the *IBM Netezza System Administrator's Guide*.

Accessing Other Databases

You can execute queries that reference tables, views, and synonyms in other databases on the same Netezza server. This means that you can use these references in the SELECT statement FROM clauses in queries that can include INSERT, DELETE, UPDATE, CREATE TABLE AS, joins, set operations, aggregations, subselects, view definitions, and so on.

When specifying reference objects, keep in mind the following:

- You must specify reference objects that reside on the same Netezza server.
- You cannot specify reference objects that are under control of third-party applications.
- You cannot specify a cross-reference object in the SELECT portion of a CREATE MATERIALIZED VIEW statement.

Referencing Database Objects

To refer to objects in other databases on the Netezza system, you must use three-level naming, which consists of the following components:

- The database — The database or catalog name.
- The schema — The schema, which is the name of the database owner. Note that Netezza currently supports only one schema per database. If you specify a schema, the system uses the owner name of the database in which you are creating the object, unless you have explicitly configured the system to return an error for an invalid schema. For more information, see "Invalid Schema Name Processing" on page 2-9.
- The object — The name of the object, table, view, or synonym.

Database Object Naming

You specify objects in the FROM clause of SQL statements in the three-level form. You can use the standard form or the shorthand notation.

- The database-name.schema.object-name — The three-level or fully qualified form.

- The database-name..object-name — A convenient way of specifying a fully qualified object name. The system supplies the schema name by internally inserting the current schema name.
- The schema.object-name — The two-level form, which you can use only when referring to the current database.

Note: When using existing scripts that contain two- and three-level names, the schema must be the name of the database owner of the applicable database to be compatible with later releases of the software. By default, the system does not print a warning or error message if that name does not match the name of the database owner. For more information, see "Invalid Schema Name Processing" on page 2-9.

Invalid Schema Name Processing

If you specify a schema that does not match the database owner name, the schema is invalid. The enable_schema_dbo_check variable specifies the actions the system will take when processing a query with an invalid schema:

- 0 (the default) causes the system to ignore the user-specified schema and use the default schema.
- 1 causes the system to ignore the user-specified schema and use the default; the system displays the warning "Schema '*schema_name*' does not exist."
- 2 causes the system to fail the query with the error "Schema '*schema_name*' does not exist."

You can set the enable_schema_dbo_check variable in the postgresql.conf file, or you can change the behavior on a session level using the using the SET enable_schema_dbo_check = *value* SQL command.

Cross-Database Usage Examples

In the following examples, the Netezza system has two databases: DEV and PROD. Both databases have tables named EMP. A client program connected to DEV is able to reference tables in PROD in the FROM clause of SQL queries. This is referred to as a cross-database query.

Note: The following examples use the implicit schema represented by ".." between the database name and the table or view name.

To retrieve all rows from the table EMP in connected-to database DEV, enter:

```
dev(admin)=>SELECT * FROM DEV..EMP;
```

To retrieve all rows from the table EMP in the database PROD, enter:

```
dev(admin)=>SELECT * FROM PROD..EMP;
```

To truncate the contents of DEV.EMP.

```
dev(admin)=>TRUNCATE TABLE EMP;
```

To insert the contents of PROD..EMP into DEV..EMP.

```
dev(admin)=>INSERT INTO EMP SELECT * FROM PROD..EMP;
```

To join tables from both PROD and DEV.

```
dev(admin)=>SELECT COUNT (*) FROM DEV..EMP DE, PROD..EMP PE WHERE
DE.ID = PE.ID;
```

To retrieve rows from PROD.EMP, enter:

```
dev(admin)=>SELECT * FROM PROD..EMP WHERE PROD..EMP.DEPTNO IN (SELECT
DE.DEPTNO FROM DEV..EMP DE WHERE DE.ID < 10);
```

To create a table from PROD.EMP and DEV.EMP, enter:

```
dev(admin)=>CREATE TABLE KEYEMPS AS SELECT * FROM PROD..EMP INTERSECT
SELECT * FROM DEV..EMP;
```

Common Error Messages

Note that you cannot use cross-database INSERT, UPDATE, or DELETE statements. If you attempt to do so, the system displays an error message.

For example, if you attempt to insert data into a table that does not reside in the current database (the database you are logged in to), the system displays an error message:

```
dev(admin)=>INSERT INTO PROD..EMP SELECT * FROM EMP;
Cross Database Access not supported for this type of command.
```

For this type of query, consider changing the query to a cross-database SELECT statement (which is supported) while logged in to the target database. For example:

```
prod(admin)=>INSERT INTO EMP SELECT * FROM DEV..EMP;
```

You cannot use CREATE, ALTER, or DROP commands to change objects in a database outside your current database. If you attempt to do so, the system displays an error message.

For example, if you attempt to create an object in a different database, the system displays an error message:

```
dev(admin)=>CREATE PROD..PAYROLL;
Cross Database Access not supported for this type of command.
```

Qualified Column Names

When a query involves multiple tables, it is sometimes difficult to know which column belongs to which table especially if the tables have the same column names. To help distinguish among column names, SQL allows you to fully qualify column names by specifying the column as: database.schema.table.column.

When referring to column names in cross-database access, the system expects the qualified column name to be in the form of exposed-table-reference.column-name; where the exposed-table-reference is any of the acceptable tables references in the FROM clause.

For example, emp, admin.emp, dev.admin.emp and dev..emp are all equivalent forms for the same table:

```
FROM emp WHERE dev.admin.emp.id = 10;
FROM dev.admin.emp WHERE emp.id = 10;
FROM emp WHERE admin.emp.id = 10;
FROM emp WHERE dev.admin.emp.id = 10;
FROM dev..emp WHERE admin.id = 10;
```

Table and Column Aliases

Aliases are like synonyms in that they are alternate names for tables or columns. Aliases differ from synonyms in that they exist only for the duration of the query.

Aliases can be single letters or words, but when using aliases in cross-database access, the system expects the column name to be in the form of the alias.column-name.

For example, the following are correct notations, where E is the alias:

```
dev(admin)=>FROM emp E WHERE E.id =10
dev(admin)=>FROM admin.emp E WHERE E.id =10
```

Using Synonyms

You can create SQL synonyms as an alternate way of referencing tables or views that reside in the current or other databases on the Netezza system. Synonyms allow you to create easily typed names for long table or view names. They also allow you to create a level of abstraction for the database objects and thereby enable you to swap the underlying objects without affecting the code that references these objects.

Synonyms share the same naming restrictions as tables and views, that is, they must be unique within a database and their names cannot be the same as global objects such as those of databases, users, or groups.

You can use the following synonym commands:

- CREATE SYNONYM — To create a synonym
- DROP SYNONYM — To drop/delete a synonym
- ALTER SYNONYM — To rename or change the owner of a synonym
- GRANT SYNONYM — To grant permission to create, alter, or drop a synonym to a user or group
- REVOKE SYNONYM — To revoke permission to create, alter, or drop a synonym to a user or group

Creating Synonyms

You cannot create synonyms for temporary tables, remote databases, or other synonyms (also called chaining). Because synonyms share the same namespace as tables and views, you cannot create a synonym with the same name as a table or view that already exists in the same database. Conversely, you cannot create a table or view with a name that matches an existing synonym.

Note: A namespace is the structure underlying SQL schemas. The namespace contains all the objects within the database plus all global objects (databases, users, groups, and system objects). There is only one namespace for each database.

All synonyms are public and viewable by all users.

The syntax for the CREATE SYNONYM command is:

```
CREATE SYNONYM synonym_name FOR table_reference;
```

The synonym_name is a name that follows the table and view naming conventions. You can create a synonym for a non-existent table or view. At runtime, the system expands the table_reference to its fully qualified form. If the referenced object does not exist, the system displays an error message.

The table_reference can be one of the following:

- Plain name (table or view name)

- Database qualified name (database_name.schema_name.table or view name)

For example, to create a synonym for EMP in database PROD, enter:

```
dev(admin)=>CREATE SYNONYM pemp FOR prod..emp;
```

For more information, see "CREATE SYNONYM" on page B-54.

Dropping Synonyms

If you no longer need a synonym, you can drop it. Note that you can only drop a synonym in the current database.

The syntax for the DROP SYNONYM command is:

```
DROP SYNONYM synonym_name;
```

If you drop or rename a table or view that has an associated synonym, the synonym becomes an orphan. To re-use the synonym for another object with a different name, drop and re-create it.

For example, to drop the synonym for EMP in database PROD, enter:

```
dev(admin)=>DROP SYNONYM pemp;
```

For more information, see "DROP SYNONYM" on page B-80.

Altering Synonyms

You can use the ALTER SYNONYM command to rename or change the owner of a synonym.

The syntax for the ALTER SYNONYM command is:

```
ALTER SYNONYM synonym_name RENAME TO new_synonym_name;
ALTER SYNONYM synonym_name OWNER TO new_owner;
```

For example, to rename the synonym for PEMP in database PROD, enter:

```
dev(admin)=>ALTER SYNONYM pemp RENAME TO p;
```

For example, to change the owner of the synonym for EMP in database PROD, enter:

```
dev(admin)=>ALTER SYNONYM pemp OWNER TO leslie;
```

For more information, see "ALTER SYNONYM" on page B-19.

Synonym Privileges

Synonyms use the same security scheme as other Netezza database objects. As the admin user, you have all privileges on synonyms and thus do not need to explicitly grant yourself privileges to manage them. The owner of the database has all privileges on all synonyms within the database. The synonym owner has all privileges on the synonym.

Table 2-3 lists the privilege rules for all other users:

Table 2-3: Synonym Privileges

To execute this command:	You need this privilege:
CREATE SYNONYM	Grant the user Create Synonym administration privilege.
ALTER SYNONYM	Grant the user Alter privilege for a synonym or the Synonym object class.

Table 2-3: Synonym Privileges

To execute this command:	You need this privilege:
DROP SYNONYM	Grant the user Drop privilege for a synonym or the Synonym object class.
LIST synonyms	All users have List privilege by default and can see all synonyms.
SELECT synonyms	All users have Select privilege by default to select all synonyms.

You grant privileges to create, alter, and drop synonyms with the GRANT SYNONYM command. For more information, see “GRANT” on page B-91.

The syntax for the GRANT SYNONYM command is:

```
GRANT [CREATE] SYNONYM TO user_or_group;
GRANT ALTER, DROP ON synonym_name TO user_or_group;
```

You revoke privileges to create, alter, and drop synonyms with the REVOKE SYNONYM command. For more information, see “REVOKE” on page B-98.

The syntax for the REVOKE SYNONYM command is:

```
REVOKE [CREATE] SYNONYM FROM user_or_group;
REVOKE ALTER, DROP ON synonym_name FROM user_or_group;
```

Managing Tables

You can create, access, and manipulate tables using SQL commands.

Creating a Table

To create a new table, specify the table name, the column names, and their types. You can enter a table line by line. The system recognizes that the command is not terminated until it encounters the semicolon (;). An example follows.

```
system(admin)=> CREATE TABLE weather (
city            varchar(80),
temp_lo         int,           -- low temperature
temp_hi         int,           -- high temperature
prcp                           --precipitation
real,
date                           --date
);
```

The following rules apply when entering data:

- Whitespace — You can use white space (that is, spaces, tabs, and new lines) within SQL commands.
- Comments — Two dashes ("--") introduce comments. The system ignores whatever follows a comment, up to the end of that line.

- Variable characters — The varchar(80) specifies a data type that can store arbitrary character strings up to 80 characters in length.
- Data types — An int is the normal integer type. A real is a type for storing single precision floating-point numbers. For more information about data types, see "Netezza SQL Basics" on page 3-1.

Note: In addition, Netezza SQL considers the following system attributes reserved words: ctid, oid, xmin, cmin, xmax, cmax, tableoid, rowid, datasliceid, createxid, and deletexid.

Using Constraints

When you create a table you can specify column and/or table constraints. For example, a table_constraint can be:

```
{ PRIMARY KEY ( column_name [, ... ] ) |  FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable (refcolumn ) [ MATCH matchtype ] [ ON DELETE action ] [ ON UPDATE action ] [ [ NOT ] DEFERRABLE ] [ INITIALLY checktime ] } [, ...]
```

Note: The system permits and maintains primary key, default, foreign key, unique, and references. Because Netezza does not support constraint checking and referential integrity, you must ensure your own constraint checking and referential integrity.

If you have permission to create a table, you can specify a constraint. If you have permission to alter a table, you can add or drop a table constraint.

You cannot change constraint names or directly change the owner of the constraint. The owner of the constraint is always the owner of the table. Thus, if you change the owner of the table, the system changes the owner of all associated constraints.

For more information about constraints, see "CREATE TABLE" on page B-55.

Removing a Table

If you are the administrator, the owner of the table, or have Drop privilege, you can drop a table. You might need to drop a table to implement a new design or to free space in your database.

Dropping a table causes its structural definition, data, and constraints to be permanently deleted from the database. The system makes the space that was used to store the table available to other tables.

- To remove a table that you no longer need, use the DROP TABLE command:

```
system(admin)=> DROP TABLE weather;
```

Note: This command, along with TRUNCATE TABLE and ALTER TABLE requires exclusive access to the table before executing, and will wait until it has exclusive access.

Truncating a Table

To empty a table, use the TRUNCATE command. The TRUNCATE command frees up all the disk space allocated to a table, making the space available for reuse.

Note: You cannot execute the TRUNCATE command inside a transaction block (begin/commit pair).

```
system(admin)=> TRUNCATE TABLE weather;
```

Note: This command, along with DROP TABLE and ALTER TABLE requires exclusive access to the table before executing, and will wait until it has exclusive access.

Renaming a Table

You can change the name of a table without changing any of the data. The data remains the same type and size. You must be the owner of the table or have the Alter privilege on tables to change the table's name.

▼ To rename a table, use the ALTER TABLE command:

```
system(admin)=> ALTER TABLE weather RENAME TO forecast;
```

Note: This command, along with DROP TABLE and TRUNCATE TABLE requires exclusive access to the table before executing, and will wait until it has exclusive access.

Changing Table Ownership

If you are the owner of the table or have the Alter privilege on tables, you can change the owner of a table without changing any of its data. The new owner must be a current system user.

▼ To change the owner of a table, enter:

```
system(admin)=> ALTER TABLE weather OWNER TO jane;
```

Inserting Rows Into a Table

To populate a table, use the INSERT command. The INSERT command adds new rows to a table. You can insert using column positions, column names, or by inserting rows from one table to another.

The following example inserts a row into the table weather:

```
system(admin)=> INSERT INTO weather VALUES ('San Francisco', 46, 50,
0.25, '1994-11-27');
```

To insert by using column names, enter:

```
system(admin)=> INSERT INTO weather (city, temp_lo, temp_hi, prcp,
date) VALUES ('San Francisco', 43, 57, 0.0, '1994-11-29');
```

When you use columns names, you can list them in any order and you can omit columns for which you do not have data:

```
system(admin)=> INSERT INTO weather (date, city, temp_hi, temp_lo)
VALUES ('1994-11-29', 'Hayward', 54, 37);
```

Generating Table Statistics

After you initially load a table or any time your table's data has changed, you should run the GENERATE STATISTICS command. This ensures that your system has the most up-to-date information on the table, and thus creates optimal execution plans for any queries on that table.

▼ To update the table's statistics, enter:

```
system(admin)=> GENERATE STATISTICS ON emp;
```

Querying a Table

To query a table, use the SELECT command. The select command is divided into three parts:

- The columns to return
- The tables from which to retrieve data
- Any restrictions

For example, to list all the rows of the table weather, enter:

```
system(admin)=> SELECT * FROM weather;
```

To specify expressions in the target list, enter:

```
system(admin)=> SELECT city, (temp_hi+temp_lo)/2 AS temp_avg, date
FROM weather;
```

In this example, you rename the columns temp_hi and temp_lo as temp_avg by using an AS clause.

You can use Boolean operators (AND, OR, and NOT) in the qualification of a query. For example, to retrieve the weather of San Francisco on rainy days, enter:

```
system(admin)=> SELECT * FROM weather WHERE city = 'San Francisco' AND
prcp > 0.0;
```

To specify that the result of a select is returned in sorted order, enter:

```
system(admin)=> SELECT DISTINCT city FROM weather ORDER BY city;
```

Updating Table Rows

To change the value in a table's existing rows, use the UPDATE command. You can update specific rows or all the rows in a table. You cannot update columns which are used as distribution keys for a table. When you update rows, you specify the table, columns, and an optional search condition (WHERE).

- To update the temperature readings as of November 28, 1994, enter:

```
system(admin)=> UPDATE weather SET temp_hi = temp_hi - 2, temp_lo =
temp_lo - 2 WHERE date > '1994-11-28';
```

Deleting Rows from Tables

To remove rows from a table, use the DELETE command. You can use the DELETE command to remove specific rows or all rows from a table. To delete rows, specify the table, and an optional search condition (WHERE) that specifies which rows to delete.

Unlike the INSERT and UPDATE commands, the DELETE command requires no column names because it removes entire rows unless you use a qualification.

While the DELETE command removes rows from a table, it does not delete the table definition. Even if you remove all the rows from a table, the table itself still exists. If you want to delete a table definition (and all its associated data) use the DROP TABLE command.

- To delete any row containing the city Hayward, enter:

```
system(admin)=> DELETE FROM weather WHERE city = 'Hayward';
```

Changing or Dropping a Column Value

You can use the ALTER TABLE command to change or drop a column's default value, which will affect future inserts into the table.

- To change a column's default value, enter:

```
system(admin)=> ALTER TABLE weather ALTER col1 SET DEFAULT 100;
```

- To drop a column's default value, enter:

```
system(admin)=> ALTER TABLE weather ALTER col1 DROP DEFAULT;
```

Changing the Length of a Varchar Column

You can use the ALTER table command to change the length of a varchar column. You can increase the length of a varchar column to a maximum size of 64,000.

- To change the length of a varchar column, enter:

```
system(admin)=> ALTER TABLE t3 MODIFY COLUMN (col1 VARCHAR(6));
```

Changing a Column's Name

You can change the name of table columns without changing any of the data. The data remains the same type and size. You must be the owner of the table or have the Alter privilege on tables to change the name of table columns.

- To change name of a column, enter:

```
system(admin)=> ALTER TABLE distributors RENAME COLUMN address to
city;
```

Adding or Dropping a Column

You can add or drop table columns. If you add a column, you can provide a default value. You must be the owner of the table or have the Alter privilege on tables to add or drop columns. You cannot add or drop a column in a transaction block. For more information about adding or dropping a column, see "ALTER TABLE" on page B-20.

- To add a column, enter:

```
system(admin)=> ALTER TABLE table_name ADD [COLUMN] column_name
type [ column_constraint [ constraint_characteristics ] ] [, ... ];
```

- To drop a column, enter:

```
system(admin)=> ALTER TABLE table_name DROP [COLUMN] column_name [,
... ];
```

Joining Tables

Queries can access multiple tables at once, or access the same table in such a way that multiple rows of the table are being processed at the same time. A query that accesses multiple rows of the same or different tables at one time is called a join query. For more information, see Appendix C, "Join Overview."

For example, to list all the weather records together with the location of the associated city, you might compare the city column of each row of the weather table with the name column of all rows in the cities table, and select the pairs of rows where these values match:

```
system(admin)=> SELECT * FROM weather, cities WHERE city = name;
```

Using Inner Join Queries

An inner join uses a comparison operator to match rows from two tables based on the values in common columns from each table. Inner joins are the most common type of joins.

To pose an inner join query, enter:

```
system(admin)=> SELECT * FROM weather INNER JOIN cities ON
(weather.city = cities.name);
```

The two columns contain the city name, because the weather and the cities tables are concatenated. To avoid the duplication, you can list the output columns explicitly rather than using an asterisk (*):

```
system(admin)=> SELECT city, temp_lo, temp_hi, prcp, date, location
     FROM weather, cities WHERE city = name;
```

Because the columns all have different names, this example worked as expected. It is good style, however, to fully qualify column names in join queries, as follows:

```
system(admin)=> SELECT weather.city, weather.temp_lo, weather.temp_hi,
weather.prcp, weather.date, cities.location FROM weather, cities WHERE
cities.name = weather.city;
```

Using Left-Outer Join Queries

A left-outer join query returns all the rows from the left table, not just the ones in which the joined column match. If a row in the left table has no matching rows in the right table, the associated result row contains nulls for all select clause columns coming from the right table:

```
system(admin)=> SELECT * FROM weather LEFT OUTER JOIN cities ON
(weather.city = cities.name);
```

Using Self-Join Queries

A self-join query joins a table against itself. For example, to find all the weather records that are in the temperature range of other weather records, you need to compare the temp_lo and temp_hi columns of each weather row to the temp_lo and temp_hi columns of all other weather rows:

```
system(admin)=> SELECT W1.city, W1.temp_lo AS low, W1.temp_hi AS high,
     W2.city, W2.temp_lo AS low, W2.temp_hi AS high
     FROM weather W1, weather W2
     WHERE W1.temp_lo < W2.temp_lo
     AND W1.temp_hi > W2.temp_hi;
```

The example query relabels the weather table as W1 and W2 to distinguish between the left and right side of the join.

Combining Tables with UNION, INTERSECT, and EXCEPT

The standard set operations UNION, INTERSECT, and EXCEPT/MINUS let you combine the results from two or more SELECT statements to construct more complex queries.

Netezza supports three classes of set operations:

- UNION [DISTINCT] and UNION ALL
- INTERSECT [DISTINCT] and INTERSECT ALL
- EXCEPT [DISTINCT] or MINUS [DISTINCT] and EXCEPT ALL, MINUS ALL

SQL statements that contain set operators are called *compound queries* and each SELECT statement in a compound query is called a *component query*. You can combine two or more SELECT statements into a compound query if they satisfy the following *union compatibility* conditions:

- The result sets of both queries must have the same number of columns.
- The corresponding columns in the two queries must have the same data type or must be implicitly convertible to the same data type.

  For example, you can have a column in the first component query be the data type CHAR that corresponds to the VARCHAR column in the second component query or vice versa. You cannot, however, have a column in the first component that is a DATE and the corresponding column in the component be a CHAR. For more information about data type conversions, see Table 2-4 on page 2-22 and Table 2-6 on page 2-23.

The syntax for a set operation is:

```
<SELECT-statement>
{UNION | INTERSECT | EXCEPT | MINUS} [ALL | DISTINCT]
<SELECT-statement>
{UNION | INTERSECT | EXCEPT | MINUS} [ALL | DISTINCT]
<SELECT-statement>]*
[ORDER BY …]
[LIMIT …]
```

Set operations have the following restrictions:

- If the names of the corresponding columns match, SQL uses that column name in the result. If the corresponding column names differ, SQL uses the column name from the first query in the set statement. If you want to rename a column in the result, use an AS clause in the first query.
- You can specify an optional ORDER BY clause only in the final query in the set statement. SQL applies the sort to the final combined result.
- You can specify an optional LIMIT clause after the ORDER BY. SQL applies the limit to the final combined result.
- You can specify GROUP BY and HAVING only in individual queries. You cannot use them to affect the result.

Using the UNION Operation

The UNION operation combines the results of two subqueries into a single result that comprises the rows returned by both queries. (This operation differs from a join, which combines columns from two tables.) A UNION expression (optional keyword DISTINCT) removes duplicate rows from the result; a UNION ALL expression does not remove duplicates.

UNION

In a UNION [DISTINCT] operation, if a tuple t appears m (>= 0) times in the first input table, and the same tuple t appears n (>= 0) times in the second input table, then that tuple t appears only once in the output table if (m + n) > 0.

```
{0,1,2,2,2,2,3,N,N} UNION {1,2,2,3,5,5,N,N,N}
➔{0,1,2,3,5,N}
```

UNION ALL

In UNION ALL operation, if a tuple t appears m (>= 0) times in the first input table, and the same tuple t appears n (>= 0) times in the second input table, then that tuple t appears (m + n) times in the output table.

```
{0,1,2,2,2,2,3,N,N} UNION ALL {1,2,2,3,5,5,N,N,N}
➔{0,1,1,2,2,2,2,2,2,3,3,5,5,N,N,N,N,N}
```

Using the INTERSECT Operation

The INTERSECT operation combines the results of two queries into a single result that comprises all the rows common to both queries. Whereas a UNION operation is a logical OR, INTERSECT is a logical AND.

INTERSECT

In an INTERSECT [DISTINCT] operation, if a tuple t appears m (> 0) times in the first input table, and the same tuple t appears n (> 0) times in the second input table, then that tuple t appears only once in the output table.

```
{0,1,2,2,2,2,3,N,N} INTERSECT {1,2,2,3,5,5,N,N,N}
➔{1,2,3,N}
```

INTERSECT ALL

In an INTERSECT ALL operation, if a tuple t appears m (> 0) times in the first input table, and the same tuple t appears n (> 0) times in the second input table, then that tuple appears the lesser of m and n times in the output table

```
{0,1,2,2,2,2,3,N,N} INTERSECT ALL {1,2,2,3,5,5,N,N,N}
➔{1,2,2,3,N,N}
```

Using the EXCEPT Operation

The EXCEPT/MINUS operation finds the difference between the two queries and the result comprises the rows that belong only to the first query.

Note: EXCEPT and MINUS are synonyms. You can use either word in SQL statements.

To contrast INTERSECT and EXCEPT:

- A INTERSECT B contains rows from table A that are duplicated in table B.
- A EXCEPT B contains rows from table A that do not exist in table B.

EXCEPT

In an EXCEPT [DISTINCT], or a MINUS [DISTINCT] operation, if a tuple t appears m (> 0) times in the first input table, and same tuple t appears n (>= 0) times in the second input table, then that tuple t appears only once in the output table if (m > 0 && n == 0).

```
{0,1,2,2,2,2,3,N,N} EXCEPT {1,2,2,3,5,5,N,N,N}
➔{0}
```

EXCEPT ALL

In an EXCEPT ALL, or in a MINUS ALL operation, if a tuple t appears m (> 0) times in the first input table, and same tuple t appears n (>= 0) times in the second input table, then that tuple t should appear (m - n) times in the output table if (m - n) > 0.

```
{0,1,2,2,2,2,3,N,N} EXCEPT ALL{1,2,2,3,5,5,N,N,N}
➔{0,2,2}
```

Understanding Precedence Ordering

UNION and EXCEPT/MINUS have the same precedence order. If these operators appear in the same query expression, SQL executes them from left to right. INTERSECT, however takes higher precedence than the other set operations. Thus, if you use INTERSECT with other set operators, SQL executes the INTERSECT operation first.

In the following examples, S1, S2, S3, and S4 represent union-compatible SELECT statements.

```
S1 UNION S2 EXCEPT S3 UNION S4 => (((S1 UNION S2) EXCEPT S3) UNION S4)
S1 UNION S2 INTERSECT S3 MINUS S4 => ((S1 UNION (S2 INTERSECT S3))
EXCEPT S4)
```

To avoid confusion or to force a certain execution order, use parentheses:

```
(S1 UNION S2) INTERSECT (S3 MINUS S4)
```

Handling NULLS

SQL treats NULLS differently in set operations (UNION, INTERSECT, EXCEPT) than it does when handing NULLS in joins and scans.

When comparing rows, SQL treats set operations NULL values as equal to each other; that is, the evaluation expression (NULL = NULL) produces the result TRUE. Whereas in join or scan operations, the same NULL equality expression evaluates to UNKNOWN.

Understanding Data Type Promotion

In a set operation, if the corresponding columns in the input tables are not exactly the same, Netezza promotes the data type. Table 2-4 lists the possible promotions.

Note that in Table 2-4, Table 2-5, and Table 2-6, the table cell represents the data type of the output table and a dash (-) means that the pair has no data type promotion logic.

Table 2-4: Data Type Promotion with Numbers and Characters for UNION

	INT1	INT2	INT4	INT8	NUMERIC	REAL	DOUBLE	CHAR	VARCHAR	NCHAR	NVARCHAR
INT1	INT1	INT2	INT4	INT8	NUMERIC	REAL	DOUBLE	—	—	—	—
INT2	INT2	INT2	INT4	INT8	NUMERIC	REAL	DOUBLE	—	—	—	—
INT4	INT4	INT4	INT4	INT8	NUMERIC	REAL	DOUBLE	—	—	—	—
INT8	INT8	INT8	INT8	INT8	NUMERIC	REAL	DOUBLE	—	—	—	—
NUMERIC	NUMERIC	NUMERIC	NUMERIC	NUMERIC	NUMERIC	DOUBLE	DOUBLE	—	—	—	—
REAL	REAL	REAL	REAL	REAL	DOUBLE	REAL	DOUBLE	—	—	—	—
DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE	—	—	—	—
CHAR	—	—	—	—	—	—	—	CHAR	VARCHAR	NCHAR	NVAR-CHAR
VARCHAR	—	—	—	—	—	—	—	VARCHAR	VARCHAR	NCHAR	NVAR-CHAR
NCHAR	—	—	—	—	—	—	—	NCHAR	NCHAR	NCHAR	NCHAR
NVARCHAR	—	—	—	—	—	—	—	NVAR-CHAR	NVAR-CHAR	NVAR-CHAR	NVAR-CHAR

Table 2-5: Data Type Promotion with Numbers and Characters for Operators

	INT1	INT2	INT4	INT8	NUMERIC	REAL	DOUBLE	CHAR	VARCHAR	NCHAR	NVARCHAR
INT1	INT1	INT2	INT4	INT8	NUMERIC	DOUBLE	DOUBLE	—	—	—	—
INT2	INT2	INT2	INT4	INT8	NUMERIC	DOUBLE	DOUBLE	—	—	—	—
INT4	INT4	INT4	INT4	INT8	NUMERIC	DOUBLE	DOUBLE	—	—	—	—
INT8	INT8	INT8	INT8	INT8	NUMERIC	DOUBLE	DOUBLE	—	—	—	—
NUMERIC	NUMERIC	NUMERIC	NUMERIC	NUMERIC	NUMERIC	DOUBLE	DOUBLE	—	—	—	—
REAL	REAL	REAL	REAL	REAL	DOUBLE	REAL	DOUBLE	—	—	—	—
DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE	—	—	—	—
CHAR	—	—	—	—	—	—	—	CHAR	VARCHAR	NCHAR	NVAR-CHAR
VARCHAR	—	—	—	—	—	—	—	VARCHAR	VARCHAR	NVAR-CHAR	NVAR-CHAR
NCHAR	—	—	—	—	—	—	—	NCHAR	NVAR-CHAR	NCHAR	NVAR-CHAR
NVARCHAR	—	—	—	—	—	—	—	NVAR-CHAR	NVAR-CHAR	NVAR-CHAR	NVAR-CHAR

Table 2-6 displays the data type promotion for noninteger values.

Table 2-6: Data Type Promotion with Non-integers

	UNKNOWN	BOOL	DATE	TIME	TIMESTAMP	TIMETZ	INTERVAL
BOOL	BOOL	BOOL	—	—	—	—	—
DATE	DATE	DATE	—	DATE	—	—	—
TIME	TIME	—	TIME	TIME	TIME	—	—
TIMESTAMP	TIMESTAMP	—	TIMESTAMP	TIMESTAMP	TIMESTAMP	TIMESTAMP	—
TIMETZ	TIMETZ	—	—	TIMETZ	TIMETZ	TIMETZ	—
INTERVAL	INTERVAL	—	—	—	—	—	INTERVAL

Managing Views

You can use views to focus, simplify, and customize each user's perception of the database. You can use views as a security mechanism by allowing users to access data through the view, without granting the users permission to access the view's underlying base tables directly.

Creating Views

The CREATE VIEW command lets you define a view of a table. The view is not physically created, but rather the Netezza RDBMS automatically generates a query rewrite to support retrieve operations on views.

▼ To create a view, enter:

```
system(admin)=> CREATE VIEW viewname AS SELECT query;
```

Replacing Views

The CREATE OR REPLACE VIEW command transfers the permissions (ACL data) from one view to another. You should also use this command to recompile your views after you rename a database.

▼ To transfer view permissions, enter:

```
system(admin)=> CREATE OR REPLACE VIEW viewname AS SELECT query;
```

Dropping Views

The DROP VIEW command drops (or removes) a view. You must be the owner of the view or have been granted the Drop privilege on views to drop an existing view from the database.

▼ To drop a view, enter:

```
system(admin)=> DROP VIEW emp;
```

Renaming Views

You can change the name of a view without changing any of the data. The data remains the same type and size. You must be the owner of the view or have the Alter privilege on views to change the view's name.

▼ To rename a view, enter:

```
system(admin)=> ALTER VIEW emp RENAME TO employees;
```

Changing View Ownership

If you are the owner or have the Alter privilege on views, you can change the owner of a view without changing any of its data. The new owner must be a current user of the system.

▼ To change view ownership, enter:

```
system(admin)=> ALTER VIEW emp OWNER TO john;
```

Using Materialized Views

Sorted, projected, and materialized views (SPM) are views of user data tables (base tables) that project a subset of the base table's columns and are sorted on a specific set of the projected columns. When you create a materialized view, the system materializes and stores the sorted projection of the base table's data in a unique (materialized) table on disk. You can query the SPM view directly or use it to increase the query performance against the base table.

The query planner/optimizer automatically checks for materialized views and takes advantage of them when they exist and when they would be faster to use than the original table. This means that you can add materialized views to your database and gain the performance benefits without having to rewrite any of your applications.

Materialized views improve query performance by reducing the amount of data the system transfers from the disk during scans, and by the fact that the data is sorted, which results in a better zone map for the ORDER BY columns.

Materialized views also provide significant performance benefits when you use them for single-record or few-record lookups. When you create materialized views, the system automatically adds another column to each SPM view record that defines where in the base table the SPM view record originated (that is, the base table's block number). You can use the SPM view as an index for the base table. A base table query targeted at a single record or a few records can use the sorted nature of an SPM view and its corresponding automatically created zone map to retrieve the block location of the records in the base table quickly.

Creating Materialized Views

When you use SQL to create a materialized view from a base table, the system stores the view definition for the lifetime of the SPM view and the view is visible as a materialized view. Records associated with the SPM view are materialized into a Netezza unique table. SPM data slices are co-located on the same data slices as the corresponding base table data slices and persistently stored on the user disk partitions.

▼ To create a materialized view, enter:

```
system(admin)=> CREATE MATERIALIZED VIEW customers_mview AS SELECT
customer_name, customer_id FROM customers ORDER BY customer_id;
```

The following restrictions apply to creating a materialized view:

- You can only specify one base table in the FROM clause.
- You cannot use the WHERE clause.
- The columns in the projection list must be columns in the base table and no expressions (aggregates, mathematical operators, casting, DISTINCT operator, and so on) are allowed.
- You must specify at least one column in the projection list.
- The columns in the optional ORDER BY clause must be one or more columns in the projection list. If you do not specify ORDER BY, the materialized view retains the same sort order as the base table.
- You cannot specify NULLS LAST or DESC in the ORDER BY expression.
- You cannot specify an external, temporary, system, or a clustered base table (CBT) as a base table for the view.

Note: As you insert new records into the base table that has an associated SPM view, the system appends the new records to the materialized view table as well. Thus, there are two areas in the materialized table: one area contains the sorted records generated when the view was created, the other area contains unsorted records that have been inserted into the base table subsequent to the SPM view's creation. If your query performance depends on the sorted nature of the SPM view, you should periodically manually refresh the SPM view by suspending and refreshing it. For more information, see "Altering Materialized Views" on page 2-26.

Viewing Materialized Views

If you have List permission for the object or object class, you can view information about materialized views. Use the **nzsql** \dm command or use the NzAdmin tool.

For more information about the \d commands, see Table D-2 on page D-2. For more information about using the GUI, see the *IBM Netezza System Administrator's Guide*.

Replacing Materialized Views

If the base table for a materialized view changes in a way that affects a materialized view for that table, users could see the following error for queries that use the view:

```
ERROR:  Base table/view 'WEATHER' attr 'CITY' has changed (precision);
rebuild view 'WEATHER_V'
```

This error indicates that the column named CITY in the base table WEATHER has changed. In this example, the column changed from a VARCHAR(80) to a VARCHAR(100). As a result, the materialized view must be rebuilt to reflect the current base table definition.

To rebuild a view after a base table change, use the CREATE OR REPLACE MATERIALIZED VIEW command to update the view, as follows:

```
MYDB(ADMIN)=> CREATE OR REPLACE MATERIALIZED VIEW weather_v AS SELECT
city, temp_lo, temp_hi FROM weather ORDER BY city;
CREATE MATERIALIZED VIEW
```

As a best practice, do not drop and recreate the materialized view because those steps result in a new view with a different object ID, which could impact other objects that reference the materialized view.

Dropping Materialized Views

When you use SQL to drop a materialized view, the system removes the view definition, the materialized table containing the materialized records, and frees the disk storage allocated to the table.

- To drop a materialized view, enter:

```
system(admin)=> DROP VIEW customers_mview;
```

If you drop the base table from which a materialized view is derived, the system drops the materialized table, but retains the view definition, which reverts to a regular view. All subsequent accesses to the SPM view result in an error message.

Note: You cannot drop the materialized table directly with the DROP command. If you attempt to drop the table, the system reports an error message and prevents the operation.

Altering Materialized Views

You can use SQL to alter the materialize property of an SPM view, which can be ACTIVE, or SUSPEND. REFRESH allows you to change from the SUSPEND to the ACTIVE state.

- Using the SUSPEND option marks a materialized view and its associated table as not eligible for use in queries or transactions. The system truncates the materialized table and redirects all queries against the materialized view to the base table.

  Use SUSPEND to temporarily defer updates to materialized tables, such as when you are running reclaims, restores, or loads.

- Using the REFRESH option re-materializes the SPM view, which re-creates the materialized table from the base table. Although normally you use the REFRESH option on suspended materialized views, you can also use it on ordered unsuspended materialized views to re-sort the views for better performance. You would also use the REFRESH option to update the materialized views after an insert to the base table.

- To change the properties of an SPM view, enter:

```
system(admin)=> ALTER VIEW customers_mview MATERIALIZE REFRESH;
```

Setting the Refresh Threshold

If you have administration privilege, you can use the SET command to set a refresh threshold that allows you to refresh all the materialized views associated with a base table. The threshold specifies the percentage of unsorted data in the materialized view.

You can set the threshold from 1 to 99. The default is 20.

- To set the materialization refresh threshold, enter:

```
system(admin)=> SET SYSTEM DEFAULT MATERIALIZE THRESHOLD TO
<number>;
```

When you use the ALTER VIEWS ON MATERIALIZE REFRESH command, the system refreshes all suspended views, and all non-suspended views whose unsorted data has exceeded the refresh threshold.

- To change the properties of all SPM views associated with a base table, enter:

```
system(admin)=> ALTER VIEWS ON customers MATERIALIZE REFRESH;
```

Changing Materialized Views

You cannot perform direct record inserts, updates, or truncates on materialized views or their associated materialized tables.

- If you delete records in the base table, either as part of a record update or record delete operation, the system propagates the change to all the appropriate materialized records in the unsuspended SPM views.
- If you truncate records in the base table, the system truncates the materialized tables for the associated SPM views.

Querying Materialized Views

You can only pose read-only queries to materialized views. Also, if you direct a query to the base table, the optimizer checks for the existence of any associated materialized views. If materialized views exist, the optimizer determines whether to use a view based on its predicted cost (query time) and performance.

Memory Usage

The memory utilization to support the creation of SPM views is equal to that required of CTAS statements. Queries that rely on the sorted nature of SPM may require additional memory to sort the unsorted records added after the materialized views were created.

Mirroring and Regeneration of Materialized Views

The materialized records in an SPM view exist in a Netezza unique table associated with the SPM view. The system mirrors this table in the mirror partition just like any other user table.

If a disk that contains a data slice of an SPM view fails, the system handles the failover and regeneration just like any other user table or view.

Reclamation and Materialized Views

The GROOM TABLE command is used to reclaim records or blocks from a table, and applies only to base tables, not to SPM views.

Note: Since using GROOM TABLE on a base table returns an error if there are any active materialized views on the table, these views must be suspended before running the command.

Loading and Materialized Views

When you use the **nzload** command to load data into a base table, the system automatically updates the associated materialized views, which can slow your load process.

For better performance, suspend the materialized views before beginning a long load process. When you have completed the load, refresh your materialized views.

Backing Up and Restoring Materialized Views

When you use the **nzbackup** command to back up your database, the system backs up the SPM view definition (that is, the schema) just like it does with any other view. The system does not back up any SPM view-specific record data.

If you restore the database, the system automatically restores any base tables and re-creates the associated materialized views unless the SPM views were suspended when you created the backup. In that case, the system restores their state as SUSPEND.

If you perform a table-level restore on a base table that has associated SPM views, the system restores the SPM views in their original materialized state, unless you have suspended the materialized views, in which case they remain suspended.

Zone Maps and Materialized Views

The system creates zone maps for all columns in SPM views that have data types integer, date, and timestamp. The system also creates zone maps for all ORDER BY columns in the SPM view, except for columns of numeric types that are larger than 8 bytes (19 decimal digits or more).

The system creates zone maps for the following data types in the ORDER BY clause:

- integers — 1-byte, 2-byte, 4-byte, and 8-byte
- date
- timestamp
- char — all sizes, but only the first 8 bytes are used in the zone map
- varchar — all sizes, but only the first 8 bytes are used in the zone map
- nchar — all sizes, but only the first 8 bytes are used in the zone map
- nvarchar — all sizes, but only the first 8 bytes are used in the zone map
- numeric — all sizes except 128-bit numerics
- float
- double
- bool
- time
- time w/timezone
- interval

Assigning Privileges to Use Materialized Views

The admin user and the owner of the database have all privileges on materialized views by default. For all other users, Table 2-7 lists the privileges you must assign:

Table 2-7: Materialized View Privileges

Task	Privilege
Create an SPM view	Assign the Create Materialized View administration privilege.

Table 2-7: Materialized View Privileges (continued)

Task	Privilege
Alter an SPM view	Assign the Alter object privilege for a specific view or the View object class.
Drop an SPM view	Assign the Drop object privilege for a specific view or the View object class.
Select from an SPM view	Assign the Select object privilege for a specific view or the View object class.
Alter Views on a table	Assign the Insert object privilege for a specific view or the Table object class.
List on SPM views	Assign the List object privilege for a specific view or the View object class.

Tips for Creating Materialized Views

Keep in mind the following guidelines when choosing to use materialized views:

- Most frequently used columns — If you have a few sets of columns that you use frequently in the queries (found by analysis), then create materialized views with those columns.
- Most restrictive column — If there is a column that participates in a filter clause and usually filters out most of the table (for example, temporal columns), then use this column for the ORDER BY clause in the materialized view creation.
- Materialized view index — Create very thin materialized views (with as few columns as possible) that contain the most restrictive columns (for example, temporal columns) for use as indexes.
- Fewer materialized views — For the best performance, create as few materialized views per each table as possible. Each materialized view that you create for a base table causes the system to analyze the performance of using each view over the base table, which can add time to the query performance.

Understanding Subqueries

A subquery is a select expression that is enclosed in parentheses as a nested query block in a query statement. These nested query blocks can appear in any of the following SQL statements: SELECT, INSERT, DELETE, UPDATE, CREATE TABLE AS, INSERT INTO, and SELECT INTO. You can nest subqueries to any arbitrary depth.

The parent query that contains the subquery is often referred to as a *super query* or *outer query*. Subqueries in the same parent are used to derive sets of results that can be evaluated in conjunction with the parent query.

Subqueries can be further divided into the following categories:

- Row subquery — Returns one row (or zero rows) and multiple columns, and can occur in a SELECT list or in a condition expression; for example, as an argument of a comparison operator.
- Table subquery — Returns multiple rows (0—n rows) and multiple columns, and can exist in a FROM clause or as an argument of an EXISTS, IN, ANY, or ALL test.

  For example, "List all stores where sales are more than one percent of all company sales," can be written in the SQL query-within-a-query form using a scalar subquery:

Example 1

```
SELECT StoreId FROM Stores
WHERE TotalSale > 0.01*
(SELECT SUM(TotalSales) FROM Stores);
```

  The system calculates the sum of sales of the inner subquery first and then uses it when running the outer query.
- Singleton subquery — Returns exactly one value in the format of one row (or zero rows), one column table.

For all these subqueries the system evaluates them once and calculates and stores their select expression. When you execute the super query, the system substitutes the computed values in place of the subquery.

A *correlated subquery*, on the other hand, is a query within a query that references (or correlates) with the outer query. With correlated subqueries the system evaluates the subquery repeatedly, once for every row selected from the outer table.

Understanding Correlated Subqueries

Correlated subqueries are not self contained, but instead refer to columns from the parent query.

For example, the query, "Find all stores where the sales of dairy products is more than 10 percent of the total sales," can be written in SQL as:

Example 2

```
SELECT StoreID FROM Stores S
WHERE S.TotalSales*0.1 <
(SELECT SUM(1.Price) FROM Item i
WHERE S.StoreID = I.StoreId and I.ItemType = "Dairy");
```

This is a correlated subquery, because the inner query uses the outer column StoreID from the Stores table in the parent query in order to calculate dairy totals for specific stores.

In many cases, you can also write correlated queries as JOINS. For example, you could rewrite Example 2 using a JOIN between the Stores table and a table subquery on Items.

Example 3

```
SELECT S.StoreId FROM Stores S,
(SELECT I.StoreId, Sum(Price) DairySales FROM Items I
WHERE I.ItemType = "Dairy" GROUP BY I.StoreId) D
WHERE S.StoreId = D.StoreId AND S.TotalSales *0.1 < D.DairySales;
```

In this example, the subquery creates a results table as part of the outer FROM clause that is then joined with the Stores tables so that the Dairy sales test can be performed.

Using Correlated Subqueries in Netezza SQL

Netezza supports both regular and correlated subqueries. Whenever Netezza SQL encounters a regular subquery, it precalculates the subquery once as in Example 1. When the system encounters correlated subqueries in WHERE restrictions, it transforms them internally to equivalent join formats as in Example 3.

If you choose to use correlated subqueries, keep in mind the following restrictions on the form and placement of correlated subqueries:

- You can use correlated subqueries in WHERE clauses.
- You can use correlated subqueries in inner join conditions and with the equal join condition operator.
- You can use correlated subqueries in mixed correlated expressions only in the form as follows:

```
expr(corr_columA, corr_columnB,...) = expr(local_columnX, local_
columnY,...)
```

- You cannot use correlated subqueries in SET operations (UNION, INTERSECT, EXCEPT, and MINUS).
- You cannot use correlated subqueries in aggregates with GROUP BY and HAVING clauses.
- You cannot use correlated subqueries in ORed clauses or in CASE/WHEN expressions.
- You cannot use correlated subqueries in IN lists.
- You can not use correlated subqueries in SELECT lists.

Note: Because correlated subqueries can drastically affect query performance, whenever possible you should consider replacing them with joins for more efficient code.

Using Aggregate Functions

An aggregate function computes a single result from multiple input rows. For example, you can use aggregates to compute the count, sum, avg (average), max (maximum), and min (minimum) over a set of rows.

There are two main categories of aggregate functions: grouped and window aggregates. The following sections provide an overview of these aggregate functions.

Grouped Aggregates

Grand grouped aggregates return a single result for all the data. For example, the following query shows a grand grouped aggregate; it returns the highest low-temperature reading in the table weather:

```
system(admin)=> SELECT max(temp_lo) FROM weather;
 MAX
-----
   55
```

As shown in this example, the grouped aggregates provide a summary (in this case, the maximum value) of a set of rows, but they do not preserve any detail of the information. They simply return group column and aggregate values.

When aggregates are used in combination with the GROUP BY clause, the query returns a result for each group. For example, the following query displays the maximum low temperature observed in each city in the table:

```
system(admin)=> SELECT city, max(temp_lo) FROM weather GROUP BY city;
     CITY      | MAX
---------------+-----
 San Francisco |  55
 Boston        |  40
(2 rows)
```

The system computes an aggregate for each city. You can filter these grouped rows using HAVING:

```
system(admin)=> SELECT city, max(temp_lo) FROM weather GROUP BY city
HAVING max(temp_lo) < 42;
  CITY  | MAX
--------+-----
 Boston |  40
(1 row)
```

The following example uses a subquery to identify the city or cities where the max temp_lo value occurred. The subquery is necessary because the aggregate max cannot be used in a WHERE clause.

```
system(admin)=> SELECT city FROM weather WHERE temp_lo = (SELECT
max(temp_lo) FROM weather);
      CITY
---------------
 San Francisco
(1 row)
```

For more information about the available aggregate functions, see "Aggregate Functions" on page 3-16.

In the 6.0 Netezza release, there is support for enhanced SQL GROUP BY clause syntax of rollup, cube, and grouping sets, to be used on user tables.

Note: To avoid undesired ordering of results, control the order by using an explicit ORDER BY. You may also need to include the GROUPING function, if nulls are in the data.

Note: At this time, the Netezza system does not support the use of CUBE, ROLLUP, or GROUPING SET syntax with hypothetical set functions or inverse distribution functions. Doing so results in an error, as in the following example:

```
SELECT percentile_disc (10) WITHIN GROUP ( order by col1_Srno) FROM
advanced_analytic_1 GROUP BY CUBE(col1_int1,col1_int2);

ERROR: CUBE, ROLLUP, GROUPING SETS, GROUPING not permitted with WITHIN
GROUP
```

Rollup

The rollup syntax gives aggregation results at multiple grouping levels in a single result set. For example, the following returns counts at three grouping levels:

```
SELECT <col1>, <col2>, COUNT(*) FROM <table> GROUP BY ROLLUP (col1,
col2);
```

The group levels returned are as follows, with results as though the group bys were specified simultaneously:

```
GROUP BY col1, col2
```

```
GROUP BY col1
GROUP BY ()
```

Note that the syntax of group by () is equivalent to specifying a grand aggregate (as though there were no group by at all).

The result of the above rollup operation is equivalent to the following UNION result:

```
SELECT col1, col2, COUNT(*) FROM <table> GROUP BY col1, col2 UNION ALL
SELECT col1, null as col2, count(*) FROM <table> GROUP BY col1 UNION
ALL
SELECT null as col1, null as col2, count(*) FROM <table>;
```

Consider the following example:

```
SELECT state, city, COUNT(*) FROM citizens GROUP BY ROLLUP (state,
city);
```

This gives the following results:

- A count for each state/city
- A count for each state
- A grand count

Cube

The cube syntax also gives aggregation results at multiple grouping levels in a single result set. For example, the following returns counts at four grouping levels:

```
SELECT col1, col2, COUNT(*) FROM <table> GROUP BY CUBE (col1, col2);
```

The group levels returned are as follows, with results as though the group bys were specified simultaneously:

```
GROUP BY col1, col2
GROUP BY col1
GROUP BY col2
GROUP BY ()
```

The result of the above cube operation is equivalent to the following UNION result:

```
SELECT col1, col2, count(*) FROM <table> GROUP BY col1, col2 UNION all
SELECT col1, null AS col2, COUNT(*) FROM <table> GROUP BY col1 UNION
all
SELECT null as col1, col2, count(*) FROM <table> GROUP BY col2 UNION
all
SELECT null as col1, null AS col2, count(*) FROM <table>;
```

Consider the following example:

```
SELECT size, color, SUM(sales) FROM citizens GROUP BY CUBE (size,
color);
```

This gives the following results:

- Total sales for each size/color combination
- Total sales for each size
- Total sales for each color

- Grand total sales

Grouping Sets

The grouping sets () syntax offers a generalization of the rollup and cube capability. Any rollup() or cube() operation can be translated into a grouping sets specification. The syntax is as follows:

```
GROUP BY [<set quantifier>] GROUPING SETS(<grouping-set-list>);
```

Where <set quantifier> is either DISTINCT or ALL and defaults to ALL, and <grouping-set-list> is a comma separated list of grouping sets, each of which is a parenthesized <grouping-col-list>, as in the following example:

```
SELECT col1, col2, col3, COUNT(*) FROM <table> GROUP BY GROUPING SETS
((col1,col2), (col2,col3), (col2), ());
```

This is equivalent to grouping by the following groupings simultaneously:

```
GROUP BY (col1, col2)
GROUP BY (col2, col3)
GROUP BY col2
GROUP BY ()
```

The result of the above SELECT is equivalent to the result of doing a UNION ALL for the four selects aggregating at the four group levels. So a rollup() or cube() operation can be simply translated into a grouping sets specification.

Note that an entry in a <grouping-set-list> can itself be a <grouping-set-list>, and so could also be a cube() or rollup(). Any such complex specified grouping-set-list can always be expanded into a simple grouping-set-list, as in the following example:

```
GROUPING SETS (ROLLUP(col1,col2), CUBE(col1,col2))
```

Which is equivalent to the following:

```
GROUPING SETS ((col1,col2), (col1), (), (col1,col2), (col1), (col2),
())
```

With the following distinct set quantifier:

```
DISTINCT GROUPING SETS (ROLLUP(col1,col2), CUBE(col1,col2))
```

Which is equivalent to the following, showing that CUBE(<list>) is a superset of ROLLUP(<list>):

```
GROUPING SETS ((col1,col2), (col1), (col2), ())
```

Also note that the grouping set (col1, col2) is equivalent to (col2, col1), so the elimination of duplicates in the grouping-set-list is done by putting list entries into a canonical form that lists unique entries in the order in which they occur in the grouping sets clause.

Multiple grouping sets at the same level (not nested) are handled as in the following example:

```
GROUP BY GROUPING SETS ((A), (B)), GROUPING SETS ((X, Y), (Z))
```

Which is equivalent to the following:

```
GROUP BY GROUPING SETS ((A, X, Y), (A, Z), (B, X, Y), (B, Z))
```

Another example is the following:

```
GROUP BY A, GROUPING SETS ((X,Y), (Z))
```

Which is equivalent to the following:

```
GROUP BY GROUPING SETS ((A, X, Y), (A, Z))
```

Using the GROUPING function helps to distinguish differences. The following is an example of a rollup query:

```
STATE  |     CITY      |  SUM

_____________________________
 CA    | Los Angeles   |  600
 CA    | San Diego     |  225
 CA    | San Francisco |  450
 CA    |               | 1275
 MA    | Boston        |  460
 MA    | Springfield   |  345
 MA    |               |  805
       |               | 2080
```

Since some group column values will be NULL in superaggregate rows, if there was a NULL city, it might be difficult to tell an aggregate row from a superaggregate. By using GROUPING (city), the result would return a 0 if the city is included in the “group by,” and would return a 1 if the city is not included in the “group by.”The following example shows the result:

```
STATE  |     CITY      |  SUM  | GROUPING(city)

_____________________________________________
 CA    | Los Angeles   |  600  |       0
 CA    | San Diego     |  225  |       0
 CA    | San Francisco |  450  |       0
 CA    |               | 1275  |       1
 MA    | Boston        |  460  |       0
 MA    | Springfield   |  345  |       0
 MA    |               |  805  |       1
       |               | 2080  |       1
```

By using GROUPING (state), the result would return a 0 if the state is included in the “group by,” and would return a 1 if the state is not included in the “group by.”The following example shows the result:

```
STATE  |     CITY      |  SUM  | GROUPING(city)  | GROUPING(state)

_____________________________________________________________________
 CA    | Los Angeles   |  600  |       0         |       0
 CA    | San Diego     |  225  |       0         |       0
 CA    | San Francisco |  450  |       0         |       0
 CA    |               | 1275  |       1         |       0
 MA    | Boston        |  460  |       0         |       0
 MA    | Springfield   |  345  |       0         |       0
 MA    |               |  805  |       1         |       0
       |               | 2080  |       1         |       1
```

Using Grouping Sets With Window Aggregates

You can use window aggregates in grouping selects, as in the following example:

```
SELECT grp, COUNT(*), sum(COUNT(*)) OVER (ORDER BY COUNT(*) rows
unbounded preceding) FROM emp GROUP BY grp;
```

Such a query is handled by converting it to a window aggregating query done on a subquery doing the grouped aggregates. So the example becomes the following:

```
SELECT grp, ct, SUM(ct OVER (ORDER BY ct rows unbounded preceding) FROM
(SELECT grp, COUNT(*) AS ct FROM emp GROUP BY grp) as s;
 GRP       | COUNT | SUM
--------------+-------+-----
 gone          |     1 |   1
 dev           |     1 |   2
 mkt           |     1 |   3
 hdev          |     1 |   4
 adm           |     3 |   7
 sdev          |     7 |  14
```

So you can use enhanced group by options in combination with window aggregates, as in the following example:

```
SELECT grp, COUNT(*) AS grpcnt, SUM(COUNT(*)) OVER (ORDER BY COUNT(*)
rows unbounded preceding) FROM emp GROUP BY ROLLUP(grp);
```

This becomes the following:

```
SELECT grp, ct, SUM(ct OVER (ORDER BY ct rows unbounded preceding) FROM
(SELECT grp, COUNT(*) AS ct FROM emp GROUP BY ROLLUP(grp)) AS s;
 GRP       | COUNT | SUM
--------------+-------+-----
 gone          |     1 |   1
 dev           |     1 |   2
 mkt           |     1 |   3
 hdev          |     1 |   4
 adm           |     3 |   7
 sdev          |     7 |  14
               |    14 |  28
```

Window Aggregates

Window aggregates (also called window analytic functions) allow you to compute an aggregate value based on a group of rows, which are defined by a window. The window determines the range of rows the system uses to perform the calculations. Window sizes can be based on a physical number of rows or a logical interval such as year-to-date, quarterly, and so on. You can use window aggregates to compute cumulative, moving, centered, and reporting aggregates. Unlike the grouped aggregates, window aggregates can preserve the row information.

For example, the following is a table of monthly sales information called monthlysales:

```
 YEAR | MONTH | SALESK
------+-------+--------
 2007 |    10 |     20
 2007 |    11 |     22
 2007 |    12 |     25
 2008 |     1 |     30
 2008 |     2 |     35
 2008 |     3 |     50
 2008 |     4 |     70
(7 rows)
```

With window aggregates, you can compute various aggregations over moving time frames. For example, the following query shows a three-month moving average of the sales total:

```
system(admin)=> SELECT year, month, salesk, avg(salesk) OVER
(PARTITION BY year ORDER BY month ROWS BETWEEN 1 PRECEDING AND 1
FOLLOWING) FROM monthlysales;
 YEAR | MONTH | SALESK |    AVG
------+-------+--------+-----------
 2007 |    10 |     20 | 21.000000
 2007 |    11 |     22 | 22.333333
 2007 |    12 |     25 | 23.500000
 2008 |     1 |     30 | 32.500000
 2008 |     2 |     35 | 38.333333
 2008 |     3 |     50 | 51.666667
 2008 |     4 |     70 | 60.000000
(7 rows)
```

The output has a result for each row and the AVG value is a moving average of the previous, current, and following month's sales values. In the case of the first row, the average is based only on the current and following month, as there is no previous month. Likewise, the last row is the average of only the previous and current month.

The following example shows a running total of the sales summary:

```
system(admin)=> SELECT *, sum(salesk) OVER (PARTITION BY year ORDER BY
month ROWS UNBOUNDED PRECEDING) FROM monthlysales;
 YEAR | MONTH | SALESK | SUM
------+-------+--------+-----
 2007 |    10 |     20 |  20
 2007 |    11 |     22 |  42
 2007 |    12 |     25 |  67
 2008 |     1 |     30 |  30
 2008 |     2 |     35 |  65
 2008 |     3 |     50 | 115
 2008 |     4 |     70 | 185
(7 rows)
```

The output has a result for each row and the SUM value is the total of the sales for each month in the table (all the previous rows plus the current row).

For more information about these functions, see "Netezza SQL Analytic Functions" on page 5-1.

Executing Scripts

Scripts allow you to bundle all your queries into a single file, which you can then run automatically and repeatedly. You can create a script by using any standard editor and typing regular SQL commands.

There are three ways that you can execute scripts:

- You can use the redirect command on the command line to specify that the system use a file instead of stdin:

  ```
  NZSQL < script_file
  ```

- You can use the **nzsql** command line argument **-f**, which allows you to specify a file:

  ```
  NZSQL -f script_file
  ```

- You can specify the script from within the **nzsql** command interpreter:

  ```
  EMP(USER)=> \i script_file
  ```

CHAPTER 3

Netezza SQL Basics

What's in this chapter

- Data Types
- Functions and Expressions
- Netezza SQL Extensions
- Netezza SQL Functional Categories

This chapter contains descriptions of data types, functions and expressions, Netezza SQL extensions, and Netezza SQL functional categories.

Data Types

A data type represents a set of values. Using data types in your databases offers the following benefits:

- Consistent results — Having columns of a uniform type produces consistent results. Database operations, such as displaying, sorting, aggregating, and joining, produce consistent results. There is no conflict over how different types are compared or displayed.
- Data validation — Having columns of a uniform type ensures that only properly formatted data is entered.
- Compact storage — Having columns of uniform type ensures that data is stored efficiently. The system does not need to allocate more storage than necessary.
- Performance — Having columns of uniform type allows the system to process the queries efficiently.

Each column in a relational database can hold only one type of data. You cannot mix data types within a column.

This section describes the data types that Netezza SQL supports. It notes type alias where they are available. The first type name listed is the preferred form and is the form that the Netezza SQL saves with the table definition. Note that the type alias is not saved with the table definition.

Exact Numeric Data Types

Numeric types allow you to store numbers. Table 3-1 describes the integer numeric types in various ranges. The larger the range, the more storage it requires.

Choose integer data types for distribution and join columns. Whenever possible use integer data types to benefit from the added performance of zone maps.

Table 3-1: Integer Types

Type	Value	Disk Usage
byteint (alias int1)	8-bit values in range –128 to 127,	1 byte
smallint (alias int2)	16-bit values in range –32,768 to 32,767	2 bytes
integer (alias int and int4)	32-bit values in range –2,147,483,648 to 2,147,483,647	4 bytes
bigint (alias int8)	64-bit values in range -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	8 bytes

Fixed-point numeric data types allow you to define the numeric rounding to a specific decimal place. Table 3-2 describes the fixed-point numeric data types.

Table 3-2: Fixed-Point Numeric Types

Type	Value	Disk Usage
numeric(p, s)	Numeric with precision p and scale s. Precision can range from 1 to 38, scale from 0 to the precision.	Precision of 9 or less — 4 bytes Precision of 10-18 — 8 bytes Precision of 19-38 — 16 bytes
numeric(p)	Equivalent to numeric(p, 0).	
numeric	Equivalent to numeric(18, 0).	8 bytes
decimal	Although decimal is sometimes a distinct SQL data type, Netezza SQL treats it as an alias for numeric.	4 bytes-16 bytes

Always use the smallest integer or fixed-point numeric whenever possible. When converting source data to the Netezza system, you may need to analyze the data to determine the smallest data type that you can use.

- To determine the smallest data size you can use for integer and fixed point numerics, type the following SQL command:

```
SELECT MIN(column_name), MAX(column_name) FROM table_name;
```

Approximate Numeric Data Types

Approximate numeric data types allow you to store floating-point values. The system stores numbers using 6 or 15 digits of precision. The system stores the location of the decimal point separately so that it can represent large values. Approximate numeric types may be an option for space considerations—because real and double-precision numeric data types are stored compactly—but they can produce imprecise rounding during computations.

Use caution when using approximate numerics. Do not use approximate numerics for distribution columns, join columns, or columns that require mathematical operations such as SUM, AVG, and so on.

Floating point data types also have inherent performance implications. For example, the system cannot perform a fast hash join on a floating point data type, but instead must perform a slower sort merge join.

Table 3-3 describes the approximate numeric data types.

Table 3-3: Approximate Numeric Data Types

Type	Value	Disk Usage
float(p)	Floating point number with precision p, from 1 to 15. Precision less than 6 are equivalent to 6. Precision between 7 and 15 are equivalent to 15.	Precision of 6 or less — 4 bytes. Precision between 7-15 — 8 bytes
real	Equivalent to float(6).	4 bytes
double precision	Equivalent to float(15).	8 bytes

Netezza SQL prefers type names real and double precision, with float(p) being closer to an alias for one or the other of the preferred forms.

Note: Not only is floating point summation approximate, but more importantly, it is non-associative; that is, the result depends on the order in which the partial sums are combined. This is very different from integer and numeric summations that are precise and always produce the same result irrespective of any reordering.

In the massively parallel Netezza, sums and averages are partially evaluated on the SPUs and then combined at the host to produce the final result. Because SPUs return results asynchronously to the host, floating point summations produce different results from run to run.

This effect is particularly noticeable if the values span a large dynamic range and/or there are large values of differing signs tending to cancel each other out.

Character String Data Types

Character strings are the most commonly used data types. They can hold any sequence of letters, digits, punctuation, and other valid characters. Typical character strings are names, descriptions, and mailing addresses. Although you can store any value in a character string, you should use character strings only if other data types are inappropriate. Other data types provide better data validation and more compact storage.

Table 3-4 describes the character string data types.

Table 3-4: Character String Data Types

Type	Value	Disk Usage
Fixed length, character(n) (alias char(n))	Fixed length, blank padded to length n. The default value of n is 1. The maximum character string size is 64,000.	If n is equal to 16 or less — n bytes. If n is greater than 16, disk usage is the same as varchar(n).
Variable length, character varying(n) (alias varchar(n))	Variable length to a maximum length of n. No blank padding, stored as entered. The maximum character string size is 64,000.	N+2 or fewer bytes depending on the actual data.
Fixed length Unicode (alias nchar(n))	Fixed length, blank padded to length n. The maximum length of 16,000 characters.	For more information, see "The Data Types" on page 6-3.
Variable length, Unicode (alias nvarchar(n))	Variable length to a maximum length of n. The maximum length of 16,000 characters.	For more information, see "The Data Types" on page 6-3.

- To determine the optimal character data type, type the following SQL command:

```
system(admin)=> SELECT MAX(LENGTH(TRIM(column_
name))),AVG(LENGTH(TRIM(column_name)))FROM table_name;
```

When selecting a character data type, consider the following:

- If the data is exclusively numeric, use an integer data type instead of a character data type. For example, 11212345 could be defined as a VARCHAR or a bigint. Select a bigint, especially if you are using the column for distribution or joins.
- If, when converting source date, the MAX length is less than the CHAR size, use a CHAR instead of VARCHAR. If the AVG length +2 is less than the CHAR size, use a VARCHAR instead of a CHAR.
- Comparing numbers with string data types can sometimes produce unpredictable results. As a best practice, use the to_number conversion function to convert the string to a number, for example:

```
where to_number(<varchar-column>, '9999') > <integercolumn>
```

Logical Data Types

The only logical data type is boolean. A boolean field can store true, false, and null. Note that the boolean data type is not standard SQL. Table 3-5 describes the logical data type.

Table 3-5: Logical Data Types

Type	Value	Disk Usage
boolean (alias bool)	With value true (t) or false (f).	1 byte

You can use the following words to specify booleans: true or false, on or off, '0' or '1', "true' or 'false', 't' or 'f', 'on' or 'off', 'yes' or 'no'.

Note: Never use a boolean data type for distribution columns because your table would be distributed to only two data slices in the Netezza.

Temporal Types

Temporal data types allow you to store date, time, and time-interval information. Although you can store this data in character strings, it is better to use temporal types for consistency and validation. Note that the time values provide storage accuracy in microseconds (one millionth of a second — six decimal places).

Table 3-6 describes the temporal data types.

Table 3-6: Temporal Data Types

Type	Value	Disk Usage
date	Ranging from January 1, 0001, to December 31, 9999.	4 bytes
time	Hours, minutes, and seconds to 6 decimal positions. Ranging from 00:00:00.000000 to 23:59:59.999999. For more information, see "Conversion Functions" on page 3-29.	8 bytes
time with time zone (alias timetz)	Hours, minutes, seconds to 6 decimal positions, and time zone offset from GMT. Ranging from 00:00:00.000000+13:00 to 23:59:59.999999-12:59.	12 bytes
timestamp	Has a date part and a time part, with seconds stored to 6 decimal positions. Ranging from January 1, 0001 00:00:00.000000 to December 31, 9999 23:59:59.999999.	8 bytes
interval (alias timespan)	An interval of time. This is a nonstandard implementation. For more information, see the next section, "Netezza SQL Interval Support."	12 bytes

Netezza SQL Interval Support

Netezza SQL interval support is nonstandard. Table 3-7 describes how it differs from standard SQL interval support.

Table 3-7: Interval Comparison

Standard SQL	Netezza SQL
Declares intervals as having particular units; for example, colA interval year to month, or colB interval hour.	Accepts this syntax, but ignores the unit specification. All intervals are the same, and can contain values of any combination of units. The hours field of the interval is stored as an integer and thus has a maximum value of 2147483647.
Does not include units in interval literals. For example, an interval year to month column's values might be "13-4," meaning 13 years and 4 months.	Requires that all literal values include the units, as in "13 years 4 months," because interval units pertain to a particular value rather than to a particular column's declaration.
Disallows declaring intervals as having both units smaller than a month and units greater than a day; for example, interval month to day, because this is ambiguous.	Internally normalizes all intervals to units of seconds. Considers a month to be thirty days for the purposes of interval comparisons. To avoid inaccuracies introduced by this approximation, use only intervals in units smaller than months.

Note: You cannot load the interval data type from an external table.

Binary Data Types

Netezza supports two types of binary data types for use in tables and external tables. Table 3-8 describes the binary data types.

Table 3-8: Binary Data Types

Type	Value
VARBINARY(n)	Variable length field from 1 up to 64,000 bytes. Use the VARBINARY type to store binary data in a type-specific field and apply restricts or other processing against the columns as needed.
ST_GEOMETRY(n)	Variable length field from 1 up to 64,000 bytes. The ST_GEOMETRY data type holds geometric binary information. The ST_ GEOMETRY type is useful for storing geometric data as processed by spatial analysis functions and other geometric analysis to identify columns with binary geometry-oriented data.

While it is possible to store binary data in a VARCHAR column, it is difficult to differentiate between VARCHAR data and binary data in the database and for processing and restricts.

Columns that use the binary data types do not support some of the common query processing operations. For example, binary data type columns cannot be used in ordering, grouping, or magnitude comparisons, or in aggregates such as sum(), avg(), distinct(), or min/max comparisons. The binary data cannot be implicitly or explicitly cast to other types.

You can insert the data for the binary objects using tools such as user-defined functions (UDFs) that create the binary data, or you can specify the content using hexadecimal string literal notation.

The hexadecimal string literal representation has the following format:

```
x'hexValue'
```

A valid hexadecimal string literal must begin with the letter x in uppercase or lowercase followed by a single-quoted string of hexadecimal characters. Each hexadecimal character is formed as a pair of two characters from the numbers 0 through 9 and the letters A through F (uppercase or lowercase). For example, the string 'hello' appears as x'68656c6c6f' in hexadecimal string format. A sample INSERT statement with a hexadecimal string literal follows:

```
insert into my_table values (1, x'68656c6c6f');
```

If you do not specify the x prefix, enclose the string in single-quotes, or if the string contains an odd number of characters, the hexadecimal string literal is invalid and the system returns an error.

Netezza Internal Data Types

Netezza supports three internal datatypes: rowid, transaction ID, and dataslice.

Table 3-9 describes the internal data types.

Table 3-9: Internal Data Types

Type	Column Name	Value	Disk Usage
rowid	rowid	100,000-9,223,372,036,854,775,807	8 bytes
transaction ID	createxid deletexid	1024-9,223,372,036,854,775,807	8 bytes
dataslice	datasliceid	1-2,147,483,647	4 bytes

- rowid — Identifies a specific record in the database and is guaranteed to be unique, but not necessarily sequential within a table. At installation, the initial rowid value is 100,000. The Netezza host assigns a range of sequential rowids to each SPU. When you insert records, the system assigns them rowids. When the initial allotment of rowids is depleted, the system assigns another range, which is why the rowids may not be sequential.
- transaction ID — Identifies the transaction ID that created the record (createxid) and the transaction ID that deleted the record (deletexid), which is 0 if the record has not been deleted. When the system updates a record, it updates the deletexid on the original row with the current transaction ID, inserts a record with the updated value and preserves the rowid.

  An xid is an 8-byte integer of which 48 bits are significant. At installation, the initial xid value is 1024. The size of the xid allows for over 100 trillion transaction IDs.
- data slice — Identifies that portion of the database stored on each disk. At installation, the system is divided into a logical number of data slices. When the system creates a record, it assigns it to a logical data slice (and thus a physical disk) based on its distribution key. Although the system dynamically generates datasliceid values, it does not store them with each individual record.

  You can use the datasliceid keyword in a query to identify which data slice the records are coming from. For example:

  ```
  SELECT DATASLICEID, name FROM employee_table;
  ```

  For more information about data slices, see the *IBM Netezza System Administrator's Guide*.

Note: These internal data type column names are reserved words, which means that you cannot use them in DDL statements.

Calculating Row Size

For every row of every table, there is a 24-byte fixed overhead of the rowid, createxid, and deletexid. If you have any nullable columns, a null vector is required and it is N/8 bytes where N is the number of columns in the record. The system rounds up the size of this header to a multiple of 4 bytes.

In addition, the system adds a record header of 4 bytes if any of the following is true:

- Column of type VARCHAR
- Column of type CHAR where the length is greater than 16 (stored internally as VARCHAR)
- Column of type NCHAR
- Column of type NVARCHAR

  Using UTF-8 encoding, each Unicode code point can require 1-4 bytes of storage. A 10-character string requires 10 bytes of storage if it is ASCII and up to 20 bytes if it is Latin, or as many as 40 bytes if it is Kanji.

The only time a record does not contain a header is if all the columns are defined as NOT NULL, there are no character data types larger than 16 bytes, and no variable character data types.

Table 3-10 describes header storage.

Table 3-10: Calculating Row Size

Create XID	Delete XID	Row ID	Null Vector	Record Length	Column_1 ...Column_n
8 bytes	8 bytes	8 bytes	N/8 bytes	4 bytes	Number of bytes

Functions and Expressions

Netezza SQL provides many functions and operators. Functions are operations that take a value, whereas operators are symbols. In many cases, you can use functions and operations to perform the same task, so the difference is commonly one of syntax.

Netezza SQL supports the following types of functions:

- Numeric — Perform mathematical operations on numeric data
- Text — Manipulate strings of text
- Date and time — Manipulate data and times values and to extract specific components from these values
- System — Return information specific to the RDBMS being used
- Fuzzy search and phonetic matching — Provide approximate string matching based on defined techniques or algorithms.
- User-defined — Perform actions that are defined by the function developer (described in the *IBM Netezza User-Defined Functions Developer's Guide*, which is available from Netezza for users who are participating in the Netezza Developer Network)

Operators

Operators differ from functions in the following ways:

- Operators are symbols not names.
- Operators usually take two arguments.

- Arguments usually appear to the left and right of the operator symbol.

The standard arithmetic operators — addition, subtraction, multiplication, division, exponentiation, and modulo — use the standard precedence rules. That is, exponentiation is performed first; multiplication, division, and modulo second; and addition and subtraction last. You can use parentheses to alter this precedence. Netezza SQL evaluates operators of the same precedence in a left-to-right manner unless you use parentheses.

Table 3-11 describes the Netezza SQL operators.

Table 3-11: Operators

Operator	Symbol
Binary Arithmetic Operators	
Addition	+
Subtraction	-
Multiplication	*
Division	/
Exponentiation	^ or **
Modulo	%
Unary Arithmetic Operators	
Plus	+
Minus	-
Factorial	!
Binary Text Operator	
Concatenate	||
Relational Operators	
Equal	=
Not equal	<> or !=
Greater than	>
Greater than or equal	>=
Less than	<
Less than or equal	<=

Netezza follows Postgres operator precedence rules. Table 3-12 lists the precedence and associativity of the available operators.

Table 3-12: Operator Precedence

Operator/Element	Associativity	Description
.	Left	Table/column name separator
::	Left	Typecast (PostgreSQL extension to standard SQL)
-	Right	Unary minus
^	Left	Exponentiation
* / %	Left	Multiplication, division, modulo
+ –	Left	Addition, subtraction
IS		IS TRUE, IS FALSE, IS UNKNOWN, IS NULL, IS NOT NULL
ISNULL		Test for null (PostgreSQL extension to standard SQL)
NOTNULL		Test for not null (PostgreSQL extension to standard SQL.)
(any other)	Left	All other native operators
IN		Set membership
BETWEEN		Containment
OVERLAPS		Time interval overlap
LIKE		String pattern matching
< <= > >=		Less than, less than or equal to, greater than, greater than or equal to
=	Right	Equality, assignment
NOT	Right	Logical negation
AND	Left	Logical conjunction
OR	Left	Logical disjunction

Functions

Functions allow you to access specified routines from SQL. They take one or more arguments and return a result. Table 3-13 describes the functions.

Table 3-13: Functions

Name	Description

case	*Searched* form:
	`CASE`
	`  WHEN <search-condition-1> THEN <result-1>`
	`  WHEN <search-condition-2> THEN <result-2>`
	`...`
	`  WHEN <search-condition-n> THEN <result-n>`
	`  ELSE <default-result>`
	`END`
	Search conditions can be arbitrarily complex and results can be expressions.

case	*Value* form:
	`CASE <test-value>`
	`  WHEN <comparand-value-1> THEN <result-1>`
	`  WHEN <comparand-value-2> THEN <result-2>`
	`...`
	`  WHEN <comparand-value-n> THEN <result-n>`
	`  ELSE <default-result>`
	`END`
	Test values, comparand values, and results can be expressions.

nullif(a,b)	Returns a null value if a=b, otherwise it returns a.
coalesce(arg1, arg2, ...)	Returns its first non-null argument or null if all arguments are null. isnull is a synonym for SQL Server compatibility.
cast (<value> as <type>)	Is available to convert from one data type to another data type. For more information, see "Cast Conversions" on page 3-14.
extract (field from <datetime value>)	Extracts a numeric datetime or time zone field from a datetime or interval value. For example, extract(year from <datetime-value>). For a list of valid arguments, see "Extract Date-Time Value" on page 3-14. For syntax, see Table B-122 on page B-140.
date_part ('field', <datetime value>)	Similar to extract, extracts a numeric datetime or time zone field from a datetime or interval value. For example, date_time('day', <datetime-value>). For a list of valid arguments, see "Extract Date-Time Value" on page 3-14. For syntax, see Table B-122 on page B-140.
nvl(x,y)	Returns the first argument if it is not null, otherwise it returns the second argument. For example, nvl(hire_date, current_date) returns the current_date if the hire_date is null. nvl is equivalent to the SQL coalesce function, and is short hand for the case expression "case when x is not null then x else y end." For more information, see "NVL Example" on page 3-15.

Table 3-13: Functions (continued)

Name	Description
nvl2(x,y,z)	Returns the second argument if the first argument is not null, otherwise it returns the third argument. nvl2 is short hand for the case expression "case when x is not null then y else z end." For more information, see "NVL2 Example" on page 3-16.
decode(<expr>, <search1>,<result1>, ...<search N>, <result N>, <default>)	Compares the expr to each search value. If the expr is equal to the search, decode returns the result. If there is no match, decode returns the default, or if the default is omitted, returns null. For more information, see "Decode Example" on page 3-16.

CASE

The SQL standard disallows the use of non-deterministic functions, like the NPS random() function, in CASE expression WHEN clauses. But, as an extension to the standard, NPS SQL permits such usage. Use caution with this type of expression, as the behavior might not be what is expected.

Example:

```
SELECT CASE WHEN random() = .1 THEN 'A' WHEN random() = .2 THEN 'B'
ELSE 'C' END FROM tblA
```

The system evaluates the expression in the following manner:

- Generate a random number
- If the generated number is .1 then return 'A'
- Generate a second random number
- If the newly generated number is .2 then return 'B'
- Otherwise, return 'C'

So the system evaluates the random() function separately when evaluating each WHEN expression. This is important when non-deterministic functions like random() are involved, and each execution can return a different value.

If you wanted a different behavior in the previous example, you can use a subquery, as in the following example:

```
SELECT CASE WHEN rand = .1 THEN 'A' WHEN rand = .2 THEN 'B' ELSE 'C'
END
FROM (SELECT random() rand FROM tblA LIMIT ALL) subset
```

The LIMIT ALL in the subquery prevents it from being pulled up into the parent query, and the random() function is invoked only once for each row of tblA, and therefore the same random() result is tested in each WHEN clause.

The previous example used a form of CASE expression that the standard calls a searched CASE. But SQL offers another form of CASE expression, called simple CASE.

Original example in simple CASE form:

```
SELECT CASE random() WHEN .1 THEN 'A' WHEN .2 THEN 'B' ELSE 'C' END
FROM tblA
```

In this form, it looks like the random() function is invoked once. But the standard says that the simple CASE expression means exactly the same thing as the more verbose searched CASE equivalent. So the system handles both of these examples in the same way, and the same cautions apply.

There are other CASE variants like NULLIF, NVL, NVL2, COALESCE, and DECODE that are converted to CASE expressions, and so show the same behavior and call for the same caution.

Cast Conversions

You can use cast(<value> as <datatype>) to cast a data type from one type to another type. For example, you could convert any numeric data type (byteint, smallint, int, bigint, numeric/decimal, float, double) to any other numeric datatype. The <value> field can be a column or an expression.

In addition to the cast function, Netezza offers additional datatype conversions as described in Table 3-14.

Table 3-14: Datatype Conversions

Data Type	Function
numeric to a string	to_char(numeric,text)
real or double precision to a string	to_char(double precisions, text)
timestamp to a string	to_char(timestamp,text)
string to date	to_date(text,template)
string to numeric	to_number(text,template)
string to timestamp	to_timestamp(text,text)

You can also convert from one date, time, or timestamp to another. For example,

```
NZSQL nzdw -a -x <  x.sql
```

To convert a timestamp to a date, use CAST. For example,

```
CAST(<timestamp_value> AS DATE);
```

Extract Date-Time Value

Table 3-15 describes the date and time values:

Table 3-15: Date-Time Values

Value	Description
epoch	The number of seconds since 1970-01-01 00:00:00-00 The value can be positive or negative.
year/years	The year field, such as 2007

Table 3-15: Date-Time Values (continued)

Value	Description
quarter	The quarter of the year (1 to 4) that the specified day is in.
month/months	The number of the month within the year, from 1 to 12.
week	The number of the week of the year (1-53) that the specified day is in. The value uses the ISO-8601 definition of a week, which begins on Monday; as a result, some years may have 53 weeks, and sometimes the first few days of January could be included as part of the 52nd or 53rd week of the previous year.
day/days	The day of the month, from 1 to 31.
dow	The day of the week, from 1 (Sunday) to 7 (Saturday).
doy	The day of the year, from 1 to 366.
hour/hours	The hour of the day, from 0 to 23.
minute/minutes	The minute of the hour, from 0 to 59.
second/seconds	The second or the minute, from 0 to 59
millisecond/milliseconds	The seconds field, including fractional parts, multiplied by 1000. Note that this includes full seconds.
microsecond/microseconds	The microsecond field, including fractional parts, multiplied by 1000000. Note that this includes full seconds.

For data type intervals, see “Netezza SQL Interval Support” on page 3-6.

Note: Netezza SQL does not support timezone* values.

For example:

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2007-02-14 20:38:40');
```
Result: 14

```
SELECT EXTRACT(MILLISECONDS FROM TIME '12:15:06');
```
Result: 6000
```
SELECT DATE_PART('DAY', DATE '2007-02-18');
```
Result: 18

NVL Example

In this example, when selecting the title and price for all books, if the price for a title is NULL, the price displays as 0.00

```
SELECT title, nvl(price, 0.00) AS price FROM titles;
```

Note: The nvl function is equivalent to the SQL coalesce function. The result of this expression is type compatible with both expr1 and expr2, just as in the coalesce function.

NVL2 Example

In this example, the first argument is not null, so it returns the second argument.

```
SELECT nvl2(1,2,3);
NVL2(1,2,3)
-----------
          2
```

Note: The nvl function is equivalent to a case expression. The result of the expression is type compatible with the second and third arguments, just as it is in the corresponding case function.

Decode Example

Use the decode function to create an if-then-else statement. In this example, if the color ID is 1000 the result is red, if 1001, blue, if 1002, yellow, otherwise it is none.

```
SELECT color_id,
DECODE (color_id, 1000, 'red', 1001, 'blue', 1002, 'yellow', 'none')
AS color_name
FROM colors;
```

Note: Netezza SQL implements the decode function as a variant of a simple case expression, and it is equivalent to the expression "case x when val1 then result 1 when val2 then result 2 else default end." Except If both x and val1 contain NULL values, unlike a case expression, decode considers the NULL values to be equal.

Aggregate Functions

Aggregates compute a result value from a set of values. Table 3-16 describes the five Netezza SQL aggregate functions.

Table 3-16: Aggregate Functions

Name	Description
count	Counts all rows
sum	Provides a total
max	Provides the maximum value
min	Provides the minimum value
avg	Averages the values

Table 3-17 describes data type functions and their associated aggregates.

Table 3-17: Data Types for Aggregates

Type	Name
All types	count, max, min

Table 3-17: Data Types for Aggregates (continued)

Type	Name
Exact numeric	avg, max, min, sum, count **Note:** The averages of exact numeric types are computed to 6 decimal places.
Approximate numeric	avg, max, min, sum, count

Standard String Functions

String functions allow you to manipulate text strings. Table 3-18 describes the standard string functions.

Table 3-18: Standard String Functions

Function	Description
Case conversion	Done through lower (<character-value>) and upper (<character-value>). Note that for NFC characters, upper and lower use the data specified in UnicodeData.txt. They do not use the conversion rules in the Unicode Consortium's SpecialCasing rules (http://www.unicode.org/Public/4.1.0/ucd/SpecialCasing.txt).
trim	Removes leading or trailing occurrences of characters from a string. For example, trim({leading | trailing | both} [<character> | ' '] from {<character-value>}).
Position	Finds one string within another. The value 0 indicates not found. For example, position(<character-value> in <character-value>).
Substring	Extracts one string from another. For example, Substring(<character-value> from <start-position> [for <length>]). The first character is at position 1. • If you do not specify <length>, then the rest of <character-value> is implied. • If <start-position> is beyond the end of <character-value>, then Netezza SQL returns an empty string. • If <start-position> is negative or zero, then the start is at an imaginary position before the first character of <character-value>.
Like/not like	Provides pattern matching comparisons. Netezza SQL supports the standard pattern characters: %, _ and the escape character by default. For more information about using like, see "Pattern Matching" on page 3-18.
Character length	Character_length(<character-value>) returns the length of the string in <character-value>.

Note: The Netezza SQL string comparison ignores trailing spaces.

Pattern Matching

Every pattern defines a set of strings. The LIKE expression returns true if the string is contained in the set of strings represented by pattern. Consequently, the NOT LIKE expression returns false if LIKE returns true, and vice versa. An equivalent expression is NOT (string LIKE pattern).

If pattern does not contain percent signs or underscores, then the pattern only represents the string itself; in that case LIKE acts like the equals operator. An underscore (_) in pattern stands for (matches) any single character; a percent sign (%) matches any string of zero or more characters.

For example:

```
'abc' LIKE 'abc'    true
'abc' LIKE 'a%'     true
'abc' LIKE '_b_'    true
'abc' LIKE 'c'      false
```

LIKE pattern matches always include the entire string. To match a pattern anywhere within a string, the pattern must therefore start and end with a percent sign.

To match a literal underscore or percent sign without matching other characters, you must precede the respective character in pattern with the escape character. The default escape character is the backslash, but you can choose a different character by using the ESCAPE clause. To match the escape character itself, enter two escape characters.

- To escape the % character, use a backslash, for example:

```
SELECT * FROM table WHERE col LIKE '%90\%%'
```

- If you cannot use the backslash character, then designate another ASCII character as the escape character, for example:

```
SELECT * FROM table WHERE col LIKE '%90#%%' escape '#'
```

It is also possible to select no escape character by entering ESCAPE '' (empty single quotes). In this case, there is no way to turn off the special meaning of underscore and percent signs in the pattern.

Note: Netezza SQL does not support ILIKE (case insensitive search) SQL operators. However you can always use UPPER() or LOWER() to do case insensitive searches. For example:

```
WHERE UPPER(first_name) LIKE 'PAT%'
```

Fuzzy String Search Functions

The Netezza SQL language supports two fuzzy string search functions: Levenshtein Edit Distance and Damerau-Levenshtein Edit Distance. A fuzzy string search is a form of approximate string matching based on defined techniques or algorithms. These functions compare two strings to show how similar or different they are. These functions support VARCHAR and CHAR data types (Latin9 encoding) only.

Levenshtein Edit Distance SQL syntax:

```
<int4 value> = le_dst(<str_expr_1>, <str_expr_2>)
```

The return value indicates how different the two input strings are, calculated according to the Levenshtein edit distance algorithm. A value of 0 indicates that the strings are equivalent without any modifications. The algorithm computes the number of modifications that

are required to change the first string into the second string. The strings are case-sensitive. A modification is a change such as an addition, deletion, letter case-change, or substitution of a single character.

For example, `le_dst('sow','show')` returns a value of 1 (the addition of the character h); `le_dst('hello','Hollow')` returns a value of 3 (the substitution of e for o, the capitalization of H, and the addition of w).

Because the string comparisons are case-sensitive, you can use functions such as upper() and lower() to change the letter casing of strings prior to the comparison and ignore case-change modifications. For example, `select le_dst('Smith','SMYTH')` returns a value of 4 (three uppercase letter changes and a letter substitution). The function `select le_dst(upper('Smith'),'SMYTH')` returns a value of 1 (the I/Y letter substitution).

Damerau-Levenshtein Edit Distance SQL syntax:

```
<int4 value> = dle_dst (<str_expr_1>, <str_expr_2>)
```

The value returned indicates how different the two input strings are, calculated according to the Damerau-Levenshtein edit distance algorithm. The strings are case-sensitive. Similar to the Levenshtein algorithm, a modification is a change such as an addition, deletion, letter case-change, or substitution of a single character. However, in the Damerau-Levenshtein algorithm, a character transposition change such as 'two' to 'tow' counts as one change, not two. A value of 0 indicates the strings are equivalent without any modifications. Similar to the le_dst() function, the string comparisons are case-sensitive; you can use functions such as upper() and lower() to change the case of the input strings prior to the comparison and ignore case-change modifications.

Phonetic Matching Functions

The Netezza SQL language supports two phonetic matching functions that allow you to encode names into phonetic representations using the SoundEx NYSIIS or Double Metaphone algorithms. By encoding names phonetically, you can match names based on their pronunciation and reduce misses that might result from spelling variations. The phonetic matching functions are case-insensitive comparisons; the phonetic representations are the same for two strings that have the same spelling but different letter casing. These functions support VARCHAR and CHAR data types — and specifically the ASCII subset of Latin9 encoding — only. The functions ignore any characters outside the ASCII subset.

Note: If you use strings that call the phonetic functions with characters beyond the ASCII range, you should transliterate the strings to convert accented characters to their ASCII unaccented versions.

SoundEx NYSIIS SQL syntax:

```
<varchar(6) value> = nysiis(<str_expr>)
```

Soundex is a well-known phonetic algorithm for indexing names by sound as pronounced in English. This function converts a string into its Soundex representation using the New York State Identification and Intelligence System (NYSIIS) variation of Soundex. The return value is a string of up to 6 characters that identifies the pronunciation of the input string. For example, the function `nysiis('Washington')` returns the string 'wasang', while the function `nysiis('brown')` returns the value 'bran'.

Double Metaphone SQL syntax:

```
<int4 value> = dbl_mp(<str_expr>)
```

Double Metaphone is another phonetic algorithm for indexing strings by their pronunciation. Similar to Soundex, it uses a different set of rules for English as well as alternate pronunciation. The function returns two 4-character string encodings—a primary key and secondary (or alternate) key—for pronunciation of the input string. Similar sounding words share the same keys, though they may be of variable length and spelling. For example the double metaphone primary and secondary keys for the name 'washington' are 'AXNK' and 'FXNK'.

For improved performance, the dbl_mp function maps the 4-character keys to 16-bit numbers and returns a composite 32-bit value (Netezza type int4) that holds both the 16-bit primary and secondary keys. So, the function `dbl_mp('washington')` returns the value 781598358.

There are three helper functions (pri_mp, sec_mp, and score_mp) that you can use to extract the primary and secondary keys as strings from the return value, as well as to perform key comparisons for scoring relevance.

Primary Metaphone SQL Syntax:

```
<varchar(4) value> = pri_mp(<int4 dbl_mp return value>)
```

This helper function takes the value returned by a call to the dbl_mp function and returns the corresponding 4-character primary metaphone string. For example, `pri_mp(781598358)` returns the primary key AXNK.

Secondary Metaphone SQL Syntax:

```
<varchar(4) value> = sec_mp(<int4 dbl_mp return value>)
```

This helper function takes the value returned by a call to the dbl_mp function and returns the corresponding 4-character secondary metaphone string. For example, `sec_mp(781598358)` returns the secondary key FXNK.

Score Metaphones SQL Syntax:

```
<varchar(4) value> = score_mp(<int4 dbl_mp value 1>, <int4 dbl_mp
value 1>, <int4 strong match value>, <int4 normal match value>,
<int4 minor match value>, <int4 no match value>)
```

This helper function takes two values returned by the dbl_mp function and compares them to determine how closely they match. The last four arguments are the values to return for the four possible scoring or matching outcomes. With double-metaphone encoding, the two input values are said to be matches when one or more of their primary or secondary keys match; the strength of the match depends upon which keys match. For example, the following chart shows how to evaluate the strength of the match:

- **Strongest Match:** Primary Key (1) = Primary Key (2)
- **Normal Match:** Secondary Key (1) = Primary Key (2)
  Primary Key (1) = Secondary Key (2)
- **Minimal Match:** Secondary Key (1) = Secondary Key (2)

For the four match value arguments, you can specify values such as 1, 2, 3, and 4 (for strongest, normal, minimal, or no match). You could also use weighted values such as 100, 50, 25, and 0 to return more points for better match results.

For example, if you compare the double metaphone encodings for ‘washington’ (781598358) and ‘wachingten’ (7815963100), you could use the following score_mp function to determine how closely they match:

```
score_mp(781598358,781596310,1,2,3,4)
```

The function returns the value 1, which indicates a strongest match.

If you compare the encodings for ‘washington’ and ‘vachingten’ (1050031766):

```
score_mp(781598358,1050031766,100,50,25,0)
```

The function returns the value 50, which indicates a normal match.

Value Functions and Reserved/Nonreserved Keywords

The value functions are built-in functions. They are niladic (or nullary; that is, they have no arguments), and return information about the system. You can use built-in functions anywhere that you would use a constant.

SQL makes a distinction between reserved and nonreserved keywords. Reserved keywords are never allowed as regular identifiers. Nonreserved keywords have a special meaning only in a particular context and can be used as identifiers in other contexts. Most nonreserved keywords are actually the names of built-in tables and functions. For a list of all the reserved and nonreserved words, see Table A-1 on page A-1.

Table 3-19 lists the value functions and compares whether each is reserved or nonreserved in Netezza SQL and SQL-92.

Table 3-19: Key Words

Key Word	Netezza SQL	SQL-92
current_date	Reserved	Reserved
current_time(p)	Reserved	Reserved
current_timestamp(p)	Reserved	Reserved
current_catalog	Reserved	Reserved
current_user	Reserved	Reserved
session_user	Reserved	Reserved
current_db	Reserved	Nonreserved
current_userid	Reserved	Nonreserved
current_useroid	Reserved	Nonreserved

Note: Current_time and current_timestamp which return microseconds, can return extra digits of precision by giving (p) a value of 0-6, with zero as the default.

Netezza SQL Extensions

The sections describes the Netezza SQL extensions. Note that in cases where the standard command arguments and return values apply, this section lists only the function names.

Math Functions

The following sections describe the trigonometric, random number, miscellaneous, and binary math functions.

Trigonometric Functions

Table 3-20 describes the trigonometric functions.

Table 3-20: Trigonometric Functions

Function	Description
acos(x)	inverse cosine
asin(x)	inverse sine
atan(x)	inverse tangent
atan2(x,y)	inverse tangent of x/y
cos(x)	cosine
cot(x)	cotangent
degrees(dp)	radians to degrees
pi()	pi constant
radians(dp)	degrees to radians
sin(x)	sine
tan(x)	tangent

Random Number Functions

Table 3-21 describes the random number functions.

Table 3-21: Random Number Math Functions

Function	Return Type	Description	Example	Result
random()	dp	random value between 0.0 and up to but not including 1.0	random()	—
setseed(dp)	integer	set seed for subsequent random() calls	setseed(0.54823)	1177314959

Miscellaneous Math Functions

Table 3-22 describes the miscellaneous math functions.

Table 3-22: Miscellaneous Math Functions

Function	Return Type	Description	Example	Result
N! (factorial) n can be any size integer	int8	Computes N x (N-1) x... x 1	5 !	120
abs(x)	same as x	absolute value	abs (-17.4)	17.4
ceil(numeric)	numeric	smallest whole number no less than argument	ceil (-42.8)	-42
dceil(double)	double	smallest whole number no less than argument	dceil (42.8)	43
dfloor(double)	double	largest integer not greater than argument	floor(42.8)	42
exp(dp or numeric)	same as input	exponential	exp(1.0)	2.71828182845905
floor(numeric)	numeric	largest integer not greater than argument	floor(-42.8)	-43
fpow(a real, b real)	real	a raised to the power of b	pow(9.0, 3.0)	729
ln(double)	double	This is natural logarithm	ln(2.0)	0.693147180559945
log(numeric)	numeric	This is base 10 logarithm	log(100.0)	2

Table 3-22: Miscellaneous Math Functions (continued)

Function	Return Type	Description	Example	Result
mod(x,y)	If either x or y is a double precision or a real, then the result is a double precision; otherwise, if either x or y is a numeric, then the result is a numeric; otherwise, the two args are some integer type and the result is the same as the wider of the two arguments	Computes the remainder of x/y	mod(9,4)	1
numeric_sqrt(numeric)	numeric	square root	numeric_sqrt(2)	1.4142
pow(a dp, b dp)	dp	a raised to the power of b	pow(9.0, 3.0)	729
round(dp or numeric)	same as input	round to nearest integer	round(42.4)	42
round(v numeric, s integer)	numeric	round to s decimal places	round(42.4382,2)	42.44
sign(numeric)	numeric	sign of the argument(-1,0,+1)	sign(-8.4)	-1
sqrt(double)	double	square root	sqrt(2.0)	1.4142135623731
trunc(dp or numeric)	same as input	truncate toward zero	trunc(42.8)	42
trunc(v numeric,s integer)	numeric	truncate to s decimal places	trunc(42.4382,2)	42.43

Note: The result of the pow function must fit in a FLOAT8 (be a double precision floating point value). A FLOAT8 can hold positive or negative values with a magnitude as large as 1.798×10^{308} and as small as 4.941×10^{-324}(approximately).

Binary Math Functions

Table 3-23 describes the binary math functions. In this table *N* represents 1,2,4, or 8 and denotes the byte size of the integer data type that Netezza operates on and returns.

Table 3-23: Binary Math Functions

Function	Description
intNand(arg1, arg2)	Bitwise AND of arg1 and arg2
intNor(arg1, arg2)	Bitwise OR of arg1 and arg2
intNxor(arg1, arg2)	Bitwise exclusive OR of arg1 and arg2
intNnot(arg1)	Bitwise NOT of arg1
intNshl(arg1, arg2 [,arg3])	Shift left, with optional mask. arg1 is ANDed with arg3 (if present) then shifted left by arg2 bits
intNshr(arg1, arg2 [,arg3])	Shift right, with optional mask. arg1 is ANDed with arg3 (if present) then shifted right by arg2 bits

Character Functions

Table 3-24 describes the character functions.

Table 3-24: Character Functions

Name	Description
ascii(s)	Returns numeric ASCII value of first character in s. For the NCHAR version, see unicode(s).
btrim(s)	Trims spaces from both ends of string s.
btrim(s,t)	Trims occurrences of the characters in t string from both ends of string s.
chr(n)	Returns the character with ASCII value n. For NCHAR version, see unichar(n).
initcap(s)	Capitalizes the first character of each word of string s.
instr(s1,s2[,n[,m]])	Returns the location of a substring in a string. The function also supports nchar and nvarchar character strings. s1 specifies the string to search. s2 specifies the substring to search for within s1. n is an optional argument that specifies the position in s1 where the search will start. When positive, n specifies the count from the start of the string and searches left to right. When negative, n specifies the count backward from the end of the string, and searches right to left. m is an optional argument that specifies to search for *m*th occurrence of the substring s2 in s1. The return value is an integer that specifies the position of substring s2 in s1.

Table 3-24: Character Functions (continued)

Name	Description
length(s)	Returns the length of string s.
lower(s)	Converts string s to lowercase.
lpad(s, n)	Spaces pad string s on left to length n. There is an optional third argument (t) that specifies the pad char. If the length argument is shorter than the string being padded, the system truncates the string to the specified length.
ltrim(s)	Trims spaces from left end of string s.
ltrim(s,t)	Trims occurrences of the characters in t string from left end of string s.
repeat(s,n)	Repeats string s n times. If the resulting string is greater than the maximum varchar length of 64,000 characters, Netezza truncates the result to 64,000.
rpad(s, n)	Spaces pad string s on right to length n. There is an optional third argument (t) that specifies the pad char. If the length argument is shorter than the string being padded, the system truncates the string to the specified length.
rtrim(s)	Trims spaces from right end of string s.
rtrim(s,t)	Trims occurrences of the characters in t string from right end of string s.
strpos(s, b)	Specifies starting position of substring b in string s.

Table 3-24: Character Functions (continued)

Name	Description
substr(s,p,l)	Returns a substring of strings that begin at position p and is the size of l characters. The ansi_substring configuration variable controls the behavior of the substr() built-in function and its start position when the value is negative. The variable has the following values: • When ansi_substring = 1 (set to true), a negative start position causes the substring function to "insert" empty characters before the start of the string. The number of empty characters inserted is 1 plus the absolute value of the start position. This is the default value. • When ansi_substring = 0 (set to false), a negative start position value causes the string to start on the right-most character of the string, and to include those characters that continue to the left for the specific length. A start position of 0 is treated as a start position of 1. The following is an example of ansi_substring usage. In all cases, one row is returned, though the result may vary: • SELECT substr ('left and right', 1, 4): – Result if True: left – Result if False: left • SELECT substr ('left and right', 0, 4): – Result if True: lef – Result if False: left • SELECT substr ('left and right', -5, 1): – Result if True: – Result if False: r
translate(s,from, to)	Replaces any character in *s* that matches a character in the *from* set with the corresponding character in the *to* set. For example, translate('12345','14','ax') returns 'a23x5'.
upper(s)	Converts string s to uppercase.
unichr(n)	Returns the character with the ASCII value n. Equivalent to the chr() function. The function verifies that the codepoints are in the valid ranges, and displays an error if the codepoints are in the invalid range of U+D800-U+DFFF or in decimal 55,296-57,343.
unicode(s)	NCHAR version of ascii(). Returns the Unicode value of the first character in the string s. A separate function is defined because six characters have different values between Latin9 and Unicode.

Table 3-24: Character Functions (continued)

Name	Description
unicodes(s, unit, base)	Returns the Unicode value for every character in the string s. By default, if you specify only the string, the function returns the representation in UTF-32 hex digits. The unit value specifies 8, 16, or 32 to return the value in UTF-8, UTF-16, or UTF-32 protocol. The base value specifies 'oct', 'dec', or 'hex' in upper- or lowercase (or 8, 10, or 16) to control the number base.

Date/Time Functions

Table 3-25 describes the date/time functions. For syntax, see Table B-122 on page B-140. For interval datatypes, see "Netezza SQL Interval Support" on page 3-6. For data/time units, see "Extract Date-Time Value" on page 3-14.

Table 3-25: Date/Time Functions

Type	Description
add_months (d, n)	The function returns the date d plus the n months. You can use any integer for n months. If d is the last day of the month, or if the resulting month has fewer days than the day component, then the result is the last day of the resulting month. Otherwise, the result has the same day component as d.
age(t,t)	• Returns the interval between two timestamps. If you use a single timestamp, the age function returns the interval between the current time and the timestamp. • The interval returned by the age function can include year/month data as well as day/time data. For example, select age('10-22-2003', '7-6-2002') returns 1 year 3 months 16 days. • If you use the age function with a single argument, as in age(<timestamp>) then that is equivalent to age(CURRENT_TIMESTAMP, <timestamp>). Note that Netezza SQL interval support is nonstandard.
date_part(units,col)	Extracts the subfield from date/time value or extracts the subfield from interval value. It returns the units part of col.
date_trunc(units, col)	Truncates the date to a specified precision. It returns col rounded to units.
extract(units FROM col)	Extracts the subfield from date/time value or the subfield from interval value. Same as date_part().
last_day(date)	Returns the last day of the month that is specified in the date value.

Table 3-25: Date/Time Functions (continued)

Type	Description
months_between(d1, d2)	The function returns the number of months between dates d1 and d2. • If d1 is later than d2, the result is positive. • If d1 is earlier than d2, the result is negative. • If d1 and d2 are either the same days of the month or both the last days of months, the result is always an integer. • Otherwise, the function calculates the fractional portion of the result based on a 31-day month and considers the difference in time components of d1 and d2.
next_day(*date*, *weekday*)	Returns the date of the weekday following a particular date. The *date* value specifies a date, either as a date or a timestamp. The returned value matches the format of the date value. The *weekday* value is a day of the week ('SUN', 'MON', 'TUE', 'WED', 'THU', 'FRI', 'SAT'). It is a string literal and must be enclosed in quotes, and you can use either uppercase or lowercase letters. Any characters specified after the third character of the weekday string are ignored.
now()	The function is the same as current_timestamp. For more information, see "Conversion Functions" on page 3-29.
overlaps	The function determines if two time intervals overlap. Netezza SQL supports the standard SQL overlaps predicate.
timeofday()	The function is the verbose string version of current_timestamp. For example, Mon Dec 01 16:12:05 2003 EST. For more information, see "Conversion Functions" on page 3-29.

Conversion Functions

You can use the Netezza SQL formatting functions to convert data types (date/time, integer, floating point, numeric) to formatted strings and to convert from formatted strings to specific data types. These functions all use a common calling convention: the first argument is the value to be formatted, and the second argument is a template that defines the output or input format.

Note: The to_char type casts the *date* datatype to a *timestamp* datatype internally.

Table 3-26 describes the conversion functions.

Table 3-26: Conversion Functions

Type	Description	Examples
to_char(timestamp, text)	Converts time stamp to string.	to_char(now(),'HH12:MI:SS')

Table 3-26: Conversion Functions (continued)

Type	Description	Examples
to_char(double precision, text)	Converts real/double precision to a string.	to_char(125, '999')
to_char(numeric, text)	Converts numeric to a string.	to_char(numeric '-125.8', '"999D99S"')
to_date(text,template)	Converts a string to date.	to_date('05 Dec 2000','DD Mon YYYY')
to_number(text,template)	Converts a string to a number. Note that the output of to_number is a numeric. Its precision and scale are indicated by the digit and decimal point placeholders in the format string.	to_number('12,454.8-', '99G999D9S')
to_timestamp(text,text)	Converts a string to time stamp.	to_timestamp('05 Dec 2000', 'DD Mon YYYY')

Template Patterns for Date/Time Conversions

Netezza SQL recognizes certain patterns in the output template and replaces them with the appropriately formatted data. Any text in the format string of a to_char function that is not in a template pattern is passed through if it is enclosed in double quotes, for example, to_char (125, '"The number is "999').

Table 3-30 describes the date/time conversions.

Table 3-27: Template for Date/Time Conversions

Pattern	Description
HH	Hour of day (01:12).
HH12	Hour of day (01:12).
HH24	Hour of day (00:23).
MI	Minute (00:59).
SS	Second (00:59).
SSSS	Seconds past midnight (0:86399).
MS	Milliseconds (00:00.999)
US	Microseconds (00:00.999999)
AM or A.M. or PM or P.M.	Meridian indicator (uppercase).
am or a.m. or pm or p.m.	Meridian indicator (lowercase).

Table 3-27: Template for Date/Time Conversions (continued)

Pattern	Description
Y,YYY	Year (4 and more digits) with a comma.
YYYY	Year (4 and more digits).
YYY	Last 3 digits of the year.
YY	Last 2 digits of the year.
Y	Last digit of the year.
BC or B.C. or AD or A.D.	Era indicator (uppercase).
bc or b.c. or ad or a.d.	Era indicator (lowercase).
MONTH	Full uppercase month name (blank-padded to 9 chars).
Month	Full mixed case month name (blank-padded to 9 chars).
month	Full lowercase month name (blank-padded to 9 chars).
MON	Abbreviated uppercase month name (3 chars).
Mon	Abbreviated mixed case month name (3 chars).
mon	Abbreviated lowercase month name (3 chars).
MM	Month number (01:12).
DAY	Full uppercase day name (blank-padded to 9 chars).
Day	Full mixed case day name (blank-padded to 9 chars).
day	Full lowercase day name (blank-padded to 9 chars).
DY	Abbreviated uppercase day name (3 chars).
Dy	Abbreviated mixed case day name (3 chars).
dy	Abbreviated lowercase day name (3 chars).
DDD	Day of the year (001:366).
DD	Day of the month (01:31).
D	Day of the week (1:7; SUN=1).
W	Week of the month (1:5) where first week start on the first day of the month.
WW	Week number of the year (1:53) where the first week starts on the first day of the year.
IW	ISO week number of the year (the first Thursday of the new year is in week 1).
CC	Century (2 digits).

Table 3-27: Template for Date/Time Conversions (continued)

Pattern	Description
J	Julian Day (days since November 24, 4714 BC).
Q	Quarter
RM	Month in Roman Numerals (I-XII; I=January) — uppercase.
rm	Month in Roman Numerals (i-xii; i=January) — lowercase.

You can apply the following modifiers to any template pattern to alter its behavior. Table 3-28 describes these modifiers.

Table 3-28: Template Modifiers

Modifier	Description	Example
FM prefix	Fill mode (suppresses padding blanks and zeroes).	FMMonth
TH suffix	Add uppercase ordinal number suffix.	DDTH
th suffix	Add lowercase ordinal suffix.	DDth
FX prefix	Fixed format global option.	FX Month DD Day

Usage notes:

- The FM prefix suppresses leading zeroes or trailing blanks that Netezza SQL would otherwise add to make the output of a pattern be fixed width.
- Normally the to_timestamp and to_date types skip multiple blank spaces. If you specify the FX prefix, Netezza SQL does not skip blank spaces. Note that you must specify the FX prefix as the first item in the template.
- To output pattern keywords as literal text, put the substring in double quotes; for example '"Hello Year:"YYYY'. Netezza SQL will replace YYYY with the year data, but will not interpret the single Y.
- To output a double quote, precede it with a backslash; for example, '\"YYYY Month\"'.

Table 3-29 describes the template patterns for numeric conversions.

Table 3-29: Template Patterns for Numeric Conversions

Pattern	Description
9	Value with the specified number of digits.
0	Value with leading zeros.
. (period)	Decimal point.
, (comma)	Group (thousand) separator.

Table 3-29: Template Patterns for Numeric Conversions (continued)

Pattern	Description
PR	Negative value in angle brackets.
S	Negative value with minus sign (uses locale).
L	Currency symbol (uses locale).
D	Decimal point (uses locale).
G	Group separator (uses locale).
MI	Minus sign in the specified position (if number < 0).
RN	Roman numeral (input between 1 and 3999).
V	Shift n digits (see notes).

Usage notes:

- A sign formatted using the MI pattern is not an anchor in the number; for example, to_char(-12, 'S9999') produces ' -12', but to_char(-12, 'MI9999') produces '- 12'.
- The V pattern effectively multiplies the input values by 10^n, where n is the number of nines following V. The to_char type does not support the use of the V pattern combined with a decimal point. For example, 99.9V99 is not allowed.

Miscellaneous Functions

Table 3-30 describes the miscellaneous functions.

Table 3-30: Miscellaneous Functions

Type	Description
isfalse(c)	Is condition c not true or unknown?
isnotfalse(c)	Is condition c true or unknown?
istrue(c)	Is condition c not false or unknown?
isnottrue(c)	Is condition c false or unknown?
version()	Specifies the Netezza SQL database version string?
get_viewdef('v')	Displays the SQL code that created the view?
width_bucket	Takes a range and divides it into equi-width intervals, specified by num-buckets.

Netezza SQL Functional Categories

All SQL commands belong to one of the following functional categories:

- Data Definition Language (DDL)

- Data Control Language (DCL)
- Data Manipulation Language (DML)
- Transaction Control
- Miscellaneous commands

During database initialization, Netezza SQL uses DDL, DCL, and DML commands to create and populate an initial database.

- DDL creates the system databases (system and master_db) and users (admin and public), tables, and views.
- DCL grants privileges on database objects.
- DML retrieves and places values in the database tables.

The remainder of this section provides descriptions of these functional categories.

Data Definition Language

The Netezza SQL Data Definition Language (DDL) allows you to define, modify, and delete databases objects, such as databases, tables, and views. Netezza SQL uses DDL to manage (create, alter and drop) all the objects in SQL databases. The database objects that the DDL manipulates fall into two categories:

- Global objects — objects global to all databases. Database, user, and group objects are examples of global objects.
- Local objects — objects that reside in a particular database. Table and view objects are examples of local objects.

When you create a database object, you must name it. Database object names can be up to 128 bytes in length. For a description of the valid identifier characters and formatting rules, see "Handling SQL Identifiers" on page 2-7. You cannot use a global object name for a user-defined object. You can create local objects with the same name in different databases.

The Netezza SQL system tables are called the *system catalog*, which is global in scope. The system catalog contains all the metadata for all objects within all databases (global and local). When you enter DDL commands, Netezza SQL changes the system catalog to reflect the request.

When you create a database, Netezza SQL copies the template database master_db. Master_db is a special database and should not be modified, altered, or have user objects created within it. Note that only one person at a time can be connected to the master_db database to create databases.

User and group database objects are global in scope; that is, they are not tied to a particular database. There is a predefined group called *public*. As you create users, they are automatically added to the group public. You cannot remove users from the group public, or drop the group public.

Groups are designed to allow security administrators to group users by department or functionality. Groups are used to control user privileges (refer to DCL for more information). Users can be members of many groups; however, groups cannot be members of other groups.

Table 3-31 describes the Netezza SQL DDL, which includes SQL commands and clauses.

Table 3-31: Data Definition Language

Component	Description
Database	
alter	Sets the default character set and changes the name of the database. Refer to "ALTER DATABASE" on page B-5.
create	Creates a database. Refer to "CREATE DATABASE" on page B-38.
drop	Drops a database. Refer to "DROP DATABASE" on page B-73.
Group	
alter	Changes a group's limits, drops a user from a group, changes the group's owner, or name. Refer to "ALTER GROUP" on page B-6.
create	Creates a group. Refer to "CREATE GROUP" on page B-40.
drop	Drops a group. Refer to "DROP GROUP" on page B-75.
User	
alter	Alters a user's account. Changes the owner, password, optional expiration time, rowset limits, and name. Refer to "ALTER USER" on page B-24.
create	Creates a user. Refer to "CREATE USER" on page B-65.
drop	Drops a user. Refer to "DROP USER" on page B-82.
Table	
alter	Changes the definition of a table. Refer to "ALTER TABLE" on page B-20.
create	Creates a table. Refer to "CREATE TABLE" on page B-55.
create external	Creates an external table. Refer to "CREATE EXTERNAL TABLE" on page B-40.
create table as	Creates a new table based on query results. Refer to "CREATE TABLE AS" on page B-61.
create temp table	Creates a temporary table. Refer to "CREATE TABLE" on page B-55.
drop	Drops a table. Refer to "DROP TABLE" on page B-81.
View	
alter	Changes the owner or name of the view. Refer to "ALTER VIEW" on page B-28.
create	Creates a view. Refer to "CREATE VIEW" on page B-69.
drop	Drops a view. Refer to "DROP VIEW" on page B-83.

Table 3-31: Data Definition Language (continued)

Component	Description
Index	
create, alter, drop	**Not Supported**.

Data Control Language

As database security administrator, you use DCL SQL commands to control user access to database objects and their contents. Security starts with the user, admin. As the admin user, you must create and authorize other users. When you first create users, they cannot see or do anything. As you grant users more privileges, they can access more database objects.

When you create new users, by default they have access only to system views. With these views, they can retrieve lists of user database objects and select data within those objects. Because security is also built into these system views, the list of database objects a user can see depends on the user's security privileges.

Table 3-32 describes the Netezza SQL DCL.

Table 3-32: Data Control Language

Component	Description
grant	Grants privileges. Refer to "GRANT" on page B-91.
revoke	Revokes privileges. Refer to "REVOKE" on page B-98.

Types of Privileges

There are two types of privileges that you can grant: administrator and object.

- Administrator privileges control creation of objects and system administration.
- Object privileges control access to specific database objects.

Some administrator privileges are global in scope, regardless of the current database. For example, the database, user, group, system and hardware administrator privileges are global in scope. All other administrative privileges can be either global or local depending on the current database. Table 3-33 describes the administrative privileges.

Table 3-33: Administrator Privileges

Privilege	Description
Backup	Allows user to perform backups. The user can run the command **nzbackup**.
[Create] Database	Allows the user to create databases. Permission to operate on existing databases is controlled by object privileges.
[Create] External Table	Allows the user to create external tables. Permission to operate on existing tables is controlled by object privileges.

Table 3-33: Administrator Privileges (continued)

Privilege	Description
[Create] Group	Allows the user to create groups. Permission to operate on existing groups is controlled by object privileges.
[Create] Index	For system use only. Users cannot create indexes.
[Create] Materialized View	Allows the user to create materialized views.
[Create] Sequence	Allows the user to create database sequences.
[Create] Synonym	Allows the user to create synonyms.
[Create] Table	Allows the user to create tables. Permission to operate on existing tables is controlled by object privileges.
[Create] Temp Table	Allows the user to create temporary tables. Permission to operate on existing tables is controlled by object privileges.
[Create] User	Allows the user to create users. Permission to operate on existing users is controlled by object privileges.
[Create] View	Allows the user to create views. Permission to operate on existing views is controlled by object privileges.
[Manage] Hardware	Allows the user to perform the following hardware-related operations: view hardware status, manage SPUs, manage topology and mirroring, and run diagnostics. The user can run these commands: **nzhw** and **nzds**.
Restore	Allows the user to restore the system. The user can run the **nzrestore** command.
[Manage] System	Allows the user to perform the following management operations: start/stop/pause/resume the system, abort sessions, view the distribution map, system statistics, and logs. The user can use these commands: **nzsystem**, **nzstate**, **nzstats**, and **nzsession**.

Object privileges can also be local or global in scope. The procedure to define global object privileges is different than that of defining local object privileges. Another difference is that global object privileges are broader and not particular to a specific object, but instead to a class of objects. Table 3-34 describes the object privileges.

Table 3-34: Object Privileges

Privilege	Description
Abort	Allows the user to abort sessions. Applies to groups and users.
Alter	Allows the user to modify object attributes. Applies to all objects.
Delete	Allows the user to delete table rows. Applies only to tables.
Drop	Allows the user to drop all objects.

Table 3-34: Object Privileges (continued)

Privilege	Description
Execute	Allows the user to run user-defined functions, user-defined aggregates, or stored procedures.
GenStats	Allows the user to generate statistics on tables or databases. The user can run the GENERATE STATISTICS command.
Groom	Allows the user to reclaim disk space for deleted or outdated rows, and reorganize a table by the organizing keys, or to migrate data for tables that have multiple stored versions. **Note:** Grooming a table is done as a user, not an administrator, so to run GROOM TABLE requires that you have object privileges on that table as well.
Insert	Allows the user to insert rows into a table. Applies only to tables.
List	Allows the user to display an object's name, either in a list or in another manner. Applies to all objects.
Select	Allows the user to select (or query) rows within a table. Applies to tables and views.
Truncate	Allows the user to delete all rows from a table with no rollback. Applies only to tables.
Update	Allows the user to modify table rows, such as changing field values or changing the next value of a sequence. Applies to tables only.

Granting Object Privileges

When you grant object privileges to a user or group, you must decide if the privilege should be for a particular database or for all databases, for example:

- If the privilege is global, log on to the system database and grant the privilege for the object class to the user or group.
- If the privilege is for a particular database, log on to that database and grant the privilege for the object to the user or group.

Whoever creates an object becomes the owner of that object, and as owner has full access to the object. There is a special case when a user creates a database. The owner of a database has full access to all objects within the database, even if he or she did not create those objects. In essence, the database owner becomes the superuser for that database. In addition to the owner, the admin user has full access to the database and all its objects.

Listing User's Privileges

You can use the nzsql command or the NzAdmin tool to list a user's privileges. To use the NzAdmin tool, see the *IBM Netezza System Administrator's Guide*.

- To list a user's privileges within nzsql, enter:

```
SYSTEM(ADMIN) =>\dpu john
```

Data Manipulation Language

Data Manipulation Language of SQL allows you to access and modify database data using the select, update, insert, delete, truncate, begin, commit, and rollback commands.

Table 3-35 describes the Netezza SQL DML.

When using commands to manipulate data, you can use row values. Netezza SQL supports SQL-92 row values. For example:

```
SELECT * FROM emp WHERE (id, name,grp)=(3,'John','sdev');
```

Table 3-35: Data Manipulation Language

Component	Description
DELETE	Removes rows from a table. Refer to “DELETE” on page B-71.
INSERT	Adds new rows to a table. Refer to “INSERT” on page B-95.
SELECT	Retrieves rows from a table or view. Refer to “SELECT” on page B-102.
TRUNCATE	Empties a table. Refer to “TRUNCATE” on page B-134.
UPDATE	Replaces values of columns in a table. Refer to “UPDATE” on page B-135.

Transaction Control

Transaction control enforces database integrity by ensuring that batches of SQL operations execute completely or not at all. The transaction control commands are BEGIN, COMMIT, and ROLLBACK.

Netezza SQL supports auto-commit transaction mode. In this mode, all SQL commands commit when you execute them. If the system encounters a SQL command before a BEGIN SQL command, it executes the SQL command in auto-commit transaction mode. If the system encounters a BEGIN SQL command, it executes all successive SQL commands within the transaction. To end a transaction, you must issue a COMMIT or ROLLBACK SQL command.

Some SQL commands are prohibited within the BEGIN/COMMIT transaction block. For example:

- BEGIN
- TRUNCATE TABLE
- [CREATE | DROP] DATABASE
- ALTER TABLE [ADD | DROP] COLUMN operations
- SET AUTHENTICATION
- [SET | DROP] CONNECTION
- GROOM TABLE
- GENERATE STATISTICS
- SET SYSTEM DEFAULT HOSTKEY

- [CREATE | ALTER|DROP] KEYSTORE
- [CREATE | DROP] CRYPTO KEY

  These SQL commands are also prohibited within the body of a Netezza stored procedure. If you use one of these commands within a transaction block or stored procedure, the system displays an error similar to the following:

  ERROR: CREATE DATABASE: may not be called in a transaction block or stored procedure

Isolation Level

There are four levels of transaction isolation that are defined by the ANSI/ISO SQL. Netezza SQL supports the SQL grammar for defining all four isolation levels:

- read committed
- read uncommitted
- repeatable read
- serializable

The only isolation level that Netezza SQL implements, however, is serializable, which provides the highest possible level of consistency.

These isolation levels prevent the following occurrences between concurrent transactions:

- Dirty reads — A transaction reads data written by concurrent uncommitted transactions.
- Nonrepeatable reads — A transaction re-reads data it previously read and finds that the data has been modified by another transaction (that committed since the initial read).
- Phantom read — A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows has changed due to another recently committed transaction.

Table 3-36 describes the four isolation levels.

Table 3-36: Isolation Levels

Isolation Level	Dirty Read	Nonrepeatable	Phantom
read uncommitted	Possible	Possible	Possible
read committed	Not possible	Possible	Possible
repeatable read	Not possible	Not possible	Possible
serializable	Not possible	Not possible	Not possible

Locks and Concurrency

Netezza SQL does not use conventional locking to enforce consistency among concurrently executing transactions. Instead, it automatically uses a combination of the following mechanisms. Note that there is no need for user intervention, commands, or hints.

- Multiversioning — Each transaction sees a consistent state that is isolated from other transactions that have not been committed. Because of the Netezza architecture, the hardware can quickly provide the correct view to each transaction.
- Serialization dependency checking — Concurrent executions that are not serializable are not permitted. If two concurrent transactions attempt to modify the same data, the system automatically rolls back the latest transaction. This is a form of optimistic concurrency control that is suitable for low-conflict environments.

Table Locking

As a user, you cannot explicitly lock tables. The Netezza SQL, however, implicitly locks a table when there is a DDL operation on it. For example, a drop table command is blocked if somebody is running a select command on the same table (or vice versa).

For concurrent DML operations (select, insert, update, and delete commands), Netezza SQL uses serialization graph checking, which is a form of optimistic concurrency control that does not use locks. Instead, if there is a concurrency conflict, Netezza SQL rolls back one (or sometimes several) of the affected transactions.

- A select command on a given table can proceed concurrently with an update, delete, insert, or select command on the same table. Invisibility lists, and other mechanisms, ensure that each transaction sees a consistent state.
- More than one concurrent insert command can proceed against the same table, provided no more than one is also selecting from the same table.
- Concurrent update or delete commands against different tables are permitted, with some restrictions that are needed to ensure serializability. For example:
  - If transaction 1 selects from table A and updates (or deletes from) table B, while transaction 2 selects from table B and updates table A, Netezza SQL rolls back one or the other (typically the transaction that started more recently). This is called the *cross-update* case.
  - If there is a cycle of three or more transactions (transaction 1 selects from A and updates B, transaction 2 selects from B and updates C, transaction 3 selects from C and updates A), the Netezza SQL rolls back one of the transactions in the cycle.

Read-Only Sessions

The Netezza SQL provides SQL commands that define sessions as read-only:

```
SET SESSION { READ ONLY | READ WRITE }
```

Read-only sessions are efficient and reduce the overhead in processing SQL commands. If you define a session as read-only and the system encounters an insert, update, delete, truncate SQL, a DDL or DCL command, it generates an error.

CHAPTER 4

SQL Statement Grammar

What's in this chapter

- Netezza SQL Lexical Structure
- Grammar Overview
- Implicit and Explicit Casting

This chapter contains descriptions of the Netezza SQL lexical structure.

Netezza SQL Lexical Structure

A Netezza SQL statement is a sequence of commands. Each command consists of a sequence of tokens terminated by an optional semicolon with the end of the input terminating the command.

Tokens can be:

- Keywords — Words that have a fixed meaning in SQL. For a list of keywords, see Appendix A, "SQL Reserved Words and Keywords."
- Identifiers — Names of users, tables, columns, or other database objects.
- Constants — Strings, integers, and floating-point numbers.
- Comments — An arbitrary sequence of characters beginning with double dashes and extending to the end of the line.
- Constraints — A rule that restricts the value for one or more columns in a table. (Netezza does not support constraint checking and referential integrity.)

Keywords

Keywords are words that have a special meaning in SQL. There are two types of keywords: reserved and non-reserved keywords. A reserved keyword cannot be used as a regular identifier. Although you can use non-reserved keywords as regular identifiers, it is typically not a good practice to do so. Keywords are case-insensitive, so for example, the keyword **select** has the same meaning as **SELECT** or **Select**. For a list of the SQL keywords, see Appendix A, "SQL Reserved Words and Keywords."

Identifiers

You use identifiers to name database objects, such as users, tables, and columns. Identifiers can either be unquoted (regular identifiers) or quoted (delimited identifiers). For more information about identifiers, their naming requirements, and other restrictions, see "Handling SQL Identifiers" on page 2-7.

Constants

Constants are symbols that represent specific data values. The format of a constant depends on the data type of the value it represents. Constants are also called *literals*.

Constants can be either implicit or explicit. Implicitly-typed constants can be strings, integers, numeric, or floating-point numbers. Explicit constants enable more accurate representation and more efficient system handling.

String Constants

A string constant is an arbitrary sequence of characters enclosed in single quotes ('); for example, 'This is a string'. You can embed single quotes in strings by typing two adjacent single quotes.

If you have two string constants that are separated only by whitespace with at least one newline, the system concatenates them and effectively treats them as if the strings had been written as one constant.

Integer Constants

Integer constants are sequences of decimal digits (0 though 9) with no decimal point and no exponent. The range of legal values depends on which integer data type you use, but the plain integer type accepts values ranging from -2147483648 to +2147483647.

Floating Point Constants

Floating-point constants are accepted in two general forms:

- digits.[digits][e[+-]digits]
- [digits].digits[e[+-]digits]

The digits value is one or more decimal digits. At least one digit must be before or after the decimal point. At least one digit must follow the exponent delimiter (e) if that field is present. A floating-point constant is distinguished from an integer constant by the presence of either the decimal point or the exponent clause (or both). You cannot embed a space or other characters in the constant.

The following are some examples of valid floating-point constants:

- 2.5
- 6.
- .001
- 5e2
- 1.925e-3

Floating-point constants are of double precision type. You can explicitly specify real type by using SQL string notation: real'1.23' — string style.

Explicit Constants

You can enter a constant of an arbitrary type by using either of the following notations:

type 'string'

CAST ('string' AS type)

The system passes the string text to the input conversion routine for the type called *type*. The result is a constant of the indicated type. You can omit the explicit type cast if there is no ambiguity as to the type the constant must be (for example, when it is passed as an argument to a non-overloaded function), in which case it is automatically coerced.

You can also specify a type coercion by using a function-like syntax:

typename ('string')

You can use CAST() and function-call syntaxes to specify runtime type conversions of arbitrary expressions. But, you can only use the form type 'string' to specify the type of a literal constant.

Date and Time Constants

Table 4-1 describes the date and time constants.

Table 4-1: Date and Time Constants

Notation	Description
MM	One- or two-digit month
DD	One- or two-digit day
YY	Two-digit year
YYYY	Four-digit year
MON	Three-letter month (JAN, FEB, and so on)

You can use the following date formats:

- MM-DD-YY
- MM-DD-YYYY
- MM/DD/YY
- MM/DD/YYYY
- MM.DD.YY
- MM.DD.YYYY
- YY-MM-DD
- YYYY-MM-DD
- YY/MM/DD
- YYYY/MM/DD
- YY.MM.DD

- YYYY.MM.DD
- YYYY.DDD (year and day number in year)
- DD-MON-YY

Note: For date values, Netezza assumes that the first value MM represents the number of the month. However, if the MM value is greater than 12, Netezza treats the first value as the day of month (DD) and treats the next value as MM. For example, 01/07/2007 is January 7, 2007, but 14/07/2007 is treated as July 14, 2007. As a best practice, use consistent date formats within your queries.

You can use three-character month abbreviations or full month names, for example:

- July 4, 1776
- Jul 4, 1776
- 4 July 1776
- 4 Jul 1776

You can specify time as one or two-digit hours and minutes with optional one or two-digit seconds (and seconds can include 0 to 6 digits after the decimal point). For example, 01:30:45 or 12:14:66.123456

You can use optional am/AM or pm/PM suffixes. If you omit the suffix, Netezza SQL assumes a twenty-four hour notation (also called military time).

You specify time zones as signed hours with optional minutes; for example -HH or +HH:MM, which indicate the offset of the local time zone from GMT. For example, EST is -05 hours from GMT. The minus sign means west of Greenwich. The range of time zones is -12:59 to +13:00.

Quoted and Mixed Literals

SQL statements can contain quoted literals '...' (single quote - series of characters - single quote). A quoted literal is a constant that is expressed as itself rather than as a result of an expression, such as an arithmetic formula.

Netezza SQL considers quoted literals typeless unless they have an associated data type keyword, or are explicitly typecast. When Netezza SQL cannot readily determine the datatype, it deduces the type from the context.

- Quoted literals with a keyword — date '2004-Mar-09'
- Quoted literals with explicit typecasting — cast('2004-Mar-09' as date) or '2004-Mar-09'::date
- Quoted literals whose type is determined by context — date_column = '2004-Mar-09'

  This is an unknown type until Netezza SQL compares it to date column, at that time Netezza SQL assigns it to the date type and parses the string as a date constant.

When literals mix integers and numerics in a statement, Netezza SQL has to determine how to parse them.

- Quoted literals that mix integers and numerics — integer_column = '5' OR integer_column = '5.4'

  Netezza SQL treats quoted literals with decimal points as numerics when comparing them to integers and thus allows this syntax.

- Unquoted literals mixing integers and numerics — integer_column = 5 OR integer_column = 5.4

  The parser recognizes the constants as integer and numeric. To execute the clause <integer_column = 5.4>, the parser promotes the integer_column to numeric type, because the built-in promotion rules states that the common datatype of integer and numeric is numeric. Netezza SQL allows this syntax and executes the clause as if it is <integer_column::numeric = 5.4>.

Comments

A comment is an arbitrary sequence of characters beginning with double dashes and extending to the end of the line, for example:

```
-- This is a standard SQL92 comment
```

Before analyzing the command syntax, the system removes the comment from the input stream and replaces it with whitespace.

Grammar Overview

Table 4-2 describes the key to reading Netezza SQL statement grammar.

Table 4-2: Grammar Components

Syntax element...	Named...	Performs this function...
<name>	Name	Specifies the name of the instance.
[]	Square brackets	Specifies an optional parameter.
[, ...]	Square brackets enclosing several items	Specifies that the prior item repeats, separated by commas.
[, ()...]	Square brackets enclosing items in parenthesis	Specifies that items within parenthesis repeat, separated by commas.
{}	Curly braces	Specifies a required parameter.
item	Item	Specifies the default value.
|	OR	Specifies an OR expression.

Implicit and Explicit Casting

The Netezza SQL language is aligned with the SQL Standard with respect to implicit type casting. An example follows:

```
SELECT * FROM tbl WHERE <character expression> > <integer expression>;
```

The two expressions can be column names, literals, or more complex expressions. Because a character expression is being compared to an integer expression, the system tries to parse each character expression value as an integer. If a particular value cannot be parsed as an integer, a pg_atoi error occurs, similar to the following:

ERROR: pg_atoi: error in "*abc*": can't parse "*abc*"

The error can also occur in situations such as the following:

```
SELECT * FROM tbl WHERE <date expression> = <character expression>;
```

In this case, Netezza tries to parse the character expression as a date value. If the expression does not parse, then you may receive an error such "ERROR: Bad date external representation 'abc'".

The implicit casting can also result from other queries, such as:

```
INSERT INTO <table having one column of type date> SELECT <character
expression> FROM <another table>
```

In this case, since you are inserting a character expression into a date column, Netezza implicitly casts the character values to dates, and can get the same parse error as in the prior example.

The error can be somewhat cryptic; the value that appears is the character data that cannot be parsed as a value of the implicit cast's target data type. If you receive a pg_atoi "can't parse" error message or a "bad external representation" error, look for a situation where a value is invalid for the operation that you are attempting.

In Release 4.0 and later, the Netezza system attempts to handle mixed-types expressions with implicit casting, but if the mixed types are very different, these actions can often yield undesired results or errors. For example, an expression could attempt to multiply a numeric(24,4) by a varchar(50). For this type of expression, the Netezza system attempts to interpret the expression by implicit casts of the arguments; in this case, by trying to cast the varchar to a numeric(24,4) to match the left-side operand. This cast may not be the desired approach, and could result in a numeric precision range error (a resulting numeric that has more than 38 digits).

In examples of this mixed-types case, the best practice is either to use explicit cast operations to force the datatype conversions that you need, or to change the cause of such a mixed-type multiplication, perhaps by a table change. Fixing the cause may be more efficient, since an explicit cast still requires string-to-number conversions for every expression evaluated.

Table 4-3 shows the supported implicit and explicit casts. A summary of Netezza casting best practices follow the table.

Table 4-3: Supported Implicit and Explicit Casts

From: \ To:	Char	Varchar	Nchar	Nvarchar	Byteint	Smallint	Integer	Bigint	Numeric	Real	Double	Timestamp	Date	Time	TimeTz	Interval	Boolean
Char		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Varchar	X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Nchar	X	X		X	X	X	X	X	X	X	X	X	X	X	X	X	X
Nvarchar	X	X	X		X	X	X	X	X	X	X	X	X	X	X	X	X
Byteint	X	X	X	X		X	X	X	X	X	X						
Smallint	X	X	X	X	X		X	X	X	X	X						
Integer	X	X	X	X	X	X		X	X	X	X						
Bigint	X	X	X	X	X	X	X		X	X	X						

Table 4-3: Supported Implicit and Explicit Casts

From: \ To:	Char	Varchar	Nchar	Nvarchar	Byteint	Smallint	Integer	Bigint	Numeric	Real	Double	Timestamp	Date	Time	TimeTz	Interval	Boolean
Numeric	X	X	X	X	X	X	X	X		X	X						
Real	X	X	X	X	X	X	X	X	X		X						
Double	X	X	X	X	X	X	X	X	X	X							
Timestamp	X	X	X	X									X	X	X		
Date	X	X	X	X								X					
Time	X	X	X	X												X	
TimeTz	X	X	X	X													
Interval	X	X	X	X										X			
Boolean				X													

Note the following points about the table and implicit and explicit casting:

- To perform an explicit cast, use the CAST operator as follows:

  ```
  CAST (<from-type> AS <to-type>)
  ```

  For example:

  ```
  CAST ( <int2-column> AS NUMERIC(12,3) )
  ```

- The Netezza system could perform an implicit cast in an expression involving functions or operators. For example, the Netezza built-in function SQRT takes a float8 (that is, double) argument. For example, consider the following expression:

  ```
  SQRT ( <int4-column> )
  ```

  The system performs an implicit cast, converting the sample expression to the following:

  ```
  SQRT ( CAST ( <int4-column> AS FLOAT8 ) )
  ```

  As another example, consider the following expression:

  ```
  <varchar-column> + <int4-column>
  ```

  The system implicitly casts the varchar column to an int4 type, transforming the expression to the following:

  ```
  CAST ( <varchar-column> AS INT4 ) + <int4-column>
  ```

- In general, it is better to use explicit casts rather than rely on the implicit casting behavior of the system. The implicit cast choice made by the system might not yield the behavior you want. Also, implicit casting behavior could change from one release of Netezza to another. In the previous example where a varchar is added to an int4, the system chose an implicit cast from varchar to int4. But if your varchar column contained strings representing numbers with decimal points, as in '25.7', the cast to integer would generate an error.

- When executing an INSERT or UPDATE statement, the Netezza system implicitly casts the values being inserted into table columns to the types of those columns, as needed.

- When comparing a string to a non-string value such as a numeric, integer, timestamp, and so on, the Netezza system implicitly casts the string value to the type of the non-string value.
- When comparing a string to a NUMERIC type, or performing an arithmetic operation on a string and a NUMERIC type, the Netezza implicitly casts the string to a NUMERIC of the same precision and scale as those of the second operand. Therefore, if your string contains data requiring a different precision and scale, you should use explicit casting.
- When concatenating a string and a non-string type such as a numeric, integer, timestamp, and so on, the Netezza implicitly casts the non-string to a varchar.
- When casting from a Unicode type (nchar or nvarchar) to an 8-bit Latin 9 character type (char or varchar), the Netezza system converts any characters that do not exist in Latin 9 encoding into a question mark character ('?').
- Arithmetic operations on a temporal type (timestamp, date, interval, and so on) along with a string type can produce unexpected results. Therefore, you should use explicit casts as needed in such expressions.

CHAPTER 5

Netezza SQL Analytic Functions

What's in this chapter

- Overview of Analytic Functions
- Window Analytic Functions
- Examples

Netezza SQL supports many of the ANSI SQL:1999 standard analytic functions and some extensions. Using analytic functions in your business intelligence queries provides the following benefits:

- Improved query processing — Using these functions results in better performance, because the system no longer must perform complex procedural processing and instead can perform simple SQL queries.
- Enhanced productivity — You can perform complex analysis with clearer, more concise code. The code is quicker to formulate and easy to maintain.
- Easy to learn — The functions utilize existing aggregate functions and thus leverage known syntax.
- Standardized syntax — Because these functions are part of the ANSI standard, they are supported in many software packages.

The Netezza system also supports the development and use of user-defined functions and aggregates. This feature is described in the *IBM Netezza User-Defined Functions Developer's Guide*, which is available from Netezza for users who are participating in the Netezza Developer Network.

Overview of Analytic Functions

Netezza SQL analytic functions compute an aggregate value based on a group of rows. But unlike aggregate functions, they return multiple rows for each group. You use analytic functions to compute cumulative, moving, centered, and reporting aggregates. Analytic functions answer questions such as the following:

- What is the running total?
- What are the percentages within a group?
- What are the top *n* queries?
- What is the moving average?

The Netezza SQL functions can be divided into the following families. Some functions are used in more than one family:

- Reporting/Window aggregate family — Reporting functions enable you to compare values at different levels of aggregation, because they can perform multiple passes over the data in a single query block and then return the same aggregate value for every row in a partition. You can use reporting functions in percent-of-total and market share calculations. For instance, you might want to know regional sales levels as a percent of national sales.

  Window functions answer questions such as "what is the 12-week moving average of a stock price?" or "what was the cumulative sum of sales per each region?"

  For all SQL aggregate functions, reporting functions provide reporting aggregate processing, while window functions provide moving and cumulative processing. These SQL aggregate functions include: avg, sum, min, max, count, variance, and stddev.
- Lag/Lead family — These functions allow you to compare different rows of a table by specifying an offset from the current row. You can use these functions to analyze change and variation. The functions include lag and lead.
- First/Last family — These functions allow you to specify sorted aggregate groups and return the first or last value of each group. For example, you could query bank statements for beginning and ending monthly balances. These functions include first_value and last_value.
- Ranking family — These functions help to answer questions such as "what are the top 10 and bottom 10 selling items?" The functions examine the entire output before displaying an answer. These functions include ntile, dense_rank, percent_rank, cume_dist, and rank.
- Row count family — The row_number function assigns a number to each row, based on its position within the window partition. It is similar to dense_rank and rank, but unlike the rank functions, it counts ties (peer rows, that is, rows that match on the order by column).
- Hypothetical set family — These functions give the rank or percentile that a row would have if inserted into a specified data set. These functions include dense_rank, percent_rank, cume_dist, and rank.
- Inverse distribution functions family — These functions give the value in a data set that corresponds to a specified percentile. These functions include percentile_cont and percentile_disc, however, these two functions are not supported as window aggregates.

Processing Order

Analytic functions compute an aggregate value based on a group of rows, which are defined by a window. The window determines the range of rows the system uses to perform the calculations. Window sizes can be based on a physical number of rows or a logical interval.

The system processes queries with analytic functions in three stages:

- Processes all JOINS, WHERE, and HAVING clauses.

- Makes the result set available to the analytic functions that perform the calculations, which include finding the partition boundaries, ordering the rows within each partition, framing, and performing the aggregate calculations where appropriate.
- If the query has an ORDER BY clause, orders the output.

Result Set Partitions

You can divide query results into groups of rows called *partitions* that are based on column values. You can partition query results into one partition, a few large partitions, or many small partitions. For more information, see “Window Partitioning” on page 5-4.

Windows

For each row in a partition, you can define a sliding window of data. The window determines the range of rows to use for calculations for the current row. You can base windows on a physical number of rows or a logical interval such as time. Windows have a starting row and an ending row. You can define the window to move at one or both ends. You can define a window to be as large as all the rows in a partition or as small as a sliding window of one row in a partition. For more information, see “Window Framing” on page 5-4.

Current Row

Each analytic calculation is based on the current row within a partition. The current row is the reference point for the start and the end of the window. For example, you could have a calculation based on the preceding five rows and the following four rows, which would result in a window of ten rows.

Using Windowing

Analytic functions perform analysis on a window of data. A window is a user-specified selection of rows (or a logical partition of a query that determines the set of rows) used to perform certain calculations with respect to the current row.

Using windowing you can determine ranking, distribution of values, or moving averages and sums.

A window has three components: partition, order, and frame.

- Window partitioning — Groups all rows that have partition column values that are equal to the values in the specified row. The system returns each row of a partition that has an equal value on the set of specific rows in a table, rather than collapsing them into a single representative row as is the case with grouped aggregate functions.
- Window ordering — Allows you to order rows within each partition. Because all rows in a partition have equal values in their partitioning columns, you usually order the rows by values in other columns.
- Window framing — Defines the size of the window within a window partition. You can express it in physical terms (the number of rows) or in logical terms (a range of values). Window framing is also called *window aggregation grouping*.

  Framing can be either row-based or range-based. Framing can specify anchored or floating frame endpoints. If the frame is anchored, you can specify unbounded preceding and/or unbounded following. If the frame is floating, you can specify an absolute

row offset (for row-based frames) or as a delta from the current row's order column value (for range-based frames).

The frames for functions in the Window Aggregation family can also specify an exclusion as: exclude no others, exclude current row, exclude ties, or exclude group. For details about which functions support the exclude clause in syntax, see "Netezza SQL Analytic Functions" on page 7.

Window Partitioning

You can divide a table or rowset into partitions based on the values of specific columns. For example, you could partition sales data by the store ID. If you do not specify partitioning, the system considers the entire table or rowset to be a single partition.

The window partition syntax allows you to name one or more columns.

<window partition clause> ::= partition by <column reference list>

<column reference list> ::= <value_expression>

The resulting window is partitioned into one or as many partitions as there are rows in the input virtual table, based on the values of the one or more columns referenced in the <window partition clause>.

Each partition has one or more rows, and the values of the partition columns are equal among all rows of the partition.

Window Ordering

When you order data in a window, you are actually ordering the data in each of its partitions. To order data in a window, use the ORDER BY clause to specify the columns known as the *ordering columns*.

<window order clause> ::= ORDER BY <sort specification list>

<sort specification list> ::= <sort specification> [{ , <sort specification> } ...]

<sort specification> ::= <sort key> [<ordering specification>]

<sort key> ::= <value_expression>

<ordering specification> ::= [ASC | DESC] [NULLS {FIRST | LAST}]

If your data contains nulls, Netezza SQL considers all null values to be lower than any non-null value. This means that for an ascending sort, nulls appear first in the output, or for a descending sort, they appear last. You can control the null sorting using NULLS {FIRST | LAST}.

Window Framing

The window frame or window aggregation grouping defines the boundary within the window partition for which some computation is being done.

<window frame clause> ::= <window frame units> <window frame extent> [<window frame exclusion>]

<window frame units> ::= ROWS | RANGE

<window frame extent> ::=

    <window frame start> | <window frame between>

<window frame start> ::=

unbounded preceding | <unsigned value specification> preceding | current row

<window frame between> ::=

between <window frame bound 1> and <window frame bound 2>

<window frame bound 1> ::= <window frame bound>

<window frame bound 2> ::= <window frame bound>

<window frame bound> ::=

<window frame start>

| unbounded following

| <unsigned value specification> following

<window frame exclusion> ::=

exclude current row

| exclude group

| exclude ties

| exclude no others

You can specify a frame as either row-based or range-based:

- Use rows for physical rows — Only if the input data is dense; that is, there are no gaps in the sequence of ordering column values, but not too dense, such as having multiple rows for a given month. In that case you can have a tie, which can result in nondeterministic results. You can have more than one ordering column and you are not limited to a simple formula for computing the preceding and succeeding ordering column values.
- Use range for logical rows — Only when there is a simple formula that allows you to add or subtract a value from the ordering column. Your order column must be a numeric, datetime, or an interval data type, because these are the only types for which addition and subtraction are defined. Although, you cannot use more than one ordering column, your queries are not limited to dense data and ties do not lead to nondeterminism.

You can specify bounded or unbounded frames:

- Bounded frames — The frame's boundary is determined by a specified number or rows, or by a range of values.
- Unbounded frames — The frame extends to partition boundary in the given direction, that is, it can be unbounded preceding the current row and/or unbounded following the current row.
- If you do not specify window framing and window ordering, then the effect is as though you had specified between unbounded preceding and unbounded following.
- If you do not specify a window framing, but you do specify a window ordering, then the implicit window framing is range unbounded preceding.

Window Aggregates

At a high level, a window aggregate is expressed as:

<aggr-type> (<arglist>) over (<window-spec>)

- aggr-type can be the functions count, sum, avg, min, max, stddev, stddev_pop, stddev_samp, variance, var_pop, var_samp, row_number, rank, dense_rank, lead, lag, first_value, and last_value.
- arglist specifies zero or more arguments to the aggregation. Note that only lead and lag can take more than one argument.
- window-spec specifies the partition columns, order columns, and framing.

Window Analytic Functions

The Netezza SQL analytic functions include window aggregation, reporting aggregation, lag and lead, first and last, ranking, and row count families. Table 5-1 describes the analytic function keywords.

Table 5-1: Analytic Function Keywords

Syntax	Description
ALL	Applies the analytic function to all values. This is the default and you do not need to specify it.
ASC | DESC	Specifies the ordering sequence, either ascending or descending.
BETWEEN ... AND	Specifies starting and ending points for the window. The first expression (before and) specifies the start; the second expression (after and) specifies the end.
CURRENT ROW	As a starting point, specifies that the window begins at the current row or value. As an ending point, specifies that the window ends at the current row or value.
DISTINCT	Specifies that the function should aggregate results for each unique value. Note that DISTINCT is not supported for the FIRST_VALUE, LAST_VALUE, LEAD, or LAG functions. It is also not supported for STDDEV, STDDEV_POP, STDDEV_SAMP, VARIANCE, VAR_POP, OR VAR_SAMP functions in either a grouped or windowed aggregate.
EXCLUDE CURRENT ROW	Specifies excluding the current row.
EXCLUDE GROUP	Specifies excluding the current row and all rows tied with it. Ties occur when there is a match on the order column or columns.
EXCLUDE NO OTHERS	Specifies not excluding any rows. This is the default if you specify no exclusion.
EXCLUDE TIES	Specifies excluding all rows tied with the current row (peer rows), but retaining the current row.

Table 5-1: Analytic Function Keywords

Syntax	Description
NULLS {FIRST | LAST}	Specifies whether nulls appear before or after non-null values in the sort ordering. By default, null values sort as if they are lower than any non-null value; that is, NULLS FIRST is the default for DESC order, and NULLS LAST otherwise.
ORDER BY	Specifies how the data is ordered within the partition. You can order the values on multiple keys, each defined by an ordering sequence.
OVER	Indicates that the function operates on a query result set that is computed after the from, where, and having clauses. Use over to define the window of rows to include in the function.
PARTITION BY	Partitions the query result into groups based on one or more columns. If you omit this clause, the query handles the query result as a single partition.
ROWS | RANGE	Defines the window as physical rows or as a logical range. To use range between, you must have specified the order by clause.
UNBOUNDED FOLLOWING	Specifies that the window ends at the last row of the partition. If there is no partition by clause, the end is the last row of the dataset.
UNBOUNDED PRECEDING	Specifies that the window starts at the first row of the partition. If there is no partition by clause, the start is the first row of the dataset.

Netezza SQL Analytic Functions

The following describes the notation used in this section:

- ( ) — Parentheses are interpreted literally.
- | — A pipe character separates a list of items.
- [ ] — Square braces denote an optional clause, and may contain a list.
- { } — Curly braces group a non-optional list of choices (you must choose one).
- [,...] — Indicates previous item(s) can be repeated, because Netezza SQL uses curly braces for something else, see above.
- <> — Angle braces contain category names.
- Keywords are in caps and category names are in lower case.

Report Aggregation Family Syntax

The syntax for the report aggregation family is as follows:

```
Func( [DISTINCT] value_expression ) OVER ( )
```

Or

```
Func( [DISTINCT] value_expression) OVER (PARTITION BY value_expression
[, …] )
```

Functions are detailed in the following section, as they are similar to those in the Window Aggregate Family. Reporting functions appear only in the SELECT or ORDER BY clauses. They enable you to make multiple passes over a single query block without doing a self-join. Because the OVER clause can only contain a partition column list, a report aggregate has the same value in each row of the partition.

Window Aggregation Family Syntax

The syntax for the window aggregation family is as follows:

```
Func( value_expression) OVER ( <partition_by_clause> <order_by_clause>
[<frame_spec_clause>] )
```

Or

```
Func( value_expression) OVER (<order_by_clause> [<frame_spec_clause>])
```

Where

Func is an expression:

```
<partition_by_clause> = PARTITION BY <value_expression> [, ...]+

<order_by_clause> = ORDER BY <value_expression> [asc | desc] [nulls
{first|last}] [, ...]+

<frame_spec_clause> = <frame_extent> [<exclusion clause>]

<frame_extent> =

     ROWS  UNBOUNDED PRECEDING

    |ROWS  <constant> PRECEDING

    |ROWS   CURRENT ROW

    |RANGE  UNBOUNDED PRECEDING

    |RANGE  <constant> PRECEDING

    |RANGE  CURRENT ROW

    |ROWS BETWEEN {UNBOUNDED PRECEDING| <constant> PRECEDING | CURRENT
ROW } AND { UNBOUNDED FOLLOWING | <constant>  FOLLOWING | CURRENT ROW }

    |RANGE BETWEEN {UNBOUNDED PRECEDING| <constant> PRECEDING | CURRENT
ROW } AND { UNBOUNDED FOLLOWING | <constant>  FOLLOWING | CURRENT ROW }

<exclusion_clause> =  EXCLUDE CURRENT ROW | EXCLUDE TIES | EXCLUDE
GROUP | EXCLUDE  NO OTHERS
```

- avg — Returns the average value of the expression.
- count — Returns the number of rows in the query.
  - If you specify value expression, count returns the number of rows where the expression is not null.
  - If you specify an asterisk, count returns all rows, including duplicates and nulls. Otherwise count never includes nulls.
- max — Returns the maximum value of the expression.
- min — Returns the minimum value of the expression.
- sum — Returns the sum of the expression.

- stddev — Returns the standard deviation of the expression, which is the square root of variance. When variance returns null, this function returns null.
- stddev_pop — Computes the population standard deviation. This function is the same as the square root of the var_pop function. When var_pop returns null, this function returns null.
- stddev_samp — Computes the sample standard deviation, which is the square root of var_ samp. When var_samp returns null, this function returns null.
- variance — Variance = (sum(expr**2) - (sum(expr)**2) / count(expr)) / (count(expr) - 1) Returns the variance of the expression. If you apply this function to an empty set, it returns null. Netezza SQL calculates the variance of expression as follows:
  - 0 if the number of rows in expression = 1
  - var_samp if the number of rows in expression > 1
- var_pop — Var_pop = (sum(expr**2) - sum(expr)2 / count(expr)) / count(expr) Returns the population variance of a set of numbers after discarding the nulls in this set. If you apply this function to an empty set, it returns null.
- var_ samp — var_samp = (sum (expr**2) - ((sum (expr) **2) / (count (*)))) / (count (*) - 1)
  Returns the sample variance of a set of numbers after discarding the nulls in this set. If you apply this function to an empty set, it returns null.

Lag and Lead Family Syntax

The syntax for the lag and lead family is as follows:

```
{lag | lead}
   (<value expression>, [<offset> [, <default>]]) OVER
   ([partition by <value_expression>[,...]]
   ORDER BY <value_expression> [asc | desc] [nulls {first | last}]
[,...])
```

- lag — Provides access to more than one row of a table at the same time without a self-join. The lag function provides access to a row at a given physical offset prior to that position. If you do not specify the offset, the default is one. The system returns the value in the optional default column, if the offset is beyond the scope of the window. If you do not specify the default, the value is null.
- lead — Provides access to more than one row of a table at the same time without a self-join. The lead function provides access to a row at a given physical offset beyond that position. If you do not specify the offset, the default is one. The system returns the value in the optional default column, if the offset is beyond the scope of the window. If you do not specify the default, the value is null.

The following is an example to illustrate the lag window function. The table monthly_sales is as follows:

```
 YEAR | MONTH_NUM |  SALES
------+-----------+---------
 2011 |         1 | 1000.00
 2011 |         2 |  900.00
```

```
 2011 |         3 | 1200.00
 2011 |         4 | 1200.00
 2011 |         5 | 1250.00
 2011 |         6 | 1200.00
 2011 |         7 |  800.00
 2011 |         8 | 1200.00
 2011 |         9 | 1400.00
 2011 |        10 | 1450.00
 2011 |        11 | 1500.00
 2011 |        12 | 1800.00
 2012 |         1 | 1600.00
 2012 |         2 | 1500.00
 2012 |         3 | 1400.00
 2012 |         4 | 1700.00
```

Within a given year, you can compare a month's sales to the prior month's sales, as in the following example:

```
SELECT year, month_num, sales as current_month_sales, LAG(sales,1)
OVER (partition by year order by month_num) prior_month_sales,
ROUND(100*(current_month_sales - prior_month_sales)/prior_month_
sales,1) percentage_increase from monthly_sales ORDER BY year, month_
num;
YEAR | MONTH_NUM | CURRENT_MONTH_SALES | PRIOR_MONTH_SALES |
PERCENTAGE_INCREASE
------+-----------+---------------------+-------------------+--------
 2011 |         1 |             1000.00 |                   |
 2011 |         2 |              900.00 |           1000.00 | -10.0
 2011 |         3 |             1200.00 |            900.00 | 33.3
 2011 |         4 |             1200.00 |           1200.00 | 0.0
 2011 |         5 |             1250.00 |           1200.00 | 4.2
 2011 |         6 |             1200.00 |           1250.00 | -4.0
 2011 |         7 |              800.00 |           1200.00 | -33.3
 2011 |         8 |             1200.00 |            800.00 | 50.0
 2011 |         9 |             1400.00 |           1200.00 | 16.7
 2011 |        10 |             1450.00 |           1400.00 | 3.6
 2011 |        11 |             1500.00 |           1450.00 | 3.4
 2011 |        12 |             1800.00 |           1500.00 | 20.0
 2012 |         1 |             1600.00 |                   |
 2012 |         2 |             1500.00 |           1600.00 | -6.3
 2012 |         3 |             1400.00 |           1500.00 | -6.7
 2012 |         4 |             1700.00 |           1400.00 | 21.4
(16 rows)
```

This shows how the LAG function can access data from an earlier row in the current partition. Similarly, the LEAD window function can access data from a later row.

First and Last Family Syntax

The syntax for the first and last family is as follows:

```
{first_value | last_value}
({<column reference> | <value expression>} [IGNORE NULLS] ) OVER
([partition by <value_expression>[,...]]
ORDER BY{<value_expression> [asc | desc] [nulls {first | last}]}
[,...]] [ rows | range
{{ current row | unbounded preceding | <literal value> preceding} |
between
{current row | unbounded preceding | <literal value> preceding}
and {current row | unbounded following | <literal value> following}}])
```

- first_value — Returns the first value in an ordered set of values. If the first value in the set is null, the function returns NULL unless you specify IGNORE NULLS. If you specify IGNORE NULLS, FIRST_VALUE returns the first non-null value in the set, or NULL if all values are null.
- last_value — Returns the last value in an ordered set of values. If the last value in the set is null, the function returns NULL unless you specify IGNORE NULLS. If you specify IGNORE NULLS, LAST_VALUE returns the last non-null value in the set, or NULL if all values are null.

Ranking Family Syntax

The syntax for the ranking family is as follows:

```
Function ( ) OVER (
    ([partition by <value_expression> [,...]]
   ORDER BY <value_expression> [asc|desc] [nulls {first | last}]
[,...]))
```

- rank — Calculates the rank of a value in a group of values. Rows with equal values for the ranking criteria receive the same rank. If there are ties, Netezza SQL adds the number of tied rows to the tied rank to calculate the next rank. For example if three people tie for second place, all three would be in second place and the next person would be in fifth place.
- dense_rank — Returns the rank of a row in an ordered group of rows. The ranks are consecutive integers beginning with one. The largest rank value is the number of unique values the query returns. If there are ties, Netezza SQL does not skip rank values, but rather assigns rows with equal values the same rank. For example, if three people tie for second place all three would be in second place and the next person would be in third place.
- percent_rank — This is zero, if the number of rows in the partition is one. Otherwise, it is the (rank of row R)-1, divided by the number of rows in the partition of row R-1.
- cume_dist — This is the number of rows preceding, or is peer with current row R, divided by the number of rows in the partition of current row R.

- ntile (expr) — Divides an ordered set of rows into a number of bins, which are numbered 1 to <expr>. The count of rows in all the bins differs at most by one. This function returns the bin number assigned to the current row. The argument (expr) should evaluate to a numeric constant. If it is not an integral constant, it is rounded to an integer.

Hypothetical Set Family Syntax

The syntax for the hypothetical set family is as follows:

```
SELECT fn(<exprlist>)
WITHIN GROUP ( ORDER BY <value_expression> [asc|desc] [nulls {first |
last}] [,...])) FROM <from_expr>[GROUP BY <group_expr>]
```

These functions are called hypothetical set functions. They take an argument and an order expression. The order expression is evaluated to produce an ordered collection of values. The argument list produces a row R, which is a hypothetical row in this ordered collection.

- dense_rank — Returns the dense rank of a hypothetical row within a group of rows. The ranks are consecutive integers beginning with one. The number of arguments should equal the number of expressions in the ORDER BY clause of WITHIN GROUP. The arguments to dense_rank are treated as if they are of the same data type (any necessary casting happens automatically) as that of the corresponding order-by expression of WITHIN GROUP.
- rank — Calculates the rank of a hypothetical row within a group of values. Rows with equal values for the ranking criteria receive the same rank.
- percent_rank — Calculates the percent rank (between 0 and 1) of a hypothetical row within a group of values. Returns the percent-rank of row R if it were a hypothetical row in the aggregating group (the rank of row R-1, divided by the number of rows in the augmented collection).
- cume_dist — Calculates the cumulative distribution of a hypothetical row within a group of values. Returns the cume_dist of R if it were a hypothetical row in the aggregating group (the number of rows preceding or peers of R, divided by the number of rows in the aggregating group).

The argument expression/sub-expression needs to match the GROUP BY expression exactly. The following example is not exact, and returns an error:

```
SELECT dense_rank(b+a) WITHIN GROUP( order by col) FROM T GROUP BY a+b;
```

The correct expression would be the following:

```
SELECT dense_rank(a+b) WITHIN GROUP( order by col) FROM T GROUP BY a+b;
```

The following example is not exact, and returns an error:

```
SELECT dense_rank(a+b+c) WITHIN GROUP(order by col) FROM T GROUP BY a,
b+c;
```

The correct expression would be the following:

```
SELECT dense_rank(a+(b+c)) WITHIN GROUP(order by col) FROM T GROUP BY
a, b+c;
```

Inverse Distribution Functions Family Syntax

The syntax for the inverse distribution functions family is as follows:

```
SELECT fn(<expr>) WITHIN GROUP(ORDER BY <value_expression> [asc|desc]
[nulls {first | last}]) FROM <from_expr>[GROUP BY <group_expr>];
```

- percentile_cont (expr) — This is computed by considering the pair of consecutive rows that are indicated by the argument, treated as a fraction of the total number of rows in the group, and interpolating the value of the value expression evaluated for these rows. The interpolated value x is obtained using argument y. It is the one sided inverse of percent_rank, and can be used only with numeric and interval datatypes.
  This is computed as follows:
  X = xf(rc-r) + xc(r –rf)
  Where
  Y is the argument to the function
  r = Y *(N-1) + 1
  N is the number of values in the ordered partition/group
  rf = floor(r), rc=ceil(r)
  xf = value of order by column at row rf
  xc = value of order by column at rc

  Note: percentile_cont(0.5) is commonly known as median.

- percentile_disc (expr) — This is computed by treating the group as a window partition of the CUME_DIST window function, using the specified ordering of the value expression as the window ordering, and returning the first value expression whose cumulative distribution value is greater than or equal to the argument. Works with all datatypes.
  X =xc , where r = Y*N , rc = ceil(r), xc is value of order by column at row rc.

  Note: percentile_disc(0) is null.

Miscellaneous Functions Family Syntax

The syntax for the inverse distribution functions family is as follows:

```
width_bucket(<expression>,<beginning of range>, <end of range>,
<number of buckets>)
```

- width_bucket — This function is used to create equi-width histograms. It takes a range and divides it into equi-width intervals, where the number of intervals is specified by num-buckets. It returns the bucket to which the expression belongs. Note that each bucket is a half-closed interval [a, b) on the real line.
  - If the expression is less than the beginning, it is assigned to an underflow bucket (zero (0)).
  - If the expression is greater than the end, it is assigned to the bucket numbuckets+1.
  - All the arguments should evaluate to a numeric value and num-buckets should evaluate to an integral constant.

Examples

This section provides some examples describing how to use analytic functions.

- Window aggregation on a grouping select
  - Selecting the top n within each partition

- Inverse distribution functions
  - Median calculation
- Ranking function examples
- Hypothetical set function
- Ntile function
- Width_bucket function

Sample Table

The following is a sample sales_tbl table:

```
SELECT city, state, region, quarter, amt, profit_margin FROM sales_tbl
ORDER BY city,state, region;
```

```
 CITY         | STATE |  REGION   | QUARTER | AMT  | PROFIT_MARGIN
--------------+-------+-----------+---------+------+---------------
 Atlanta      | GA    | Central   |       4 | 1700 |   16
 Atlanta      | GA    | Central   |       2 | 1500 |   12
 Atlanta      | GA    | Central   |       3 | 1600 |   15
 Atlanta      | GA    | Central   |       1 | 1500 |   12
 Baltimore    | MD    | Central   |       2 | 2500 |   21
 Baltimore    | MD    | Central   |       3 | 2000 |   22
 Baltimore    | MD    | Central   |       4 | 2500 |   19
 Baltimore    | MD    | Central   |       1 | 2000 |   12
 Boston       | MA    | Northeast |       1 | 2000 |   10
 Boston       | MA    | Northeast |       2 | 1500 |   20
 Boston       | MA    | Northeast |       3 | 1700 |   21
 Boston       | MA    | Northeast |       4 | 2400 |   20
 Los Angeles  | CA    | Southwest |       4 | 5300 |   22
 Los Angeles  | CA    | Southwest |       2 | 3000 |   15
 Los Angeles  | CA    | Southwest |       3 | 3500 |   17
 Los Angeles  | CA    | Southwest |       1 | 3000 |   15
 New York     | NY    | Northeast |       4 | 5000 |   20
 New York     | NY    | Northeast |       1 | 3000 |   8
 New York     | NY    | Northeast |       3 | 4300 |   22
 New York     | NY    | Northeast |       2 | 3700 |   20
 Seattle      | WA    | Northwest |       2 | 2200 |   15
 Seattle      | WA    | Northwest |       4 | 2300 |   17
 Seattle      | WA    | Northwest |       3 | 2200 |   18
 Seattle      | WA    | Northwest |       1 | 2000 |   15
(24 rows)
```

Examples — Window Aggregation on a Grouping Select

In this section, we go through the steps for writing a SQL query that finds the top city in each region, based on annual sales totals.

The following example ranks the cities based on their sales amount:

```
SELECT *, RANK() OVER (ORDER BY amt DESC) AS ranking FROM sales_tbl
WHERE region = 'Northeast' or region = 'Central' ORDER BY amt DESC;
```

```
city       | state |  region   | quarter | amt  | ranking
-----------+-------+-----------+---------+------+---------
 New York  | NY    | Northeast |       4 | 5000 |       1
 New York  | NY    | Northeast |       3 | 4300 |       2
 New York  | NY    | Northeast |       2 | 3700 |       3
 New York  | NY    | Northeast |       1 | 3000 |       4
 Baltimore | MD    | Central   |       2 | 2500 |       5
 Baltimore | MD    | Central   |       4 | 2500 |       5
 Boston    | MA    | Northeast |       4 | 2400 |       7
 Boston    | MA    | Northeast |       1 | 2000 |       8
 Baltimore | MD    | Central   |       1 | 2000 |       8
 Baltimore | MD    | Central   |       3 | 2000 |       8
 Atlanta   | GA    | Central   |       4 | 1700 |      11
 Boston    | MA    | Northeast |       3 | 1700 |      11
 Atlanta   | GA    | Central   |       3 | 1600 |      13
 Boston    | MA    | Northeast |       2 | 1500 |      14
 Atlanta   | GA    | Central   |       2 | 1500 |      14
 Atlanta   | GA    | Central   |       1 | 1500 |      14
(16 rows)
```

The following example throws an error, as aggregates are not allowed in the WHERE clause:

```
SELECT *, RANK() OVER (ORDER BY amt DESC) AS ranking FROM sales_tbl
WHERE ranking <= 2;
```

```
ERROR:  Aggregates not allowed in WHERE clause
```

Get the top two by selecting cities with a rank of less than 2:

```
SELECT * FROM ( SELECT *, RANK() OVER (ORDER BY amt DESC) AS ranking
FROM sales_tbl WHERE region = 'Northeast' or region = 'Central') AS
subset WHERE ranking <= 2;
```

```
CITY   | STATE |  REGION   | QUARTER | AMT  | PROFIT_MARGIN | RANKING
-------+-------+-----------+---------+------+---------------+---------
New York | NY | Northeast |       4 | 5000 |            20 |       1
New York | NY | Northeast |       3 | 4300 |            22 |       2
(2 rows)
```

Now we want to rank the cities based on total sales for the year. So first compute total sales for each city for each year:

```
SELECT city, state, region, SUM(amt) AS yr_sales FROM sales_tbl WHERE
region = 'Northeast' or region = 'Central' GROUP BY region, state,
city;
```

```
CITY      | STATE |  REGION   | YR_SALES
----------+-------+-----------+----------
Boston    | MA    | Northeast |     7600
New York  | NY    | Northeast |    16000
Baltimore | MD    | Central   |     9000
```

```
Atlanta   | GA    | Central   |     6300
(4 rows)
```

Now rank the cities on total sales. Note the ORDER BY SUM(amt):

```
SELECT city, state, region, SUM(amt) AS yr_sales, RANK() OVER
(PARTITION BY region ORDER BY SUM(amt)) FROM sales_tbl WHERE region =
'Northeast' or region = 'Central' GROUP BY region, state, city;
CITY    | STATE |  REGION   | YR_SALES | RANK
----------+-------+-----------+----------+------
 Boston    | MA    | Northeast |     7600 |    1
 New York  | NY     | Northeast |    16000 |    2
 Atlanta   | GA    | Central   |     6300 |    1
Baltimore | MD    | Central   |     9000 |    2
(4 rows)
```

Note: When a window aggregate and a GROUP BY clause appear in a single query, the GROUP BY is evaluated first, and the window aggregate function is evaluated on the result of the GROUP BY.

The following example puts it all together to find the top two cities based on total sales:

```
SELECT * FROM (SELECT city, state, region, SUM(amt) AS yr_sales, RANK
() OVER (PARTITION BY region ORDER BY SUM(amt) DESC) AS ranking FROM
sales_tbl WHERE region = 'Northeast' or region = 'Central' GROUP BY
region, state, city) subset WHERE ranking = 1 ORDER BY yr_sales DESC;
city    | state |  region   | yr_sales | ranking
----------+-------+-----------+----------+---------
 New York  | NY    | Northeast |    16000 |       1
 Baltimore | MD    | Central   |     9000 |       1
(2 rows)
```

Examples — Inverse Distribution Functions

The following are examples of inverse distribution functions. Percentile_cont and percentile_disc are not supported as window aggregates.

In this example, for each value of column "grp" find the salary at the 40th percentile. Here we use the percentile_cont function:

```
SELECT grp, percentile_cont(0.4) WITHIN GROUP (ORDER BY sal) AS
fortieth FROM pctest GROUP BY grp;
 GRP | FORTIETH
-----+----------
   1 |     3000
   2 |      980
   3 |     1300
(3 rows)
```

This example uses the percentile_disc function:

```
SELECT grp, percentile_disc(0.4) WITHIN GROUP (ORDER BY sal) AS
fortieth FROM pctest GROUP BY grp;
 GRP | FORTIETH
-----+----------
```

```
   1 |     3000
   2 |      950
   3 |     1250
(3 rows)
```

Note: percentile_disc returns a value for the data set, while percentile_cont returns an interpolated value.

The following calculates the median sales for each region:

```
SELECT region, percentile_cont(0.5) WITHIN GROUP(ORDER BY amt) FROM
sales_tbl GROUP BY region;
 REGION    | PERCENTILE_CONT
-----------+-----------------
Central    |            1850
Northeast |            2700
Northwest |            2200
Southwest |            3250
(4 rows)
```

Examples — Ranking Function

The following is an example of a ranking function:

```
SELECT grp, a, sal, rank() OVER(partition by grp ORDER BY sal), dense_
rank() OVER( partition by grp ORDER BY sal) FROM tab;
GRP  A   SAL RANK PERCENT_RANK DENSE_RANK
--- ---  --- ---- ----------   ----------
1   1    3000 1   0           1
1   2    3000 1   0           1
2   3    800  1   0           1
2   4    950  2   .333333333  2
2   5    1100 3   .666666667  3
2   6    1300 4   1           4
3   7    1250 1   0           1
3   8    1250 1   0           1
3   9    1500 3   .5          2
3   10   1600 4   .75         3
3   11   2500 5   1           4
(11 rows)
```

Note: In line 9 in the example above, a rank is not skipped for the row when dense_rank is computed.

Examples — Hypothetical Set Functions

The following is an example of a hypothetical set function:

```
SELECT grp, percent_rank(1500) WITHIN GROUP(ORDER BY sal) percent_
rankPR, rank(1500) WITHIN GROUP(ORDER BY sal) rank, COUNT(*) FROM
salary_info GROUP BY grp (ORDER BY grp);
```

```
GRP PR  RANK COUNT(*)
--- --- ---- --------
1   0.0000000000000000 1    2
2   1.0000000000000000 5    4
3   0.5000000000000000 3    4
(3 rows)
```

Let's work out the steps. Look at the data from table salary_info:

```
GRP A  SAL
--- -- ----
1   1  3000
1   2  3000
2   3  800
2   4  950
2   5  1100
2   6  1300
3   7  1250
3   8  1250
3   9  1500
3   10 1600
```

For group with grp value =3, the "sal" set ordered by the column value is {1250, 1250, 1500, 1600}. percent_rank's argument provides a hypothetical row. This is "added" to the "sal" set. Now the ordered "sal" set is {1250, 1250, 1500, 1500, 1600}.

percent_rank(1500) is (rank of "1500" in the "sal" set -1)/ (number of rows in group -1) = 3 – 1 / 4 = 2/4 = 0.5

The following is an example of a hypothetical set function with multiple arguments:

```
SELECT region, rank(1800, 15) WITHIN GROUP(ORDER BY amt, profit_
margin), dense_rank(1800, 15) WITHIN GROUP(ORDER BY amt, profit_
margin) FROM sales_tbl GROUP BY region;
 REGION    | RANK | DENSE_RANK
-----------+------+------------
 Central   |    5 |          4
 Northeast |    3 |          3
 Northwest |    1 |          1
 Southwest |    1 |          1
(4 rows)
```

Example — Ntile Function

The following is an example of an ntile function that divides the total sales of each city into four buckets (because there are six data points, the first two buckets each get an extra value):

```
SELECT city, sum(amt), ntile(4) over(ORDER BY sum(amt)) FROM sales_tbl
GROUP BY city;
 CITY         |  SUM   | NTILE
```

```
-------------+-------+-------
Atlanta      |  6300 |     1
Boston       |  7600 |     1
Seattle      |  8700 |     2
Baltimore    |  9000 |     2
Los Angeles | 14800 |     3
New York     | 16000 |     4
(6 rows)
```

Example — Width_Bucket Function

The following is an example of a width_bucket function:

```
SELECT quarter, city, profit_margin, width_bucket(profit_margin, 15,
21, 3) FROM sales_tbl where quarter = 2 or quarter = 3 ORDER BY
quarter, city;
 QUARTER |    CITY     | PROFIT_MARGIN | ?COLUMN?
---------+-------------+---------------+----------
       2 | Atlanta     |            12 |        0
       2 | Baltimore   |            21 |        4
       2 | Boston      |            20 |        3
       2 | Los Angeles |            15 |        1
       2 | New York    |            20 |        3
       2 | Seattle     |            15 |        1
       3 | Atlanta     |            15 |        1
       3 | Baltimore   |            22 |        4
       3 | Boston      |            21 |        4
       3 | Los Angeles |            17 |        2
       3 | New York    |            22 |        4
       3 | Seattle     |            18 |        2
(12 rows)
```

Note: In the example above, in the first row, the profit margin of 12 is less than the beginning value, so it is assigned to the underflow bucket. The row with a profit margin of 22 are greater than the end value, so they are assigned to the overflow bucket.

If you create a hsitogram of the profit margins for each city in quarters 2 and 3, the buckets are as follows: [15, 17) [17, 19) and [19, 21)

CHAPTER 6

Using National Character Sets

What's in this chapter

You can use Netezza SQL to store national characters based on the syntax extensions to SQL:1999, which use Unicode and ISO standards. Using these extensions, you can store Latin and national characters, including Kanji.

Overview

The best known and most widely used character encoding standard is ASCII, which is based on 7-bit byte character strings and has enough characters to encode English text, but no other major written languages.

ISO has standardized several 8-bit extensions of ASCII for various groups of Latin-based writing systems. The ISO standard 8859 also has sections for Cyrillic, Arabic, Greek, Hebrew, and Thai.

- Latin-1 supports Western European languages and is widely used.
- Latin-9 has been adopted (with minor changes to Latin-1) to include the Euro sign (€).

Netezza SQL treats char and varchar as Latin-9 encoding. Latin-9 has replaced Latin-1 as the preferred 8-bit encoding for western European character data.

Latin-9 covers most Western European written languages such as French, Spanish, Catalan, Galician, Basque, Portuguese, Italian, Albanian, Afrikaans, Dutch, German, Danish, Swedish, Norwegian, Finnish, Faroese, Icelandic, Irish, Scottish, English, but none of the central European languages like Polish, Czech, Hungarian, Romanian, and so on. Unicode is the problem-free way to handle written languages that are not in the Latin-9 list.

The Unicode Standard

The Unicode Standard is an effort to encode all the world's characters in one standard. Unicode 4.1 encodes just under 100,000 characters. The Unicode Standard specifies a numeric value and a name for each of its characters. In this respect, it is similar to other character encoding standards such as ASCII.

The range of integers used to code the characters is called the *codespace*. A particular integer in this range is called a *code point*. When a character is mapped or assigned to a particular code point in the codespace, it is referred to as a *coded* character.

The Unicode Standard defines three encoding forms that allow the same data to be stored and transmitted in a byte, word, or double-word-oriented format (that is, in 8-, 16-, or 32-bits per code unit). All three encoding forms encode the same common character repertoire (the actual collection of characters) and can be efficiently transformed into one another without data loss.

The three encoding forms are:

- UTF-8 stores each code point as a single 8-bit unit (the ASCII characters), or as two, three, or four 8-bit sequences.
- UTF-16 stores each code point using either a single 16-bit unit or as a two 16-bit units.
- UTF-32 stores each code point as a 32-bit unit.

All three encoding forms need at most 4 bytes (or 32-bits) of data for each character.

Different writing systems also vary in how they handle collation. Netezza uses binary collation to determine sorting order, which means collating char and nchar data according to the binary character codes

Encoding and Normalization

ASCII characters take 4, 2, and 1 bytes respectively in UTF-32, UTF-16, and UTF-8. The various non-ASCII characters from the ISO Latin character sets take 4, 2, and 2 bytes in these encodings. The common Han (Chinese) characters however, use 3 bytes in UTF-8 and 2 bytes (or, more correctly, 16 bits) in UTF-16. Some rare Han characters take 4 bytes in both UTF-16 and UTF-8.

Unicode allows some characters to be encoded in more than one way — the character À could be the single Unicode character 'Latin Capital Letter A with Accent Grave' or two characters, 'Latin Capital Letter A followed by Combining Grave Accent'. Unicode defines these as canonically equivalent sequences. Because Netezza must treat these two sequences as identical, it does not allow these equivalent sequences in the database.

In previous releases, Netezza would not load data that used combining characters; thus, Netezza could not support languages such as Arabic, Thai, Urdu, and Hindi. Starting in Netezza Release 4.0.3 and later, the nzconvert command has an -nfc switch that allows you to convert input that is in UTF-8, -16, or -32 format to Normalization Form C (NFC) format using the International Components for Unicode (ICU) routines. Netezza will load data that is in NFC format.

To avoid ambiguity, Unicode defines two normalization forms: **Normalization Form C (NFC)** and **Normalization Form D (NFD)**. (For a description, see Unicode Standard Annex #15 on http://www.unicode.org/reports/tr15/ for the specification.) NFD is essentially 'always decompose' and NFC is 'precompose where possible'.

Netezza actually supports a slight superset of NFC, referred to as NFC'. The superset allows the Netezza to support singleton decomposition characters as well, because sometimes the standard conversions from some legacy character encodings result in singletons. For a description of singletons, refer to the Unicode Standard Annex #15.

Netezza Extensions

To allow the usage of national character sets, Netezza has extended its character set support in two ways. Note that the features described below for nchar are the same as those for char.

- The char/varchar data type supports the ISO Latin-9 code set. Sort comparisons use the 8-bit ISO code point values 0-255 for string comparisons. Netezza does not support other ISO/Latin code sets or language-specific collations.
- The ANSI SQL standard nchar/nvarchar (or national char) datatype supports Unicode using the UTF-8 encoding. Collation is in Unicode code point order.

The Data Types

The data types nchar and nvarchar are stored as UTF-8 encoded Unicode (Unicode Version 4.1.0, ISO/IEC 10646:2003).

- The nchar data type specifies fixed-length Unicode data with a maximum length of 16000 characters.
- The nvarchar data type specifies variable-length Unicode data with a maximum length of 16000 characters.

You declare these data types just as you would char/varchars. For example,

```
nchar ( <char-length> ) and nvarchar ( <char-length> )
```

The <char-length> is the maximum number of Unicode characters that the column can hold as distinct from the number of bytes of storage available for the column data.

Using UTF-8 encoding, each Unicode code point can require 1-4 bytes of storage. So a 10-character string requires 10-bytes of storage if it is ASCII, up to 20 bytes if it is Latin, or as many as 40 bytes if it is pure Kanji (but typically 30 bytes).

Netezza supports nchar and nvarchar with declared character widths of up to 16000 characters. Whereas char and varchar can have a maximum character width of 64000 characters.

The system checks for errors based on the lengths declared for nchar-class columns just as it does for char columns. If you attempt to insert or load character data longer than the declared column character width, the system displays an error message.

Syntax Shorthand

The types nchar and nvarchar are shorthand for other syntax.

- nchar(m) is equivalent to:
  - national character(m)
  - national char(m)
- nvarchar(m) is equivalent to:
  - national character varying(m)
  - national char varying(m)

Data Definition Language Effects

Netezza uses the default character and collation (SQL:1999) when you create databases.

```
CREATE DATABASE dbname WITH DEFAULT CHARACTER SET latin9 [ COLLATION
binary ]
```

A binary collation means that the sort sequence is based on the numeric values of the characters defined by the character encoding scheme. Binary sorts are the fastest type of sort.

Data Manipulation Language Effects

Using the nchar and nvarchar character sets affects ordering, sorting, comparing, joining, and aggregating.

- ORDER BY and GROUP BY — The collation is binary.
- Comparisons — All comparisons — joining, filtering, general expressions, and those implicit in grouping and aggregations — are done through binary collation.
- JOIN — Join comparisons use binary collation.
- Aggregations — You can perform min, max, and distinct aggregations of character columns through binary collation.
- Casting — You can cast between char and nchar data types. Since every character that can be represented in char can also be represented in nchar, casting from char to nchar is lossless. For example, cast (<char-col> as nchar(10)). Casting from nchar to char may have characters that cannot be represented in a char column since nchar can store all Unicode and char stores Latin-9. Characters that do not have a Latin-9 representation are converted to a question mark.
- SQL pattern matching — You can perform standard like predicate pattern matching on nchar class data.
- Mixing char and nchar — You can join across char and nchar columns.
- String functions for nchar data — You can use the standard string manipulation functions, such as to_char, to_date, upper, lower, and so on. For more information about these functions, see Table 3-13 on page 3-12 and Table 3-24 on page 3-25.

  Note: Note that string conversion functions such as upper and lower do not use the Unicode Organization SpecialCasing rules; instead, they use the rules in UnicodeData.txt. The SpecialCasing rules sometimes cause undesirable results, such as altering the length of a string.

Loading and Unloading through nzload and External Tables

You can use the nzload command to load national character set data. The source dataobject must contain char-class data in Latin-9 and nchar-class data in correct UTF-8 encoding. For the nzload command, you use the -encoding option to specify the type of data in the file. The -encoding option has three values:

- The default value is 'latin9', which indicates that the whole file is in Latin-9 char/varchar data and has no nchar/nvarchar data. (If the file contains any nchar/nvarchar data, it will be rejected by the load operation.)

- A value of ‘utf8’ indicates the whole file is in UTF-8 encoding and has only nchar/nvarchar data and no char/varchar data. (If the file contains any char/varchar data, it will be rejected by the load operation.)
- The value ‘internal’ indicates that the file could have either or both Latin-9 and UTF-8 data using any or all of the char, varchar, nchar, or nvarchar data types. **As a best practice, Netezza recommends that you always use ‘internal’.**

You can also use the CREATE EXTERNAL TABLE command to load and unload national character set data. The source dataobject must contain char-class data in Latin-9 and nchar-class data in correct UTF-8 encoding. The CREATE EXTERNAL TABLE command uses the same -encoding option and values as the nzload command to identify the content of the file.

External Table Loading

```
CREATE EXTERNAL TABLE xTBL SAMEAS TBL USING (dataobject
('/tmp/TBL.data') delimiter '|' encoding 'internal');
```

Note: If a table definition contains either or both Latin-9 and UTF-8 char/varchar and nchar/nvarchar data, make sure that you specify the encoding as 'INTERNAL'. Also, load and unload using external tables does not perform any codeset conversions. If your data is in a legacy format and needs to be converted to UTF-8, use the **nzconvert** command. For more information about this command, see “Converting Legacy Formats” on page 6-7.

During loading, the system verifies that the UTF-8 data is correctly encoded, which means that a Unicode value is represented in the smallest possible number of bytes, that values do not exceed the maximum length of four UTF-8 bytes, and that the character sequences are well formed. The system also verifies that the data does not include any disallowed characters.

Unloading to an external table copies unmodified column data to the external table’s dataobject.

Understanding Loading Log File Errors

If the system identifies incorrect data during loading (either with the nzload command or the CREATE EXTERNAL TABLE command), it writes a message to the log file. There are two types of errors:

- Malformed UTF-8 — The message displays multiple pairs of hexadecimal digits containing 1-4 bytes. The system encloses the incorrect bytes with square brackets and provides a clue as to why it was rejected.
- Unassigned and unsupported characters — If the characters you attempted to load were unassigned in Unicode or unsupported by Netezza, the system displays the error message as “U + 4 or 5 digits.”

Avoiding Illegal Character Data

It is important to note that various functions could cause you to create illegal character data. For example, you could produce non-NFC' data by using functions such as trim(), substring(), or others to manipulate the data, or by inserting any codepoints using the unichr function, as in the following example:

```
INSERT INTO test values (unichr(101)||unichr(775));
```

These actions could result in illegal character content in the database. If you unload that data, you will not be able to reload it using nzload. Unsupported character data could also prevent future upgrades until the characters are changed to supported values.

Displaying Non-ASCII Characters

You can use Netezza SQL on terminals using the code sets Latin1 or Latin9, UTF8, and code sets supported by the International Components for Unicode library (ICU), specifically Shift-JIS (sjis), Extended Unix Code-JP (euc-jp), Extended Unix Code-CN (euc-cn), and Extended Unix Code-KR (euc-kr) formats.

Netezza SQL converts all data from char, varchar, nchar, and nvarchar columns to the terminal's code set for display. If the selected character does not exist in the terminal's code set then the system displays a question mark.

Specifying the Encoding

To be able to enter and read non-ASCII character sets on the Netezza system, you must specify the encoding. There are two ways to specify the encoding you want Netezza SQL to use:

- Setting the environmental variable, NZ_ENCODING by exporting the environment variable or setting it for the SQL session.
- Setting the locale (Linux **locale** command) that specifies a set of language and cultural rules. You can specify the locale in any of the following ways:
  - You can set the locale LC_ALL setting that sets all the language rules.
  - You can set the locale LC_CTYPE setting that specifies the character handling.
  - You can use the default system locale setting. For the Netezza host running Linux LAS 4.0, the default is en_US.utf8. On the Netezza host running Linux LAS 2.1, the default is en_US that has an encoding of Latin1.

After you have set the encoding, start a terminal emulator by typing the **xterm** command. For more information about using the **xterm** command, see the Red Hat documentation.

Examples

The following examples show how to set encoding for SJIS (Japanese).

- To set the environmental variable for SJIS, enter:

  ```
  export NZ_ENCODING=SJIS
  ```

  If you have set the environment to a valid ICU converter name, Netezza SQL uses it, otherwise it displays an error message and exits.

- To set the session for Japanese, enter:

  ```
  set nz_encoding=sjis
  ```

  The command sets the locale variable for the current session only.

- To set the locale for EUC-JP, enter:

  ```
  export LC_CTYPE=japanese.eucjp
  ```

  The command sets the locale variable.

When you start a Netezza SQL session, the system checks your environment. The system checks the locale codeset setting for the terminal window in which the **nzsql** command is invoked. To display the locale codeset setting, use the Linux command **locale charmap**.

Of course, you can always change the nz_encoding session variable value with an explicit SET command.

- If you set the encoding through the **nzsql** command, you can use mixed case; that is, SET nz_encoding=latin9.
- If you set the encoding at the UNIX shell using the environment variable, use upper case for the variable name; that is, export NZ_ENCODING=LATIN9. The name NZ_ENCODING must be in upper case. The value Latin9 can be in mixed case.

To see all the data, the nz_encoding must be the same as the xterm display; that is, utf8 on a utf-8 xterm and latin9 on a latin9 or latin1 xterm. Netezza SQL converts the output from a char or varchar column to utf-8 when the nz_encoding to set to utf8, and nchar or nvarchar columns to latin9 when nzencoding is set to latin9, substituting "?" if the conversion is not valid.

ODBC Character Set Behavior

You can use the ODBC driver to bind column data to either 8-bit character buffers (ASCII mode) or 16-bit wide character buffers (WIDE mode). For more information, see the *IBM Netezza ODBC, JDBC, and OLE-DB Installation and Configuration Guide*.

Converting Legacy Formats

You can use the **nzconvert** command to convert between any two encodings, between these encodings and UTF-8, and from UTF-32, -16, or -8 to NFC, for loading with the **nzload** command or external tables.

Note: Although the **nzconvert** command can handle all the encodings in the ICU libraries, Netezza has specifically tested the conversion of Shift-JIS (sjis), Extended Unix Code-JP (euc-jp), and Extended Unix Code-KR (windows949) formats.

Using nzconvert

The **nzconvert** command converts the *from* encoding to the *to* encoding. If the program cannot map a legacy character to Unicode, it inserts a user-supplied conversion character. You can specify the maximum substitutions allowed before the system aborts the conversion. The system only logs substitution errors.

nzconvert Options

Table 6-1 describes the options you can use with the **nzconvert** command.

Table 6-1: nzconvert Options

Option	Description
-a[lias] *name*	Reports whether *name* is a valid converter name and lists all its aliases. You can specify any number of -a options.

Table 6-1: nzconvert Options (continued)

Option	Description
-bf *filename*	Specifies the log file for errors only. If there are no errors, the system does not create a log file. The default is standard error.
-bom	Removes any initial Byte Order Mark (BOM) (UTF-8, -16 or -32) from the start of a UTF-8 output file and all other instructed transforms. See "Byte Order Mark."
-df *filename*	Specifies the input filename. The default is standard input.
-f[rom]	Specifies the encoding that you are converting from. You cannot use it with the -l or -a options. If -t is specified, the default is UTF-8.
-h or -?	Prints this help text.
-l[ist]	Lists the primary name of all supported codeset converters. You can use it with any number of -a options.
-maxErrors *n*	Aborts after encountering <n> characters that cannot be converted. The default is 1. Use 0 to cause the load to never abort.
-nfc	For input that is in UTF-8, -16, or -32 format, calls the International Components for Unicode (ICU) routines to convert data to NFC format.
-of *filename*	Specifies the output filename. The default is standard output.
-[rR]ev	Prints the **nzconvert** command's revision number.
-s[ubs] *char*	Sets the substitution character to use when a conversion cannot be done and the number of maxErrors is greater than 1.
-t[o]	Specifies the encoding that you are converting to. You cannot use this option with the -l or -a options. If -f is specified, the default is UTF-8.

Byte Order Mark

Unicode in the 16-bit UTF-16 form has no prescribed endian orientation for interchange. This requires communication processes to evaluate the endian orientation correctly. To aid in this, the character U+FEFF ZERO WIDTH NO-BREAK SPACE may be used as a Byte Order Mark (BOM). When interpreted in the incorrect endian orientation, it evaluates to U+FFFE, which is defined as NOT A CHARACTER.

Some applications, particularly on Windows systems, write a BOM character to the start of a file. In UTF-8, the BOM is the sequence of bytes EF BB BF. As a byte-oriented encoding, there are no endian issues with UTF-8, but some applications – again, primarily on Windows – write the BOM to the start of a UTF-8 encoded file. A Netezza system does not load data that begins with the BOM code point; you can use the -bom switch to remove that code point and load the data.

You can remove a BOM from the start of a UTF-8 file using the nzconvert command, as in the following example:

```
nzconvert -f utf8 -t utf8 -bom -df input_file -of output_file
```

When converting from or to UTF-16, you can use one of three converters: UTF16, UTF16be, or UTF16le as the input (-f option) and output (-t option):

- UTF16 - As input, Netezza checks for a BOM to indicate endianness; otherwise, Netezza interprets the input as big-endian. As output, Netezza writes a BOM and outputs in the native endianness of the machine. When converting from UTF-16 to any other encoding, such as UTF-8, the BOM is removed.
- UTF16le - As input, interprets the input as little-endian. As output, Netezza outputs as little-endian without a BOM. Any BOM is treated as data and converted, such as to UTF-8.
- UTF16be - As input, interprets all input as big-endian. As output, Netezza converts as big-endian without a BOM. Any BOM will be treated as data and converted, such as to UTF-8.

nzconvert Examples

You can use the **nzconvert** command with pipes to convert existing data in legacy code to UTF-8. The following examples show sample commands for nzconvert running on a UNIX system. The command can be used on UNIX as well as Windows Netezza clients.

- To convert and load a data file in sjis, enter:

```
>cat datafile | nzconvert -f sjis | nzload -t table -encoding
internal
```

- To convert and load data from external tables, enter:

```
>mkfifo namedpipe
>cat data file | convert -f sjis > namedpipe&
>nzload -df namedpipe -t jploadtest -encoding internal
```

- To list the full set of codeset aliases supported by nzconvert, enter:

```
>nzconvert -l | grep -v 'Available converters' | awk '{print
"nzconvert -a "  $0}' | bash
```

  All hyphens, underscores, and case differences are ignored in alias names.

CHAPTER 7

Sequences

What's in this chapter

Netezza supports the SQL:2003 standard that allows users to create, alter, and drop named user sequences that exist within a containing database.

Overview of Sequences

A sequence is a named object in a database that supports the *get next value* method. By using sequences, you can generate unique numbers that can be used as surrogate key values for primary key values, where the identification of rows within a table would involve a large, compound primary key, or for other purposes.

A sequence value is an integer that you can use wherever you would use numeric values. Netezza supports user sequences for the four integer types: byteint, smallint, integer, and bigint.

You create a sequence with an initial value, an increment, a minimum and a maximum value. You also specify whether the sequence cycles, which determines whether the sequence starts over when the endpoint is reached.

Note: Sequences always provide a unique sequence number; however, you are not guaranteed that sequence numbers will be predictable, monotonically increasing values. Gaps and unexpected sequence numbers occur as a result of the Netezza topology and how it processes sequences for query performance. For more information, see "Using Sequences in a Distributed System" on page 7-3.

Sequences do not support cross-database access; you cannot obtain a sequence value from a sequence defined in a different database.

Creating a Sequence

Use the CREATE SEQUENCE statement to create a sequence, which is a database object from which users can generate unique numbers.

To create a sequence use the CREATE SEQUENCE statement and specify the options in any order.

Create Sequence

```
CREATE SEQUENCE <sequence name> [as <data type> <options>]
```

Where the options are the following:

```
START WITH <start value>
INCREMENT BY <increment>
no minvalue | minvalue <minimum value>
no maxvalue | maxvalue <maximum value>
cycle | no cycle
```

The options have the following parameters:

- The default minvalue is no minvalue, which is defined to be 1.
- The default maxvalue is no maxvalue and is the largest value by data type that the sequence can hold.
- The default start value is the minvalue for an increasing sequence, and the maxvalue for a decreasing sequence. The startvalue has to be within the range of the minvalue and maxvalue.
- The default increment is 1.
- By default, sequences do not cycle.

When a user generates a sequence number, the system increments the sequence independently of the transaction committing or rolling back. Therefore, a rollback does not return the value to the sequence object. If two users concurrently increment the same sequence, the sequence numbers each user acquires may have gaps because the sequence numbers are being generated by the other user.

Sequences also can have gaps because the Netezza caches sequence values on the host and SPUs for efficient operation. For more information, see "Using Sequences in a Distributed System" on page 7-3.

Sample Creating Sequences

The following are some sample cases of specifying sequences.

- To create a sequence that produces values 1 through 100 in order and the 101st "next value for" results in an error:

```
CREATE SEQUENCE sequence1 as integer
START WITH 1 increment by 1
minvalue 1 maxvalue 100 no cycle
```

- To create a sequence that generates odd values 11 through 99 then cycles and starts again at the minvalue 1. Note that the sequence does not restart at the starting value.

```
CREATE SEQUENCE sequence2 as integer
START WITH 11 increment by 2
minvalue 1 maxvalue 100 cycle
```

- To create a sequence that decreases by 10 and then cycles. Note that the first and second passes return different sets of values — 93, 83,73 ...100, 90, 80 ... 10, 100.

```
CREATE SEQUENCE sequence3 as integer
START WITH 93 increment by -10
minvalue 1 maxvalue 100 cycle
```

Caching Sequences

Although the SQL:2003 standard does not allow you to specify a cache size, some database applications encourage you to change the cache size for better performance. Netezza, on the other hand, derives the cache size of a user sequence based the following characteristics:

- The number of sequence values between the minvalue and the maxvalue
- The number of SPUs in the system

Using Sequences in a Distributed System

Netezza uses the number of sequence values and the number of SPUs to create cache pools of sequence numbers that are distributed to the Netezza host and to each SPU. In the Netezza topology, this is a performance benefit because there is no unnecessary traffic between the SPUs and the host for the next value call.

When a query that runs on the host selects the next value in the sequence, the next available sequence number in the host's cache pool will be allocated. Likewise, when a query is running on a SPU, the next available sequence number in the SPU's cache pool will be allocated. Thus, the next sequence value can sometimes be an unexpected value, because the next value depends upon where the query is running in the Netezza topology.

Sequences can be declared as 8-, 16-, 32-, or 64-bit integers and the cache size is a function of the number of distinct values in one complete cycle of the sequence (so it depends on the declared minvalue, maxvalue, and increment of the sequence).

Understanding the Effects of Sequence Caching

In the following example, assume that sequence *seq* has the default type of 64-bit integer, and each SPU has its own cache of 100,000 values. The select statement retrieves *emp* table rows, where emp is distributed across several data slices. For each row, the nextval sequence number is obtained from the cache on the host or SPU. The output shows how the nextval numbers can differ depending upon which SPU contained the rows of the table, and how there are typically gaps between the sequence numbers.

Sequence caching

```
dev(admin)=> SELECT *, next value for seq FROM emp;
id |     name    |     grp      | nextval
---+-------------+--------------+---------
 4 | John        | mkt          |      49
 8 | Jim K       | sdev         |      52
12 | Jane        | adm          |      54
 1 | Julie       | dev          |  310079
 5 | Jackie      | hdev         |  310080
 9 | Mike        | hdev         |  310088
13 | Jill        | adm          |  310114
 2 | Tom         | adm          |  198235
 6 | Dan         | sdev         |  198312
```

```
10 | Craig       | sdev         |  198331
14 | Judy        | sdev         |  199010
 3 | Chuck       | sdev         |  243522
 7 | Dave        | sdev         |  256673
11 | David       | sdev         |  262004
```

Flushing Cached Values

Sometimes the system must flush the cache, which means that the unused values remaining in the cache are no longer available.

The system flushes the cache when any of the following occurs:

- The system is stopped.
- The system or a SPU crashes.
- You use certain ALTER SEQUENCE statements. For more information, see "Flushing the Cache When Altering a Sequence" on page 7-4.

Altering a Sequence

You can alter a user sequence by resetting any sequence option, including the name and owner of the sequence. To change the starting value, use the RESTART WITH option.

To alter a sequence, use the ALTER SEQUENCE statement and specify the options in any order.

Alter Sequence

```
ALTER SEQUENCE <sequence name> <options>
```

Where the options are the following:

```
OWNER to <new owner>
RENAME TO <new sequence name>
RESTART WITH <start value>
INCREMENT BY <increment>
no minvalue | minvalue <minimum value>
no maxvalue | maxvalue <maximum value>
cycle | no cycle
```

If you alter a sequence while a sequence is in use by a running query, the system waits for the running query's transaction to complete before altering the sequence. This is similar to the way the system handles attempts to drop a table that is in use by a running query.

Flushing the Cache When Altering a Sequence

When you alter a sequence, the system attempts to preserve existing cache loads on the host and the SPUs. The system, however, flushes the cache under the following circumstances:

- The ALTER SEQUENCE statement includes RESTART WITH.
- The ALTER SEQUENCE statement changes the sequence's INCREMENT BY value.
- The ALTER SEQUENCE statement raises the MINVALUE of an ascending sequence.
- The ALTER SEQUENCE statement lowers the MAXVALUE of a descending sequence.

Note: When changing a sequence flushes the cache, you should also change the starting value to ensure more sequential NEXT VALUE results.

Altering a Sequence Increment

When you alter the sequence increment, the system flushes the cache. The following example shows altering the sequence increment from 1 to 2, and the resulting next value.

```
dev(admin)=> CREATE SEQUENCE s;
CREATE SEQUENCE

dev(admin)=> SELECT NEXT VALUE FOR s;
 NEXTVAL
---------
       1
(1 row)

dev(admin)=> ALTER SEQUENCE s INCREMENT BY 2;
ALTER SEQUENCE

dev(admin)=> SELECT NEXT VALUE FOR s;
 NEXTVAL
---------
  100002
(1 row)
```

In the first NEXT VALUE call, the system gave the host a cache load of sequences from 1-100,000. On the ALTER SEQUENCE statement, the system flushed the cache. On the second next value call, the new increment of 2 was added to the last value allocated (100,000), so the system returned the value of 100,002.

Altering the Sequence Sign

If you change the sequence increment from positive to negative or vice versa, the system flushes the cache and the subsequent next value request can return an unexpected result. To avoid this, specify a new value through the RESTART WITH option on an ALTER SEQUENCE statement that changes the increment sign.

For example, suppose you have been using your sequence generator for some time with the increment of 1 and the last cache load the system dispensed was 300,001-400,00. If you were to issue an ALTER SEQUENCE statement setting the increment to -1, your next value would be 399999.

This value is equivalent to a final next value call having returned the last value in the cache (400,000). After you change the increment to-1, the next value call applies to the new increment (-1) to the last value returned (400,000) and so it is 399,999.

Dropping a Sequence

Although you can drop a sequence, remember that system locking occurs if you attempt to drop a sequence that is in use by a running query. In addition, if the sequence is referenced by other metadata, subsequent use of that metadata results in an error, for example if the sequence is referenced in a view definition.

To drop a sequence, use the DROP SEQUENCE statement and specify the sequence name.

Drop Sequence

```
DROP SEQUENCE <sequence name>
```

Sequences and Privileges

The privileges to create, alter, drop, select, and update sequences are as follows:

- The admin user has all privileges on all user sequences. There is no need to grant any privileges to the admin user.
- The owner of the database has all privileges on all user sequences in that database. There is no need to grant any privileges to the owner.

Table 7-1 lists the privileges that you must grant for all other users.

Table 7-1: Sequence Privileges

Command	Privileges
CREATE SEQUENCE	Create Sequence administration permission
ALTER SEQUENCE	Alter object privilege for a specific sequence or the Sequence object class
DROP SEQUENCE	Drop object privilege for a specific sequence or the Sequence object class
NEXT VALUE FOR SEQUENCE	Update object privilege for a specific sequence or the Sequence object class For example, "SELECT NEXT VALUE FOR <sequence name>."

Getting Values from Sequences

After you have established a sequence, you can use the NEXT VALUE FOR and the NEXT <integer expression> VALUES FOR statement to retrieve sequence values.

- The NEXT VALUE FOR statement returns the next available value.
- The NEXT <integer expression> VALUES FOR statement returns the first of a set of contiguous values for the sequence.

Note: Sequences generate unique numbers, but the next number the system generates for any sequence may not be the next number you would expect for that progression. There may or may not be a gap between the last sequence number generated and the next. For example, you are using a sequence where 2 is the common difference, as in "3, 5, 7, 9." The next number you would expect is 11, but the system may return the number 17. This is expected behavior for sequencing.

Getting the Next Value of a Sequence

The syntax for NEXT VALUE is:

```
NEXT VALUE FOR <sequence name>
```

The system returns the next available sequence number.

If there is no next value without overshooting the maxvalue for ascending sequences or undershooting the minvalue for descending sequences, the system does the following:

- For a noncycling sequence, the system displays an error.
- For a cycling sequence, the next value wraps to the minvalue for ascending sequences and wraps to maxvalue for descending sequences.

You cannot use NEXT VALUE in the following statements:

- CASE expressions
- WHERE clauses
- ORDER BY clauses
- aggregate functions
- window functions
- grouped queries
- SELECT distinct

You can use the next value of a sequence of one precision (for example, bigint) to supply a value for a column of a different precision (such as smallint).

Next Value Example

```
CREATE TABLE small_int_table (col1 smallint);
CREATE SEQUENCE bing_int_seq as bigint;
INSERT INTO small_int_table SELECT NEXT VALUE FOR big_int_seq;
```

Note: If the actual value being inserted cannot fit into the lower-precision column, the system displays an error.

Getting Batch Values for a Sequence

To get a batch of values for a sequence, use the NEXT <integer expression> VALUES FOR statement.

```
NEXT <integer expression> VALUES FOR <sequence name>
```

The system returns the single value that is the earliest of a set of <integer expression> contiguous values available for the sequence. Note that the two adjacent values from the batch differ by the increment value.

- If the sequence increment is positive, the system returns the lowest value of the batch of values.
- If the sequence increment is negative, the system returns the highest of the batch values.
- If the sequence does not cycle and the requested batch size exceeds the number of available sequence values, the system displays an error.
- If the sequence cycles and there are not enough values between the current base value and the endpoint of the cycle, the system wraps the base value and assigns the batch from that point, which results in wasting the values skipped to wrap.
- If the batch size exceeds the cycle size of the sequence or the total number of possible sequence values, the system displays an error.

You might want to request batch values for a sequence if you are using ETL tools that require a large number of sequence values and want to avoid the performance impact of requesting sequence values individually. Another advantage of requesting batch sequence values is that when you get values in advance of adding a table row, rows of related tables can cross-reference each other.

Backing Up and Restoring Sequences

Because sequences are part of the scope of a database, backing up the database also backs up the sequence.

Note: You should, however, run backups even when there are no changes to sequences. Because sequences are outside standard transaction, changes to them could be reflected in a concurrent backup.

APPENDIX A

SQL Reserved Words and Keywords

What's in this appendix

- SQL Common Reserved Words
- Nonreserved Keywords

This appendix contains the SQL reserved words and the nonreserved keywords.

SQL Common Reserved Words

The SQL language is composed of reserved words, that is, special words that perform SQL operations. Do not use these reserved words when you name databases, tables, columns, or any other database objects. Table A-1 contains a list of the common reserved words

Table A-1: Reserved Words

ABORT	DEC	LEADING	RESET
ADMIN	DECIMAL	LEFT	REUSE
AGGREGATE	DECODE	LIKE	RIGHT
ALIGN	DEFAULT	LIMIT	ROWS
ALL	DEFERRABLE	LISTEN	ROWSETLIMIT
ALLOCATE	DESC	LOAD	RULE
ANALYSE	DISTINCT	LOCAL	SEARCH
ANALYZE	DISTRIBUTE	LOCK	SELECT
AND	DO	MATERIALIZED	SEQUENCE
ANY	ELSE	MINUS	SESSION_USER
AS	END	MOVE	SETOF
ASC	EXCEPT	NATURAL	SHOW
BETWEEN	EXCLUDE	NCHAR	SOME
BINARY	EXISTS	NEW	SUBSTRING
BIT	EXPLAIN	NOT	SYSTEM
BOTH	EXPRESS	NOTNULL	TABLE
CASE	EXTEND	NULL	THEN
CAST	EXTERNAL	NULLIF	TIES
CHAR	EXTRACT	NULLS	TIME
CHARACTER	FALSE	NUMERIC	TIMESTAMP
CHECK	FIRST	NVL	TO
CLUSTER	FLOAT	NVL2	TRAILING

Table A-1: Reserved Words (continued)

COALESCE	FOLLOWING	OFF	TRANSACTION
COLLATE	FOR	OFFSET	TRIGGER
COLLATION	FOREIGN	OLD	TRIM
COLUMN	FROM	ON	TRUE
CONSTRAINT	FULL	ONLINE	UNBOUNDED
COPY	FUNCTION	ONLY	UNION
CROSS	GENSTATS	OR	UNIQUE
CURRENT	GLOBAL	ORDER	USER
CURRENT_CATALOG	GROUP	OTHERS	USING
CURRENT_DATE	HAVING	OUT	VACUUM
CURRENT_DB	IDENTIFIER_CASE	OUTER	VARCHAR
CURRENT_SCHEMA	ILIKE	OVER	VERBOSE
CURRENT_SID	IN	OVERLAPS	VERSION
CURRENT_TIME	INDEX	PARTITION	VIEW
CURRENT_ TIMESTAMP	INITIALLY	POSITION	WHEN
CURRENT_USER	INNER	PRECEDING	WHERE
CURRENT_USERID	INOUT	PRECISION	WITH
CURRENT_USEROID	INTERSECT	PRESERVE	WRITE
DEALLOCATE	INTERVAL	PRIMARY	RESET
	INTO		REUSE

Note: In addition, Netezza SQL considers the following system attributes reserved words: CTID, OID, XMIN, CMIN, XMAX, CMAX, TABLEOID, ROWID, DATASLICEID, CREATEXID, and DELETEXID.

Nonreserved Keywords

Nonreserved keywords have a special meaning only in particular contexts and can be used as identifiers in other contexts. Most nonreserved keywords are actually the names of built-in tables and functions. Netezza SQL uses nonreserved keywords to attach a predefined meaning to a word in a specific context. Table A-2 contains the list of nonreserved keywords.

Table A-2: Non-reserved Keywords

ABSOLUTE	ACTION	ADD	ADMIN
AFTER	AGGREGATE	ALIAS	ALL
ALLOCATE	ALLOWED	ALTER	AND
ANY	ARE	ARRAY	AS
ASC	ASSERTION	AT	AUTHORIZATION
BEFORE	BEGIN	BINARY	BIT
BLOB	BOOLEAN	BOTH	BREADTH

Table A-2: Non-reserved Keywords

BY	CALL	CASCADE	CASCADED
CASE	CAST	CATALOG	CHAR
CHARACTER	CHECK	CLASS	CLOB
CLOSE	COLLATE	COLLATION	COLUMN
COMMIT	COMPLETION	CONNECT	CONNECTION
CONSTRAINT	CONSTRAINTS	CONSTRUCTOR	CONTINUE
CORRESPONDING	CREATE	CROSS	CUBE
CURRENT	CURRENT_DATE	CURRENT_PATH	CURRENT_ROLE
CURRENT_TIME	CURRENT_ TIMESTAMP	CURRENT_USER	CURSOR
CYCLE	DATA	DATE	DAY
DEALLOCATE	DEC	DECIMAL	DECLARE
DEFAULT	DEFERRABLE	DEFERRED	DELETE
DEPTH	DEREF	DESC	DESCRIBE
DESCRIPTOR	DESTROY	DESTRUCTOR	DETERMINISTIC
DIAGNOSTICS	DICTIONARY	DISCONNECT	DISTINCT
DOMAIN	DOUBLE	DROP	DYNAMIC
EACH	ELSE	END_EXEC	END
EQUALS	ESCAPE	EVERY	EXCEPT
EXCEPTION	EXEC	EXECUTE	EXTERNAL
FALSE	FENCED	FETCH	FINAL
FIRST	FLOAT	FOR	FOREIGN
FOUND	FREE	FROM	FULL
FUNCTION	GENERAL	GET	GLOBAL
GO	GOTO	GRANT	GROUP
GROUPING	HAVING	HOST	HOUR
IDENTITY	IGNORE	IMMEDIATE	IN
INDICATOR	INITIALIZE	INITIALLY	INNER

Table A-2: Non-reserved Keywords

INOUT	INPUT	INSERT	INT
INTEGER	INTERSECT	INTERVAL	INTO
IS	ISOLATION	ITERATE	JOIN
KEY	LANGUAGE	LARGE	LAST
LATERAL	LEADING	LEFT	LESS
LEVEL	LIKE	LIMIT	LOCAL
LOCALTIME	LOCALTIMESTAMP	LOCATOR	MAP
MATCH	MINUTE	MODIFIES	MODIFY
MODULE	MONTH	NAMES	NATIONAL
NATURAL	NCHAR	NCLOB	NEW
NEXT	NO	NONE	NOT
NULL	NUMERIC	OBJECT	OF
OFF	OLD	ON	ONLY
OPEN	OPERATION	OPTION	OR
ORDER	ORDINALITY	OUT	OUTER
OUTPUT	PAD	PARALLEL	PARAMETER
PARAMETERS	PARTIAL	PATH	POSTFIX
PRECISION	PREFIX	PREORDER	PREPARE
PRESERVE	PRIMARY	PRIOR	PRIVILEGES
PROCEDURE	PUBLIC	READ	READS
REAL	RECURSIVE	REF	REFERENCES
REFERENCING	RELATIVE	RESTRICT	RESULT
RETURN	RETURNS	REVOKE	RIGHT
ROLE	ROLLBACK	ROLLUP	ROUTINE
ROW	ROWS	SAVEPOINT	SCHEMA
SCOPE	SCROLL	SEARCH	SECOND
SECTION	SELECT	SEQUENCE	SESSION

Table A-2: Non-reserved Keywords

SESSION_USER	SET	SETS	SIZE
SMALLINT	SOME	SPACE	SPECIFIC
SPECIFICTYPE	SQL	SQLEXCEPTION	SQLSTATE
SQLWARNING	START	STATE	STATEMENT
STATIC	STRUCTURE	SYSTEM_USER	TABLE
TEMPORARY	TERMINATE	THAN	THEN
TIME	TIMESTAMP	TIMEZONE_HOUR	TIMEZONE_ MINUTE
TO	TRAILING	TRANSACTION	TRANSLATION
TREAT	TRIGGER	TRUE	UNDER
UNION	UNIQUE	UNKNOWN	UNNEST
UPDATE	USAGE	USER	USING
VALUE	VALUES	VARCHAR	VARIABLE
VARYING	VIEW	WHEN	WHENEVER
WHERE	WITH	WITHOUT	WORK
WRITE	YEAR	ZONE	

APPENDIX B

Netezza SQL Command Reference

This appendix includes reference information for Netezza SQL commands and functions. Table B-1 describes the commands supported.

The Netezza SQL commands support the SQL-92 standard grammar. Note that although no database product supports the full SQL-92 specification, Netezza SQL supports the broadly implemented portions handled by other common database products. Netezza SQL ensures data integrity while allowing and controlling concurrent access to the data. It also includes SQL additions that take advantage of the Netezza hardware.

Table B-1: Netezza SQL Commands

Command	Description	More Information
ALTER DATABASE	Changes the database name, owner, or the default character set.	See "ALTER DATABASE" on page B-5.
ALTER GROUP	Adds or removes users from a group.	See "ALTER GROUP" on page B-6.
ALTER HISTORY CONFIGURATION	Modify the configuration of query history logging.	See "ALTER HISTORY CONFIGURATION" on page B-10.
ALTER SEQUENCE	Changes the sequence.	See "ALTER SEQUENCE" on page B-15.
ALTER SESSION	Changes the priority of a session and also aborts the active transaction in a session.	See "ALTER SESSION" on page B-17.
ALTER SYNONYM	Changes the owner or renames a synonym.	See "ALTER SYNONYM" on page B-19.
ALTER TABLE	Changes the definition of a table.	See "ALTER TABLE" on page B-20.
ALTER USER	Changes a database user account.	See "ALTER USER" on page B-24.
ALTER VIEW	Changes the owner or name of the view.	See "ALTER VIEW" on page B-28.
BEGIN	Starts a transaction block.	See "BEGIN" on page B-30.

Table B-1: Netezza SQL Commands (continued)

Command	Description	More Information
COMMENT	Defines or change an object's comment.	See "COMMENT" on page B-32.
COMMIT	Commits the current transaction.	See "COMMIT" on page B-34.
COPY	Copies data between files and tables.	See "COPY" on page B-35.
CREATE DATABASE	Creates a new database.	See "CREATE DATABASE" on page B-38.
CREATE EXTERNAL TABLE	Creates an external table for metadata.	See "CREATE EXTERNAL TABLE" on page B-40.
CREATE GROUP	Defines a new user group.	See "CREATE GROUP" on page B-40.
CREATE HISTORY CONFIGURATION	Create a configuration for query history logging.	See "CREATE HISTORY CONFIGURATION" on page B-44.
CREATE MATERIALIZED VIEW	Defines a materialized view.	See "CREATE MATERIALIZED VIEW" on page B-49.
CREATE SEQUENCE	Creates a sequence.	See "CREATE SEQUENCE" on page B-51.
CREATE SYNONYM	Creates a synonym.	See "CREATE SYNONYM" on page B-54.
CREATE TABLE	Defines a new table.	See "CREATE TABLE" on page B-55.
CREATE TABLE AS	Creates a new table based on query results.	See "CREATE TABLE AS" on page B-61.
CREATE USER	Defines a new database user account.	See "CREATE USER" on page B-65.
CREATE VIEW	Defines a new view.	See "CREATE VIEW" on page B-69.
DELETE	Deletes rows of a table.	See "DELETE" on page B-71.
DROP DATABASE	Removes a database.	See "DROP DATABASE" on page B-73.
DROP CONNECTION	Removes a Netezza connection.	See "DROP CONNECTION" on page B-72.
DROP GROUP	Removes a user group.	See "DROP GROUP" on page B-75.

Table B-1: Netezza SQL Commands (continued)

Command	Description	More Information
DROP HISTORY CONFIGURATION	Drop the configuration for query history logging.	See “DROP HISTORY CONFIGURATION” on page B-76.
DROP SEQUENCE	Removes a sequence.	See “DROP SEQUENCE” on page B-77.
DROP SESSION	Stops and removes a session from the Netezza.	See “DROP SESSION” on page B-78.
DROP SYNONYM	Drops a synonym.	See “DROP SYNONYM” on page B-80.
DROP TABLE	Removes a table.	See “DROP TABLE” on page B-81.
DROP USER	Removes a database user account.	See “DROP USER” on page B-82.
DROP VIEW	Removes a view.	See “DROP VIEW” on page B-83.
EXPLAIN	Shows the execution plan of a statement.	See “EXPLAIN” on page B-84.
GENERATE EXPRESS STATISTICS	This command is deprecated starting in Release 4.6.	See “GENERATE EXPRESS STATISTICS” on page B-87.
GENERATE STATISTICS	Collects information on a database, table, or individual column.	See “GENERATE STATISTICS” on page B-89.
GRANT	Defines access privileges.	See “GRANT” on page B-91.
GROOM TABLE	Reclaims and reorganizes tables.	See “GROOM TABLE” on page B-93.
INSERT	Creates new rows in a table.	See “INSERT” on page B-95.
RESET	Restores the value of a runtime parameter to its default value.	See “RESET” on page B-97.
REVOKE	Removes access privileges.	See “REVOKE” on page B-98.
ROLLBACK	Aborts the current transaction.	See “ROLLBACK” on page B-101.
SELECT	Retrieves rows from a table or view.	See “SELECT” on page B-102.
SET	Changes a runtime parameter.	See “SET” on page B-110.

Table B-1: Netezza SQL Commands (continued)

Command	Description	More Information
SET AUTHENTICATION	Configures the user authentication method for users logging on to the system.	See "SET AUTHENTICATION" on page B-112.
SET CONNECTION	Defines connection records for Netezza client users who log on to the Netezza system.	See "SET CONNECTION" on page B-116.
SET HISTORY CONFIGURATION	Specify a configuration for query history logging to take effect after next Netezza software restart.	See "SET HISTORY CONFIGURATION" on page B-117.
SET SESSION	Sets session characteristics including compatibility.	See "SET SESSION" on page B-119.
SET SYSTEM DEFAULT	Sets the system defaults for session timeout, rowset query timeout, and priority.	See "SET SYSTEM DEFAULT" on page B-120.
SET TRANSACTION	Sets the isolation level of the current transaction.	See "SET SYSTEM DEFAULT" on page B-120.
SHOW	Shows the value of a runtime parameter.	See "SHOW" on page B-123.
SHOW AUTHENTICATION	Displays the current user authentication method.	See "SHOW AUTHENTICATION" on page B-124.
SHOW CONNECTION	Displays the Netezza connections that use SSL.	See "SHOW CONNECTION" on page B-126
SHOW HISTORY CONFIGURATION	Display query history configuration settings.	See "SHOW HISTORY CONFIGURATION" on page B-127.
SHOW SESSION	Displays one or more sessions in summary or detail format.	See "SHOW SESSION" on page B-130.
SHOW SYSTEM DEFAULT	Shows the system defaults.	See "SHOW SYSTEM DEFAULT" on page B-132.
TRUNCATE	Empties a table.	See "TRUNCATE" on page B-134.
UPDATE	Updates rows of a table.	See "UPDATE" on page B-135.
VACUUM	Replaced by the generate statistics and the **nzreclaim** commands.	See "GENERATE STATISTICS" on page B-89.

ALTER DATABASE

Use the ALTER DATABASE command to set the default character set for the database, change its name, or change the owner of the database.

Synopsis

Syntax for specifying the default character set:

```
ALTER DATABASE <database_name> SET DEFAULT CHARACTER SET LATIN9
```

Syntax for changing the database owner:

```
ALTER DATABASE <database_name> OWNER TO <user_name>
```

Syntax for renaming the database:

```
ALTER DATABASE <database_name> RENAME TO <new_name>
```

Inputs

The ALTER DATABASE command has the following inputs:

Table B-2: ALTER DATABASE Inputs

Input	Description
database_name	The name of the original database.
new_name	The changed name.
OWNER TO	Specifies the new owner.
RENAME TO	Specifies the new name.
SET DEFAULT CHARACTER SET	If you upgraded before Netezza release 2.2, the default 8-bit character set is specified as UNDECLARED. Set the character set to LATIN9 to be able to compare, join, or cast char/nchar class data.
user_name	The name of the new database owner.

Outputs

The ALTER DATABASE command has the following output:

Table B-3: ALTER DATABASE Output

Output	Description
ALTER DATABASE	The message that the system returns if the command is successful.
ERROR: ALTER DATABASE: database "db" is being accessed by other users	The message that the command returns if you try to alter a database which is being used by other users.

Description

The ALTER DATABASE command has the following characteristics:

Privileges Required

You must be an administrator, or have the Alter Database privilege on the specific database being altered, to use this command.

Common Tasks

Use the ALTER DATABASE command to change the default 8-bit character set, the name, or the owner of the database.

Related Commands

Use the "SET SYSTEM DEFAULT" on page B-120 command to set system-wide limits. See also, "CREATE DATABASE" on page B-38.

Usage

The following provide sample usage:

- To change the default 8-bit character set in the emp database, enter:

```
system(admin)=> ALTER DATABASE emp set default character set
latin9;
```

- To rename the emp database, enter:

```
system(admin)=> ALTER DATABASE emp RENAME TO employees;
```

- To change the owner of the emp database, enter:

```
system(admin)=> ALTER DATABASE emp OWNER TO admin3;
```

ALTER GROUP

Use the ALTER GROUP command to change properties of a group.

Synopsis

Syntax for adding a user:

```
ALTER GROUP <group_name> ADD USER <user_name> [, ... ]
```

Syntax for changing limits:

```
ALTER GROUP <group_name>
[WITH
[ROWSETLIMIT [integer ]
[SESSIONTIMEOUT [integer ]
[QUERYTIMEOUT [integer ]
[RESOURCE MINIMUM resourcepercent]
[RESOURCE MAXIMUM resourcepercent]
[JOB MAXIMUM limit]
[DEFPRIORITY [critical|high|normal|low|none]]
[MAXPRIORITY [critical|high|normal|low|none]]
```

Syntax for dropping a user:

```
ALTER GROUP <group_name> DROP USER <user_name> [, ... ]
```

Syntax for changing the group's owner:

```
ALTER GROUP <group_name> OWNER TO <user_name>
```

Syntax for changing the group's name:

```
ALTER GROUP <group_name> RENAME TO new_group_name
```

Syntax for other inputs, including advanced security features:

```
| COLLECT HISTORY { ON | OFF | DEFAULT }
| CONCURRENT SESSIONS <limit>
| ALLOW CROSS JOIN [TRUE|FALSE|NULL]
| ACCESS TIME { ALL | DEFAULT | ( <access-time>,... )
<access-time> ::= DAY { ALL | <day>, ... } [ <time-bound> ]
<time-bound> ::= START <time-literal> END <time-literal> ]
```

Inputs

The ALTER GROUP command takes the following inputs:

Table B-4: ALTER GROUP Inputs

Input	Description
ADD USER	Adds a user to the group.
DEFPRIORITY	Specifies the default priority for the group. The valid priorities are critical, high, normal, and low.
DROP USER	Drops a user from a group.
group_name	Specifies the name of the group to modify.
MAXPRIORITY	Specifies the maximum priority for the group.
new_group_name	Specifies the new name of the group.
OWNER TO	Specifies the owner of the group.
QUERYTIMEOUT	Specifies the amount of time a query can run before the system sends the administrator a message. You can specify from 1 to 35,791,394 minutes or zero for unlimited. Note that to receive a message, you must enable the Run-AwayQuery event rule. For more information, see the *IBM Netezza System Administrator's Guide*.
RENAME TO	Specifies the group's new name.
ROWSETLIMIT	The rowset limit specifies the maximum number of rows any query run by this user (or group) can return. You can specify from 1 to 2,147,483,647 rows or zero for unlimited.
SESSIONTIMEOUT	Specifies the amount of time a session can be idle before the system terminates it. You can specify from 1 to 35,791,394 minutes or zero for unlimited.

Table B-4: ALTER GROUP Inputs (continued)

Input	Description
RESOURCE MINIMUM <resourcepercent>	This is the minimum amount of the system that a resource group will use when it has jobs. Values range from 0 to 100, representing the percent of total system resources allowed for that group. This corresponds to the previous RESOURCELIMIT setting. For general information, see the "GRA Ceilings" section in the *6.0.x Netezza Release Notes*.
RESOURCE MAXIMUM <resourcepercent>	This limits the amount of the system that a resource group can use. Values range from 1 to 100, representing the percent of total system resources allowed for that group. A group with a maximum of 100 is said to be unlimited. For general information, see the "GRA Ceilings" section in the *6.0.x Netezza Release Notes*.
JOB MAXIMUM <limit>	Limits the number of concurrent jobs run by a single resource group. The maximum number allowed is 48. You can submit more than this value, but they will queue in order. Possible values: • OFF or 0 — Either of these mean there is no limit. • AUTOMATIC or -1 — If the value is Automatic, the value depends on the Resource Maximum. • A positive integer For general information, see the "GRA Ceilings" section in the *6.0.x Netezza Release Notes*.
user_name	Specifies the name of the user to add to the group.
COLLECT HISTORY [ON | OFF | DEFAULT]	Determines whether this group's sessions will collect history. ON indicates history will be collected for this group when connected to a database that also has COLLECT HISTORY ON. OFF indicates history will not be collected for this group. DEFAULT means to examine groups this group is a member of to determine whether to collect history. If any group has COLLECT HISTORY ON, then history is collected when connected to a database that also has COLLECT HISTORY ON. If no group has COLLECT HISTORY ON, but a group has COLLECT HISTORY OFF, then no history is collected. If all groups have DEFAULT history collection, the history is collected. DEFAULT is the default for a group, if the COLLECT HISTORY clause is not specified.
CONCURRENT SESSIONS <limit>	Sets the maximum number of concurrent sessions this group can have. A value of 0 means no limit to the number of concurrent sessions, unless a limit is imposed by a group. In that case, the minimum limit of concurrent sessions across all such groups is used.

Table B-4: ALTER GROUP Inputs (continued)

Input	Description
ALLOW CROSS JOIN [TRUE | FALSE | NULL]	Sets user or group permission to allow explicit cross joins. If NULL is defined for a user, the system checks against the group permission, and takes the lowest non-null value, where FALSE is lower than TRUE. **Note:** This setting involves a system-wide change, so notify all affected users before making this change.
ACCESS TIME ALL	Indicates that this group may start sessions on the Netezza system at any time on any day.
ACCESS TIME DEFAULT	Indicates that access time restrictions are taken from the groups. If no groups have access time restrictions, then the user may start sessions at any time on any day. The access time restriction is evaluated for every group that has one. If any group restricts access, the user may not create a session. That is, the most restrictive access policy is applied.
access-time	Specifies one access time sub-clause; several may be specified. An access time sub-clause defines one or more days by the standard SQL day number (1 = Sunday, 7 = Saturday). The keyword ALL can be use to specify all days of the week; it is equivalent to 1,2,3,4,5,6,7. An access time sub-clause optionally contains one time bound. If no time bound is specified, then the group may create a session at any time on the specified day.
time-bound	Specifies a time range from a start time to an end time. The times can be specified as any valid SQL time literal. It is possible to repeat the same day specification multiple times with different time bounds.

Outputs

The ALTER GROUP command has the following output

Table B-5: ALTER GROUP Output

Output	Description
ALTER GROUP	The message that the system returns if the command is successful.

Description

The ALTER GROUP command has the following characteristics:

Privileges Required

You must be an administrator, or an administrator must have given you the Alter object privilege to use this command.

Common Tasks

Use the ALTER GROUP command to:

- Add users to a group.
- Remove users from a group.
- Alter session idle time, rowset limits, query timeout, and priority.

Note: Users you add or remove from a group must already exist. Using the ALTER GROUP command to remove a user from a group does not drop the user from the system, but only from the specified group.

Related Commands

Use "CREATE GROUP" on page B-40 to create a new group.

Use "DROP GROUP" on page B-75 to remove a group.

Use "SET SYSTEM DEFAULT" on page B-120 to set system-wide limits.

Usage

The following provides sample usage:

- To add the users Karl and John to the group staff, enter:

```
system(admin)=> ALTER GROUP staff ADD USER karl, john;
```

- To change the session idle time, enter:

```
system(admin)=> ALTER GROUP staff WITH SESSIONTIMEOUT 300;
```

- To remove the user Beth from the group workers, enter:

```
system(admin)=> ALTER GROUP workers DROP USER beth;
```

- To change the group's maximum priority, enter:

```
system(admin)=> ALTER GROUP workers WITH MAXPRIORITY critical;
```

ALTER HISTORY CONFIGURATION

Use the ALTER HISTORY CONFIGURATION command to modify a configuration for query history logging. Note that you cannot alter the current configuration.

Synopsis

Syntax for altering the history configuration:

```
ALTER HISTORY CONFIGURATION <config-name> <hist-clause> …
<hist-clause> ::=
| HISTTYPE {QUERY | NONE}
| NPS { LOCAL }
| DATABASE <dbname>
| USER <username>
| PASSWORD <writer-password>
| COLLECT  <history-item> ,...
| LOADINTERVAL {number }
| LOADMINTHRESHOLD {number}
| LOADMAXTHRESHOLD {number}
```

```
| DISKFULLTHRESHOLD {number}
| STORAGELIMIT {number}
| LOADRETRY {number}
| ENABLEHIST {boolean}
| ENABLESYSTEM {boolean}
| VERSION <version>

<history-item>
  QUERY
| PLAN
| TABLE
| COLUMN
```

Inputs

The ALTER HISTORY CONFIGURATION command has the following inputs:

Table B-6: ALTER HISTORY CONFIGURATION Inputs

Input	Description
<config-name>	Specifies the name of the configuration to alter. The configuration must exist on the Netezza system. You cannot alter the current configuration, nor can you change the name of the configuration. (To change a configuration's name, you must drop the configuration and create a new one.) This is a delimited identifier. If not delimited, the system converts the name to the host case.
HISTTYPE	Specifies the type of the database to create, which can be QUERY or NONE. Specify NONE to disable history collection. If you do not specify this input option, the current configuration value is retained.
NPS LOCAL	Store the query history logging information on the local Netezza system. If you do not specify this input option, the current configuration value is retained.
DATABASE <dbname>	Specifies the history database to which the captured data should be written. The database must exist and must have been created with the nzhistcreatedb script command on the Netezza. If you do not specify this input option, the current configuration value is retained. This is a delimited identifier. If not delimited, the system converts the name to the host case.
USER <username>	Specifies the user name for accessing and inserting data to the query history database. (This is the user name specified in the nzhistcreatedb command.) If you do not specify this input option, the current configuration value is retained. This is a delimited identifier. If not delimited, the system converts the name to the host case.

Table B-6: ALTER HISTORY CONFIGURATION Inputs

Input	Description
PASSWORD <writer password>	The password for the database user account. If you do not specify this input option, the current configuration value is retained. This is a single quoted string, and the password is stored as an encrypted string. If the user's password changes, you must update the history configuration with the new password as well, or the loader process will fail.
COLLECT	Specifies the history data to collect. After enabling query history collection, the system *always* collects login failure, session creation, session termination, and the startup of the log capture (alcapp) process. You can specify additional information to collect using this clause: • QUERY—collect the query data. • PLAN—collect plan data from queries. If you specify PLAN, you automatically collect QUERY as well. • TABLE—collect table detail data from queries. If you specify TABLE, you automatically collect QUERY as well. • COLUMN —collect column detail data from queries. If you specify COLUMN, you automatically collect QUERY and TABLE as well. • SERVICE — collect CLI commands • STATE — collect state changes You can specify multiple values using comma-separated values. If you do not specify this input option, the current configuration value is retained. For more information, refer to the chapter on query history in the *IBM Netezza System Administrator's Guide*.
LOADINTERVAL	Specifies the number of minutes to wait before checking the staged area for history data to transfer to the loading area. The valid values are 0 (to disable the timer), or 1 to 60 minutes. There is no default value. If you do not specify this input option, the current configuration value is retained. **Note:** This value works in conjunction with LOADMINTHRESHOLD and LOADMAXTHRESHOLD to configure the loading process. For more information about the settings, refer to the chapter on query history in the *IBM Netezza System Administrator's Guide*.

Table B-6: ALTER HISTORY CONFIGURATION Inputs

Input	Description
LOADMINTHRESHOLD	Specifies the minimum amount of history data in MB to collect before transferring the staged batch files to the loading area. A value of 0 disables the min threshold check. The maximum value is 102400MB (100GB). **Note:** This value works in conjunction with the LOADINTERVAL and LOADMAXTHRESHOLD inputs to configure the loading process timers. For more information about the settings, see the chapter on query history in the *IBM Netezza System Administrator's Guide*..
LOADMAXTHRESHOLD	Specifies the amount of history data in MB to collect before automatically transferring the staged batch files to the loading area. A value of 0 disables the max threshold check. The maximum value is 102400MB (100GB). **Note:** This value works in conjunction with the LOADMINTHRESHOLD and LOADINTERVAL inputs to configure the loading process timers. For more information about the settings, refer to the chapter on query history in the *IBM Netezza System Administrator's Guide*.
DISKFULLTHRESHOLD	This option is reserved for future use. Any value you specify will be ignored. The default value is 0.
STORAGELIMIT	Specifies the maximum size of the history data staging area in MB. If the size of the staging area reaches or exceeds this threshold, history data collection stops until disk space can be freed. The maximum value is 102400MB (100GB). The STORAGELIMIT value must be greater than LOADMAXTHRESHOLD. If you do not specify this input option, the current configuration value is retained. Valid values are 1 to any positive integer.
LOADRETRY	Specifies the number of times that the load operation will be retried. The valid values are 0 (no retry), 1 or 2. If you do not specify this input option, the current configuration value is retained.

Table B-6: ALTER HISTORY CONFIGURATION Inputs

Input	Description
ENABLEHIST	Specifies whether to log information about queries to the query history database. A value of TRUE enables history collection for these queries, and FALSE disables the history collection. If you do not specify this input option, the current configuration value is retained. If you specify FALSE, note that any queries against the history database which have syntax errors will be captured.
ENABLESYSTEM	Specifies whether to log information about system queries. A system queries accesses at least one system table but no user tables. A value of TRUE enables history collection for these queries, and FALSE disables the history collection. If you do not specify this input option, the current configuration value is retained. If you specify FALSE, note that any queries against system tables which have syntax errors will be captured.
VERSION <version>	Specifies the query history schema version of the configuration. By default, this is the query history schema version of the current image. For Release 4.6, the version number is 1. **Note:** The version must match the version number specified in the nzhistcreatedb command; otherwise, the loader process will fail.

Outputs

The ALTER HISTORY CONFIGURATION command has the following outputs:

Table B-7: ALTER HISTORY CONFIGURATION Output

Output	Description
ALTER HISTORY CONFIGURATION	Message returned if the command is successful.
ERROR: permission denied	You must have Manage Security permission to modify query history configuration settings.
ERROR: <config-name> not found.	The specified configuration name could not be found.
ERROR: database <dbname> not found.	The query history database was not found on the Netezza system.

Description

This command changes a query history configuration on a Netezza system. You cannot modify the current/active configuration. Any changes you make to a configuration are saved but they take effect only after you SET to that configuration and restart the Netezza server using the nzstop and nzstart commands.

The ALTER command is logged to the current query history log. The target query database does not need to be empty.

The ALTER HISTORY CONFIGURATION command has the following characteristics:

Privileges Required

You must have Manage Security permissions to alter query history configurations.

Related Commands

See the "CREATE HISTORY CONFIGURATION" on page B-44 to create a new configuration.

See the "DROP HISTORY CONFIGURATION" on page B-76 to drop configurations.

See the "SET HISTORY CONFIGURATION" on page B-117 to specify a configuration for query history logging.

See the "SHOW HISTORY CONFIGURATION" on page B-127 to display information about a configuration.

Usage

The following command changes the type of history data captured to only Query data (the other settings remain unchanged):

```
SYSTEM(ADMIN)=> ALTER HISTORY CONFIGURATION all_hist COLLECT QUERY;
```

ALTER SEQUENCE

Use the ALTER SEQUENCE command to reset a user sequence option, including the name and owner of the sequence. This statement affects only future sequence numbers.

Synopsis

Syntax for altering a sequence

```
ALTER SEQUENCE <sequence name> <options>
```

Options

The ALTER SEQUENCE command takes the following options:

Table B-8: ALTER SEQUENCE Inputs

Input	Description
INCREMENT BY	Changes the increment value.
NO CYCLE | CYCLE	Specifies whether to cycle the sequence.

Table B-8: ALTER SEQUENCE Inputs

Input	Description
NO MAXVALUE | MAXVALUE	Changes the maximum value.
NO MINVALUE | MINVALUE	Changes the minimum value.
OWNER TO	Changes the owner of the sequence.
RENAME TO	Changes the name of the sequence.
RESTART WITH	Retains the original starting value.
sequence_name	Specifies the name of the sequence.

Outputs

The ALTER SEQUENCE command produces the following output:

Table B-9: ALTER SEQUENCE Output

Output	Description
ALTER SEQUENCE	The message that the system returns if the command is successful.
ERROR: ALTER on system sequence not allowed.	The message that the system returns if you attempt to alter a system sequence.

Description

The ALTER SEQUENCE command has the following characteristics:

Privileges Required

The privileges to alter sequences are as follows.

- The admin user has all privileges on all user sequences. There is no need to grant any privileges to the admin user.
- The owner of the database has all privileges on all user sequences in that database. There is no need to grant any privileges to the owner.
- All others must have the Alter object privilege for a specific sequence or the Sequence object class.

Notes

If you change the increment — raise the minvalue for ascending sequences or lower the maxvalue for descending sequences — the system clears the cache, which results in non-sequential sequence numbers. Thus, if you want to retain the original starting value, specify the RESTART WITH option when you use the ALTER SEQUENCE statement.

Related Commands

See “CREATE SEQUENCE” on page B-51 and “DROP SEQUENCE” on page B-77 for related sequence commands.

Usage

The following provides sample usage:

- To change the maximum value, enter:

```
system(admin)=> ALTER SEQUENCE sequence1 MAXVALUE 1000;
```

ALTER SESSION

Use the ALTER SESSION command to abort the active transaction in a session or to set the priority of a session.

Synopsis

Syntax for aborting a transaction in a session:

```
ALTER SESSION [ <session-id> ] ROLLBACK TRANSACTION
```

Syntax for setting the priority of a session:

```
ALTER SESSION [<session_id>] SET PRIORITY TO <priority>
```

Inputs

The ALTER SESSION command takes the following inputs:

Table B-10: ALTER SESSION Inputs

Input	Description
session_id	A number identifying an active session. If <session-id> is not specified, the command alters the current session.
priority	The priority level to set for the session. The priority can be one of: CRITICAL, HIGH, NORMAL or LOW.

Outputs

The ALTER SESSION command produces the following output:

Table B-11: ALTER SESSION Output

Output	Description
ALTER SESSION	The message that the system returns if the command is successful.
ERROR: permission denied	The user does not have permission to change the session priority of the session specified by <session-id>.

Table B-11: ALTER SESSION Output

Output	Description
ERROR: request exceeds maximum priority limit.	The user attempted to raise the session priority beyond their maximum priority limit.
ERROR: id '<session-id>' does not correspond to an existing session.	The specified session ID does not exist.
ERROR: system session id '<session-id>' cannot be aborted	The specified session id refers to a system session. Users cannot rollback transactions of system sessions.
ERROR: access denied. You must have ABORT privileges to perform this action	You do not have permission to rollback the transaction of the session specified by <session-id>.
ERROR: session abort failed for session <session-id>; reason is '<reason>'	The attempt to rollback the transaction in session <session-id> failed; the reason for the failure is provided in the <reason> string.

Description

The ALTER SESSION command has two functions. You can use it to adjust the priority of a session and to rollback the active transaction of a session.

Privileges Required

You need no special privileges to roll back your own session's active transaction or to set its priority up to the maximum priority.

You must be the admin user or have Manage System privilege to change the priority of someone else's session or to exceed the maximum priority of your own session.

Each session is owned by a user. You must be granted Abort privileges on the user in order to rollback their transactions.

Common Tasks

Use the ALTER SESSION command to change the priority of a session.

Related Commands

See "ALTER SESSION" on page B-17 to change a session's priority or to abort a transaction in a session.

See "SHOW SESSION" on page B-130 to display session information.

See "DROP SESSION" on page B-78 to abort and remove a session.

Usage

The following provides sample usage:

- To change the session from normal to critical, enter:

```
system(admin)=> ALTER SESSION SET PRIORITY TO critical;
```

ALTER SYNONYM

Use the ALTER SYNONYM command to rename or change the owner of a synonym.

Synopsis

Syntax for altering a synonym:

```
ALTER SYNONYM <synonym_name> RENAME TO <new_synonym_name>
ALTER SYNONYM <synonym_name> OWNER TO <new_owner>
```

Inputs

The ALTER SYNONYM command takes the following inputs:

Table B-12: ALTER SYNONYM Inputs

Input	Description
RENAME TO	Specifies the new name of the synonym.
OWNER TO	Specifies the name of the new owner.
synonym_name	The name of the synonym.
new_synonym_name	The new name of the synonym.
new_owner	The new owner of the synonym.

Outputs

The ALTER SYNONYM command produces the following output:

Table B-13: ALTER SYNONYM Output

Output	Description
ALTER SYNONYM	The message that the system returns if the command is successful.

Description

The ALTER SYNONYM command has the following characteristics:

Privileges Required

You can alter your own synonyms. To alter other synonyms, you must be the admin user or have been granted the Alter privilege for synonyms.

Common Tasks

Use the ALTER SYNONYM command to rename or change the owner of a synonym.

Related Commands

See "CREATE SYNONYM" on page B-54, "DROP SYNONYM" on page B-80, and "GRANT" on page B-91.

Usage

The following provides sample usage:

- To rename the synonym payroll to pr, enter:

```
system(admin)=> ALTER SYNONYM payroll RENAME TO pr;
```

- To change the owner of the synonym pr, enter:

```
system(admin)=> ALTER SYNONYM pr OWNER TO accounting;
```

ALTER TABLE

Use the ALTER TABLE command to change the structure of an existing table. If the table is in use by an active query, the ALTER command will wait until that query completes.

Synopsis

Syntax for changing a column default value, or dropping a default value:

```
ALTER TABLE <table_name> ALTER [ COLUMN ] <column> { SET DEFAULT
<value> | DROP DEFAULT };
```

Syntax for modifying the column length of a VARCHAR column:

```
ALTER TABLE <table_name> MODIFY COLUMN (column VARCHAR(maxsize));
```

Syntax for renaming a column:

```
ALTER TABLE <table_name> RENAME [ COLUMN ] <column> TO <new_column>;
```

Syntax for organizing columns:

```
ALTER TABLE <table_name> <options> [ORGANIZE ON {(<columns>) | NONE}];
```

Syntax for adding a column:

```
ALTER TABLE <table_name> ADD [COLUMN] column_name type [ column_
constraint [ constraint_characteristics ] ] [, ... ];
```

Column_constraint can be:

```
{ NOT NULL | NULL | UNIQUE | PRIMARY KEY | DEFAULT value |
REFERENCES table [ ( column [, ... ] ) ]
[ MATCH match_type ]
[ ON UPDATE referential_action ]
[ ON DELETE referential_action ] }
```

Match_type can be:

```
{ FULL | PARTIAL }
```

Referential_action can be:

```
{ CASCADE | RESTRICT | SET NULL | SET DEFAULT | NO ACTION }
```

Syntax for dropping a column:

```
ALTER TABLE table_name DROP [COLUMN] column_name [, …] {CASCADE |
RESTRICT };
```

Syntax for renaming a table:

```
ALTER TABLE <table_name> RENAME TO <new_table>;
```

Syntax for changing the table owner:

```
ALTER TABLE <table_name> OWNER TO <name>;
```

Syntax for copying the privileges from one table to another:

```
ALTER TABLE <table_name> SET PRIVILEGES TO <table>;
```

Syntax for adding a constraint:

```
ALTER TABLE <table_name> ADD [constraint_name] <table_constraint>
[constraint_characteristics] ;
```

Syntax for dropping a constraint:

```
ALTER TABLE <table_name> DROP CONSTRAINT name {CASCADE | RESTRICT};
```

Constraint_name can be:

```
 { CONSTRAINT name }
```

Table_constraint can be:

```
{ UNIQUE ( column_name [, ... ] ) |
PRIMARY KEY ( column_name [, ... ] ) |
FOREIGN KEY ( column_name [, ... ] )
REFERENCES table [ ( column [, ... ] ) ]
[ MATCH match_type ]
[ ON UPDATE referential_action ]
[ ON DELETE referential_action ]
}
```

Match_type can be:

```
{ FULL | PARTIAL }
```

Referential_action can be:

```
{ CASCADE | RESTRICT | SET NULL | SET DEFAULT | NO ACTION }
```

Constraint_characteristics can be:

```
{ [ [ NOT ] DEFERRABLE ] { INITIALLY DEFERRED | INITIALLY IMMEDIATE }
|[ INITIALLY DEFERRED | INITIALLY IMMEDIATE ] [ NOT ] DEFERRABLE }
```

Inputs

The ALTER TABLE command takes the following inputs:

Table B-14: ALTER TABLE Inputs

Input	Description
ADD COLUMN	Adds a column. This cannot be used in a transaction block.
ADD CONSTRAINT	Adds a column constraint.
ALTER [COLUMN]	Changes a column default.

Table B-14: ALTER TABLE Inputs (continued)

Input	Description
column	Specifies the name of a new or existing column. You can further specify <column> as follows: <name><datatype> [default <value>] <column_constraint>
DROP COLUMN	Drops a column. You cannot drop a distribution column, an "organize on" column, or the last remaining column in a table. This cannot be used in a transaction block. **Note:** If you drop a column, and wish to reuse the dropped column name, first run the GROOM TABLE <tablename> VERSIONS command to prevent the dropped name from causing errors.
DROP CONSTRAINT	Drops a constraint. If a column is dropped form a table, constraints that rely on the column are automatically dropped as well.
MODIFY COLUMN	Changes a column length.
name	Specifies the new user name.
new_column	Specifies a new name for an existing column.
new_table	Specifies a new name for a table.
ORGANIZE ON {(<columns>) | NONE}]	Specifies which columns (from one to four) the table is to be organized on. Not available for external tables. If columns are specified, the columns cannot be dropped, the table cannot have any materialized views, and all specified column data types must be zone-mappable. When NONE is specified, any organizing key definitions are removed form the host catalog. The table data reorganization takes effect when GROOM TABLE is run. For more information, see "Using Clustered Base Tables" in the *IBM Netezza System Administrator's Guide.*
OWNER TO	Changes the table's owner.
RENAME [COLUMN]	Renames a column name.
RESTRICT	Prevents deletion of referenced rows. Cannot be deferred.
SET PRIVILEGES	Copies the privileges from one table to another.
table_constraint	Specifies the name of the constraint.
table_name	Specifies the name of an existing table to alter.

For more information about constraint options, see "CREATE TABLE" on page B-55.

Outputs

The ALTER TABLE command produces the following output:

Table B-15: ALTER TABLE Output

Output	Description
ALTER	The message that the system returns upon the successful renaming of a table or column.
ERROR	The message that the system returns if the table or column you specify is not available.

Description

The ALTER TABLE command has the following characteristics:

Privileges Required

You must be an administrator, the owner of a table, or an administrator must have given you the Alter object privilege to use this command.

Common Tasks

Use the ALTER TABLE command to:

- Change or drop a column default. Defaults you set only apply to subsequent INSERT commands, not to rows already in the table.
- Rename a column or a table without changing the data type or size within the column or table.

  Note: You can omit the keyword column.
- When you change the name of a table, all views based on the table will cease to work, because views use name binding.
- Add or drop a table or column constraint. Note that you cannot change the name of a constraint. You must instead drop the constraint and create new one.
- Modify the length of a varchar column.

Note: If a table is referenced by a stored procedure, adding or dropping a column is not allowed. You must first drop the stored procedure prior to running the ALTER TABLE command, and then recreate the stored procedure after the table is altered.

Related Commands

See also, “CREATE TABLE” on page B-55.

Usage

The following provides sample usage:

- To change a default, enter:

```
system(admin)=> ALTER TABLE distributors ALTER COLUMN address DROP
DEFAULT;
```

- To modify the length of a varchar column, enter:

```
system(admin)=> ALTER TABLE t3 MODIFY COLUMN (col1 VARCHAR(6));
```

- To rename the column address, enter:

```
system(admin)=> ALTER TABLE distributors RENAME COLUMN address TO
city;
```

- To rename the existing table distributors, enter:

```
system(admin)=> ALTER TABLE distributors RENAME TO suppliers;
```

- To change the table owner, enter:

```
system(admin)=> ALTER TABLE distributors OWNER TO carmen;
```

- To set privileges, enter:

```
system(admin)=> ALTER TABLE distributors SET PRIVILEGES TO carmen;
```

- To add a constraint, enter:

```
system(admin)=> ALTER TABLE distributors ADD CONSTRAINT empkey
PRIMARY KEY(col1) INITIALLY IMMEDIATE;
```

- To drop a constraint, enter:

```
system(admin)=> ALTER TABLE distributors DROP CONSTRAINT empkey
CASCADE;
```

ALTER USER

Use the ALTER USER command to modify a user account.

Synopsis

Syntax for modifying a user's account, including owner, password, optional expiration time for the password, rowset limits, and name:

```
ALTER USER user_name
[WITH
[PASSWORD {'string' | NULL}]
[IN GROUP 'group' [,...]]
[VALID UNTIL 'date' ]
[ROWSETLIMIT [integer ]
[SESSIONTIMEOUT [integer ]
[QUERYTIMEOUT [integer ]
[DEFPRIORITY [critical|high|normal|low|none]]
[MAXPRIORITY [critical|high|normal|low|none]]
[IN RESOURCEGROUP resourcegroupname]

ALTER USER user_name OWNER TO username
ALTER USER user_name RESET ACCOUNT
ALTER USER user_name RENAME TO newname
```

Syntax for other inputs, including advanced security features:

```
| COLLECT HISTORY { ON | OFF | DEFAULT }

| CONCURRENT SESSIONS <limit>

| ALLOW CROSS JOIN [TRUE|FALSE|NULL]
```

```
| ACCESS TIME { ALL | DEFAULT | ( <access-time>,... )
<access-time> ::= DAY { ALL | <day>, ... } [ <time-bound> ]
<time-bound> ::= START <time-literal> END <time-literal> ]
[EXPIRE PASSWORD]
[AUTH [LOCAL | DEFAULT]]
```

Inputs

The ALTER USER command takes the following inputs:

Table B-16: ALTER USER Inputs

Input	Description
date	Specifies the date (and, optionally, the time) when this user's account expires.
DEFPRIORITY	Specifies the default priority for the user. The valid priorities are critical, high, normal, low.
IN GROUP	Specifies the group into which to add a user.
IN RESOURCEGROUP	Specifies the resource group.
MAXPRIORITY	Specifies the maximum priority for the user.
OWNER TO	Specifies the new owner of this account.
QUERYTIMEOUT	Specifies the amount of time a query can run before the system sends the administrator a message. You can specify from 1 to 35,791,394 minutes or zero for unlimited. Note that to receive a message, you must enable the RunAwayQuery event rule. For more information, see the *IBM Netezza System Administrator's Guide*.
RENAME TO	Specifies the new user name for this account.
RESET ACCOUNT	Unlocks the account after the maximum number of logons has been exceeded.
ROWSETLIMIT	Specifies the number of rows a query can return. You can specify from 1 to 2,147,483,647 rows or zero for unlimited. The rowset limit specifies the maximum number of rows any query run by this user (or group) can return.
SESSIONTIMEOUT	Specifies the amount of time a session can be idle before the system terminates it. You can specify from 1 to 35,791,394 minutes or zero for unlimited.

Table B-16: ALTER USER Inputs (continued)

Input	Description
string	Specifies a password for this account. Note the following: • You can set a password when using either LOCAL or LDAP authentication, but the password is used only for LOCAL authentication. • When using LOCAL authentication, the user must have a password to log on. If you change authentication from LDAP to LOCAL, use the ALTER USER command to specify a password for each user so that the user can access the database.
NULL	You can specify WITH PASSWORD NULL to explicitly set a user's password to NULL. A user with a NULL password cannot log on when authentication is set to LOCAL. **Note:** The system stores an empty password as a null password.
COLLECT HISTORY [ON | OFF | DEFAULT	Determines whether this group's sessions will collect history. ON indicates history will be collected for this group when connected to a database that also has COLLECT HISTORY ON. OFF indicates history will not be collected for this group. DEFAULT means to examine groups this group is a member of to determine whether to collect history. If any group has COLLECT HISTORY ON, then history is collected when connected to a database that also has COLLECT HISTORY ON. If no group has COLLECT HISTORY ON, but a group has COLLECT HISTORY OFF, then no history is collected. If all groups have DEFAULT history collection, the history is collected. DEFAULT is the default for a group, if the COLLECT HISTORY clause is not specified.
CONCURRENT SESSIONS <limit>	Sets the maximum number of concurrent sessions this group can have. A value of 0 means no limit to the number of concurrent sessions, unless a limit is imposed by a group. In that case, the minimum limit of concurrent sessions across all such groups is used.
ALLOW CROSS JOIN [TRUE | FALSE | NULL]	Sets user or group permission to allow explicit cross joins. If NULL is defined for a user, the system checks against the group permission, and takes the lowest non-null value, where FALSE is lower than TRUE. **Note:** This setting involves a system-wide change, so notify all affected users before making this change.
ACCESS TIME ALL	Indicates that this group may start sessions on the Netezza system at any time on any day
ACCESS TIME DEFAULT	Indicates that access time restrictions are taken from the groups. If no groups have access time restrictions, then the user may start sessions at any time on any day. The access time restriction is evaluated for every group that has one. If any group restricts access, the user may not create a session. That is, the most restrictive access policy is applied.

Table B-16: ALTER USER Inputs (continued)

Input	Description
access-time	Specifies one access time sub-clause; several may be specified. An access time sub-clause defines one or more days by the standard SQL day number (1 = Sunday, 7 = Saturday). The keyword ALL can be use to specify all days of the week; it is equivalent to 1,2,3,4,5,6,7. An access time sub-clause optionally contains one time bound. If no time bound is specified, then the group may create a session at any time on the specified day.
time-bound	Specifies a time range from a start time to an end time. The times can be specified as any valid SQL time literal. It is possible to repeat the same day specification multiple times with different time bounds.
EXPIRE PASSWORD	Expires the password of the user, requiring them to change their password the next time they log in. If the user is already logged in, this does not affect the current session.
AUTH [LOCAL | DEFAULT]	Overrides the authentication for the user to LOCAL if specified. DEFAULT is the connection setting or whatever authentication is set.

Outputs

The ALTER USER command produces the following output:

Table B-17: ALTER USER Output

Output	Description
ALTER USER	The message that the system returns upon successfully altering the user's account.
ERROR: ALTER USER: user 'username' does not exist	The message that the system returns if the database does not recognize the user.

Description

The ALTER USER command has the following characteristics:

Privileges Required

You must be an administrator to alter the passwords or limits of others, or an administrator must have given you the Alter object privilege. Ordinary users can change only their own passwords and limits.

Common Tasks

Use the ALTER USER command to change attributes of a user's account:

- Set or change a password.
- Set an expiration time for the password.

- Set session timeout, query timeout, and rowset limits.
- Unlock an account after the maximum number of logon attempts has been exceeded.

Note: Whatever attributes you do not mention remain unchanged. For example, if you do not specify an expiration time, the previously set expiration time applies.

The ALTER USER command cannot change a user's group memberships.

Related Commands

Use "CREATE USER" on page B-65 to create a new user.

Use "DROP USER" on page B-82 to drop a user.

Use "ALTER GROUP" on page B-6 to change a user's group memberships.

Use "SET AUTHENTICATION" on page B-112 to set authentication to and from LDAP and LOCAL.

Use "SHOW AUTHENTICATION" on page B-124 to display how the Netezza system is currently configured for authentication.

Use "SET CONNECTION" on page B-116 to specify which Netezza system connections should use an SSL connection.

Use "SHOW CONNECTION" on page B-126 to display the Netezza connections that use SSL.

Use "DROP CONNECTION" on page B-72 to drop a Netezza connection.

Usage

The following provides sample usage.

- To change a user password, enter:

```
system(admin)=> ALTER USER davide WITH PASSWORD 'hu8jmn3';
```

- To set the user's password expiration, enter:

```
system(admin)=> ALTER USER manuel WITH VALID UNTIL 'Jan 31 2030';
```

- To set the user's rowset limits, enter:

```
system(admin)=> ALTER USER mark WITH ROWSETLIMIT 10000;
```

ALTER VIEW

Use the ALTER VIEW command to change the name or the owner of a view.

Synopsis

Syntax for altering a view:

```
ALTER VIEW <viewname> RENAME TO <newname>;
ALTER VIEW <viewname> OWNER TO <newowner>;
ALTER VIEW <view> SET PRIVILEGES TO <view>;
```

To alter materialized views:

```
ALTER VIEW <view> MATERIALIZE REFRESH
ALTER VIEW <view> MATERIALIZE SUSPEND
```

To alter all the materialized views for a base table:

```
ALTER VIEWS ON <table> MATERIALIZE {REFRESH|SUSPEND}
```

Inputs

The ALTER VIEW command takes the following inputs:

Table B-18: ALTER VIEW Input

Input	Description
MATERIALIZE	Specifies a materialized view. You can either refresh it, that is, re-create the materialized table from the base table, or suspend it, that is, truncate the materialized table and redirect all queries against the materialized view to the base table
newname	Specifies the new name of the view.
newowner	Specifies the new owner of the view.
OWNER TO	Specifies the new owner.
RENAME TO	Specifies the new name.
SET PRIVILEGES TO	Copies the privileges from one view to another.
viewname	Specifies the name of the view.
VIEWS ON	Specifies refreshing or suspending all materialized views associated with the base table.

Outputs

The ALTER VIEW command has the following output:

Table B-19: ALTER VIEW Output

Output	Description
ALTER VIEW	Message returned if the command successfully renames or changes the owner of the view.

Description

The ALTER VIEW command has the following characteristics:

Privileges Required

You must be an administrator, or an administrator must have given you the appropriate object privileges, to use this command.

Privileges Required for Materialized Views

You must be an administrator, or the owner of the database. For all other users, Table B-20 lists the privileges you must assign.

Table B-20: Materialized View Privileges

Task	Privilege
Create an SPM view	Assign the Create Materialized View administration privilege.
Alter an SPM view	Assign the Alter object privilege for a specific view or the View object class.
Drop an SPM view	Assign the Drop object privilege for a specific view or the View object class.
Select from an SPM view	Assign the Select object privilege for a specific view or the View object class.
Alter Views on a table	Assign the Insert object privilege for a specific table or the Table object class.
List on SPM views	Assign the List object privilege for a specific view or the View object class.

Common Tasks

Use the ALTER VIEW command to rename or change the owner of a view.

Note: Views are read-only. The system does not allow an insert, update, or delete on a view.

Related Commands

See "DROP VIEW" on page B-83 to drop views.

Usage

The following provides sample usage.

- To rename a view, enter:

```
system(admin)=> ALTER VIEW emp RENAME TO employees
```

- To change the owner of a view, enter:

```
system(admin)=> ALTER VIEW emp OWNER TO john
```

BEGIN

Use the BEGIN command to start a transaction block.

Synopsis

Syntax for starting a transaction block:

```
BEGIN [ WORK | TRANSACTION ]
```

Inputs

The BEGIN command takes the following inputs:

Table B-21: BEGIN Input

Input	Description
WORK	These are optional keywords that have no effect.
TRANSACTION	

Outputs

The BEGIN command has the following output:

Table B-22: BEGIN Output

Output	Description
BEGIN	The message that the system returns when a new transaction has been started.
NOTICE: BEGIN: already a transaction in progress	The message that the system returns indicating a transaction was already in progress. The current transaction is not affected.

Description

The BEGIN command has the following characteristics.

Privileges Required

You need no special privileges to use the BEGIN command.

Common Tasks

Use the BEGIN command to Initiate a user transaction in chained mode. The system executes all user commands after a BEGIN command in a single transaction until an explicit commit, rollback, or execution abort. The system executes commands in chained mode more quickly because transaction start/commit requires significant CPU and disk activity. Chained mode allows consistency when you are executing multiple commands inside a transaction while changing several related tables.

Note: By default, Netezza SQL executes transactions in unchained mode (also known as autocommit). The system executes each user statement in its own transaction, and performs an implicit commit at the end of the statement (if execution was successful, otherwise the system does a rollback).

If the transaction is committed, Netezza SQL ensures either that all updates are done or else that none of them is done. Transactions have the standard ACID (atomic, consistent, isolatable, and durable) property.

Related Commands

Use the "COMMIT" on page B-34 or "ROLLBACK" on page B-101 commands to terminate a transaction.

Usage

The following provides sample usage:

- To begin a user transaction, enter:

```
system(admin)=> BEGIN WORK;
```

COMMENT

Use the COMMENT command to define or change an object's comment.

Synopsis

Syntax for adding a comment:

```
COMMENT ON
[[ DATABASE | SEQUENCE | TABLE | VIEW ]
 object_name | COLUMN table_name.column_name] IS 'text'];
```

Inputs

The COMMENT command takes the following inputs:

Table B-23: COMMENT Input

Input	Description
column_name	Specifies the name of the column.
DATABASE	Specifies the database.
object_name	Specifies the name of the object.
SEQUENCE	Specifies the sequence.
TABLE	Specifies the table.
table_name	Specifies the name of the table.
text	Specifies the comment text.
VIEW	Specifies the view.

Outputs

The COMMENT command has the following output:

Table B-24: COMMENT Output

Output	Description
Comment	The message that the system returns to indicate commenting was successful.

Description

The COMMENT command has the following characteristics:

Privileges Required

You can change comments for objects you own.

You must be an administrator, or an administrator must have given you the Alter object privilege to use this command.

Common Tasks

Use the COMMENT command to:

- Store a comment about a database object. You can retrieve comments with the CLI command \dd.
- Modify a comment by issuing a new COMMENT command for the same object.
- Remove a comment by entering null in place of the text string.

Note: Only one comment string is stored for each object.The system automatically drops comments when you drop an object. Do not put security-critical information into comments. Any user connected to a database can view all comments for objects in the database.

Usage

The following provides sample usage.

- To add a comment to the table mytable, enter:

```
system(admin)=> COMMENT ON my_table IS 'This is my table.';
```

- To add comments to a database, sequence, view, and column, enter:

```
system(admin)=> COMMENT ON DATABASE my_database IS 'Development
Database';
system(admin)=> COMMENT ON SEQUENCE my_sequence IS 'Used to
generate primary keys';
system(admin)=> COMMENT ON VIEW my_view IS 'View of departmental
costs';
system(admin)=> COMMENT ON COLUMN my_table.my_field IS 'Employee ID
number';
```

COMMIT

Use the COMMIT command to commit the current transaction.

Synopsis

Syntax for committing a transaction:

```
COMMIT [ WORK | TRANSACTION ];
```

Inputs

The command takes the following inputs:

Table B-25: COMMIT Input

Input	Description
WORK	These are optional keywords that have no effect.
TRANSACTION	

Outputs

The command has the following output:

Table B-26: COMMIT Output

Output	Description
COMMIT	The message that the system returns to indicate the commit was successful.
NOTICE: COMMIT: no transaction in progress	The system returns this message if there is no transaction in progress. **Note:** Always precede a COMMIT command with a BEGIN command.

Description

The command has the following characteristics:

Privileges Required

You must be an administrator, or an administrator must have given you the appropriate object privileges, to use this command.

Common Tasks

Use the COMMIT command to commit the current transaction. All changes made by the transaction become visible to other users.

Related Commands

Precede a COMMIT command with a BEGIN command. Use the ROLLBACK command to abort a transaction.

Usage

The following provides sample usage.

- To make all changes permanent, enter:

```
COMMIT WORK;
```

- To use the COMMIT command within (at the end of) a transaction, enter:

```
system(admin)=> begin;
BEGIN
system(admin)=> insert into cities values ('Boston',
'Massachusetts');
INSERT 0 1
system(admin)=> insert into cities values ('Houston', 'Texas');
INSERT 0 1
system(admin)=> commit;
COMMIT
system(admin)=>
```

COPY

Use the COPY command to copy data between files and tables. Note that there is no COPY command in SQL92.

Netezza does not recommend using the COPY command. For load/unload operations use the **nzload** command or CREATE EXTERNAL TABLE commands. These commands are faster and more stable than the COPY command. The command is documented here because it is used for internal operations.

Synopsis

Syntax for input coming from a client application:

```
COPY [BINARY] table
    FROM { 'file name' | stdin }
    [ [USING] DELIMITERS 'delimiter' ]
    [ WITH NULL AS 'null string' ];
```

Syntax for output going to a client application:

```
COPY table
    TO { 'file name' | stdout }
    [ [USING] DELIMITERS 'delimiter' ]
    [ WITH NULL AS 'null string' ];
```

Inputs

The COPY command takes the following inputs:

Table B-27: COPY Input

Input	Description
BINARY	Changes the behavior of field formatting. Forces all data to be stored or read in binary format rather than text.
delimiter	Specifies the character that separates fields within each row of the file.
file name	Specifies the absolute path name of the input or output file.
null string	The string that represents a NULL value. The default is "\N" (backslash-N). **Note:** On a copy in, any data item that matches this string is stored as a NULL value. Ensure you use the same string as you used on copy out.
stdin	Specifies that input comes from the client application.
stdout	Specifies that output goes to the client application.
table	Specifies the name of an existing table.

Outputs

The COPY command has the following output

Table B-28: COPY Output

Output	Description
COPY	The system returns this message if the copy completes successfully.
ERROR: reason	If a copy fails, the system returns this message with a reason for the failure.

Description

The command has the following characteristics:

Privileges Required

You must have Select privilege on any table whose values are read by the COPY command.

You must have either Insert or Update privilege to a table into which values are being inserted by the COPY command.

The database server also needs appropriate permissions for any file read or written by the COPY command.

Common Tasks

The COPY command moves data between Netezza tables and standard file-system files. Use the COPY command with a file name to read directly from or write to a file.

- Use the COPY TO command to copy the entire contents of a table to a file. The COPY TO command does not act on column defaults.
- Use the COPY FROM command to copy data from a file to a table. Note that, if there is data in the table already, the command appends to the table.

By default, a text copy uses a tab ("\t") character as a delimiter between fields. You can change the field delimiter to any other single character with the keyword phrase using delimiters. The system backslash quotes characters in data fields that happen to match the delimiter character.

Note: You can use the COPY command with plain tables, but not with views.

The COPY command stops operation at the first error. In the case of the COPY FROM command, this is not an issue. In the case of the COPY TO command, the target relation will already have received earlier rows. These rows are not visible or accessible, but they still occupy disk space. This might amount to a considerable amount of wasted disk space if the failure happens well into a large copy operation.

When using a file name, always specify an absolute path. The database server enforces an absolute path in the case of the COPY TO command, but for the COPY FROM command you have the option of reading from a file specified by a relative path. The system interprets the path relative to the database server's working directory, not the Netezza SQL working directory.

File Formats

The following subsections describe file formats for the COPY command.

Text Format

When you use a COPY command, the system reads or writes a text file with one line per table row. The delimiter character separates columns (attributes) in a row. The attribute values themselves are strings that the output function generates for each attribute's data type, or that are acceptable to the input function. The system uses the specified null-value string in place of attributes that are null.

You can represent end-of-data by a single line containing just backslash-period (\.). An end-of-data marker is not necessary when reading from a UNIX file, but you must provide an end marker when copying data to or from a client application.

You can use backslash characters (\) in the COPY command data to quote data characters that might otherwise be assumed to be row or column delimiters. Specifically, the following characters must be preceded by a backslash if they appear as part of an attribute value: backslash itself, newline, and the current delimiter character.

Table B-29 describes the COPY FROM command backslash sequences.

Table B-29: COPY FROM Backslash Sequences

Sequence	Represents
\b	Backspace (ASCII 8)

Table B-29: COPY FROM Backslash Sequences (continued)

\f	Form feed (ASCII 12)
\n	New line (ASCII 10)
\r	Carriage return (ASCII 13)
\t	Tab (ASCII 9)
\v	Vertical tab (ASCII 11)
\digits	Backslash followed by one to three octal digits specifies the character with that numeric code.

The COPY TO command does not output an octal-digits backslash sequence, but it does use the other sequences listed in Table B-29 for those control characters.

Note: Do not add a backslash before a data character N or period (.). Such pairs are mistaken for the default null string or the end-of-data marker, respectively. Any other backslashed character that is not mentioned in Table B-29 is taken to represent itself.

Netezza recommends that applications generating COPY command data convert data newlines and carriage returns to the \n and \r sequences respectively.

The end of each row is marked by a UNIX-style newline ("\n"). The COPY FROM command does not work properly with a file containing DOS- or Mac-style newlines.

Related Commands

See "CREATE TABLE AS" on page B-61.

Usage

The following provides sample usage.

- To copy a table to standard output, using a vertical bar (|) as the field delimiter, enter:

```
system(admin)=> COPY country TO stdout USING DELIMITERS '|';
```

- To copy data from a UNIX file into the table country, enter:

```
system(admin)=> COPY country FROM '/usr1/proj/bray/sql/country_
data';
```

- This is a sample of data suitable for copying into a table from stdin (so it has the termination sequence on the last line):

```
AF      AFGHANISTAN
AL      ALBANIA
DZ      ALGERIA
ZM      ZAMBIA
ZW      ZIMBABWE
\.
```

CREATE DATABASE

Use the CREATE DATABASE command to create a new database.

Synopsis

Syntax for creating a new database:

```
CREATE DATABASE name
```

```
CREATE DATABASE name
[ WITH
[ DEFAULT CHARACTER SET charset ]
[ DEFAULT CHARACTER SET charset COLLATION collation ]
]
```

Inputs

The CREATE DATABASE command takes the following inputs:

Table B-30: CREATE DATABASE Input

Input	Description
COLLATION	The collation is binary. You cannot specify other values.
DEFAULT CHARACTER SET	Specifies the default character set and collation. The default and only supported value is Latin9. Do not specify other values. For more information on character encoding on the Netezza, see Chapter 6, “Using National Character Sets.”
name	Specifies the name of the database to create.

Outputs

The CREATE DATABASE command has the following output:

Table B-31: CREATE DATABASE Output

Output	Description
CREATE DATABASE	The system returns this message if the command is completed successfully.
ERROR: User 'username' is not allowed to create/drop databases.	The system returns this message if you do not have the correct access to create a database. You must have the Create Database privilege to create databases. See “CREATE USER” on page B-65.
ERROR: Createdb: database ‘name’ already exists.	The system returns this message if a database with the name you specified already exists.
ERROR: Create database: may not be called in a transaction block.	If you have an explicit transaction block in progress, you cannot call create database. You must finish the transaction first.

Table B-31: CREATE DATABASE Output (continued)

Output	Description
ERROR: Unable to create database directory 'path'.	The system returns these messages if you have: • Insufficient permissions on the data directory; you must have access to the location. • A full disk. • Other file system problems.
ERROR: Could not initialize database directory.	

Description

The CREATE DATABASE command has the following characteristics:

Privileges Required

You must be an administrator, or an administrator must have given you the appropriate object privileges, to use this command.

Common Tasks

Use the CREATE DATABASE command to create and become the owner of a new database.

Related Commands

Use "DROP DATABASE" on page B-73 to remove a database.

Usage

The following provides sample usage.

▼ To create a new database, enter:

```
system(admin)=> CREATE DATABASE customers;
```

CREATE EXTERNAL TABLE

Use the CREATE EXTERNAL TABLE command to create a database object that is stored outside of the database. For a complete description of the command and information about loading data into the Netezza system, refer to the *IBM Netezza Data Loading Guide*.

CREATE GROUP

Use the CREATE GROUP command to create a new group.

Synopsis

Syntax for creating a group:

```
CREATE GROUP name
[ WITH
[ SYSID gid ]
[ROWSETLIMIT [integer ]
```

```
[SESSIONTIMEOUT [integer ]
[QUERYTIMEOUT [integer ]
[DEFPRIORITY [critical|high|normal|low|none]]
[MAXPRIORITY [critical|high|normal|low|none]]
[RESOURCE MINIMUM resourcepercent]
[RESOURCE MAXIMUM resourcepercent]
[JOB MAXIMUM limit]
[ USER  username [, ...] ]
]
```

Syntax for other inputs, including advanced security features:

```
| COLLECT HISTORY { ON | OFF | DEFAULT }
| CONCURRENT SESSIONS <limit>
| ALLOW CROSS JOIN [TRUE|FALSE|NULL]
| ACCESS TIME { ALL | DEFAULT | ( <access-time>,... )
<access-time> ::= DAY { ALL | <day>, ... } [ <time-bound> ]
<time-bound> ::= START <time-literal> END <time-literal> ]
```

Inputs

The command takes the following inputs:

Table B-32: CREATE GROUP Input

Input	Description
<name>	Specifies the name of the group to create.
<gid>	Specifies the SYSID clause to choose the group ID of the new group. **Note:** If you do not specify SYSID, the system uses the highest assigned group ID plus one, starting at 1, as the default.
<username>	Specifies existing users to include in the group. **Note:** When specified, this argument must be the *last* argument in the SQL command.
ROWSETLIMIT	The rowset limit specifies the maximum number of rows any query run by this user (or group) can return. You can specify from 1 to 2,147,483,647 rows or zero for unlimited.
SESSIONTIMEOUT	Specifies the amount of time a session can be idle before the system terminates it. You can specify from 1 to 35,791,394 minutes or zero for unlimited.
QUERYTIMEOUT	Specifies the amount of time a query can run before the system sends the administrator a message. You can specify from 1 to 35,791,394 minutes or zero for unlimited. Note that to receive a message, you must enable the RunAwayQuery event rule. For more information, see the *IBM Netezza System Administrator's Guide*.

Table B-32: CREATE GROUP Input (continued)

Input	Description
DEFPRIORITY	Specifies the default priority for the group. The valid priorities are critical, high, normal, low.
RESOURCE MINIMUM <resourcepercent>	This is the minimum amount of the system that a resource group will use when it has jobs. Values range from 0 to 100, representing the percent of total system resources allowed for that group. This corresponds to the previous RESOURCELIMIT setting. For general information, see the "GRA Ceilings" section in the *6.0.x Netezza Release Notes*.
RESOURCE MAXIMUM <resourcepercent>	This limits the amount of the system that a resource group can use. Values range from 1 to 100, representing the percent of total system resources allowed for that group. A group with a maximum of 100 is said to be unlimited. For general information, see the "GRA Ceilings" section in the *6.0.x Netezza Release Notes*.
JOB MAXIMUM <limit>	Limits the number of concurrent jobs run by a single resource group. The maximum number allowed is 48. You can submit more than this value, but they will queue in order. Possible values: • OFF or 0 — Either of these mean there is no limit. • AUTOMATIC or -1 — If the value is Automatic, the value depends on the Resource Maximum. • A positive integer For general information, see the "GRA Ceilings" section in the *6.0.x Netezza Release Notes*.
MAXPRIORITY	Specifies the maximum priority for the group.
COLLECT HISTORY [ON | OFF | DEFAULT	Determines whether the sessions for this group will collect history. • ON collects history for any member of the group when connecting to a database that also has COLLECT HISTORY ON. • OFF specifies that history should not be collected for the group. • DEFAULT specifies to collect history when a user in the group connects to a database that also has COLLECT HISTORY ON. The COLLECT HISTORY setting specified for the user takes precedence over the group setting. When the user setting is DEFAULT, the system uses the group membership to determine the history collection for the user.
CONCURRENT SESSIONS <limit>	Sets the maximum number of concurrent sessions this group can have. A value of 0 means no limit to the number of concurrent sessions, unless a limit is imposed by a group. In that case, the minimum limit of concurrent sessions across all such groups is used.

Table B-32: CREATE GROUP Input (continued)

Input	Description
ALLOW CROSS JOIN [TRUE | FALSE | NULL]	Sets user or group permission to allow explicit cross joins. If NULL is defined for a user, the system checks against the group permission, and takes the lowest non-null value, where FALSE is lower than TRUE. **Note:** This setting involves a system-wide change, so notify all affected users before making this change.
ACCESS TIME ALL	Indicates that this group may start sessions on the Netezza system at any time on any day
ACCESS TIME DEFAULT	Indicates that access time restrictions are taken from the groups. If no groups have access time restrictions, then the user may start sessions at any time on any day. The access time restriction is evaluated for every group that has one. If any group restricts access, the user may not create a session. That is, the most restrictive access policy is applied.
access-time	Specifies one access time sub-clause; several may be specified. An access time sub-clause defines one or more days by the standard SQL day number (1 = Sunday, 7 = Saturday). The keyword ALL can be use to specify all days of the week; it is equivalent to 1,2,3,4,5,6,7. An access time sub-clause optionally contains one time bound. If no time bound is specified, then the group may create a session at any time on the specified day.

Outputs

The command has the following output:

Table B-33: CREATE GROUP Output

Output	Description
CREATE GROUP	The system returns this message if the command completed successfully.

Description

The command has the following characteristics:

Privileges Required

You must be an administrator, or an administrator must have given you the appropriate object privileges, to use this command.

Common Tasks

Use the CREATE GROUP command to create a new group in the database installation.

Related Commands

Use the ALTER GROUP command to change a group's membership, and DROP GROUP to remove a group.

Usage

The following provides sample usage.

- To create an empty group, enter:

```
system(admin)=> CREATE GROUP staff;
```

- To create a group with members, enter:

```
system(admin)=> CREATE GROUP marketing WITH USER jonathan, david;
```

- To set the group's maximum priority, enter:

```
system(admin)=> Create GROUP workers WITH MAXPRIORITY critical;
```

CREATE HISTORY CONFIGURATION

Use the CREATE HISTORY CONFIGURATION command to create a configuration for query history logging on a Netezza system.

Synopsis

Syntax for creating the history configuration:

```
CREATE HISTORY CONFIGURATION <config-name> <hist-clause> …
<hist-clause> ::=
| HISTTYPE {QUERY | NONE}
| NPS { LOCAL }
| DATABASE <dbname>
| USER <username>
| PASSWORD <writer-password>
| COLLECT  <history-item> ,...
| LOADINTERVAL {number }
| LOADMINTHRESHOLD {number}
| LOADMAXTHRESHOLD {number}
| DISKFULLTHRESHOLD {number}
| STORAGELIMIT {number}
| LOADRETRY {number}
| ENABLEHIST {boolean}
| ENABLESYSTEM {boolean}
| VERSION <version>

<history-item>
  QUERY
| PLAN
| TABLE
| COLUMN
```

Inputs

The CREATE HISTORY CONFIGURATION command has the following inputs:

Table B-34: CREATE HISTORY CONFIGURATION Inputs

Input	Description
<config-name>	Specifies the name of the configuration that you want to create. You can create more than one configuration, but names must be unique. This is a delimited identifier. If not delimited, the system converts the name to the host case.
HISTTYPE	Specifies the type of the database to create, which can be QUERY or NONE. Specify NONE to disable history collection. This is a required option which does not have a default value.
NPS LOCAL	Store the query history logging information on the local Netezza system. This is the default and only value.
DATABASE <dbname>	Specifies the history database to which the captured data will be written. The database must exist and must have been created with the nzhistcreatedb script command on the Netezza. There is no default. This is a delimited identifier. If not delimited, the system converts the name to the host case.
USER <username>	Specifies the user name for accessing and inserting data to the query history database. This is the user name specified in the nzhistcreatedb command. There is no default. This is a delimited identifier. If not delimited, the system converts the name to the host case.
PASSWORD <writer password>	The password for the database user account. There is no default. This is a single quoted string, and the password is stored as an encrypted string. If the user's password changes, you must update the history configuration with the new password as well, or the loader process will fail.

Table B-34: CREATE HISTORY CONFIGURATION Inputs

Input	Description
COLLECT	Specifies the history data to collect. After enabling query history collection, the system *always* collects login failure, session creation, session termination, and the startup of the log capture (alcapp) process. You can specify additional information to collect using this clause: • QUERY—collect the query data. • PLAN—collect plan data from queries. If you specify PLAN, you automatically collect QUERY as well. • TABLE—collect table detail data from queries. If you specify TABLE, you automatically collect QUERY as well. • COLUMN —collect column detail data from queries. If you specify COLUMN, you automatically collect QUERY and TABLE as well. • SERVICE — collect CLI commands (used with the Advanced Security feature) • STATE — collect state changes (used with the Advanced Security feature) You can specify multiple values using comma-separated values. If you do not specify this input option, the current configuration value is retained. For more information, refer to the chapter on query history in the *IBM Netezza System Administrator's Guide*.
LOADINTERVAL	Specifies the number of minutes to wait before checking the staged area for history data to transfer to the loading area. The valid values are 0 (to disable the timer), or 1 to 60 minutes. There is no default value. **Note:** This value works in conjunction with LOADMINTHRESHOLD and LOADMAXTHRESHOLD to configure the loading process. For more information about the settings, refer to the chapter on query history in the *IBM Netezza System Administrator's Guide*.
LOADMINTHRESHOLD	Specifies the minimum amount of history data in MB to collect before transferring the staged batch files to the loading area. A value of 0 disables the min threshold check. The maximum value is 102400MB (100GB). **Note:** This value works in conjunction with the LOADINTERVAL and LOADMAXTHRESHOLD inputs to configure the loading process timers. For more information about the settings, refer to the chapter on query history in the *IBM Netezza System Administrator's Guide*.

Table B-34: CREATE HISTORY CONFIGURATION Inputs

Input	Description
LOADMAXTHRESHOLD	Specifies the amount of history data in MB to collect before automatically transferring the staged batch files to the loading area. A value of 0 disables the max threshold check. The maximum value is 102400MB (100GB). **Note:** This value works in conjunction with the LOADMINTHRESHOLD and LOADINTERVAL inputs to configure the loading process timers. For more information about the settings, refer to the chapter on query history in the *IBM Netezza System Administrator's Guide*.
DISKFULLTHRESHOLD	This option is reserved for future use. Any value you specify will be ignored. The default value is 0.
STORAGELIMIT	Specifies the maximum size of the history data staging area in MB. If the size of the staging area reaches or exceeds this threshold, history data collection stops until disk space can be freed. The STORAGELIMIT value must be greater than LOADMAXTHRESHOLD. There is no default. Valid values are 1 to any positive integer. The maximum value is 102400MB (100GB).
LOADRETRY	Specifies the number of times that the load operation will be retried. The valid values are 0 (no retry), 1 or 2. There is no default.
ENABLEHIST	Specifies whether to log information about queries to the query history database. A value of TRUE enables history collection for these queries, and FALSE disables the history collection. There is no default. If you specify FALSE, note that any queries against the history database which have syntax errors will be captured.
ENABLESYSTEM	Specifies whether to log information about system queries. A system queries accesses at least one system table but no user tables. A value of TRUE enables history collection for these queries, and FALSE disables the history collection. There is no default. If you specify FALSE, note that any queries against system tables which have syntax errors will be captured.
VERSION <version>	Specifies the query history schema version of the configuration. By default, this is the query history schema version of the current image. For Release 4.6, the version number is 1. The version must match the version number specified in the nzhistcreatedb command; otherwise, the loader process will fail.

Outputs

The CREATE HISTORY CONFIGURATION command has the following outputs:

Table B-35: CREATE HISTORY CONFIGURATION Output

Output	Description
CREATE HISTORY CONFIGURATION	Message returned if the command is successful.
ERROR: permission denied	You must have Manage Security permission to configure query history logging.
ERROR: database <dbname> not found.	The query history database was not found on the Netezza system.

Description

This command creates a configuration definition for query history logging on a Netezza system. You must create at least one configuration for the current schema version to enable query history logging. This operation itself is not logged in the query history database if it is being set up for the first time for the current query history schema version or if the current history configuration points to a type NONE. The CREATE HISTORY CONFIGURATION command has the following characteristics:

Privileges Required

You must have Manage Security permissions to configure query history logging.

Related Commands

See the "ALTER HISTORY CONFIGURATION" on page B-10 to modify configurations.

See the "DROP HISTORY CONFIGURATION" on page B-76 to drop configurations.

See the "SET HISTORY CONFIGURATION" on page B-117 to specify a configuration for query history logging.

See the "SHOW HISTORY CONFIGURATION" on page B-127 to display information about a configuration.

Usage

Some sample usages of the CREATE HISTORY CONFIGURATION command follow.

The following command creates a history configuration named all_hist which enables the capture of all history information:

```
SYSTEM(ADMIN)=> CREATE HISTORY CONFIGURATION all_hist HISTTYPE QUERY
DATABASE histdb USER histusr PASSWORD histusrpw COLLECT PLAN,COLUMN
LOADINTERVAL 5 LOADMINTHRESHOLD 4 LOADMAXTHRESHOLD 20 VERSION 1;
```

The following command creates a history configuration named hist_mincollect which collects the basic level of history data (login failure, session creation, and termination, and the startup of the alcapp process):

```
SYSTEM(ADMIN)=> CREATE HISTORY CONFIGURATION hist_mincollect HISTTYPE
QUERY DATABASE histdb USER histusr PASSWORD histusrpw COLLECT
LOADINTERVAL 5 LOADMINTHRESHOLD 4 LOADMAXTHRESHOLD 20 VERSION 1;
```

The following command creates a history configuration named hist_queryonly which collects query and plan details and the basic level of information:

```
SYSTEM(ADMIN)=> CREATE HISTORY CONFIGURATION hist_mincollect HISTTYPE
QUERY DATABASE "query db" USER histusr PASSWORD histusrpw COLLECT
QUERY,PLAN LOADINTERVAL 5 LOADMINTHRESHOLD 4 LOADMAXTHRESHOLD 20
VERSION 1;
```

The following command creates a history configuration named hist_disabled that disables history collection:

```
SYSTEM(ADMIN)=> CREATE HISTORY CONFIGURATION hist_disabled HISTTYPE
NONE;
```

CREATE MATERIALIZED VIEW

Use the CREATE MATERIALIZED VIEW command to create sorted, projected, and materialized views. These are views of base tables that project a subset of the base table's columns and are sorted on a specific set of the base table's columns.

Synopsis

Syntax for creating a materialized view:

```
CREATE MATERIALIZED VIEW <viewname> AS SELECT <select_column> [,...]
FROM <table> [ORDER BY <order_columns>[,...]];
CREATE OR REPLACE MATERIALIZED VIEW <viewname> AS SELECT <select_
column> [,...]
FROM <table> [ORDER BY <order_columns>[,...]];
```

Inputs

The CREATE MATERIALIZED VIEW command takes the following inputs:

Table B-36: CREATE MATERIALIZED VIEW Input

Input	Description
CREATE	Creates a materialized view.
CREATE OR REPLACE	Replaces an existing materialized view. You should use this option to rebuild a view after its base table changes in some way or after renaming a database.
order_column	Specifies the column(s) on which to sort.
select_column	Specifies the columns that comprise this view (up to 64).
table	Specifies the table from which the view is created.

Table B-36: CREATE MATERIALIZED VIEW Input

Input	Description
viewname	Specifies the name of the materialized view.

Restrictions

The CREATE MATERIALIZED VIEW command has the following restrictions:

- You can only specify a single base table in the FROM clause.
- You cannot use the WHERE clause when creating a materialized view.
- The columns you specify in the selection list must exist in the base table.
- You must specify at least one column in the selection list.
- You can select up to 64 columns for your materialized view.
- The columns in the ORDER BY list must be specified in the selection list.
- You cannot specify an external, temporary, system, or a clustered base table (CBT) as a base table for the view.

Outputs

The CREATE MATERIALIZED VIEW command has the following output:

Table B-37: CREATE MATERIALIZED VIEW Output

Output	Description
CREATE MATERIALIZED VIEW	Message returned if the command successfully creates the view.
ERROR: Relation 'view' already exists.	Message returned if the view you specified already exists in the database.

Description

The CREATE MATERIALIZED VIEW command requires the following privileges and performs the following tasks:

Privileges Required

You must be an administrator or an administrator must have given you the Create Materialized View administration privilege to use this command.

Common Tasks

Use the CREATE MATERIALIZED VIEW command to define a sorted, projected, materialized view of a subset of the base tables' columns.

Note: The system persistently stores the view and it is visible with the \dm command.

Privileges Required for Materialized Views

You must be an administrator or the owner of the database. For all other users, see Table B-20 on page B-30.

Related Commands

See "DROP VIEW" on page B-83 to drop view. See "ALTER VIEW" on page B-28 to refresh or suspend an existing view.

Usage

The following provides sample usage.

- To create a materialized view, enter:

```
system(admin)=> CREATE MATERIALIZED VIEW kinds AS SELECT t1 FROM
emp ORDER BY name;
```

CREATE SEQUENCE

Use the CREATE SEQUENCE statement to create a sequence. A sequence is a database object from which multiple users can generate unique integers. For more information about creating and using sequences, see Chapter 7, "Sequences."

After you create a sequence, you can access its value in SQL statement with the NEXT VALUE FOR statement (which increments the sequence and returns the new value).

The system generates sequence numbers independent of whether the transaction commits or rolls back. If two users concurrently increment the same sequence, the sequence numbers each user acquires can have gaps, because the other user is generating sequence numbers. No user, however, can every acquire the sequence number generated by another user.

Synopsis

Syntax for creating a sequence:

```
CREATE SEQUENCE <sequence_name>
[ AS datatype ]
[ START WITH start value ]
[ INCREMENT BY increment ]
[ NO MINVALUE | MINVALUE minimum value ]
[ NO MAXVALUE | MAXVALUE maximum value ]
[ NO CYCLE | CYCLE ];
```

Options

The CREATE SEQUENCE command takes the following inputs:

Table B-38: CREATE SEQUENCE Options

Input	Description
datatype	Specifies the data type. The value can be any exact integer type such as byteint, smallint, integer, or bigint. If you do not specify this option, the default datatype is bigint. For a description of the datatypes and their ranges, see Table 3-1 on page 3-2.
START WITH	Specifies the starting value. Use this clause to start an ascending sequence at a value greater than its minimum or to start a descending sequence at a value less than its maximum. For ascending sequences, the default value is the minimum value of the sequence. For descending sequences, the default value is the maximum value of the sequence. This integer value must be between the sequence datatype's minvalue and maxvalue.
INCREMENT BY	Specifies the increment value. The integer value can be any positive or negative integer, but it cannot be zero. The magnitude of this value must be less than the difference of the maxvalue and minvalue. If you specify a positive value, you create an ascending sequence. If you specify a negative value, you create a descending sequence. If you do not specify this option, the default is 1.
NO MINVALUE | MINVALUE	Specifies the minimum value of the sequence. The default is NO MINVALUE, which results in a value of 1. MINVALUE must be less than or equal to START WITH and must be less than MAXVALUE.
NO MAXVALUE | MAXVALUE	Specifies the maximum value that the sequence can have. The default is NO MAXVALUE, which results in the largest value for the specified datatype. MAXVALUE must be equal to or greater than START WITH and must be greater than MINVALUE. For a description of the datatype ranges, see "Integer Types" on page 3-2.
NO CYCLE | CYCLE	Specifies whether the sequence continues to generate values after reaching either its maximum value (in an ascending sequence) or its minimum value (in a descending sequence). The default is NO CYCLE, which means that the sequence stops when it reaches its last value. If you specify CYCLE, then when an ascending sequence reaches it maximum value, it uses its minimum value next. When a descending sequence reaches its minimum value, it uses its maximum value next.

Table B-38: CREATE SEQUENCE Options (continued)

Input	Description
sequence_name	The name of the sequence.

Outputs

The CREATE SEQUENCE command produces the following output:

Table B-39: CREATE SEQUENCE Output

Output	Description
create sequence	The message that the system returns if the command is successful.

Description

The CREATE SEQUENCE command has the following characteristics:

Privileges Required

The privileges to create sequences are as follows:

- The admin user has all privileges on all user sequences. There is no need to grant any privileges to the admin user.
- The owner of the database has all privileges on all user sequences in that database. There is no need to grant any privileges to the owner.
- All others must have the Create Sequence administration permission.

Notes

To create specific sequences, do the following:

- To create an ascending sequence that increments to its maximum datatype value, omit the MAXVALUE or specify NO MAXVALUE. For descending sequences that should decrement to the minimum value of 1, omit the MINVALUE or specify NO MINVALUE.
- To create a sequence that stops at a predefined limit for ascending sequences, specify a value for the MAXVALUE parameter. For descending sequences, specify a value for the MINVALUE. Also specify NO CYCLE. Any attempt to generate a sequence number after the sequence has reached its limit results in an error.
- To create a sequence that restarts after reaching its limit, specify CYCLE.

Related Commands

See “ALTER SEQUENCE” on page B-15 and “DROP SEQUENCE” on page B-77 for related sequence commands.

Usage

The following provides sample usage:

- To create a sequence as an integer with a starting value of 11, increment of 2, min-value of 1, and maxvalue of 100, enter:

```
CREATE SEQUENCE sequence1 As integer START WITH 11 INCREMENT BY 2
MINVALUE 1 MAXVALUE 100 NO CYCLE;
```

CREATE SYNONYM

Use the CREATE SYNONYM statement to create a synonym. A synonym is an alternate way of referencing tables or views. Synonyms allow you to create easy to type names for long table or view names.

Synonyms share the same naming restrictions as tables and views, that is, they must be unique with a database and their names cannot be the same as global objects such as those of databases, users, or groups.

Synopsis

Syntax for creating a synonym:

```
CREATE SYNONYM synonym_name FOR table_reference
```

Options

The CREATE SYNONYM command takes the following inputs:

Table B-40: CREATE SYNONYM Options

Input	Description
synonym_name	Specifies the name of the synonym
table_name	Specifies the table name
FOR	Introduces the table_name

Outputs

The CREATE SYNONYM command produces the following output:

Table B-41: CREATE SYNONYM Output

Output	Description
CREATE SYNONYM	The message that the system returns if the command is successful.

Description

The command has the following characteristics:

Privileges Required

The admin user and the owner of the database can create synonyms. All other users must be granted the Create Synonym privilege.

Notes

The synonym_name is a name that follows the table and view naming conventions. You can create a synonym for a non-existent table or view. At runtime the system expands the table_ reference to its fully qualified form. If the referenced object does not exist, the system displays an error message.

Related Commands

See ALTER SYNONYM and DROP SYNONYM.

Usage

The following provides sample usage:

▼ To create a synonym pr for the table payroll, enter:

```
CREATE SYNONYM pr FOR payroll;
```

CREATE TABLE

Use the CREATE TABLE command to define a new table.

Synopsis

General syntax for the create table command:

```
CREATE [ TEMPORARY | TEMP ] TABLE table_name
(
column_name type [ [ constraint_name ] column_constraint [ constraint_
characteristics ] ] [, ... ]
[ [ constraint_name ] table_constraint [ constraint_characteristics ]
] [, ... ]
    ) [ DISTRIBUTE ON ( column [, ...] ) ]
[ORGANIZE ON {(<column>) | NONE}]
```

Where constraint_name can be:

```
{CONSTRAINT name}
```

Where column_constraint can be:

```
{ NOT NULL | NULL | UNIQUE | PRIMARY KEY | DEFAULT value |
              REFERENCES table [ ( column [, ... ] ) ]
    [ MATCH match_type ]
    [ ON UPDATE referential_action ]
    [ ON DELETE referential_action ]
}
```

Where table_constraint can be:

```
{ UNIQUE ( column_name [, ... ] ) |
PRIMARY KEY ( column_name [, ... ] ) |
FOREIGN KEY ( column_name [, ... ] )
REFERENCES table [ ( column [, ... ] ) ]
[ MATCH match_type ]
[ ON UPDATE referential_action ]
[ ON DELETE referential_action ]
}
```

Where match_type can be:

```
{FULL | PARTIAL}
```

Where referential_action can be:

```
{CASCADE | RESTRICT | SET NULL | SET DEFAULT | NO ACTION}
```

Where constraint_characteristics can be:

```
{ [ [ NOT ] DEFERRABLE ] { INITIALLY DEFERRED | INITIALLY IMMEDIATE } |
    [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ] [ NOT ] DEFERRABLE }
```

Note: The system permits and maintains primary key, default, foreign key, unique, and references. The Netezza does not support constraint checks and referential integrity. The user must ensure constraint checks and referential integrity.

Inputs

The CREATE TABLE command takes the following inputs:

Table B-42: CREATE TABLE Input

Input	Description
column_name	Specifies the name of a column create in the new table.
constraint_name	Specifies a name for a column or table constraint. The system generates a name if you do not specify one.
INITIALLY checktime	Specifies either DEFERRED (at the end of the transaction) or IMMEDIATE (at the end of each statement).
Match match type	Specifies the match type, which can be MATCH FULL, MATCH PARTIAL, and the default. • Match Full prevents one column from a multicolumn foreign key from being null if other parts of the foreign key are not null. • MATCH PARTIAL is unsupported.
NOT DEFERRABLE	Controls where the constraint can be deferred to the end of the transaction. NOT DEFERRABLE is the default. (Netezza does not support constraint checking and referential integrity.)
NOT NULL	Specifies that the column is not allowed to contain null values.
NULL	Specifies that the column is allowed to contain null values. This is the default.
ON DELETE	Specifies the action to take then a reference table is deleted. • NO ACTION produces an error if the foreign key is violated. This is the default. • RESTRICT is the same as NO ACTION. • CASCADE deletes any rows referencing the deleted row. • SET NULL sets the referencing column values to their default value. • SET DEFAULT sets the referencing column values to their default value.

Table B-42: CREATE TABLE Input

Input	Description
ON UPDATE	Specifies the action when a referenced column in the referenced table is updated to a new value. The actions are the same as those of the ON DELETE parameter.
ORGANIZE ON {(<columns>) | NONE}]	Specifies which columns (from one to four) the table is to be organized on. Not available for external tables. If columns are specified, the table cannot have any materialized views, and all specified column data types must be zone-mappable. The table data reorganization takes effect when GROOM TABLE is run. For more information, see "Using Clustered Base Tables" in the *IBM Netezza System Administrator's Guide.*
primary key (column_ constraint) primary key (column_name [, ...]) (table_ constraint)	The primary key constraint specifies that a column or columns of a table may contain only unique (non-duplicate), non-null values. **Note:** The primary key constraint is virtually a combination of the unique and not null constraints, but identifying a set of columns as a primary key also provides metadata about the design of the schema. A primary key implies that other tables may rely on this set of columns as a unique identifier for rows. You can specify only one primary key constraint for a table, whether as a column constraint or a table constraint. The primary key constraint should name a set of columns that is different from other sets of columns named by any unique constraint defined for the same table.
references table [(column)] foreign key(column_name [, ...]) references table [(column [, ...])] (table_ constraint)	Specifies that a group of one or more columns of the new table must only contain values that match against values in the referenced column(s) of the referenced table. If you omit column, the primary key of the table is used. The referenced columns must be the columns of a unique or primary key constraint in the referenced table.
table_name	Specifies the name of the table to create.
type	Specifies the data type of the column. See Chapter 3, "Netezza SQL Basics," for information about data types.

Outputs

The command has the following output:

Table B-43: CREATE TABLE Output

Output	Description
CREATE TABLE	The system returns this message if the command completes successfully.

Table B-43: CREATE TABLE Output

Output	Description
ERROR	The system returns this message if table creation fails. The error message provides descriptive text, such as: error: Relation 'table' already exists.

Description

The command has the following characteristics:

Privileges Required

The user who issues the CREATE TABLE command owns the resultant table. You must be an administrator, or an administrator must have given you the appropriate object privileges, to use this command.

Common Tasks

Use the CREATE TABLE command to create a new, initially empty table in the current database. The CREATE TABLE command automatically creates a data type that represents the tuple type (structure type) corresponding to one row of the table.

Note that a table cannot have the following:

- The same name as any existing data type.
- The same name as a system catalog table.
- More than 1600 columns. In practice, the effective limit is lower because of tuple-length constraints.
- Table or view attributes with the following names: ctid, oid, xmin, cmin, xmax, cmax, tableoid, rowid, datasliceid, createxid, and deletexid

The optional constraint clauses specify constraints (or tests) that new or updated rows must satisfy for an insert or update operation to succeed. A constraint is a named rule; that is, a SQL object that helps define valid sets of values by limiting the results of insert, update, or delete operations performed on a table. Netezza does not support constraint checks; if you specify constraints, you must perform the constraint checking and referential integrity.)

You can define table constraints and column constraints.

- A column constraint is defined as part of a column definition.
- A table constraint definition is not tied to a particular column, and it can encompass more than one column.

You can also write every column constraint as a table constraint. A column constraint is only a notational convenience if the constraint only affects one column.

Distribution Specification

Each table in a Netezza RDBMS database has only one distribution key, which consists of one to four columns. You can use the following SQL syntax to create distribution keys.

- To create an explicit distribution key, the Netezza SQL syntax is:

usage: create table <tablename> [(<column> [, …])] as <select_clause>
[distribute on [hash] (<column> [,…])]
The phrase distribute on specifies the distribution key, the word hash is optional.

- To create a round-robin distribution key, the Netezza SQL syntax is:
  usage: create table <tablename> (col1 int, col2 int, col3 int)
  distribute on random;
  The phrase distribute on random specifies round-robin distribution.
- To create a table without specifying a distribution key, the Netezza SQL syntax is:
  usage: create table <tablename> (col1 int, col2 int, col3 int);
  The Netezza chooses a distribution key. There is no guarantee what that key is and it can vary depending on the Netezza software release.

Constraint Rule Action

You can specify the following actions upon updating or deleting a constraint. Note that because the system does not enforce constraint checking, these rules are merely accepted rather than invoked.

- CASCADE — Updates the value of the referencing column to the new value of the referenced column.
- SET NULL — Sets the referencing column to the new value of the referenced column.
- SET DEFAULT — Sets the referenced column
- RESTRICT — Same as NO ACTION
- NO ACTION — Produces an error if the foreign key is violated.

Constraint Attributes

Constraints can have the following attributes that determine whether the constraint check is immediate or deferred. Note that because the system does not enforce constraint checking, these attributes are merely accepted rather than invoked.

- [NOT] DEFERRABLE — Determines whether the constraint is checked at the end of the transaction.
- INITIALLY DEFERRED — Checks the constraint only at the end of the transaction.
- INITIALLY IMMEDIATE — Checks the constraint after each statement.

Related Commands

Refer to CREATE TABLE AS and ALTER TABLE.

Usage

The following provides sample usage:

- To create a table, enter:

```
system(admin)=>
CREATE TABLE name (
    code        CHARACTER(5) CONSTRAINT firstkey PRIMARY KEY,
    title       CHARACTER VARYING(40) NOT NULL,
    did         DECIMAL(3) NOT NULL,
    date_prod   DATE,
    kind        CHAR(10),
    len         INTERVAL HOUR TO MINUTE
);
```

- To define a primary key table constraint for the table films, you can define primary key table constraints on one or more columns of the table:

```
system(admin)=>
CREATE TABLE name (
    code        CHAR(5),
    title       VARCHAR(40),
    did         DECIMAL(3),
    date_prod   DATE,
    kind        VARCHAR(10),
    len         INTERVAL HOUR TO MINUTE,
    CONSTRAINT code_title PRIMARY KEY(code,title)
);
```

- To define a primary key constraint for the table distributors, enter:

```
system(admin)=> CREATE TABLE distributors (
    did     DECIMAL(3),
    name    CHAR VARYING(40),
    PRIMARY KEY(did)
);
system(admin)=> CREATE TABLE distributors (
    did     DECIMAL(3) PRIMARY KEY,
    name    VARCHAR(40)
);
```

Note: The two examples are equivalent — the first uses the table constraint syntax, the second uses the column constraint notation.

- To define two not null column constraints on the table distributors, one of which is explicitly given a name, enter:

```
system(admin)=> CREATE TABLE distributors (
    did     DECIMAL(3) CONSTRAINT no_null NOT NULL,
    name    VARCHAR(40) NOT NULL
);
```

CREATE TABLE AS

Use the CREATE TABLE AS command to create a new table from the results of a query. Note that this command is often referred to as CTAS in the documentation.

Synopsis

Syntax for using the CREATE TABLE AS command:

```
CREATE [ TEMPORARY | TEMP ] TABLE <table> [ (<column_name> [, ...] ) ]
AS <select_clause> [ DISTRIBUTE ON ( <column> [, ...] ) ]
```

Inputs

The CREATE TABLE AS command takes the following inputs:

Table B-44: CREATE TABLE AS Input

Input	Description
AS	Specifies the select statement.
column_name	Specifies the name of a column in the new table. You can specify multiple column names using a comma-delimited list of column names. If you do not provide column names, they are taken from the output column names of the query.
DISTRIBUTE ON	Specifies the distribution column. For more information, see “Handling Distribution Keys” on page B-62.
select_clause	Refer to the select command for permitted syntax.
table	Specifies the name of the new table to create. This table must not already exist.
TEMPORARY	Specifies a temporary table.

Outputs

Refer to the CREATE TABLE and SELECT commands for a listing of possible output messages.

Description

The CREATE TABLE AS command has the following characteristics:

Privileges Required

The user who issues the CREATE TABLE AS command owns the resultant table. You must be an administrator, or an administrator must have given you the appropriate object privileges, to use this command.

Common Tasks

Use the CREATE TABLE AS command to create a table and fill it with data from a select command:

- Table columns have the names and data types associated with the output columns of the SELECT command, unless you override the column names by giving an explicit list of new column names.
- The CREATE TABLE AS command creates a new table and evaluates the query just once to fill the new table initially. The new table does not track subsequent changes to the source tables of the query. (In contrast, whenever you query a view it re-evaluates the underlying SELECT commands.)

Suppress Auto-Statistics on Small CTAS

During CTAS operations, the Netezza typically runs GENERATE STATISTICS following the CTAS operation to collect statistics on the created table. However, for shorter table queries, the GENERATE STATISTICS process can sometimes take more time to run than the CTAS operation itself.

When a CTAS operation is submitted, table creation and insert operations take place. During the insert operation, the Netezza computes the Min values, Max values, and zone maps for all of the columns. If the insert operation yields a number of rows that is less than the configured threshold (ctas_auto_stats_min_rows), Netezza skips the generate statistics operation.

There are two postgresql.conf file settings that control this feature:

- enable_small_ctas_autostats enables or disables the feature to suppress auto-statistics on small tables. The setting is enabled by default.
- ctas_autostats_min_rows specifies the threshold number for a small table. The Netezza will not calculate statistics for any tables that are under this threshold. The default value is 10000.

Handling Distribution Keys

If you do not define explicit distribution keys, a CTAS table inherits its distribution from the parent table. In general, the distribution of the target table is defined by the final node in the plan. If the final node has a valid distribution, the system assigns that distribution to the CTAS's target. Only if the final plan node has no distribution (like a node at the host) does the system default to the table's first column.

The default distribution key is the first column (hash distribution) where there is no discernible distribution key or the source stream into the CTAS table has round-robin distribution.

Table t_one inherits its distribution keys from f_one. (It does not default to first column.)

Example 1

```
CREATE TABLE t_one AS SELECT ... FROM tbl ...;
```

Table t_two inherits its distribution keys from the join table of (tbl_one+tbl_two), which would be their join keys.

Example 2

```
CREATE TABLE t_two AS SELECT ... FROM tbl_one,tbl_two ... WHERE
tbl_one.b1 = tbl_two.b2 ...
```

Table t_three inherits its distribution keys from the grouping node, which would be (b1,b2,b3).

Example 3

```
CREATE TABLE t_three AS SELECT ... FROM tbl_one, tbl_two, tbl_three...
WHERE .... GROUP BY b1,b2,b3;
```

System Default for Table Distributions

The postgresql.conf setting enable_random_table_distribute controls the default distribution behavior when tables are created. The default value 0 (disabled) specifies the following behavior, which is the default behavior for tables created in prior releases:

- For a CREATE TABLE operation:
  - If DISTRIBUTE ON is specified, use the specified distribution mechanism.
  - If DISTRIBUTE ON is not specified, use the first column as the default distribution key.
- For a CREATE TABLE AS (CTAS) operation:
  - If DISTRIBUTE ON is specified, use the specified distribution mechanism.
  - If DISTRIBUTE ON is not specified, inherit the distribution keys from the plan. If the planner cannot determine a distribution from the plan, use the first column as the default key.

If you specify enable_random_table_distribute=1 (enabled), the system behavior changes to the following:

- For a CREATE TABLE operation:
  - If DISTRIBUTE ON is specified, use the specified distribution mechanism.
  - If DISTRIBUTE ON is not specified, use random as the distribution method.
- For a CREATE TABLE AS (CTAS) operation:
  - If DISTRIBUTE ON is specified, use the specified distribution mechanism.
  - If DISTRIBUTE ON is not specified, inherit the distribution keys from the plan. If the planner cannot determine a distribution from the plan, use random as the distribution method.

In some cases, the planner may not be able to determine a distribution from the plan. For example:

- If a final join happens on the host, then the distribution of the result of that join is non-deterministic.
- If the distribution column is missing from the <select-list>, then distribution is non-deterministic.
- If the distribution of the final result node is random, then distribution is considered non-deterministic.
- If the final join is a full-outer-join, then also distribution is non-deterministic.

Note: This change obsoletes the NZ_DISABLE_SKEW_DEFENSE environment variable which controlled this behavior in earlier releases. The upgrade to Release 4.6 checks for the presence of the variable, and if it is set, uses its value to set enable_random_table_distribute to its corresponding value.

To change the postgresql.conf variable, do the following:

1. Use a standard editor to open the configuration file, nz/data/postgresql.conf.
2. Find the line containing "the enable_random_table_distribute = 0".
3. Change the variable from 0 to 1 and save the change.
4. Restart the Netezza system for the changes to take effect.

Related Commands

See "CREATE TABLE" on page B-55.

Usage

The following provides sample usage.

- To show an example of the CREATE TABLE AS command, consider the following table, named cows, enter:

  system(admin)=> **CREATE TABLE cow2 AS SELECT * FROM cows;**

```
 cnumber | cname  |   cbreed    | ckind
---------+--------+-------------+-------
       3 | Cindy  | Ayrshire    | milk
       8 | Muffin | Guernsey    | milk
       2 | Martha | Brown Swiss | milk
       7 | Joe    | Angus       | beef
       5 | Gretel | Highland    | beef
       1 | Betsy  | Holstein    | milk
       6 | Bob    | Angus       | beef
       4 | Mindy  | Hereford    | beef
       9 | Milda  | Jersey      | milk
(9 rows)
```

- To use the CREATE TABLE AS command to create a new table from two columns of the table cows, enter:

  system(admin)=> **CREATE TABLE cow2 AS SELECT cname, cbreed FROM cows;**

  The result is the new table, named cows2:

```
system(admin)=> SELECT * FROM cows2;

cname  |   cbreed
--------+-------------
 Cindy  | Ayrshire
 Muffin | Guernsey
 Mindy  | Hereford
 Milda  | Jersey
 Betsy  | Holstein
 Gretel | Highland
 Martha | Brown Swiss
```

```
 Joe     | Angus
 Bob     | Angus
(9 rows)
```

CREATE USER

Use the CREATE USER command to define a new database user account.

Synopsis

Syntax for creating a user:

```
CREATE USER username
[WITH
[PASSWORD {'string' | NULL }]
[SYSID uid]
[ROWSETLIMIT [integer ]
[IN GROUP groupname [, ...] ]
[VALID UNTIL 'date' ]
[SESSIONTIMEOUT [integer ]
[QUERYTIMEOUT [integer ]
[DEFPRIORITY [critical|high|normal|low|none]]
[MAXPRIORITY [critical|high|normal|low|none]]
[IN RESOURCEGROUP resourcegroupname]
]
```

Syntax for other inputs, including advanced security features:

```
| COLLECT HISTORY { ON | OFF | DEFAULT }
| CONCURRENT SESSIONS <limit>
| ALLOW CROSS JOIN [TRUE|FALSE|NULL]
| ACCESS TIME { ALL | DEFAULT | ( <access-time>,... )
<access-time> ::= DAY { ALL | <day>, ... } [ <time-bound> ]
<time-bound> ::= START <time-literal> END <time-literal> ]
[EXPIRE PASSWORD]
[AUTH [LOCAL | DEFAULT]]
```

Inputs

The command takes the following inputs:

Table B-45: CREATE USER Input

Input	Description
username	Specifies the name of the user.

Table B-45: CREATE USER Input

Input	Description
uid	Use the sysid clause to choose the user ID of the user you are creating. If you do not specify uid, the system uses the highest user ID and adds one (with a minimum of 100) as default.
string	Specifies a password for this account. Note the following: • You can set a password when using either LOCAL or LDAP authentication, but the password is used only for LOCAL authentication. • When using LOCAL authentication, the user must have a password to log on. If you change authentication from LDAP to LOCAL, be sure to use the ALTER USER command to specify a password for the user so that the user can access the database. **Note:** You must specify a password when authentication is LOCAL; otherwise, the user cannot connect.
NULL	You can specify WITH PASSWORD NULL to explicitly create a user with a null password. This is the same as not including the WITH PASSWORD option. **Note:** A user who has privileges to access the "_t_user" table can find all users with null passwords by executing the following: `SELECT * FROM _t_user WHERE passwd is null`
date	Specifies the date (and, optionally, the time) when this user's account expires.
ROWSETLIMIT	Specifies the number of rows a query can return. You can specify from 1 to 2,147,483,647 rows or zero for unlimited. The rowset limit specifies the maximum number of rows any query run by this user (or group) can return.
SESSIONTIMEOUT	Specifies the amount of time a session can be idle before the system terminates it. You can specify from 1 to 35,791,394 minutes or zero for unlimited.
QUERYTIMEOUT	Specifies the amount of time a query can run before the system sends the administrator a message. You can specify from 1 to 35,791,394 minutes or zero for unlimited. Note that to receive a message, you must enable the RunAwayQuery event rule. For more information, see the *IBM Netezza System Administrator's Guide*.
DEFPRIORITY	Specifies the default priority for the user. The valid priorities are critical, high, normal, low.
MAXPRIORITY	Specifies the maximum priority for the user.
IN GROUP	Specifies the group into which to insert the user as a new member.
IN RESOURCEGROUP	Specifies the resource group into which to add a user.

Table B-45: CREATE USER Input

Input	Description
group	Specifies the name of the group.
COLLECT HISTORY [ON | OFF | DEFAULT	Determines whether the sessions for this user will collect history. • ON collects history for the user when connecting to a database that also has COLLECT HISTORY ON. • OFF specifies that history is not collected for the user. • DEFAULT causes the system to review the collection settings for the groups in which the user is a member. This is the default value. – If any group has COLLECT HISTORY ON, history is collected when the user connects to a database that has COLLECT HISTORY ON. – If no group has COLLECT HISTORY ON but a group has COLLECT HISTORY OFF, then history is not collected for the user. – If all groups have the DEFAULT history collection setting, history is collected for the user when connected to a database that has COLLECT HISTORY ON.
CONCURRENT SESSIONS <limit>	Sets the maximum number of concurrent sessions this group can have. A value of 0 means no limit to the number of concurrent sessions, unless a limit is imposed by a group. In that case, the minimum limit of concurrent sessions across all such groups is used.
ALLOW CROSS JOIN [TRUE | FALSE | NULL]	Sets user or group permission to allow explicit cross joins. If NULL is defined for a user, the system checks against the group permission, and takes the lowest non-null value, where FALSE is lower than TRUE. **Note:** This setting involves a system-wide change, so notify all affected users before making this change.
ACCESS TIME ALL	Indicates that this group may start sessions on the Netezza system at any time on any day
ACCESS TIME DEFAULT	Indicates that access time restrictions are taken from the groups. If no groups have access time restrictions, then the user may start sessions at any time on any day. The access time restriction is evaluated for every group that has one. If any group restricts access, the user may not create a session. That is, the most restrictive access policy is applied.
access-time	Specifies one access time sub-clause; several may be specified. An access time sub-clause defines one or more days by the standard SQL day number (1 = Sunday, 7 = Saturday). The keyword ALL can be use to specify all days of the week; it is equivalent to 1,2,3,4,5,6,7. An access time sub-clause optionally contains one time bound. If no time bound is specified, then the group may create a session at any time on the specified day.

Table B-45: CREATE USER Input

Input	Description
EXPIRE PASSWORD	Creates a user with an expired/invalidated password. The first time the user logs in, they must change their password.
AUTH [LOCAL | DEFAULT]	Overrides the authentication for the user to LOCAL if specified. DEFAULT is the connection setting or whatever authentication is set.

Outputs

The CREATE USER command has the following output

Table B-46: CREATE USER Output

Output	Description
CREATE USER	Message returned if the command completes successfully.

Description

The CREATE USER command has the following characteristics:

Privileges Required

You must be an administrator, or an administrator must have given you the appropriate object privileges, to use this command.

Common Tasks

Use the CREATE USER command to add a new user.

Related Commands

Use the following commands:

- Use "ALTER USER" on page B-24 to change a user's password and privileges.
- Use "DROP USER" on page B-82 to remove a user.
- Use "ALTER GROUP" on page B-6 to add or remove the user from other groups.
- Use "SET AUTHENTICATION" on page B-112 to set authentication to and from LDAP and LOCAL.
- Use "SHOW AUTHENTICATION" on page B-124 to display how the Netezza system is currently configured for authentication.
- Use"SET CONNECTION" on page B-116 to specify which Netezza system connections should use an SSL connection.
- Use "SHOW CONNECTION" on page B-126 to display Netezza connections that use SSL.
- Use "DROP CONNECTION" on page B-72 to drop a Netezza connection.

Usage

The following provides sample usage.

- To create a user with a password, enter:

```
system(admin)=> CREATE USER davide WITH PASSWORD 'jw8s0F4';
```

- To create a user with a password whose user account expires on January 1, 2003, enter:

```
system(admin)=> CREATE USER miriam WITH PASSWORD 'jw8s0F4' VALID
UNTIL 'Jan 1 2003';
```

CREATE VIEW

Use the CREATE VIEW command to create a view. Use CREATE OR REPLACE VIEW to transfer the permissions (ACL data) from one view to another.

Synopsis

Syntax for creating a view:

```
CREATE VIEW <viewname> AS SELECT <query>
CREATE OR REPLACE VIEW <viewname> AS SELECT <query>
```

Inputs

The CREATE VIEW command takes the following inputs:

Table B-47: CREATE VIEW Input

Input	Description
CREATE	Creates a view.
CREATE OR REPLACE	Replaces an existing view.
query	Specifies the SQL query that provides the columns and rows of the view. Refer to the select command for information about valid arguments.
view	Specifies the name of the view to create.

Outputs

The CREATE VIEW command has the following output:

Table B-48: CREATE VIEW Output

Output	Description
CREATE VIEW	Message returned if the command successfully creates the view.
ERROR: Relation 'view' already exists	Message returned if the view you specified already exists in the database.

Description

The CREATE VIEW command has the following characteristics:

Privileges Required

You must be an administrator, or an administrator must have given you the appropriate object privileges, to use this command.

Common Tasks

Use the CREATE VIEW command to define a view of a table. The view is not physically materialized. Instead, a query rewrite retrieve rule is automatically generated to support retrieve operations on views.

Use the CREATE OR REPLACE VIEW to redefine a view and retain the permissions from the original view.

Note: Views are read-only. The system does not allow an insert, update, or delete on a view.

Related Commands

See "DROP VIEW" on page B-83 to drop views.

Usage

The following provides sample usage.

- To create a view consisting of all comedy films, enter:

```
system(admin)=> CREATE VIEW kinds AS
    SELECT *
    FROM films
    WHERE kind = 'Comedy';
```

- To display the view, enter:

```
system(admin)=> SELECT * FROM kinds;
code  | title                     | did | date_prod  | kind    | len
------+---------------------------+-----+------------+---------+--
C_701 | There's a Girl in my Soup | 107 | 1970-06-11 | Comedy |
01:36:00
UA502 | Bananas                   | 105 | 1971-07-13 | Comedy |
01:22:00
(2 rows)
```

- To re-create the view while retaining the permissions of the original view, enter:

```
system(admin)=> CREATE OR REPLACE VIEW kinds AS
    SELECT *
    FROM films
    WHERE kind = 'Action';
```

DELETE

Use the DELETE command to remove rows from a table.

Synopsis

Syntax for deleting rows from a table:

```
DELETE FROM <table> [ WHERE <condition> ]
```

Inputs

The DELETE command takes the following inputs:

Table B-49: DELETE Input

Input	Description
table	Specifies the name of an existing table.
condition	Specifies a SQL selection query that returns the rows to be deleted. Refer to the SELECT command for further description of the where clause.
WHERE	Specifies the condition.

Outputs

The DELETE command has the following output:

Table B-50: DELETE Output

Output	Description
DELETE COUNT	Message returned if items are successfully deleted. The count is the number of rows deleted. If the count is zero, no rows were deleted.

Description

The DELETE command has the following characteristics:

Privileges Required

You must be an administrator, or an administrator must have given you the appropriate object privileges, to use this command.

Common Tasks

Use the DELETE command to remove rows that satisfy the WHERE clause condition for a particular table.

If you use the DELETE command without the WHERE clause, the system deletes all rows in the table. The result is a valid, empty table.

You can use a table alias in update and delete statements. For example:

```
UPDATE tablename t1 set t1.c2='new value' where t1.c1=1;
DELETE from tablename t1 where t1.c1=2;
```

Related Commands

The TRUNCATE command provides a faster mechanism to remove all rows from a table.

Usage

The following provides sample usage.

- To remove all but musicals from the table films, enter:

```
system(admin)=> DELETE FROM films WHERE kind <> 'Musical';
```

- To display the remaining rows, enter:

```
system(admin)=> SELECT * FROM films;
code| title| did| date_prod| kind| len
----+------|-----|----------+-----+------
UA501| West Side Story| 105| 1961-01-03| Musical| 02:32:00
TC901| The King and I| 109| 1956-08-11| Musical| 02:13:00
WD101| Bed Knobs and Broomsticks| 111| | Musical | 01:57:00
(3 rows)
-----+---------------------------|------|----------+-----------
(3 rows)
```

- To clear the table films, enter:

```
system(admin)=> DELETE FROM films;
```

- To display the valid, but blank, table, enter:

```
system(admin)=> SELECT * FROM films;
code| title| did| date_prod| kind| len
----+------|-----|----------+-----+------
(0 rows)
```

DROP CONNECTION

Use the DROP CONNECTION command to drop a Netezza connection record from the system table. For more information about configuring Netezza host access for clients, refer to the *IBM Netezza System Administrator's Guide*.

Synopsis

Syntax for dropping a connection:

```
DROP CONNECTION [connection_number]
```

Inputs

The DROP CONNECTION command takes the following inputs:

Table B-51: DROP CONNECTION Input

Input	Description
connection_ number	Specifies the connection number of the connection you wish to drop. To obtain the connection number, use the SHOW CONNECTION command and review the CONNID field of the output for the associated number.

Outputs

The DROP CONNECTION command has the following output:

Table B-52: DROP CONNECTION Output

Output	Description
ERROR: permission denied	You do not have the proper permission to run this command.
DROP CONNECTION	Message returned if the command is successful.

Description

The DROP CONNECTION command has the following characteristics:

Privileges Required

To use this command, you must be an administrator, or an administrator must have granted you the Manage System privilege.

Common Tasks

Use this command to drop a Netezza connection.

Related Commands

See “SET CONNECTION” on page B-116.

See “CREATE USER” on page B-65.

See “ALTER USER” on page B-24.

See “SHOW CONNECTION” on page B-126.

DROP DATABASE

Use the DROP DATABASE command to remove an existing database.

Synopsis

Syntax dropping a database:

```
DROP DATABASE <name>
```

Inputs

The DROP DATABASE command takes the following inputs:

Table B-53: DROP DATABASE Input

Input	Description
name	Specifies the name of an existing database to remove.

Outputs

The DROP DATABASE command has the following output:

Table B-54: DROP DATABASE Output

Output	Description
DROP DATABASE	Message returned if the command is successful.
DROP DATABASE: cannot be executed on the currently open database	You cannot be connected to the database that you are about to remove. Connect to any other database, and run the command again.
DROP DATABASE: may not be called in a transaction block	You must finish the transaction in progress before you can call the command.
ERROR: Can't delete database - *num* object(s) depend on objects in it	The database contains objects such as user-defined functions or aggregates, or stored procedures, which are dependencies for *num* objects defined in other databases. This message follows up to 5 notice messages of the named objects. You must resolve all the dependencies by either dropping or altering the objects before you can drop the database.

Description

The DROP DATABASE command has the following characteristics:

Privileges Required

You must be the database owner, an administrator, or an administrator must have given you the appropriate object privileges, to use this command.

Common Tasks

Use the DROP DATABASE command to remove catalog entries for an existing database, and to delete the directory containing the data.

Note: The DROP DATABASE command cannot be undone. You cannot execute this command while you are connected to the target database.

Related Commands

See “CREATE DATABASE” on page B-38 for information on how to create a database.

Usage

The following provides sample usage:

- To delete the database emp, enter:
  system(admin)=> **DROP DATABASE emp;**

DROP GROUP

Use the DROP GROUP command to remove a group.

Synopsis

Syntax for dropping a group:

```
DROP GROUP <name>
```

Inputs

The DROP GROUP command takes the following inputs:

Table B-55: DROP GROUP Input

Input	Description
name	Specifies the name of an existing group.

Outputs

The DROP GROUP command has the following output

Table B-56: DROP GROUP Output

Output	Description
DROP GROUP	Message returned if the group is successfully deleted.

Description

The DROP GROUP command has the following characteristics:

Privileges Required

You must be an administrator, or an administrator must have given you the appropriate object privileges (including the Drop privilege), to use this command.

Common Tasks

Use the DROP GROUP command to remove the specified group from the database. The drop group command does not remove the users in the group.

Related Commands

Use "CREATE GROUP" on page B-40 to add new groups.

Use "ALTER GROUP" on page B-6 to change a group's membership.

Usage

The following provides sample usage.

▼ To drop the group staff, enter:

```
system(admin)=> DROP GROUP staff;
```

DROP HISTORY CONFIGURATION

Use the DROP HISTORY CONFIGURATION command to drop a configuration for query history logging.

Synopsis

Syntax for dropping a configuration:

```
DROP HISTORY CONFIGURATION <config-name>
```

Inputs

The DROP HISTORY CONFIGURATION command has the following inputs:

Table B-57: DROP HISTORY CONFIGURATION Inputs

Input	Description
<config-name>	Specifies the name of the configuration to drop. The configuration must exist on the Netezza system. You cannot drop the current/active configuration.

Outputs

The DROP HISTORY CONFIGURATION command has the following outputs:

Table B-58: DROP HISTORY CONFIGURATION Output

Output	Description
DROP HISTORY CONFIGURATION	Message returned if the command is successful.
ERROR: permission denied	You must have Manage Security permission to drop a configuration.
ERROR: <config-name> not found.	The specified configuration name could not be found.

Description

This command will not allow you to drop the current query history configuration. If you want to drop the current configuration, you must SET HISTORY CONFIGURATION to another configuration, restart the Netezza software, and then DROP HISTORY CONFIGURATION.

After you drop a history configuration, any existing but unloaded history files in the staging or loading area for that configuration will not be loaded. When the loading process attempts to load those files, if it cannot find the associated history configuration, it moves the batch files to the error directory ($NZ_DATA/hist/error).

Privileges Required

You must have Manage Security and Drop permissions to drop history configurations.

Related Commands

See the "CREATE HISTORY CONFIGURATION" on page B-44 to create a new configuration.

See the "ALTER HISTORY CONFIGURATION" on page B-10 to modify configurations.

See the "SET HISTORY CONFIGURATION" on page B-117 to specify a configuration for query history logging.

See the "SHOW HISTORY CONFIGURATION" on page B-127 to display information about a configuration.

Usage

The following sample command drops the basic_hist history configuration:

```
SYSTEM(ADMIN)=> DROP HISTORY CONFIGURATION basic_hist;
```

DROP SEQUENCE

Use the DROP SEQUENCE command to drop a sequence.

Synopsis

Syntax for dropping a sequence:

```
DROP SEQUENCE <sequence name>
```

Inputs

The DROP SEQUENCE command takes the following inputs:

Table B-59: DROP SEQUENCE Inputs

Input	Description
sequence name	Specifies the name of the sequence to drop.

Outputs

The DROP SEQUENCE command produces the following output:

Table B-60: ALTER SEQUENCE Output

Output	Description
drop sequence	The message that the system returns if the command is successful.

Description

The DROP SEQUENCE command has the following characteristics:

Privileges Required

The privileges to drop sequences are as follows:

- The admin user has all privileges on all user sequences. There is no need to grant any privileges to the admin user.
- The owner of the database has all privileges on all user sequences in that database. There is no need to grant any privileges to the owner.
- All others must have the Drop object privilege for a specific sequence or the Sequence object class.

Common Tasks

Use the DROP SEQUENCE command to remove a sequence.

Related Commands

See "CREATE SEQUENCE" on page B-51 and "ALTER SEQUENCE" on page B-15 for related sequence commands.

Usage

The following provides sample usage:

- To drop sequence1, enter:

```
system(admin)=> DROP SEQUENCE sequence1;
```

DROP SESSION

Use the DROP SESSION command to abort and remove a session from the Netezza.

Synopsis

Syntax for dropping a session:

```
DROP SESSION <session-id>
```

Inputs

The DROP SESSION command takes the following inputs:

Table B-61: DROP SESSION Inputs

Input	Description
session_id	A number identifying an active session. Unlike the other session commands, the <session-id> is required for this command.

Outputs

The DROP SESSION command produces the following output:

Table B-62: DROP SESSION Output

Output	Description
DROP SESSION	The message that the system returns if the command is successful.
ERROR: id '<session-id>' does not correspond to an existing session.	The specified session ID does not exist.
ERROR: system session id '<session-id>' cannot be aborted	The specified session ID refers to a system session. Users cannot rollback transactions of system sessions.
ERROR: access denied. You must have ABORT privileges to perform this action	You do not have permission to rollback the transaction of the session specified by <session-id>.
ERROR: session abort failed for session <session-id>; reason is '<reason>'	The attempt to rollback the transaction in session <session-id> failed; the reason for the failure is provided in the <reason> string.

Description

The DROP SESSION command aborts a session and removes it from the Netezza.

Privileges Required

You need no special privileges to drop your own session.

Each session is owned by a user. You must be granted ABORT privileges on the user in order to drop the user's sessions.

Common Tasks

Use the DROP SESSION command to stop a session.

Related Commands

See "ALTER SESSION" on page B-17 to change a session's priority or to abort a transaction in a session.

See "SHOW SESSION" on page B-130 to display session information.

Usage

The following provides sample usage:

▼ To drop session ID 17044:

```
system(admin)=> DROP SESSION 17044;
DROP SESSION
```

DROP SYNONYM

Use the DROP SYNONYM command to drop a synonym.

Synopsis

Syntax for dropping a synonym:

```
DROP SYNONYM <synonym_name>;
```

Inputs

The DROP SYNONYM command takes the following inputs:

Table B-63: DROP SYNONYM Inputs

Input	Description
synonym name	Specifies the name of the synonym to drop.

Outputs

The DROP SYNONYM command produces the following output:

Table B-64: DROP SYNONYM Output

Output	Description
DROP SYNONYM	The message that the system returns if the command is successful.

Description

The DROP SYNONYM command has the following characteristics:

Privileges Required

You can drop your own synonyms. To drop other synonyms, you must be the admin user or have been granted the Drop privilege for synonyms.

Common Tasks

Dropping unneeded synonyms.

Related Commands

See “CREATE SYNONYM” on page B-54 and “ALTER SYNONYM” on page B-19.

Usage

The following provides sample usage:

- To drop synonym pr, enter:

```
system(admin)=> DROP SYNONYM pr;
```

DROP TABLE

Use the DROP TABLE command to remove an existing table from a database.

Synopsis

Syntax dropping a table:

```
DROP TABLE <name> [, ...]
```

Inputs

The DROP TABLE command takes the following inputs:

Table B-65: DROP TABLE Input

Input	Description
name	Specifies the name of an existing table to drop.

Outputs

The DROP TABLE command has the following output:

Table B-66: DROP TABLE Output

Output	Description
DROP TABLE	Message returned if the table is successfully dropped.
ERROR: Relation “name” does not exist	Message returned if the specified table does not exist in the database.

Description

The DROP TABLE command has the following characteristics:

Privileges Required

You must be the table owner, an administrator, or an administrator must have given you the appropriate object privileges (including the Drop privilege), to use this command.

Common Tasks

Use the DROP TABLE command to remove tables from a database.

Related Commands

Refer to the "CREATE TABLE" on page B-55 and "ALTER TABLE" on page B-20 commands for information on how to create or modify tables.

Use the TRUNCATE command to empty a table rather than destroy it.

Usage

The following provides sample usage.

- To drop (delete) two tables, films and distributors, enter:

```
system(admin)=> DROP TABLE films, distributors;
```

DROP USER

Use the DROP USER command to remove a user.

Synopsis

Syntax for dropping a user:

```
DROP USER <name>
```

Inputs

The DROP USER command takes the following inputs:

Table B-67: DROP USER Input

Input	Description
name	Specifies the name of an existing user.

Outputs

The DROP USER command has the following output:

Table B-68: DROP USER Output

Output	Description
DROP USER	Message returned if the user is successfully deleted.
ERROR: Drop user: user "name" does not exist	Message returned if the user name is not found.
DROP USER: User "name" owns objects, cannot be removed	Message returned if the user you want to drop owns any objects. You must first change the ownership of the objects or drop them before you can drop the user.

Description

The DROP USER command has the following characteristics:

Privileges Required

You must be an administrator, or an administrator must have given you the appropriate object privileges (including the Drop privilege), to use this command.

Common Tasks

Use the DROP USER command to remove a specified user from the database.

Note: If the user owns any database objects, the system displays an error and prevents you from dropping the user. You must drop or change the ownership of those objects before you can drop the user.

Related Commands

Use “CREATE USER” on page B-65 to add new users.

Use “ALTER USER” on page B-24 to change a user's properties.

Usage

The following provides sample usage.

- To drop the user account jonathan, enter:

```
SYSTEM(ADMIN)=> DROP USER jonathan;
DROP USER
```

- If you drop a user that owns objects:

```
SYSTEM(ADMIN)=> DROP USER jonathan;
ERROR:  DROP USER: user "jonathan" owns objects, cannot be removed
```

DROP VIEW

Use the DROP VIEW command to remove an existing view from a database.

Synopsis

Syntax dropping a view:

```
DROP VIEW <name> [, ...]
```

Inputs

The DROP VIEW command takes the following inputs:

Table B-69: DROP VIEW Input

Input	Description
name	Specifies the name of an existing view.

Outputs

The DROP VIEW command has the following output:

Table B-70: DROP VIEW Output

Output	Description
DROP VIEW	Message returned if the command is successful.
ERROR: Name: No such view	Message returned if the specified view does not exist in the database.

Description

The DROP VIEW command has the following characteristics:

Privileges Required You must be the owner of the view, an administrator, or an administrator must have given you the appropriate object privileges (including the Drop privilege), to use this command.

Common Tasks Use the DROP VIEW command to drop an existing view from a database.

Related Commands See "CREATE VIEW" on page B-69 for information on how to create views.

Usage

The following provides sample usage.

- To remove the view kinds, enter:

```
system(admin)=> DROP VIEW kinds;
```

EXPLAIN

Use the EXPLAIN command to show a statement execution plan.

Synopsis

Syntax using the EXPLAIN command:

```
EXPLAIN [ VERBOSE ] <query>
EXPLAIN DISTRIBUTION <query>
EXPLAIN [ PLANTEXT ] <query>
EXPLAIN [ PLANGRAPH ] <query>
```

Inputs

The EXPLAIN command takes the following inputs:

Table B-71: EXPLAIN Input

Input	Description
VERBOSE	Specifies verbose to show a detailed query plan.

Table B-71: EXPLAIN Input

Input	Description
query	Specifies the query.
DISTRIBUTION	Requests explanation of the distribution plan. See “Common Tasks” on page B-84.
PLANTEXT	Specifies the text plan.
PLANGRAPH	Specifies the HTML plan.

Outputs

The EXPLAIN command has the following output:

Table B-72: EXPLAIN Output

Output	Description
NOTICE: query plan: plan	Message returned with an explicit query plan.
NOTICE: result-set distribution	Message returned from a distribution query.
EXPLAIN	Message returned after the system displays the query plan.

Description

The EXPLAIN command has the following characteristics:

Privileges Required

This command requires no special privileges.

Common Tasks

Use the EXPLAIN command to display the execution plan that the planner uses for a query.

- The execution plan displays how the table(s) referenced by the query will be scanned (for example, by plain sequential scan). You can specify a verbose, HTML, or text version.
- If multiple tables are referenced, the execution plan specifies what join algorithms it will use to bring together the required tuples from each input table.
- If you use the DISTRIBUTION subcommand, the EXPLAIN command prints out the distribution that the planner used on a SELECT statement in a CTAS command that did not specify DISTRIBUTE ON().

A portion of the execution plan estimates the query execution cost, which is the planner's guess at how long it will take to run the query (measured in milliseconds). Two numbers are shown:

- The start-up time before the first tuple can be returned.

- The total time to return all the tuples.

Note: If you limit the number of tuples to return with a limit clause, the planner makes an appropriate interpolation between the endpoint costs to estimate which plan is really the most efficient.

Use the verbose keyword to display the full internal representation of the plan tree, rather than just a summary. The verbose command also sends the plan to the postmaster log file. This option is useful for debugging.

Related Commands

None

Usage

The following provides sample usage.

- To display a query plan for a simple query on a table with a single int4 column and 128 rows, enter:

```
system(admin)=> EXPLAIN SELECT * FROM foo;
    NOTICE:  QUERY PLAN:
Seq Scan on foo  (cost=0.00..2.28 rows=128 width=4)
EXPLAIN
```

- To display a query plan for a join between the emp and grp tables, enter:

```
system(admin)=> EXPLAIN SELECT emp.* FROM emp,grp WHERE
emp.id=grp.grp_id;
NOTICE: QUERY PLAN:
Hash Join (cost=0.01..0.04 rows=10000 width=28)
-> Seq Scan on emp (cost=0.00..0.01 rows=1000 width=28)
-> Hash (cost=0.01..0.01 rows=1000 width=4)
-> Seq Scan on grp (cost=0.00..0.01 rows=1000 width=4)
EXPLAIN
```

- To display a query plan for TPHC Query 3, enter:

```
system(admin)=> EXPLAIN SELECT
    l_orderkey,
    sum(l_extendedprice * (1 - l_discount)) as revenue,
    o_orderdate,
    o_shippriority
FROM
    customer,
    orders,
    lineitem
WHERE
    c_mktsegment = 'BUILDING'
    and c_custkey = o_custkey
```

```
    and l_orderkey = o_orderkey
    and o_orderdate < date '1995-03-15'
    and l_shipdate > date '1995-03-15'
GROUP BY
    l_orderkey,
    o_orderdate,
    o_shippriority
ORDER BY
    revenue desc,
    o_orderdate
LIMIT 10;
```

```
NOTICE: QUERY PLAN:
Limit (cost=315799480130.29..315799480130.29 rows=10 width=28)
 -> Sort (cost=315799480130.29..315799480130.29 rows=360899981912
width=28)
     -> Aggregate (cost=19476.76..112854066.90 rows=360899981912
width=28)
         -> Group (cost=19476.76..72822.55 rows=360899981912 width=28)
             -> Hash Join (cost=19476.76..43432.67 rows=31349208 width=28)
                 -> Seq Scan on lineitem (cost=0.00..17842.17 rows=322475815
width=20)
                 -> Hash (cost=19476.76..19476.76 rows=14582120 width=12)
                     -> Hash Join (cost=1347.66..19476.76 rows=14582120
width=12)
                         -> Seq Scan on orders (cost=0.00..3606.62
rows=72910603 width=16)
                         -> Hash (cost=550.60..550.60 rows=3000000 width=4)
                             -> Seq Scan on customer (cost=0.00..550.60
rows=3000000 width=4)
EXPLAIN
```

- To display the distribution from a SELECT statement in a CTAS command, enter:

```
dev(admin)=> EXPLAIN DISTRIBUTION SELECT COUNT(*),grp FROM emp
GROUP BY grp;
NOTICE:  Result-set distribution (for "CREATE TABLE AS")
Distributed on hash: "grp"
EXPLAIN
```

GENERATE EXPRESS STATISTICS

Use the GENERATE EXPRESS STATISTICS command to generate information on a database or table.

Note: As of Netezza Release 4.6, this command has been deprecated. The GENERATE STATISTICS and GENERATE EXPRESS STATISTICS commands now perform the same

steps; refer to "GENERATE STATISTICS" on page B-89 for the complete command description. Note that in environments where the former command behavior is required, Netezza Support can restore the previous behavior.

Synopsis

Syntax for generating express statistics:

```
GENERATE EXPRESS STATISTICS ON <tablename> [(column_name [, ... ])];
```

Inputs

The GENERATE EXPRESS STATISTICS command takes the following inputs:

Table B-73: GENERATE EXPRESS STATISTICS Input

Input	Description
column_name	Specifies the column or columns.
table	Specifies the name of a table to analyze. The default includes all tables in the current database.

Outputs

The GENERATE EXPRESS STATISTICS command has the following output:

Table B-74: GENERATE EXPRESS STATISTICS Output

Output	Description
GENERATE EXPRESS STATISTICS	Message returned when the command has been completed.

Description

The GENERATE EXPRESS STATISTICS command has the following characteristics:

Privileges Required

You must be an administrator, or an administrator must have given you the GenStats privilege, to use this command.

Common Tasks

The GENERATE EXPRESS STATISTICS command collects statistics about the min/max, null, and estimated dispersion values. The dispersion statistics are not as accurate as the statistics that the system maintains when you run the GENERATE STATISTICS command. They are, however, close approximations.

The GENERATE EXPRESS STATISTICS command produces statistics for only the table or columns you specify.

The system processes full statistics ten columns at a time, whereas it processes express statistics 30 columns at a time.

To estimate how long generating statistics will take divide the number of columns by 10 or 30, which tells you the raw number of passes (full table scans) that the system will make against the data table.

Because full stats attempts to calculate the number of unique values in each column, it is affected by the cardinality in a column and the data types involved.

Because express stats uses a hash to estimate the number of unique values in each column, it is less affected by the cardinality of a column.

Related Commands

Refer to "GENERATE STATISTICS" on page B-89.

Usage

The following provides sample usage.

- To generate statistics on the table cows, enter:

```
system(admin)=> GENERATE EXPRESS STATISTICS ON cows;
GENERATE EXPRESS STATISTICS
```

GENERATE STATISTICS

Use the GENERATE STATISTICS command to generate information on a database, table, or individual column.

Note: As of Release 4.6, the GENERATE STATISTICS and GENERATE EXPRESS STATISTICS commands perform the same tasks. If you use the GENERATE EXPRESS STATISTICS command, it follows this same usage and behavior.

Synopsis

Syntax for generating statistics:

```
GENERATE STATISTICS [ ON <table> [( <column_name> [, ... ])] ];
```

Inputs

The GENERATE STATISTICS command takes the following inputs:

Table B-75: GENERATE STATISTICS Input

Input	Description
column_name	Specifies the name of a column to analyze. The default includes all columns.
table	Specifies the name of a table to analyze. If you do not specify a table, the command will generate statistics for all the tables in the current database.

Outputs

The GENERATE STATISTICS command has the following output:

Table B-76: GENERATE STATISTICS Output

Output	Description
GENERATE STATISTICS	Message returned when the command has been completed.

Description

The GENERATE STATISTICS command has the following characteristics:

Privileges Required

You must have the GenStats privilege to use this command. By default, the admin user can run this command for any database or table. The owner of a database can also run the command for the database he owns or a specific table in that database. Other users can run the command on any table that they own or to which they have List and GenStats privilege.

An admin user can also grant a non-admin user privilege to run generate statistics for all of the tables in a database. The admin user must grant the non-admin user List privilege on the table object in the system database and then GenStats privilege from the target database.

Common Tasks

The GENERATE STATISTICS command collects statistics about each column's proportion of duplicate values, and the maximum and minimum values. The optimizer uses this information to determine the most efficient way to execute a query.

You should run the GENERATE STATISTICS command when you initially load the table and any time the table's data has changed.

Because Netezza SQL uses the statistics collected during a generate statistics process to efficiently perform calculations involving numeric data types, if the statistics are not available, or if they are out of date (by as little as a change in a single row of the table), then the numeric calculations from that table are less efficient.

With no parameter, the GENERATE STATISTICS command processes every table in the current database. With a parameter, the GENERATE STATISTICS command produces statistics for only the table you specify.

After adding or deleting a large number of records, run the GENERATE STATISTICS command for the affected table. Doing so allows the query optimizer to make better choices in planning user queries.

Note: You cannot execute the GENERATE STATISTICS command inside a transaction block (begin/commit pair).

Related Commands

None

Usage

The following provides sample usage.

- To generate statistics on the table cows, column cnumber, enter:

```
system(admin)=> GENERATE STATISTICS ON cows (cnumber);
GENERATE STATISTICS
```

- To generate statistics on all the tables in database mydb, enter:

```
mydb(admin)=> GENERATE STATISTICS;
GENERATE STATISTICS
```

GRANT

Use the GRANT command to grant privileges to a user, a group, or all users.

Synopsis

Syntax for granting an object privilege:

```
GRANT <object_privilege> [, ...] ON <object> [, ...]
    TO { PUBLIC | GROUP <group> | <username> } [ WITH GRANT OPTION ]
```

Syntax for granting an administration privilege:

```
GRANT <admin_privilege> [, ...]
    TO { PUBLIC | GROUP <group> | <username> } [ WITH GRANT OPTION ]
```

Inputs

The GRANT command takes the following inputs:

Table B-77: GRANT Input

Input	Description
object	Specifies the target of the privilege. You can further define an object as one or more of the following object class types or one or more named objects of these types: {DATABASE | GROUP | USER | TABLE | VIEW | EXTERNAL TABLE | SEQUENCE | SYNONYM | SYSTEM TABLE | SYSTEM VIEW | MANAGEMENT TABLE | MANAGEMENT VIEW | FUNCTION | AGGREGATE | PROCEDURE } **Note:** TABLE represents user tables, not all tables (user, system, and management). To grant privileges for system tables, specify the SYSTEM TABLE object.
object_privilege	Specifies any of the following object privileges: ALL, ABORT, ALTER, DELETE, DROP, GENSTATS, INSERT, LIST, SELECT, TRUNCATE, UPDATE, EXECUTE

Table B-77: GRANT Input

Input	Description
admin_privilege	Specifies any of the following administration privileges: ALL ADMIN, BACKUP, [CREATE] <dbobject>, [MANAGE] <manageobject>, RECLAIM, RESTORE where <dbobject> can be: {DATABASE | GROUP | USER | TABLE | TEMP TABLE | EXTERNAL TABLE | VIEW | MATERIALIZED VIEW | SEQUENCE | SYNONYM | FUNCTION | AGGREGATE | PROCEDURE } where <manageobject> can be: {SYSTEM | HARDWARE | SECURITY }
PUBLIC	Specifies that the privileges are to be granted to all users, including users that may be created later. The public group may be thought of as an implicitly defined group that always includes all users. **Note:** Each user has a sum of privileges: • Granted directly to the user. • Granted to any group the user is presently a member of. • Granted to public.
GROUP	Specifies the group.
WITH GRANT	Allows the user to grant the privilege to other users. Note that a non-administrator user must have LIST privileges on the user when granting to that user.

Outputs

The GRANT command has the following output

Table B-78: GRANT Output

Output	Description
GRANT	Message returned when the command has been completed.

Description

The GRANT command has the following features:

Privileges Required

Administrators can access all objects regardless of object privilege settings.

Users other than the creator of an object do not have any access privileges to the object unless the creator grants permissions. There is no need to grant privileges to the creator of an object, as the creator automatically holds all privileges. The creator could, however, choose to revoke some of his own privileges for safety. Note that the ability to grant and revoke privileges is inherent in the creator and cannot be revoked. The right to drop the

object is likewise inherent in the creator, and cannot be granted or revoked. If you grant permission to other users to manage objects, make sure that they also have the LIST permission to view those objects.

Common Tasks

Use the GRANT command to give specific object or administrator permissions to one or more users or groups of users. The system adds the permissions you grant to whatever permissions the user or group already has.

Note: To grant privileges to only a few columns, you must create a view having the desired columns and then grant privileges to that view. You can use the \dp command to obtain information about privileges on existing objects.

Related Commands

Use "REVOKE" on page B-98 to revoke access privileges.

Usage

The following provides sample usage.

- To grant the insert privilege to all users on the user table films, enter:

```
system(admin)=> GRANT INSERT ON films TO PUBLIC;
```

The table

GROOM TABLE

Use the GROOM TABLE command to reclaim disk space for deleted or outdated rows, and reorganize tables based on the clustered base table organizing keys, or to migrate data for tables that have multiple stored versions. This command replaces the **nzreclaim** command. For more information, see "Grooming Tables" in the *IBM Netezza System Administrator's Guide*.

The GROOM TABLE command processes and reorganizes the table records in each data slice in a series of "steps." Users can perform tasks such as SELECT, UPDATE, DELETE, and INSERT operations while the online data grooming is taking place. The SELECT operations run in parallel with the groom operations; the INSERT, UPDATE, and DELETE operations run serially between the groom steps. For CBTs, the groom steps are somewhat longer than for non-CBT tables, so INSERT, UPDATE, and DELETE operations may pend for a longer time until the current step completes.

Note: When you specify organizing keys for an existing table to make it a CBT, the new organization could impact the compression size of the table. The new organization could create sequences of records that improve the overall compression benefit, or it could create sequences that do not compress as well. Following a groom operation, your table size could change somewhat from its size using the previous organization.

Synopsis

Syntax for granting an object privilege:

```
GROOM TABLE <name> <mode-choice> <reclaim-choice>
```

Where

```
<mode-choice>:= RECORDS READY | RECORDS ALL | PAGES ALL | PAGES START |
VERSIONS
<reclaim-choice>:= RECLAIM BACKUPSET { NONE | DEFAULT | <backupsetid>}
```

Inputs

The GROOM TABLE command takes the following inputs:

Table B-79: GROOM TABLE Input

Input	Description
RECORDS READY	Specifies to reclaim and reorganize records in the table that have not been groomed, as well as those previously groomed but marked for regrooming. This is the default for clustered base tables (CBT).
RECORDS ALL	Specifies to reclaim and reorganize all records in a table. This is the default for a non-CBT.
PAGES ALL	Identifies and marks as 'Empty' data pages in the table with no visible record, to free up disk extents.
PAGES START	Identifies and marks as 'Empty' leading data pages in the table with no visible record, stopping when it finds a non-empty data page.
VERSIONS	Specifies to migrate records from previous table versions. Dropped columns will not appear, and added columns will show default values.
RECLAIM BACKUPSET	Can be set to the following: • NONE — No backup-reclaim synchronization, which means the next backup may not be able to incrementally backup the table, and require a full backup. • DEFAULT — This is the default if no reclaim-choice is specified, and uses the default backup set, if one exists, for the given database for backup-reclaim synchronization. • <backupsetid> — Use the most recent backup in the specified backup set for backup-reclaim synchronization.

Outputs

The GROOM TABLE command has the following output

Table B-80: GROOM TABLE Output

Output	Description
NOTICE: GROOM processed <#> pages; released <#> pages; purged <#> records. Table size grew/shrunk/unchanged from <#> extents to <#> extents.	Message returned when the command has been completed.

Description

The GROOM TABLE command has the following features:

Privileges Required

While **nzreclaim** required an administrator privilege, GROOM TABLE requires a GROOM object privilege.

Common Tasks

Use the GROOM TABLE command to remove outdated and deleted records from tables while allowing access to all tables in the system.

Note: You cannot execute the GROOM TABLE command inside a transaction block (begin/commit pair) or with a stored procedure.

Note: Grooming a table is done as a user, not an administrator, so to run GROOM TABLE requires that you have object privileges on that table as well.

Related Commands

See **nzreclaim** in the *IBM Netezza System Administrator's Guide*.

Usage

The following provides sample usage.

- To migrate data for a versioned table, enter:

```
system(admin)=> GROOM TABLE <table_name> VERSIONS;
```

- To reclaim deleted records in a table (equivalent to nzreclaim -records), enter:

```
system(admin)=> GROOM TABLE <table_name> RECORDS ALL;
```

- To identify data pages containing only deleted records and to reclaim extents that are empty as a result (an extension of nzreclaim -startEndBlocks), enter:

```
system(admin)=> GROOM TABLE <table_name> PAGES ALL;
```

- To organize data not already organized in a clustered base table, enter:

```
system(admin)=> GROOM TABLE <table_name> RECORDS READY;
```

INSERT

Use the INSERT command to add new rows to a table.

Synopsis

Syntax for using the INSERT command:

```
INSERT INTO <table> [ ( <column> [, ...] ) ]
    { DEFAULT VALUES | VALUES ( <expression> [, ...] ) | SELECT <query>
};
```

Inputs

The INSERT command takes the following inputs:

Table B-81: INSERT Input

Input	Description
column	Specifies the name of a column in a table.
DEFAULT VALUES	Specifies that all columns be filled by nulls or by values you specified when you created the table using default clauses.
expression	Specifies a valid expression or value assigned to a column.
query	Specifies a valid query. Refer to the SELECT command for a further description of valid arguments.
table	Specifies the name of an existing table.

Outputs

The INSERT command has the following output:

Table B-82: INSERT Output

Output	Description
insert 0 #	Message returned if zero or more rows were inserted. The # stands for the number of rows inserted.
Error: Reload allow NULLs mismatch <table> <col>	When you INSERT data to a table from an external table, the Netezza verifies that the null setting for each column is the same between the tables. If the null settings do not match, the INSERT operaton fails and displays information about the columns which did not have matching null settings.
Error: Large table size limit on <part>, SPU <id>.	Message returned if the table reaches its maximum size (64 GB, unless large table support is enabled).

Description

The INSERT command has the following characteristics:

Privileges Required

You must be an administrator, or an administrator must have given you the appropriate object privileges, to use this command.

You must have the INSERT privilege to append a table.

You must have SELECT privilege on any table specified in a where clause.

Common Tasks

Use the INSERT command to insert new rows into a table. You can insert a single row at a time, or several rows as a result of a query. You can list the columns in the target list in any order.

The system inserts a declared default or null value for any columns not present in the target list. The system rejects the new column if you insert a null into a column declared not null.

If the expression for each column is not of the correct data type, the system attempts automatic type coercion.

Related Commands

None

Usage

The following provides sample usage.

- To insert a single row into the table films, enter:

```
system(admin)=> INSERT INTO films VALUES
    ('UA502','Bananas',105,'1971-07-13','Comedy',INTERVAL '82
minute');
```

- In this sample, the last column len is omitted, and therefore the last column assumes the default value of null:

```
system(admin)=> INSERT INTO films (code, title, did, date_prod,
kind)
    VALUES ('T_601', 'Yojimbo', 106, DATE '1961-06-16', 'Drama');
```

- To insert a single row into the table distributors, enter:

```
system(admin)=> INSERT INTO distributors (name) VALUES ('British
Lion');
```

Note: Because only the column name is specified, the omitted column is assigned its default value.

- To insert several rows into the table films from the table tmp, enter:

```
system(admin)=> INSERT INTO films SELECT * FROM tmp;
```

RESET

Use the RESET command to restore the value of a runtime parameter to its default value.

Synopsis

Syntax resetting a parameter:

```
RESET variable
```

Inputs

The RESET command takes the following inputs:

Table B-83: RESET Input

Input	Description
variable	Specifies the name of a runtime parameter. Refer to the SET command for a list.

Outputs

The RESET command has the following output:

Table B-84: RESET Output

Output	Description
ERROR: '"Is not a valid option name.	Message returned if the system does not recognize the variable that you specify.

Description

The RESET command has the following characteristics:

Privileges Required

You must be an administrator, or an administrator must have given you the appropriate object privileges, to use this command.

Common Tasks

Use the RESET command to restore runtime parameters to their default values.

Related Commands

The RESET command is an alternate form for the set variable to default command.

Usage

The following provides sample usage.

- To set DateStyle to its default value, enter:

```
system(admin)=> RESET DateStyle;
```

- To set Geqo to its default value, enter:

```
system(admin)=> RESET geqo;
```

REVOKE

Use the REVOKE command to remove access privileges for a user, a group, or all users.

Synopsis

Syntax for revoking an object privilege:

```
REVOKE [ GRANT OPTION FOR ] <object_privilege> [, ...]
ON <object> [, ...]
FROM { PUBLIC | GROUP <groupname> | <username> }
```

Syntax for granting an administration privilege:

```
REVOKE [ GRANT OPTION FOR ] <admin_privilege> [, ...]
FROM { PUBLIC | GROUP <groupname> | <username> }
```

Inputs

The REVOKE command takes the following inputs:

Table B-85: REVOKE Input

Input	Description
object	Specifies the target of the privilege. You can further define an object as one or more of the following object class types or one or more named objects of these types: {DATABASE | GROUP | USER | TABLE | VIEW | EXTERNAL TABLE | SEQUENCE | SYNONYM | SYSTEM TABLE | SYSTEM VIEW | MANAGEMENT TABLE | MANAGEMENT VIEW | FUNCTION | AGGREGATE | PROCEDURE } **Note:** TABLE represents user tables, not all tables (user, system, and management). To revoke privileges for system tables, specify the SYSTEM TABLE object.
object_privilege	Specifies any of the following object privileges: ALL, ABORT, ALTER, DELETE, DROP, GENSTATS, INSERT, LIST, SELECT, TRUNCATE, UPDATE, EXECUTE
admin_privilege	Specifies any of the following administration privileges: all admin, BACKUP, [create] <dbobject>, [manage] <manageobject>, RECLAIM, RESTORE where <dbobject> can be: {DATABASE | GROUP | USER | TABLE | TEMP TABLE | EXTERNAL TABLE | VIEW | MATERIALIZED VIEW | SEQUENCE | SYNONYM | FUNCTION | AGGREGATE | PROCEDURE } where <manageobject> can be: {SYSTEM | HARDWARE}

Table B-85: REVOKE Input

Input	Description
PUBLIC	Specifies that the privileges are to be revoked to all users, including users that may be created later. The public group may be thought of as an implicitly defined group that always includes all users. **Note:** Each user has a sum of privileges: • Granted directly to the user. • Granted to any group the user is presently a member of. • Granted to public.
GRANT OPTION FOR	Revokes the WITH GRANT OPTION.

Outputs

The REVOKE command has the following output

Table B-86: REVOKE Output

Output	Description
REVOKE	Message returned if the command is successful.

Description

The REVOKE command has the following characteristics:

Privileges Required

You must be an administrator, or an administrator must have given you the appropriate object privileges, to use this command. If you are the creator of an object, you can revoke previously granted permissions.

Common Tasks

Use the REVOKE command to revoke permissions from one or more users or groups of users.

You can use the \dp internal slash command to obtain information about privileges on existing objects. For more information about slash commands, refer to the *IBM Netezza System Administrator's Guide*.

Note: Privileges granted to a group cannot be revoked from individual members of the group.

Related Commands

See "GRANT" on page B-91 for a description of the privilege types.

Usage

The following provides sample usage.

▼ To revoke the insert privilege for the group public on the table films, enter:

```
system(admin)=> REVOKE INSERT ON films FROM PUBLIC;
```

ROLLBACK

Use the ROLLBACK command to abort the current transaction.

Synopsis

Syntax for rolling back the current transaction:

```
ROLLBACK [ WORK | TRANSACTION ]
```

Inputs

The ROLLBACK command takes the following inputs:

Table B-87: ROLLBACK Input

Input	Description
WORK	These are optional keywords that have no effect.
TRANSACTION	

Outputs

The ROLLBACK command has the following output:

Table B-88: ROLLBACK Output

Output	Description
ABORT	Message returned if the command is successful.
NOTICE: rollback: no transaction in progress	Message returned if there is no transaction in progress.

Description

The ROLLBACK command has the following characteristics:

Privileges Required

You must be an administrator, or an administrator must have given you the appropriate object privileges, to use this command.

Common Tasks

Use the ROLLBACK command to abort the current transaction. The system discards all changes made by the current transaction.

Related Commands

See "COMMIT" on page B-34 to successfully terminate a transaction.

Usage

The following provides sample usage.

▼ To abort all changes, enter:

```
system(admin)=> ROLLBACK WORK;
```

SELECT

Use the SELECT command to retrieve rows from a table or view.

Synopsis

Syntax for using the SELECT command:

```
SELECT [ ALL | DISTINCT ] * | <column> [ AS <output_name> ]
<expression> [ AS <output_name> ] [, ...]
    [ FROM <from_item> [, ...] ]
    [ WHERE <condition> ]
    [ GROUP BY <expression> [, ...] ]
    [ HAVING <condition> [, ...] ]
    [ { UNION | INTERSECT | EXCEPT | MINUS }[ ALL| DISTINCT ]
SELECT
    [ ORDER BY expression [ ASC | DESC | USING <operator>
                          [NULLS {FIRST | LAST}][, ...] ]
    [ LIMIT { <count> | ALL } ]
```

where from_item can be:

table_name

```
[ [ AS ] <alias> [ ( <column_alias_list> ) ] ]
|
( <select> )
    [ AS ] <alias> [ ( <column_alias_list> ) ]
|
    <from_item> [ NATURAL ] <join_type> <from_item>
    [ ON <join_condition> | USING ( <join_column_list> ) ]
```

Inputs

The SELECT command takes the following inputs:

Table B-89: SELECT Input

Input	Description
alias	Specifies a substitute name for the preceding table_name. Use an alias for brevity or to eliminate ambiguity for self-joins (where the same table is scanned multiple times). If you write an alias, you can also write a column alias list to provide substitute names for one or more columns of the table.

Table B-89: SELECT Input (continued)

Input	Description
column	Specifies the name of an existing column.
condition	Specifies a boolean expression giving a result of true or false.
expression	Specifies the name of a table's column or an expression.
from_item	Specifies a table reference, sub-select, or join clause.
join_column_list	A using join_column_list (a, b, ...) is shorthand for the on condition left_table.a = right_table.a and left_table.b = right_table.b.
join_condition	Specifies a qualification condition. This is similar to the where condition except that it only applies to the two from_items being joined in this join clause.
join_type	Specifies one of the following: [inner] join, left [outer] join, right [outer] join, full [outer] join, or cross join. **Note:** For inner and outer join types, you must include exactly one of natural, on join_condition, or using (join_column_list). For the cross join type, none of these items may appear.
output_name	Specifies another name for an output column using the AS clause. You typically use this name to label a column for display. You can also use it to refer to the column's value in order by and group by clauses. **Note:** You cannot use the output_name input in the where or having clauses; write out the expression instead.
select	Specifies a select command that can include all features except the order by, for update, and limit clauses. If you parenthesize the select, you can include the order by, for update, and limit clauses. When you include a sub-select in the from clause, the sub-select acts as though its output were created as a temporary table for the duration of this single select command. **Note:** You must enclose the sub-select in parentheses, and provide an alias for it.
table_name	In a from clause, specifies the name of an existing table or view.

Outputs

The SELECT command has the following output:

Table B-90: SELECT Output

Output	Description
ROWS	Returns the complete set of rows resulting from the query.
COUNT	Returns the number of rows returned by the query.

Description

The command has the following characteristics:

Privileges Required

You must be an administrator, or an administrator must have given you the appropriate object privileges, to use this command. You must have the Select privilege to a table to read its values.

Note: As of Release 4.0, in order to select data from an external table, you must have the Select privilege on the EXTERNAL TABLE class.

While connected to the "system" database, you can grant the privilege as follows:

```
GRANT SELECT ON EXTERNAL TABLE to user;
```

The example gives all users the ability to select. If you want to restrict the privilege to one user, you would replace "user" with the specific user's name.

To read about object privileges and object classes, refer to the *IBM Netezza Administrator's Guide, Chapter 8, Establishing Security and Access Control*. To understand how privileges are assigned based upon the database you are connected to, read the section, *Understanding Object Privileges*.

Common Tasks

The SELECT command returns rows from one or more tables. Use select to choose among rows that satisfy the where condition, and return rows based upon additional criteria you specify within the command. If you omit the where condition, the SELECT command chooses from all rows.

The system forms output rows by computing the select output expressions for each selected row.

- You can include * in the output list as a shorthand way of indicating all columns of selected rows. You can also indicate table_name.* as a shorthand for the columns coming from a specific table. "Querying a Table" on page 2-16 describes functions that you can use within a SELECT command.
- You can use the keyword distinct to eliminate duplicate rows from the result. The all keyword (the default) returns all candidate rows, including duplicates.

Select Clauses

FROM — The FROM clause specifies one or more source tables for the SELECT command. If you specify multiple sources, the result is the Cartesian product of all the rows in all the sources. Usually, though, you add qualifying conditions to restrict the rows the system returns to a small number of the Cartesian product.

You can parenthesize a subselect command within a from clause. Using a subselect command is the only way to get multiple levels of grouping, aggregation, or sorting in a single query.

Note: You must specify an alias for the subselect command.

A FROM item can be a join clause, which combines two simpler from items. Use parentheses, if necessary, to determine the order of nesting.

WHERE — This clause has the following general form:

```
WHERE boolean_expr
```

The boolean_expr portion of the clause can consist of any expression that produces a boolean value.In many cases, you use the expression as follows:

```
expr cond_op expr
or
log_op expr
```

where:

- cond_op can be one of: =, <, <=, >, >= or <>, a conditional operator like all, any, in, like, or a locally defined operator.
- log_op can be one of: and, or, not. select ignores all rows for which the where condition does not return true.

GROUP BY — Allows you to divide a table into groups of rows that match one or more values.Specifies a grouped table derived by applying the clause:

```
GROUP BY expression [, ...]
```

The group by clause condenses, into a single row, all selected rows that share the same values for the grouped columns. The system computes aggregate functions across all rows making up each group, producing a separate value for each group (whereas without group by, an aggregate produces a single value computed across all the selected rows). When you include the group by clause, the SELECT command output expression(s) cannot refer to ungrouped columns except within aggregate functions, because there would be more than one possible value to return for an ungrouped column.

A group by value can be:

- An input column name.
- The name or ordinal number of an output column (select expression).
- An arbitrary expression formed from input-column values. In case of ambiguity, the system interprets a group by name as an input-column name rather than an output column name.

HAVING — The optional having clause has the general form:

```
HAVING boolean_expr
```

where boolean_expr is the same as specified for the where clause.

The HAVING clause specifies a grouped table derived by the elimination of rows that do not satisfy the boolean_expr.

The having clause is different from the where clause:

- The where clause filters individual rows before application of group by.
- The having clause filters group rows created by group by.

Each column referenced in boolean_expr must unambiguously reference a grouping column, unless the reference appears within an aggregate function.

In a grouping select, the having clause can only reference expressions that are single-valued within a group. That is, you can only reference group fields, aggregates, or single-valued expressions derived from group fields or aggregates (which must include constants).

For example, to return grp and counts of grps with more than four members, enter:

```
SELECT grp, count(id) AS n FROM emp GROUP BY grp HAVING n > 4
```

In a non-grouping select where the select is conceptually grouped by zero group fields, you can only reference aggregates or expressions that are single-valued.

For example to return no rows if there are four or fewer employees in emp, or one row with the count if there are more than four employees in emp, enter:

```
SELECT count(id) AS n FROM emp HAVING n > 4
```

UNION — Operator causes the system to compute the collection of rows returned by the queries. Eliminates duplicate rows unless you specify the ALL keyword.

```
table_query UNION [ ALL ] table_query
[ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
[ LIMIT { COUNT | ALL } ]
[ OFFSET start ]
```

where table_query specifies any select expression without an ORDER BY, FOR UPDATE, or LIMIT clause.

Note: If you enclose a sub-expression in parentheses, you can include ORDER BY and LIMIT clauses. If you do not include parentheses, the clauses are taken to apply to the result of the union, not to its right-hand input expression.

The UNION operator computes the collection (set union) of the rows returned by the queries involved. The two selects that represent the direct operands of the union must produce the same number of columns, and corresponding columns must be of compatible data types.

The result of union does not contain any duplicate rows unless you specify the ALL option. The ALL option prevents elimination of duplicates.

Multiple union operators in the same SELECT command are evaluated left to right, unless you indicate otherwise by using parentheses.

ORDER BY — Clause allows you to sort returned rows in the order that you specify. The first row of each set is unpredictable unless you include the order by clause to ensure an order to the rows.

An order by clause can be one of the following:

- The name or ordinal number of an output column (select expression).
- An arbitrary expression formed from input-column values. Note that, in case of ambiguity, an order by name is interpreted as an output-column name.

The ordinal number refers to the ordinal (left-to-right) position of the result column. This feature makes it possible to define an ordering on the basis of a column that does not have a proper name. You can assign a name to a result column using the as clause. For example:

```
SELECT title, date_prod + 1 AS newlen FROM films ORDER BY newlen;
```

You can also order by arbitrary expressions, including fields that do not appear in the select result list. For example:

```
SELECT name FROM distributors ORDER BY code;
```

A limitation of this feature is that an order by clause applying to the result of a union query may only specify an output column name or number, not an expression.

Note: If an order by clause is a simple name that matches both a result column name and an input column name, order by interprets it as the result column name. This is the opposite of the choice that the group by clause makes in the same situation.

You can add the keyword DESC (descending) or ASC (ascending) after each column name in the order by clause. If you do not specify desc, asc is assumed by default. Alternatively, you can specify a specific ordering operator name:

- ASC is equivalent to using <
- DESC is equivalent to using >

  The null value sorts higher than any other value in a domain:

- With ascending sort order, nulls sort at the beginning.
- With descending sort order, nulls sort at the end.

You can use the NULLS FIRST | NULLS LAST keywords to specify a sorting order for null data. If your data contains nulls, Netezza SQL considers all null values to be lower than any non-null value. This means that for an ascending sort, nulls appear first in the output, or for a descending sort, they appear last.

LIMIT — Clause returns a subset of the rows produced by your query.

```
LIMIT { COUNT | ALL }
```

where count specifies the maximum number of rows to return.

The LIMIT clause allows you to retrieve a portion of the rows generated by the query. If you designate a limit, the system returns only that number of rows.

When using the LIMIT option, use an ORDER BY clause that constrains the result rows into a unique order. Otherwise you can get an unpredictable subset of the query's rows; for example, you may be asking for the tenth through twentieth rows, but tenth through twentieth in what order? You do not know what order unless you specify the order by clause.

The query optimizer takes the LIMIT clause into account when generating a query plan, so you are likely to get different plans (yielding different row orders) depending on what you use for limit. Using different limit values to select different subsets of a query result gives inconsistent results unless you enforce a predictable result ordering with the order by clause. SQL does not promise to deliver the results of a query in any particular order unless you use the order by clause to constrain the order.

Note: If you specify the LIMIT clause, and have also specified a rowset limit, the system returns whichever number is lower.

INTERSECT — Combines the results of two queries into a single result that comprises all the rows common to both queries. For more information, see “Using the INTERSECT Operation” on page 2-20.

EXCEPT or MINUS — Finds the difference between the two queries and the result comprises the rows that belong only to the first query. For more information, see “Using the EXCEPT Operation” on page 2-20

CROSS JOIN and **INNER JOIN** — Produces a simple Cartesian product, the same as you would get if you listed the two items at the top level of FROM.

The cross join and inner join on (true) types are equivalent; no rows are removed by qualification.

Note: These join types are just a notational convenience. You can accomplish the same results using the from and where clauses.

LEFT OUTER JOIN — Returns all rows in the qualified Cartesian product (that is, all combined rows that pass its on condition), plus one copy of each row in the left-hand table for which there was no right-hand row that passed the on condition.

Note: The system considers only the join's own on or using condition to determine those rows that have matches. It applies outer on or where conditions afterwards.

RIGHT OUTER JOIN — Returns all the joined rows, plus one row for each unmatched right-hand row (extended with nulls on the left).

Note: This is just a notational convenience. You could convert it to a left outer join by switching the left and right inputs.

FULL OUTER JOIN — Returns all the joined rows, plus one row for each unmatched left-hand row (extended with nulls on the right), plus one row for each unmatched right-hand row (extended with nulls on the left).

Note: For all the join types except cross join, you must write exactly one of on join_condition, using (join_column_list), or natural:

- on is the most general case: you can write any qualification expression involving the two tables you wish to join.
- A using column list (a, b, ...) is shorthand for the on condition left_table.a = right_table.a and left_table.b = right_table.b ... using implies that only one of each pair of equivalent columns is to be included in the join output, not both.
- natural is shorthand for a using list that mentions all similarly-named columns in the tables.

Related Commands

The WITH Clause can be used wherever SELECT can be used. See "WITH Clause" on page B-137.

Usage

The following provides sample usage.

- To join the table films with the table distributors, enter:

```
system(admin)=> SELECT f.title, f.did, d.name, f.date_prod, f.kind
FROM distributors d, films f WHERE f.did = d.did;

title  |did|name|date_prod|kind
------------------------+-----+------------------+-----------+-------
The Third Man|101|British Lion|1949-12-23|Drama
The African Queen|101|British Lion|1951-08-11|Romantic
Une Femme est une Femme|102|Jean Luc Godard|1961-03-12|Romantic
Vertigo|103|Paramount|1958-11-14|Action
Becket |103|Paramount|1964-02-03|Drama
48 Hours|103|Paramount|1982-10-22|Action
War and Peace|104|Mosfilm|1967-02-12|Drama
West Side Story|105|United Artists|1961-01-03|Musical
```

```
Bananas|105|United Artists|1971-07-13|Comedy
Yojimbo|106|Toho|1961-06-16|Drama
There's a Girl in my Soup|107|Columbia|1970-06-11|Comedy
Taxi Driver|107|Columbia|1975-05-15|Action
Absence of Malice|107|Columbia|1981-11-15|Action
Storia di una donna|108|Westward|1970-08-15|Romantic
The King and I|109|20th Century Fox|1956-08-11|Musical
Das Boot|110|Bavaria Atelier|1981-11-11|Drama
Bed Knobs and Broomsticks|111|Walt Disney||Musical
(17 rows)
```

- To sum the column len of all films and group the results by kind, enter:

```
system(admin)=> SELECT kind, SUM(len) AS total FROM films GROUP BY
kind;
kind|total
----+-------
Action|07:34
Comedy|02:58
Drama|14:28
Musical|06:42
Romantic|04:38
(5 rows)
```

- To sum the column len of all films, group the results by kind, and show those group totals that are less than 5 hours, enter:

```
system(admin)=> SELECT kind, SUM(len) AS total FROM films GROUP BY
kind HAVING SUM(len) < INTERVAL '5 HOUR';
kind| total
----+------
Comedy| 02:58
Romantic| 04:38
(2 rows)
```

- To sort the individual results according to the contents of the second column (name), use either example:

```
system(admin)=> SELECT * FROM distributors ORDER BY name;
system(admin)=> SELECT * FROM distributors ORDER BY 2;
did| name
---|--------------------
109| 20th Century Fox
110| Bavaria Atelier
101| British Lion
107| Columbia
102| Jean Luc Godard
113| Luso Films
```

```
104| Mosfilm
103| Paramount
106| Toho
105| United Artists
111| Walt Disney
112| Warner Bros.
108| Westward
(13 rows)
```

- To show how to obtain the union of the tables distributors and actors, restricting the results to those that begin with letter W in each table, enter:

```
system(admin)=> SELECT distributors.name
FROM distributors
WHERE distributors.name LIKE 'W%';
UNION
SELECT actors.name
FROM actors
WHERE actors.name LIKE 'W%';
Walt Disney
Walter Matthau
Warner Bros.
Warren Beatty
Westward
Woody Allen
```

Note: Because only distinct rows are wanted, the *all* keyword is omitted.

SET

Use the SET command to set or change a runtime parameter, the time zone, or system limits.

Synopsis

Syntax for using the SET command:

```
SET <variable> { TO | = } { value | 'value' | DEFAULT }
SET TIME ZONE { 'timezone' | LOCAL | DEFAULT }
```

Inputs

The SET command takes the following inputs:

Table B-91: SET Input

Input	Description
TIME ZONE	Specifies the time zone for the system.

Table B-91: SET Input

Input	Description
value	Specifies a new value for a parameter. **Note:** • You can use default to reset a parameter to its default value. • You can use lists of strings: Single or double quote complex constructs.
variable	Specifies a setable runtime parameter.

Outputs

The SET command has the following output:

Table B-92: SET Output

Output	Description
SET VARIABLE	Message returned if the command is successful.
ERROR: not a valid option name: name	Message returned if the parameter you try to set does not exist.
ERROR: permission denied	Message returned if you do not have proper access. **Note:** You must be an administrator to have access to certain settings.
ERROR: name can only be set at start-up	Message returned for certain parameters that are fixed once the server is started.

Description

The SET command has the following characteristics:

Privileges Required

You must be an administrator, or an administrator must have given you the appropriate object privileges, to use this command.

Common Tasks

Use the SET command to set or change runtime configuration parameters. The command allows you to change parameters for the duration of the database connection.

You can alter the following parameters:

DATESTYLE — A value for set datestyle can include an output style, a substyle, or an output and substyle separated by a comma.

Output Styles — Choose the date/time representation style. (Two separate settings are made: the default date/time output and the interpretation of ambiguous input.)

The following are date/time output styles:

- ISO — Use ISO 8601-style dates and times (YYYY-MM-DD HH:MM:SS). This is the default.
- SQL — Use Oracle-style dates and times.
- German — Use dd.mm.yyyy for numeric date representations.

Substyles — The following two options determine both a substyle of the "SQL" output formats and the preferred interpretation of ambiguous date input.

- European — Use dd/mm/yyyy for numeric date representations.
- NonEuropean/US — Use mm/dd/yyyy for numeric date representations.

You can initialize the date format by setting the pgdatestyle environment variable. If pgdatestyle is set in the frontend environment of a client based on libpq, libpq automatically sets datestyle to the value of pgdatestyle during connection start-up.

The datestyle option is really only intended for porting applications. To format your date/time values to choice, use the to_char family of functions.

RANDOM_SEED — Sets the internal seed for the random number generator.

value — The random function uses this value. You can specify floating-point numbers between 0 and 1, which are then multiplied by 2(31)-1. (This product will silently overflow if a number outside the range is used.)

You can also set the seed by invoking the setseed function: select setseed(value);

Related Commands

Use "SHOW" on page B-123 to show the current setting of a parameter.

Usage

The following provides sample usage.

- To set the style of date to the European convention, enter:

```
SET DATESTYLE TO European;
```

- To set the threshold to 20 percent, enter:

```
SET SYSTEM DEFAULT MATERIALIZE THRESHOLD 20;
```

SET AUTHENTICATION

Use the SET AUTHENTICATION command to specify how the Netezza authenticates users who log on to the system. Authentication verifies that the user has entered a correct and known username-password combination when logging on to the Netezza system.

Note: To access a database, the user must also be defined by a CREATE USER statement, and must have been granted access rights to a particular database. For more information about configuring and using LDAP authentication, refer to the *IBM Netezza System Administrator's Guide*.

Synopsis

Syntax for setting authentication.

```
SET AUTHENTICATION { LOCAL | LDAP [(<ldap-config>)] }
```

Syntax for "ldap-config" when you choose LDAP authentication.

```
BASE <base-string>
SERVER <server>
[ VERSION <version-number> ]
[ BINDDN { bind-string | NONE } ]
[ BINDPW { bind-password | NONE } ]
[ PORT <port> ]
[ SCOPE { SUB | ONE | BASE } ]
[ SSL { ON | OFF } ]
[ ATTRNAME <attrname-string>]
[ NAMECASE { LOWERCASE | UPPERCASE }]
```

Inputs

The SET AUTHENTICATION command takes the following inputs:

Table B-93: SET AUTHENTICATION Input

Input	Description
attrname-string	The attribute name, defined in the LDAP schema, for the field containing the user name. The default is "cn".
base-string	The Distinguished Name (DN) within the LDAP namespace where username searches start. An example follows. `dc=example,dc=org`
bind-password	The password that accompanies the <bind-string> for binding to the LDAP server.
bind-string	The Distinguished Name to use when binding to the LDAP server. A bind string is optional. This clause is typically not defined for performing anonymous LDAP look-ups.
LDAP	The Netezza uses an LDAP server for authentication.
LOCAL	The Netezza uses local authentication. When a user connects to the Netezza system, the system uses the username and password defined by the CREATE USER command to authenticate. **Note:** This authentication mode is also compatible with releases before Release 4.5.
LOWERCASE | UPPERCASE	Indicates whether the LDAP server stores the user name in lowercase or uppercase.
SSL	Default is OFF. If ON, SSL is used when contacting the LDAP server.
port	The port number to use when connecting to the LDAP server. The default is 389.
scope	The scope to use when searching the LDAP tree.

Table B-93: SET AUTHENTICATION Input (continued)

Input	Description
server	The name or IP address of the LDAP server.
version-number	The LDAP protocol version number to use. The default is 3.

Outputs

The SET AUTHENTICATION command has the following output:

Table B-94: SET AUTHENTICATION Output

Output	Description
ERROR: permission denied	Message returned if you do not have proper access. **Note:** You must have the Manage System privilege to set authentication.
SET AUTHENTICATION	Message returned if the command is successful.

Description

The SET AUTHENTICATION command has the following characteristics:

Privileges Required

You must be an administrator, or an administrator must have given you the Manage System privilege, to use this command.

Implementation Notes

The SET AUTHENTICATION command performs as follows.

- It verifies that the user has the correct access rights to perform the operation.
- When you change the authentication from LOCAL to LDAP, the system does the following:
  - Creates a new Pluggable Authentication Module (PAM) file for the new authentication settings, /etc/pam.d/netezza_nps.
  - Modifies the file /nz/data/pg_hba.conf, changing the authorization to LDAP. (The pg_hba.conf file controls client authentication.)
  - Copies the /etc/ldap.conf file to /nz/data/config/ldap.conf.orig and then updates the /etc/ldap.conf file for the changes in the command.
- When you change the authentication from LDAP to LOCAL, the system does the following:
  - Changes the LDAP configuration files by copying ldap.conf.orig to ldap.conf.
  - Deletes the PAM file /etc/pam.d/netezza_nps.
  - Modifies the file /nz/data/pg_hba.conf, changing the authorization to MD5 (the default).

- When the authentication is LDAP, you can issue additional or subsequent SET AUTHENTICATION LDAP commands to update the LDAP configuration parameters. The system changes the LDAP configuration file using the options that you specify in the command. Note that the command does not retain settings from any previous SET AUTHENTICATION commands; you must specify the command and all of the options that you need when you issue the command.
- The system updates the catalog to record the authentication configuration you have chosen. The system stores the parameters you enter in the _t_systemdef system table. The _t_systemdef system table stores data using a "Tag" and "value" scheme. The system stores each option you define with the SET AUTHENTICATION command in a separate row in the table, using a predefined tag and the option value. Rows for LDAP authentication follow.

  Note: If you specify LOCAL authentication, the table contains only one row which includes the tag "AUTHENTICATION METHOD" and the value "local." The following example shows a _t_systemdef table for an LDAP configuration.

```
Tag 'AUTHENTICATION METHOD' value 'LDAP'
Tag 'AUTHMTHD LDAP BASE' value <whatever value entered for base-
string>
Tag 'AUTHMTHD LDAP SERVER' value <server>
Tag 'AUTHMTHD LDAP VERSION' value <version-number>
Tag 'AUTHMTHD LDAP BINDDN' value <bind-string, if entered, or NONE>
Tag 'AUTHMTHD LDAP BINDPW' value <bind-password, if entered, or
NONE>
Tag 'AUTHMTHD LDAP PORT' value <port>
Tag 'AUTHMTHD LDAP SCOPE' value <value entered — SUB, ONE, or BASE>
Tag 'AUTHMTHD LDAP SSL' value (value entered — ON or OFF>
Tag 'AUTHMTHD LDAP ATTRNAME' value (attrname-string)
Tag 'AUTHMTHD LDAP NAMECASE' value (value entered — LOWERCASE or
UPPERCASE)
```

- The system records authentication changes in the pg.log file. All SET AUTHENTICATION and SHOW AUTHENTICATION commands are written to the pg.log file. If you use the BINDPW option, passwords are masked-out in the pg.log file.

Related Commands

See "SHOW AUTHENTICATION" on page B-124.

See "CREATE USER" on page B-65.

See "ALTER USER" on page B-24.

Usage

The following provides sample usage.

```
SET AUTHENTICATION ldap base 'dc=netezza,dc=com' server
'ldapserver.netezza.com' port '389' version '3' binddn 'ldapreader'
scope 'base' ssl 'off' attrname 'cn' namecase 'lowercase';
```

SET CONNECTION

Use the SET CONNECTION command to define the Netezza host access records for Netezza clients. The host access records define how Netezza clients connect to the Netezza system, using either secured or unsecured connections. This command allows you to insert and delete rows from the system table. For a complete description of how to configure SSL connections for Netezza client users, refer to the *IBM Netezza System Administrator's Guide*.

Synopsis

Syntax for setting a connection.

```
SET CONNECTION { LOCAL | HOST | HOSTSSL | HOSTNOSSL }
DATABASE { '<database_name>' | 'ALL'}
[ IPADDR '<ip_address>' ]
[ IPMASK '<ip_address_mask>' ]
```

Inputs

The SET CONNECTION command takes the following inputs:

Table B-95: SET CONNECTION Input

Input	Description
LOCAL	Specifies that you are defining a connection record for users who connect to the Netezza using UNIX sockets.
HOST	Specifies that you are defining a connection record for users who connect to the Netezza over IP using either secured or unsecured SSL connections.
HOSTSSL	Specifies that you are defining a connection record for users who connect to the Netezza over IP using only secured SSL connections.
HOSTNOSSL	Specifies that you are defining a connection record for users who connect to the Netezza over IP using only unsecured SSL connections.
database_name | ALL	Specifies the database_name to which the client users at the specified IP address or range have connection access. Specify ALL to match on all databases. Note that the value ALL indicates that the user can attempt to connect to any of the databases on the Netezza system; however, the user account object permissions control the database objects that the user is permitted to view.
ip_address	Specifies the IP address for the client system; applies to HOST, HOSTSSL, and HOSTNOSSL connection types.
ip_address_ mask	Specifies the IP address mask for the client system; applies to HOST, HOSTSSL, and HOSTNOSSL connection types.

Outputs

The SET CONNECTION command has the following output:

Table B-96: SET CONNECTION Output

Output	Description
ERROR: permission denied	Message returned if you do not have proper access. **Note:** You must have the Manage System privilege to set a connection.
SET CONNECTION	Message returned if the command is successful.

Description

The SET CONNECTION command has the following characteristics:

Privileges Required

You must be an administrator, or an administrator must have given you the Manage System privilege, to use this command.

Common Tasks

Specify which Netezza system connections should use an SSL connection.

Related Commands

See “CREATE USER” on page B-65.

See “ALTER USER” on page B-24.

See “SHOW CONNECTION” on page B-126.

See “DROP CONNECTION” on page B-72.

Usage

The following provides sample usage.

```
SET CONNECTION HOST DATABASE 'ALL' IPADDR '192.168.0.0' IPMASK
'255.255.255.0';
```

SET HISTORY CONFIGURATION

Use the SET HISTORY CONFIGURATION command to specify a configuration for query history logging. The new configuration does not take effect immediately. You must restart the Netezza software to start history collection using the new configuration.

Synopsis

Syntax for setting the history configuration:

```
SET HISTORY CONFIGURATION <config-name>
```

Inputs

The SET HISTORY CONFIGURATION command has the following inputs:

Table B-97: SET HISTORY CONFIGURATION Inputs

Input	Description
<config-name>	Specifies the name of the configuration to set. The configuration must exist on the Netezza system.

Outputs

The SET HISTORY CONFIGURATION command has the following outputs:

Table B-98: SET HISTORY CONFIGURATION Output

Output	Description
SET HISTORY CONFIGURATION	Message returned if the command is successful.
ERROR: permission denied	You must have Manage Security permission to set the query history configuration.
ERROR: <config-name> not found.	The specified configuration name could not be found.

Description

This command sets the history configuration to the one that will take effect after the next Netezza software restart. After the restart, the history loader process will attempt to load any existing history data in the staging or loading area for the previous configuration. This load process could fail if the previous configuration was dropped.

Privileges Required

You must have Manage Security permissions to set query history configurations.

Related Commands

See the "CREATE HISTORY CONFIGURATION" on page B-44 to create a new configuration.

See the "ALTER HISTORY CONFIGURATION" on page B-10 to modify configurations.

See the "DROP HISTORY CONFIGURATION" on page B-76 to drop configurations.

See the "SHOW HISTORY CONFIGURATION" on page B-127 to display information about a configuration.

Usage

The following sample command sets the configuration to the all_hist configuration:

```
SYSTEM(ADMIN)=> SET HISTORY CONFIGURATION all_hist;
```

SET SESSION

Use the SET SESSION command to set the characteristics of the current SQL-transaction.

Synopsis

Syntax for setting a session:

```
SET SESSION { READ ONLY | READ WRITE }
```

Inputs

The SET SESSION command takes the following inputs:

Table B-99: SET SESSION Input

Input	Description
read only	During the session, allows a user to read a database but not write to it. The system returns an error if the user attempts to write to the database. The user can create and write to temporary tables.
read write	Allows a user to read and update a database. This is the default setting.

Outputs

The SET SESSION command has the following output:

Table B-100: SET SESSION Output

Output	Description
SET VARIABLE	Message returned if the command is successful.
ERROR: Read-only session, cannot process this request	Message returned if read only has been specified and the user attempts to write to the database.

Description

The SET SESSION command has the following characteristics:

Privileges Required

You must be an administrator, or an administrator must have given you the appropriate object privileges, to use this command.

Related Commands

See “SET SYSTEM DEFAULT” on page B-120.

Usage

The following provides sample usage.

▼ To set a session to read-only, enter:

```
system(admin)=> SET SESSION READ ONLY;
```

SET SYSTEM DEFAULT

Use the SET SYSTEM DEFAULT command to the system defaults for session timeout, rowset limit, query timeout, and materialized view refresh threshold.

Synopsis

Syntax for setting the system default:

```
SET SYSTEM DEFAULT
[SESSIONTIMEOUT | ROWSETLIMIT | QUERYTIMEOUT ] TO [number |
UNLIMITED ]
[DEFPRIORITY | MAXPRIORITY ] TO [critical | high | normal | low |
SET SYSTEM DEFAULT MATERIALIZE [REFRESH] THRESHOLD TO <number>
SET SYSTEM DEFAULT PASSWORDEXPIRY TO pwdexpiry
SET SYSTEM DEFAULT PASSWORDPOLICY TO <conf>
```

Inputs

The SET SYSTEM DEFAULT command takes the following inputs:

Table B-101: SET SYSTEM DEFAULT Input

Input	Description
DEFPRIORITY	Specifies the default priority for the system. The valid priorities are critical, high, normal, low.
MATERIALIZE THRESHOLD	Specifies the refresh threshold for materialized views.
MAXPRIORITY	Specifies the maximum priority for the system.
QUERYTIMEOUT	Specifies the amount of time a query can run before the system sends the administrator a message. You can specify from 1 to 35,791,394 minutes or zero for unlimited. **Note:** To receive a message, you must enable the RunAwayQuery event rule. For more information, see the *IBM Netezza System Administrator's Guide*.
ROWSETLIMIT	Specifies the number of rows a query can return. You can specify from 1 to 2,147,483,647 rows or zero for unlimited.
SESSIONTIMEOUT	Specifies the amount of time a session can be idle before the system terminates it. You can specify from 1 to 35,791,394 minutes or zero for unlimited.
PASSWORDEX-PIRY	Sets the global password expiration default parameter to an *n* number of days. The *n* count begins with the date of the last password change. A 0 indicates that the passwords do not expire.

Table B-101: SET SYSTEM DEFAULT Input

Input	Description
PASSWORDPOLICY	Sets the configuration string parameter for the global password policy, and can take the following options, all of which take an integer: • minlen – Specifies a minimum length of *x* characters (*x* must be greater than 0). The default and minimum is 6, and a setting of less than 6 is ignored. **Note:** The following options must all be an integer. For more information on the meaning and usage of each, see the *IBM Netezza Advanced Security Administrator's Guide*. • lcredit – Specifies a lowercase credit. The default is 1. • ucredit – Specifies an uppercase credit. The default is 1. • dcredit – Specifies a digit credit. The default is 1. • ocredit – Specifies other credit. The default is 1. If options are not set, the default values determine the policy. These options are based on those for *pam.cracklib* (8). For more information, see the Linux documentation.

Outputs

The SET SYSTEM DEFAULT command has the following output:

Table B-102: SET SESSION DEFAULT Output

Output	Description
SET SYSTEM DEFAULT	The message that the system returns if the command is successful.

Description

The SET SYSTEM DEFAULT command has the following characteristics:

Privileges Required

You must be an administrator, or an administrator must have given you the appropriate object privileges, to use this command.

Common Tasks

Use the SET SYSTEM DEFAULT command to change the system defaults for session idle timeout, rowset limits, and query timeout.

The system calculates the values at session startup and them remain in effect for the duration of the session.

You can also set session timeout, rowset limits, and query timeout at the user and/or group level. The runtime resolution for these values is:

- The value assigned to the user

- The minimum values assigned to groups of which the user is a member
- The default system value.

MATERIALIZE REFRESH THRESHOLD — Sets the threshold percentage of unsorted data in a materialized view. When you refresh a base table, all associated materialized views that have exceeded this threshold are refreshed.

Related Commands

Use "SHOW" on page B-123 to show the current setting of a parameters. See the *IBM Netezza System Administrator's Guide* for more information.

Usage

The following provides sample usage.

- To set the system default timeout to five hours (300 minutes), enter:

```
system(admin)=> SET SYSTEM DEFAULT SESSIONTIMEOUT TO 300;
```

SET TRANSACTION

Use the SET TRANSACTION command to set the transaction characteristics of the current session.

Synopsis

Syntax for setting a transaction:

```
SET TRANSACTION {READ ONLY | READ WRITE}
SET TRANSACTION ISOLATION LEVEL { READ UNCOMMITTED | READ COMMITTED |
REPEATABLE READ | SERIALIZABLE }
```

Inputs

The SET TRANSACTION command takes the following inputs:

Table B-103: SET TRANSACTION Input

Input	Description
READ ONLY	During the session, allows a user to read a database but not write to it. The system returns an error if the user attempts to write to the database. The user can create and write to temporary tables.
READ WRITE	Allows a user to read and update a database. This is the default setting.
READ UNCOMMITTED READ COMMITTED REPEATABLE READ	Netezza SQL allows the syntax, but does not support these isolation levels.
SERIALIZABLE	Specifies that the current transaction can only see rows committed before the first query or data-modification statement was executed in this transaction.

Outputs

The SET TRANSACTION command has the following output:

Table B-104: SET TRANSACTION Output

Output	Description
SET VARIABLE	Message returned if the command is successful.

Description

The SET TRANSACTION command has the following characteristics:

Privileges Required

You must be an administrator, or an administrator must have given you the appropriate object privileges, to use this command.

Common Tasks

The SET TRANSACTION isolation level command sets the transaction isolation level. The isolation level of a transaction determines what data the transaction can see when other transactions are running concurrently.

Netezza SQL supports the syntax for all isolation levels, but currently implements only the serializable level.

Serializable means that the current transaction can only see rows committed before the first query or data-modification statement was executed in this transaction. Intuitively, serializable means that two concurrent transactions leave the database in the same state as if the two had been executed strictly after one another in either order.

Related Commands

None.

Usage

The following provides sample usage.

- To set the transaction isolation level command, enter:

```
system(admin)=> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET VARIABLE
```

SHOW

Use the SHOW command to display runtime parameters.

Synopsis

Syntax for using the SHOW command:

```
SHOW <name>
```

Inputs

The SHOW command takes the following inputs:

Table B-105: SHOW Input

Input	Description
name	Specifies the name of a runtime parameter.

Outputs

The SHOW command has the following output:

Table B-106: SHOW Output

Output	Description
SHOW VARIABLE	Message returned if the command is successful.

Description

The SHOW command has the following characteristics:

Privileges Required

You must be an administrator, or an administrator must have given you the appropriate object privileges, to use this command.

Common Tasks

Use the SHOW command to display the current setting for a runtime parameter.

Related Commands

Use "SET" on page B-110 to set runtime variables. Note that the system sets some variables at start-up.

Usage

The following provides sample usage.

- To show the current DateStyle setting, enter:

```
system(admin)=> SHOW DateStyle;
NOTICE:  DateStyle is ISO with US (NonEuropean) conventions
```

SHOW AUTHENTICATION

Use the SHOW AUTHENTICATION command to display the current user authentication configuration.

Synopsis

Syntax for showing current authentication configuration:

```
SHOW AUTHENTICATION [ALL]
```

Inputs

The SHOW AUTHENTICATION command takes the following inputs:

Table B-107: SHOW AUTHENTICATION Input

Input	Description
ALL	If you specify ALL, the results include all of the attributes of the authentication setting, except for the password. If you do not specify ALL, the command shows only the type of user logon authentication (that is, LOCAL or LDAP).

Outputs

The SHOW AUTHENTICATION command has the following output:

Table B-108: SHOW AUTHENTICATION Output

Output	Description
ERROR: Not Supported	You do not have the proper permission to view the results of this command.
SHOW AUTHENTICATION	Message returned if the command is successful.

Description

The SHOW AUTHENTICATION command has the following characteristics:

Privileges Required

To view the results of the command SHOW AUTHENTICATION ALL, you must be an administrator, or an administrator must have granted you the following privileges:

- You must have been granted the Manage System privilege.
- You must have been granted the List and Select privileges on the view _v_authentication_settings.

Implementation Notes

The SHOW AUTHENTICATION command performs as follows.

- It verifies that the user has the correct access rights to perform the operation.
- It transforms the request into a SELECT command.
  - If you do not specify ALL, the SELECT is against the view _v_authentication
  - If you do specify ALL, the SELECT is against the view _v_authentication_settings
- It records the request in the pg.log file.

Related Commands

See "SET AUTHENTICATION" on page B-112.

See "CREATE USER" on page B-65.

See "ALTER USER" on page B-24.

Usage

The following provides sample usage.

- If you have the proper access to show authentication settings, you can view all but the password. A sample follows.

```
system (admin) => SHOW AUTHENTICATION ALL;
AUTH_OPTION             |         AUTH_VALUE
------------------------------------------------
AUTHENTICATION METHOD   |  LDAP
AUTHMTHD LDAP ATTRNAME  |  cn
AUTHMTHD LDAP BASE      |  dc=example, dc=org
AUTHMTHD LDAP BINDDN    |  ldapreader
AUTHMTHD LDAP NAMECASE  |  lowercase
AUTHMTHD LDAP PORT      |  389
AUTHMTHD LDAP SCOPE     |  BASE
AUTHMTHD LDAP SERVER    |  myldap.netezza.com
AUTHMTHD LDAP SSL       |  ON
AUTHMTHD LDAP VERSION   |  3
```

- If you do not have the Manage System privilege to show authentication settings, you can view the type of authentication. A sample follows.

```
system (admin) => SHOW AUTHENTICATION;
AUTH_OPTION             |         AUTH_VALUE
------------------------------------------------
AUTHENTICATION METHOD   |  LDAP
```

SHOW CONNECTION

Use the SHOW CONNECTION command to display the Netezza connection records defined for the Netezza client users.

Synopsis

Syntax for showing current authentication configuration:

```
SHOW CONNECTION
```

Inputs

There are no specific options for the SHOW CONNECTION command.

Outputs

The SHOW CONNECTION command has the following output:

Table B-109: SHOW AUTHENTICATION Output

Output	Description
ERROR: permission denied	You do not have the proper permission to view the results of this command.
SHOW CONNECTION	Message returned if the command is successful.

Description

The SHOW CONNECTION command has the following characteristics:

Privileges Required

To view the results of the command SHOW CONNECTION, you must be an administrator, or an administrator must have granted you the following privileges:

- You must have been granted the Manage System privilege.
- You must have been granted the List and Select privileges on the view _v_connection.

Common Tasks

Use this command to show Netezza connections.

Related Commands

See “SET CONNECTION” on page B-116.

See “CREATE USER” on page B-65.

See “ALTER USER” on page B-24.

See “DROP CONNECTION” on page B-72.

Usage

Sample usage with output follows.

```
SYSTEM(ADMIN)=> SHOW CONNECTION;
 CONNID | CONNTYPE   | CONNDB | CONNIPADDR   |   CONNIPMASK    | CONNAUTH
--------+------------+--------+--------------+-----------------+--------
      1 | local      | all    |              |                 | trust
      2 | host       | all    | 127.0.0.1    | 255.255.255.255 | md5
      3 | host       | all    | 0.0.0.0      | 0.0.0.0         | md5
      4 | hostnossl  | ALL    | 192.168.1.2  | 255.255.255.255 | md5
(4 rows)
```

SHOW HISTORY CONFIGURATION

Use the SHOW HISTORY CONFIGURATION command to display information about a configuration for query history logging.

Synopsis

Syntax for showing a configuration:

```
SHOW HISTORY CONFIGURATION [ <config-name> | ALL ]
```

Inputs

The SHOW HISTORY CONFIGURATION command has the following inputs:

Table B-110: SHOW HISTORY CONFIGURATION Inputs

Input	Description
ALL	Displays information about all the configurations defined on the Netezza system.
<config-name>	Displays information about the specified configuration. The configuration must exist on the Netezza system. If you do not specify a name, the command displays information for the current/active configuration.

Outputs

The SHOW HISTORY CONFIGURATION command has the following outputs:

Table B-111: SHOW HISTORY CONFIGURATION Output

Output	Description
SHOW HISTORY CONFIGURATION	Message returned if the command is successful.
ERROR: permission denied	You must have Manage Security permission to display the query history settings.
ERROR: <config-name> not found.	The specified configuration name could not be found.

Description

This command displays the information about a query history configuration or all configurations.

Privileges Required

You must have Manage Security permissions to show query history configurations.

Related Commands

See the "CREATE HISTORY CONFIGURATION" on page B-44 to create a new configuration.

See the "ALTER HISTORY CONFIGURATION" on page B-10 to modify configurations.

See the "DROP HISTORY CONFIGURATION" on page B-76 to drop configurations.

See the "SET HISTORY CONFIGURATION" on page B-117 to specify a configuration for query history logging.

Usage

The following command shows information about the plan_hist configuration:

```
SYSTEM(ADMIN)=> SHOW HISTORY CONFIGURATION plan_hist;

CONFIG_NAME | CONFIG_NAME_DELIMITED | CONFIG_DBNAME | CONFIG_DBNAME_
DELIMITED | CONFIG_DBTYPE | CONFIG_TARGETTYPE | CONFIG_LEVEL | CONFIG_
HOSTNAME | CONFIG_USER | CONFIG_USER_DELIMITED | CONFIG_PASSWORD |
CONFIG_LOADINTERVAL | CONFIG_LOADMINTHRESHOLD | CONFIG_
LOADMAXTHRESHOLD | CONFIG_DISKFULLTHRESHOLD | CONFIG_STORAGELIMIT |
CONFIG_LOADRETRY | CONFIG_ENABLEHIST | CONFIG_ENABLESYSTEM | CONFIG_
NEXT | CONFIG_CURRENT | CONFIG_VERSION
----------+----------------------+---------------+------------------
PLAN_HIST | f                    |               | f
|           3 |              1 |          1 | localhost      |
| f                  |              |               -1 |
-1 |                     -1 |                     -1 |                 -
1 |             -1 | f                | f                  | f
| f              |              1
(1 row)
```

SHOW PLANFILE

Use the SHOW PLANFILE command to display the contents of a query plan file for troubleshooting or investigating query behaviors.

Synopsis

Syntax for showing a plan file:

```
SHOW PLANFILE [planId]
```

Inputs

The SHOW PLANFILE command has the following inputs:

Table B-112: SHOW PLANFILE Inputs

Input	Description
[*planId*]	Specifies the plan ID for the plan that you want to display. The command searches the two newest compressed tar archives for the requested plan. If you do not specify a plan ID, the command displays the plan files for the last plan run in the current database session.

Description

The SHOW PLANFILE command has the following characteristics:

Privileges Required

To view the results of the SHOW PLANFILE command, you must be the admin database user or you must be granted Select privilege on the _vt_plan_file table.

Common Tasks

Use this command to show Netezza connections.

Usage

Sample usage with a short exerpt of the output.

```
TESTER(USER1)=> SHOW PLANFILE;

 <<<<<<<<<<< NPS VERSION >>>>>>>>>>
-- DBOS Version:        7.0.0-0.D-1.P-0.Bld-24857
-- NPS Version :        Release 7.0, Dev 1 [Build 24857]

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
(dbosEvent) Full Plan
Execution Plan [plan id 14, job id 14, sig 0xd1ba4d02]:
SQL: select * from ne_region where r_regionkey=2
1[00]: spu ScanNode table "TESTER.USER1.NE_REGION" 211553 memoryMode=no
flags=0x0 index=0 cost=1 (o)
-- Cost=0.0..0.0 Rows=1 Width=142 Size=142 Conf=80  {(R_REGIONKEY)}
1[01]: spu RestrictNode (non-NULL)
-- (R_REGIONKEY = 2)
1[02]: spu ProjectNode, 3 cols, projectFlags=0x0
0:R_REGIONKEY 1:R_NAME 2:R_COMMENT
-- 0:R_REGIONKEY 1:R_NAME 2:R_COMMENT
1[03]: spu ReturnNode
501[00]: dbs ReturnNode
End Execution Plan
...
----------------
Time                                Duration
Plan Submit 2012-09-06 15:36:40.458393 EDT        0.002 (plan waiting)
Plan Start  2012-09-06 15:36:40.460503 EDT        0.004 (plan execution)
Plan Finish 2012-09-06 15:36:40.464997 EDT        0.007 (plan total)
----------------
...
```

SHOW SESSION

Use the SHOW SESSION command to display information about one or more sessions.

Synopsis

Syntax for showing sessions:

```
SHOW SESSION [ ALL | <session-id> ] [ VERBOSE ]
```

Inputs

The SHOW SESSION command takes the following inputs:

Table B-113: SHOW SESSION Input

Input	Description
ALL	Display information about all sessions. Depending on your privileges, specific information about other sessions may not be provided.
<session-id>	A number identifying an active session. If neither ALL nor <session-id> is specified, SHOW SESSION displays information about the current session.
VERBOSE	Display detailed information about the session or sessions.

Description

The SHOW SESSION command has the following characteristics:

Privileges Required

You must be the administrator or have the Manage System privilege to see all sessions, otherwise you can see only your own sessions.

Common Tasks

Use the SHOW SESSION command to display information about the current session, a specific session, or all sessions. Information includes the user name and DB to which the user is connected, as well as the connect time, priority, and client information.

Related Commands

See “ALTER SESSION” on page B-17 to change a session’s priority or to abort a transaction in a session.

See “DROP SESSION” on page B-78 to abort and remove a session.

Usage

The following provides sample usage.

- The following command shows information about the current session:

```
SHOW SESSION;
SESSION_ID |  PID  | USERNAME | DBNAME | TYPE |    CONNECT_TIME     | SESSION_STATE_NAME
|    SQLTEXT     | PRIORITY | CLIENT_PID | CLIENT_IP
------------+-------+----------+--------+------+---------------------+----------------
     16011 | 11809 | ADMIN    | SYSTEM | sql  | 2008-03-19 12:45:16 | active
| show session; |        3 | 11808      | 127.0.0.1
(1 row)
```

- The following command shows information about all sessions:

```
SHOW SESSION ALL;
```

```
SESSION_ID |  PID  | USERNAME | DBNAME | TYPE |    CONNECT_TIME     | SESSION_STATE_NAME
|      SQLTEXT      | PRIORITY | CLIENT_PID | CLIENT_IP
------------+-------+----------+--------+------+---------------------+----------------
     16010 | 11807 | ADMIN    | SYSTEM | sql  | 2008-03-19 12:45:12 | idle
|                   |        3 | 11806      | 127.0.0.1
     16011 | 11809 | ADMIN    | SYSTEM | sql  | 2008-03-19 12:45:16 | active
| show session all; |        3 | 11808      | 127.0.0.1
(2 rows)
```

- The following command shows verbose information about the current session:

```
SHOW SESSION VERBOSE;
```

```
SESSION_ID |  PID  |    CONNECT_TIME     | PRIORITY | PRIORITY_NAME | MAX_PRIORITY |
MAX_PRIORITY_NAME | DB_OID | SESSION_USER_ID | CURRENT_USER_ID | SESSION_USER_OID |
CURRENT_USER_OID | OPERATING_USER_OID | SESSION_USERNAME | CURRENT_USERNAME |
OPERATING_USERNAME | DBNAME |      CLIENT_HOST      | CLIENT_IP | CLIENT_PID | CLIENT_
TYPE | CLIENT_TYPE_NAME | CLIENT_ID | SESSION_SOURCE | SESSION_SOURCE_NAME | SESSION_
STATE | SESSION_STATE_NAME | ROWSET_LIMIT | SESSION_TIMEOUT | QUERY_TIMEOUT | SER_
QUEUE_TIMEOUT | MAX_QUERY_RESTARTS | RESOURCE_GROUP_OID | RESOURCE_GROUP | RESOURCE_
PERCENTAGE |        SQLTEXT
------------+-------+---------------------+----------+---------------+--------------+-
      16011 | 11809 | 2008-03-19 12:45:16 |        3 | normal        | 5            |
critical        | 1     | 10411         | 10411         | 4900          | 4900
| 4900              | ADMIN           | ADMIN           | ADMIN             | SYSTEM
| localhost.localdomain | 127.0.0.1 | 11808     |          1 | sql             |
4 | f             | user              |            4 | active            | 0
| 0              |
 0           | 5               | 1               | *****             | *****
| 100                | show session verbose;
(1 row)
```

SHOW SYSTEM DEFAULT

Use the SHOW SYSTEM DEFAULT command to display the session timeout, rowset limit and query timeout values.

Synopsis

Syntax for showing system defaults:

```
SHOW SYSTEM DEFAULT
[SESSIONTIMEOUT | ROWSETLIMIT | QUERYTIMEOUT]
[DEFPRIORITY | MAXPRIORITY ]| [MATERIALZE [REFRESH] THRESHOLD]
[PASSWORDEXPIRY | PASSWORDPOLICY]
```

Inputs

The SHOW SYSTEM DEFAULT command takes the following inputs:

Table B-114: SHOW SYSTEM DEFAULT Input

Input	Description
DEFPRIORITY	Specifies the default priority for the system. The valid priorities are critical, high, normal, low.
MATERIALIZE THRESHOLD	Specifies the refresh threshold for materialized views.
MAXPRIORITY	Specifies the maximum priority for the system.
QUERYTIMEOUT	Specifies the amount of time a query can run before the system sends the administrator a message. You can specify from 1 to 35,791,394 minutes or zero for unlimited. Note that to receive a message, you must enable the RunAwayQuery event rule. For more information, see the *IBM Netezza System Administrator's Guide*.
ROWSETLIMIT	Specifies the number of rows a query can return. You can specify from 1 to 2,147,483,647 rows or zero for unlimited.
SESSIONTIMEOUT	Specifies the amount of time a session can be idle before the system terminates it. You can specify from 1 to 35,791,394 minutes or zero for unlimited.
PASSWORDEX-PIRY	Displays the system global default settings for password expiration in *n* number of days. The *n* count begins with the date of the last password change. A 0 indicates that the passwords do not expire.
PASSWORDPOL-ICY	Displays the system default settings for the password policy parameter string. For an explanation of the settings, see the "SET SYSTEM DEFAULT" on page B-120.

Outputs

The SHOW SYSTEM DEFAULT command has the following output:

Table B-115: SHOW SYSTEM DEFAULT Output

Output	Description
NOTICE OPTION <value>	The message returned if the command completes successfully.

Description

The SHOW SYSTEM DEFAULT command has the following characteristics:

Privileges Required

You must be an administrator, or an administrator must have given you the appropriate object privileges, to use this command.

Common Tasks

Use the SHOW SYSTEM DEFAULT command to display the current setting for a session timeout, rowset limit, and query timeout.

Related Commands

Use "SET" on page B-110 to set the session timeout, rowset limit, query timeout, and materialized refresh threshold.

Usage

The following provides sample usage.

- To show the query timeout, enter:

```
system(admin)=> SHOW SYSTEM DEFAULT QUERYTIMEOUT;
NOTICE:  QUERYTIMEOUT is UNLIMITED
```

TRUNCATE

Use the TRUNCATE command to empty a table.

Synopsis

Syntax for truncating an empty table:

```
TRUNCATE [ TABLE ] <name>
```

Inputs

The TRUNCATE command takes the following inputs:

Table B-116: TRUNCATE Input

Input	Description
name	Specifies the name of the table that you want to truncate.

Outputs

The TRUNCATE command has the following output:

Table B-117: TRUNCATE Output

Output	Description
TRUNCATE TABLE	Message returned if the command is successful.

Description

The TRUNCATE command has the following characteristics:

Privileges Required

You must be an administrator, or an administrator must have given you the appropriate object privileges (Truncate), to use this command.

Common Tasks

Use the TRUNCATE command to remove all rows from a table. This has the same effect as the DELETE command, but is faster than the DELETE command for large tables. In addition, the TRUNCATE command frees up all disk space allocated to a table, making the space available for use.

Note: You cannot execute the TRUNCATE command inside a transaction block (begin/commit pair).

Related Commands

See “DELETE” on page B-71.

Usage

The following provides sample usage.

- To truncate the table bigtable, enter:

```
system(admin)=> TRUNCATE TABLE bigtable;
```

UPDATE

Use the UPDATE command to replace values of columns in a table. You cannot update columns which are used as distribution keys for a table.

Synopsis

Syntax for using the UPDATE command:

```
UPDATE <table> SET col = expression [, ...]
    [ FROM <fromlist> ]
    [ WHERE <condition> ]
```

Inputs

The UPDATE command takes the following inputs:

Table B-118: UPDATE Input

Input	Description
col	Specifies the name of a column in a table.
condition	Specifies a where condition. Refer to the SELECT command for a full description of the where clause.

Table B-118: UPDATE Input

Input	Description
expression	Specifies a valid expression or value to assign to the column.
fromlist	Specifies columns from other tables for the where condition. When using the FROM clause, the inner join is implicit and the join condition must be specified in the WHERE clause. If out joins are required, note that you might require sub-selects or staging/temporary tables to specify the necessary join conditions.
table	Specifies the name of an existing table.

Outputs

The UPDATE command has the following output:

Table B-119: UPDATE Output

Output	Description
UPDATE #	Message returned if the command is successful. The symbol # represents the number of rows updated. If the system does not update any rows, it returns zero.

Description

The UPDATE command has the following characteristics:

Privileges Required

You must be an administrator, or an administrator must have given you the appropriate object privileges (Update), to use this command.

You must have the List privilege for any table whose values are mentioned in the WHERE condition.

Common Tasks

Use the UPDATE command to change values of columns you specify for all rows that satisfy a condition.

You need only specify the column(s) you want to modify.

You can use a table alias in update and delete statements. For example:

```
UPDATE tablename t1 set t1.c2='new value' where t1.c1=1;
DELETE from tablename t1 where t1.c1=2;
```

Related Commands

None.

Usage

The following provides sample usage.

- To change the word Drama to Dramatic within the column kind, enter:

```
system(admin)=> UPDATE films SET kind = 'Dramatic' WHERE kind =
'Drama';
system(admin)=> SELECT * FROM films WHERE kind = 'Dramatic' OR kind
kind = 'Drama';
code| title| did| date_prod| kind| len
----+------+----+----------+------+------
BL101| The Third Man| 101| 1949-12-23| Dramatic| 01:44
P_302| Becket| 103| 1964-02-03| Dramatic| 02:28
M_401| War and Peace| 104| 1967-02-12| Dramatic| 05:57
T_601| Yojimbo| 106| 1961-06-16| Dramatic| 01:50
```

WITH Clause

Use the WITH Clause to improve query speed for complex subqueries, without the need for conversion. This is also called subquery factoring, and is used when a subquery is executed multiple times. The 'WITH Clause' syntax allows it to be used wherever the 'SELECT' syntax was acceptable in the past (INSERT, UPDATE, DELETE, CTAS and SELECT).

Note: Recursive queries for the WITH Clause are not supported.

Note: Before downgrading to an Netezza system version that does not support the 'With Clause' syntax, all SQL objects (views and stored procedures) that use this new syntax must be removed from the system.

Synopsis

Syntax for using WITH Clause:

```
<query expression > ::=
    [ <WITH clause>] <query expression body>;
<with list> ::=
   <with list element> [ { <comma> <with list element> }... ]

<with list element> ::=
   <query name> [ <left paren> <with column list> <right paren> ]
      AS <left paren> <query expression> <right paren> [ <search or
cycle clause> ]

<with column list> ::= <column name list>

<query expression body> ::=
   <query term>
   | <query expression body> UNION [ ALL | DISTINCT ]
      [ <corresponding spec> ] <query term>
   | <query expression body> EXCEPT [ ALL | DISTINCT ]
      [ <corresponding spec> ] <query term>
```

```
<query term> ::=
    <query primary>
    | <query term> INTERSECT [ ALL | DISTINCT ]
        [ <corresponding spec> ] <query primary>

<query primary> ::=
    <simple table>
    | <left paren> <query expression body> <right paren>

<simple table> ::=
<query specification>
| <table value constructor>
| <explicit table>

<explicit table> ::= TABLE <table or query name>

<corresponding spec> ::=
    CORRESPONDING [ BY <left paren> <corresponding column list> <right
paren> ]

<corresponding column list> ::= <column name list>
```

Inputs

The WITH Clause command takes the following inputs:

Table B-120: WITH Clause Input

Input	Description
query name	The name given to the query expression. Multiple query name and expression combinations can be expressed, separated by a comma.
expression	Specifies the name of a table's column or an expression. For more information, see the SELECT command.

Outputs

The WITH Clause command has the following output:

Table B-121: WITH Clause Output

Output	Description
ROWS	Returns the complete set of rows resulting from the query.
COUNT	Returns the number of rows returned by the query.

Table B-121: WITH Clause Output

Output	Description
ERROR: Not Supported	This usage is not currently supported in the system.

Description

The WITH Clause command has the following characteristics:

Privileges Required

NA

Common Tasks

Use the WITH Clause command to run multiple subqueries in multiple clauses in a SELECT statement.

```
WITH manager (mgr_id, mgr_name, mgr_dept) AS
   (SELECT id, name, grp
    FROM emp_copy
    WHERE mgr = id AND grp != 'gone'),
employee (emp_id, emp_name, emp_mgr) AS
   (SELECT id, name, mgr_id
    FROM emp_copy JOIN manager ON grp = mgr_dept),
mgr_cnt (mgr_id, mgr_reports) AS
    (SELECT mgr, COUNT (*)
    FROM emp_copy
    WHERE mgr != id
    GROUP BY mgr)
SELECT *
FROM employee JOIN manager ON emp_mgr = mgr_id JOIN mgr_cnt
WHERE emp_id != mgr_id
ORDER BY mgr_dept;
```

Related Commands

See “SELECT” on page B-102.

Usage

The following provides sample usage.

- To use the WITH Clause when inserting:

  ```
  system(admin)=> INSERT INTO emp_copy WITH employee AS (select * from
  emp) SELECT * FROM employee;
  ```

- To use the WITH Clause when updating:

```
system(admin)=> UPDATE emp_copy SET grp = 'gone' WHERE id =
(WITH employee AS (select * from emp) SELECT id FROM employee WHERE id
= 1);
```

- To use the WITH Clause when deleting:

```
system(admin)=> DELETE FROM emp_copy WHERE id IN
(WITH employee AS (SELECT * FROM emp_copy where grp = 'gone')
SELECT id FROM employee);
```

Functions

Table B-122 describes the Netezza SQL functions and analytic functions which appear in the nzsql command help.

Table B-122: Netezza SQL Functions

Function	Description	Syntax
AVG	Returns the average of the expression.	AVG(column reference | value expression | *) over(window_spec)
COUNT	Returns the number of rows in the query.	COUNT(column reference | value expression | *) over(window_spec)
CURRENT CATALOG	Returns the current catalog name (database name)	CURRENT_CATALOG
CURRENT DATE	Returns the current date	CURRENT_DATE
CURRENT SCHEMA	Returns the current schema name (user name)	CURRENT_SCHEMA
CURRENT TIME	Returns the current local time	CURRENT_TIME
CURRENT TIMESTAMP	Returns the current date and time	CURRENT_TIMESTAMP
CURRENT USER	Returns the current user name	CURRENT_USER
DATE PART	Similar to EXTRACT, extracts sub-field from date/time value or extracts subfield from interval value	DATE_PART('text', timestamp) DATE_PART('text', interval)
DATE TRUNC	Truncates the date to a specified precision	DATE_TRUNC(text, timestamp)
DENSE RANK	Calculates the rank of a row in an ordered group of rows.	DENSE_RANK() over(window_spec)

Table B-122: Netezza SQL Functions (continued)

Function	Description	Syntax
EXTRACT	Extracts the subfield from date/time value or the subfield from interval value	EXTRACT(identifier from timestamp) EXTRACT(identifier from interval)
FIRST VALUE	Returns the first value in an ordered set of values.	FIRST_VALUE(column reference | value expression | *) over(window_spec)
LAG	Provides access to more than one row of a table at the same time without a self-join at a given offset prior to that position.	LAG(value_expression [, offset [, default]]) over(window_spec)
LAST VALUE	Returns the last value in an ordered set of values.	LAST_VALUE(column reference | value expression | *) over(window_spec)
LEAD	Provides access to more than one row of a table at the same time without a self-join at a given offset beyond that position.	LEAD(value_expression [, offset [, default]]) over(window_spec)
LOWER	Converts a string to lower case	LOWER(string)
MAX	Returns the maximum value of the expression.	MAX(column reference | value expression | *) over(window_spec)
NOW	Returns the current date and time (equivalent to current_timestamp)	NOW()
POSITION	Locates the specified substring	POSITION(substring in string)
RANK	Calculates the rank of a value in a group of values.	RANK() over(window_spec)
ROW NUMBER	Assigns a unique number to each row to which it is applied.	ROW_NUMBER() over(window_spec)
STDDEV	Returns the standard deviation of the expression, which is the square root of the variance.	STDDEV(column reference | value expression | *) over(window_spec)
STDDEV POPULATION	Returns the population standard deviation, this is the same as the square root of the var_pop function.	STDDEV_POP(column reference | value expression | *) over(window_spec)

Table B-122: Netezza SQL Functions (continued)

Function	Description	Syntax
STDDEV SAMPLE	Returns the sample standard deviation, this is the same as the square root of the var_samp function.	STDDEV_SAMP(column reference | value expression | *) over(window_spec)
SUBSTRING	Extracts a substring from a string	SUBSTRING(string [from integer] [for integer])
SUM	Returns the sum of the expression.	SUM(column reference | value expression | *) over(window_spec)
TIMEOFDAY	Returns high-precision date and time	TIMEOFDAY()
TIMESTAMP	Converts a date to a timestamp Combines date and time into a timestamp	TIMESTAMP(date) TIMESTAMP(date, time)
TO CHAR	Converts a timestamp to string converts int4/int8 to string converts real/double precision to string converts numeric to string	to_char(timestamp, text) to_char(int, text) to_char(double precision, text) to_char(numeric, text)
TO DATE	Converts a string to a date	to_date(text, text)
TO NUMBER	Converts a string to a numeric	to_number(text, text)
TO TIMESTAMP	Converts a string to a timestamp	to_timestamp(text, text)
TRIM	Removes the longest string containing only the characters (a space by default) from the beginning/end/both ends of the string	TRIM([leading | trailing | both] [characters] from string)
UPPER	Converts a string to upper case text	UPPER(string)
VARIANCE	Returns the variance of the expression.	VARIANCE(column reference | value expression | *) over(window_spec)
VARIANCE POPULATION	Returns the population variance.	VAR_POP(column reference | value expression | *) over(window_spec)
VARIANCE SAMPLE	Returns the sample variance.	VAR_SAMP(column reference | value expression | *) over(window_spec)

APPENDIX C

Join Overview

What's in this appendix

- Creating Sample Tables
- Types of Joins
- Using the Conditions on, using, and natural
- Outer Joins and the Order of Evaluation

This chapter gives a brief overview of joins, and provides simple examples to explain join concepts. For the full syntax of the select statement, see "SELECT" on page B-102.

Creating Sample Tables

This section provides the process for creating the sample tables that later sections use to describe join features. Follow the procedure given in Table C-1 to create two sample tables.

Table C-1: Creating Sample Tables to Illustrate Join Features

Step	Action
1	Enter the following command to create the sample table cows_one: `CREATE TABLE cows_one (cnumber int, cbreed varchar(20));`

2	Insert the following records into the table:
	`INSERT INTO cows_one VALUES (1,'Holstein');`
	`INSERT INTO cows_one VALUES (2,'Guernsey');`
	`INSERT INTO cows_one VALUES (3,'Angus');`

3	Do a select to see what the table looks like:
	`system(admin)=> SELECT * FROM cows_one;`
	`cnumber | cbreed`
	`--------+----------`
	`1| Holstein`
	`2| Guernsey`
	`3| Angus`
	`(3 rows)`

4	Create another table named cows_two: `CREATE TABLE cows_two (cnumber int, breeds varchar(20));`

Table C-1: Creating Sample Tables to Illustrate Join Features (continued)

Step	Action

5	Insert records into the table:
	`INSERT INTO cows_two VALUES (2,'Jersey');`
	`INSERT INTO cows_two VALUES (3,'Brown Swiss');`
	`INSERT INTO cows_two VALUES (4,'Ayrshire');`

6	Do a select to display the table:
	`system(admin)=> SELECT * FROM cows_two;`
	`cnumber | breeds`
	`--------+---------`
	`2 | Jersey`
	`3 | Brown Swiss`
	`4 | Ayrshire`
	`(3 rows)`

Types of Joins

This section describes the types of joins you can use to obtain specific information:

- Cross joins — Returns all possible combinations of rows from two tables.
- Joins or inner joins — Uses a comparison operator to match rows from two tables based on the values in common columns from each table.
- Left join/left outer join — Returns all the rows from the left table specified in the left outer join clause, not just the rows in which the columns match.
- Right join/right outer join — Returns all the rows from the right table specified in the right outer join clause, not just the rows in which the columns match.
- Full outer join — Returns all the rows in both the left and right tables.

Cross Join

A cross join returns all possible combinations of rows of two tables (also called a cartesian product). All of the columns from one table are followed by all of the columns from the other table. The result usually does not make sense as a stand-alone. You must add conditions to further define what you want to obtain from the cross joined information.

With a cross join, the number of rows in the resultant table is equal to the number of rows in the first table times the number of rows in the second table (in this case, nine).

```
SELECT * FROM cows_one CROSS JOIN cows_two;
cnumber|cbreed    |cnumber| breeds
-------+----------+-------+-------
1      | Holstein | 2     | Jersey
1      | Holstein | 3     | Brown Swiss
1      | Holstein | 4     | Ayrshire
2      | Guernsey | 2     | Jersey
2      | Guernsey | 3     | Brown Swiss
2      | Guernsey | 4     | Ayrshire
3      | Angus    | 2     | Jersey
```

```
3       | Angus     | 3      | Brown Swiss
3       | Angus     | 4      | Ayrshire
(9 rows)
```

Note: Using a cross join is equivalent to simply specifying a select from both tables:

```
SELECT * FROM cows_one, cows_two;
```

Join/Inner Join

An inner join, also known as a simple join, returns rows from joined tables that have matching rows. It does not include rows from either table that have no matching rows in the other.

```
SELECT * FROM cows_one INNER JOIN cows_two ON cows_one.cnumber = cows_
two.cnumber;
cnumber     | cbreed     | cnumber  | breeds
------------+------------+----------+---------
2           | Guernsey   | 2        | Jersey
3           | Angus      | 3        | Brown Swiss
(2 rows)
```

Note: The keyword inner is optional.

Left Outer Join/Left Join

A left outer join selects all the rows from the table on the left (cows_one in the sample), and displays the matching rows from the table on the right (cows_two). It displays an empty row for the table on the right if the table has no matching row for the table on the left.

```
SELECT * FROM cows_one LEFT OUTER JOIN cows_two ON cows_one.cnumber =
cows_two.cnumber;
cnumber    | cbreed     |cnumber  | breeds
-----------+------------+---------+---------
1          | Holstein   |         |
2          | Guernsey   | 2       | Jersey
3          | Angus      | 3       | Brown Swiss
(3 rows)
```

Note: The keyword outer is optional.

Right Outer Join/Right Join

A right join selects all the rows from the table on the right (cows_two in our sample), and displays the matching rows from the table on the left (cows_one). It displays an empty row for the table on the left if the table has no matching value for the table on the right.

```
SELECT * FROM cows_one RIGHT OUTER JOIN cows_two ON cows_one.cnumber =
cows_two.cnumber;
cnumber     | cbreed     | cnumber  | breeds
----------- +------------+ ---------+---------
2           | Guernsey   | 2        | Jersey
3           | Angus      | 3        | Brown Swiss
            |            | 4        | Ayrshire
(3 rows)
```

Note: The keyword outer is optional.

Full Outer Join

A full outer join returns all joined rows from both tables, plus one row for each unmatched left-hand row (extended with nulls on the right), plus one row for each unmatched right-hand row (extended with nulls on the left).

```
SELECT * FROM cows_one FULL OUTER JOIN cows_two ON cows_one.cnumber =
cows_two.cnumber;
cnumber     | cbreed     | cnumber   | breeds
----------- +------------+ ---------+---------
1           | Holstein   |          |
2           | Guernsey   | 2        | Jersey
3           | Angus      | 3        | Brown Swiss
            |            | 4        | Ayrshire
(4 rows)
```

Using the Conditions on, using, and natural

You can use the join conditions on, using, and natural to specify join criteria.

- The on clause is the most flexible. It can handle all join criteria, and, in certain cases, non-join criteria.
- The using and natural clauses provide convenient ways to specify joins when the join columns have the same name.

Cross Join

You cannot use an on, using, or natural condition with a cross join.

Inner Join

The following examples show inner joins.

- On join_condition

```
SELECT * FROM cows_one INNER JOIN cows_two ON cows_one.cnumber =
cows_two.cnumber;
cnumber     | cbreed     | cnumber   | breeds
------------+------------+----------+---------
2           | Guernsey   | 2        | Jersey
3           | Angus      | 3        | Brown Swiss
(2 rows)
```

Note: Note that the following statement is equivalent:

```
SELECT * FROM cows_one, cows_two WHERE cows_one.cnumber = cows_
two.cnumber;
```

- Using join_column_list

```
SELECT * FROM cows_one INNER JOIN cows_two USING (cnumber);
 cnumber     | cbreed     | breeds
-------------+---------- -+--------------
2            | Guernsey   | Jersey
3            | Angus      | Brown Swiss
(2 rows)
```

- Natural

```
SELECT * FROM cows_one NATURAL INNER JOIN cows_two;
  cnumber      | cbreed     | breeds
---------------+------------+--------------
2              | Guernsey   | Jersey
3              | Angus      | Brown Swiss
(2 rows)
```

Left Outer Join

The following examples show left outer joins.

- On join_condition

```
SELECT * FROM cows_one LEFT JOIN cows_two ON cows_one.cnumber =
cows_two.cnumber;
cnumber    |  cbreed    |cnumber    | breeds
-----------+------------+-----------+-----------
1          | Holstein   |           |
2          | Guernsey   | 2         | Jersey
3          | Angus      | 3         | Brown Swiss
(3 rows)
```

- Using join_column_list

```
SELECT * FROM cows_one LEFT JOIN cows_two USING (cnumber);
cnumber     |  cbreed     | breeds
------------+-------------+----------------------
1           | Holstein    |
2           | Guernsey    | Jersey
3           | Angus       | Brown Swiss
(3 rows)
```

- Natural

```
SELECT * FROM cows_one NATURAL LEFT JOIN cows_two;
cnumber     |  cbreed     | breeds
------------+-------------+----------------------
1           | Holstein    |
2           | Guernsey    | Jersey
3           | Angus       | Brown Swiss
(3 rows)
```

Right Outer Join

The following examples show right outer joins.

- On join_condition

```
SELECT * FROM cows_one RIGHT JOIN cows_two ON cows_one.cnumber =
cows_two.cnumber;
cnumber     |  cbreed     | cnumber    | breeds
------------+-------------+------------+-------------
2           | Guernsey    | 2          | Jersey
3           | Angus       | 3          | Brown Swiss
            |             | 4          | Ayrshire
(3 rows)
```

- Using join_column_list

```
SELECT * FROM cows_one RIGHT JOIN cows_two USING (cnumber);
cnumber    |  cbreed     | breeds
-----------+-------------+---------------
2          | Guernsey    | Jersey
3          | Angus       | Brown Swiss
4          |             | Ayrshire
(3 rows)
```

- Natural

```
SELECT * FROM cows_one NATURAL RIGHT JOIN cows_two;
cnumber    | cbreed     | breeds
-----------+------------+---------------
2          | Guernsey   | Jersey
3          | Angus      | Brown Swiss
4          |            | Ayrshire
(3 rows)
```

Full Outer Join

The following examples show full outer joins.

- On join_condition

```
SELECT * FROM cows_one FULL OUTER JOIN cows_two ON cows_one.cnumber
= cows_two.cnumber;
cnumber   |  cbreed    | cnumber   | breeds
----------+------------+-----------+-------------
1         | Holstein   |           |
2         | Guernsey   | 2         | Jersey
3         | Angus      | 3         | Brown Swiss
          |            | 4         | Ayrshire
(4 rows)
```

- Using join_column_list

```
SELECT * FROM cows_one FULL OUTER JOIN cows_two USING (cnumber);
cnumber   | cbreed     | breeds
----------+------------+------------------
1         | Holstein   |
2         | Guernsey   | Jersey
3         | Angus      | Brown Swiss
4         |            | Ayrshire
(4 rows)
```

- Natural

```
SELECT * FROM cows_one NATURAL FULL OUTER JOIN cows_two;
cnumber    |  cbreed    | breeds
-----------+------------+------------------
1          | Holstein   |
2          | Guernsey   | Jersey
3          | Angus      | Brown Swiss
4          |            | Ayrshire
(4 rows)
```

Outer Joins and the Order of Evaluation

This section provides additional notes and samples of outer joins. The samples use the left outer join for illustrating how the system evaluates joins. The rules also apply to right and full outer joins.

Left Outer Join

In all cases, the order of evaluation is as follows:

1. Evaluate all filters in the on clause.
2. Add in rows from the outer table (in this case, the left table in the left outer join) that do not match the filters.
3. Apply the where clause to the results of the outer join.

Samples

The following are left outer join examples.

```
SELECT * FROM cows_one LEFT OUTER JOIN cows_two ON cows_one.cnumber =
cows_two.cnumber AND cows_one.cnumber < 3;
```

```
cnumber     |  cbreed     |cnumber  | breeds
------------|-------------|---------+------------
1           | Holstein    |         |
2           | Guernsey    |2        | Jersey
3           | Angus       |         |
(3 rows)
```

```
SELECT * FROM cows_one LEFT OUTER JOIN cows_two ON cows_one.cnumber =
cows_two.cnumber WHERE cows_one.cnumber < 3;
```

```
cnumber |  cbreed   |cnumber  | breeds
--------|-----------|---------|-------------
2       | Guernsey |2        | Jersey
1       | Holstein |         |
(2 rows)
```

```
SELECT * FROM cows_one LEFT OUTER JOIN cows_two ON cows_one.cnumber =
cows_two.cnumber AND cows_two.cnumber < 3;
```

```
cnumber     |  cbreed     | cnumber | breeds
------------|-------------|---------+------------
1           | Holstein    |         |
2           | Guernsey    | 2       | Jersey
3           | Angus       |         |
(3 rows)
```

```
SELECT * FROM cows_one LEFT OUTER JOIN cows_two ON cows_one.cnumber =
cows_two.cnumber WHERE cows_two.cnumber < 3;
```

```
cnumber     |  cbreed     | cnumber | breeds
------------|-------------|---------+------------
2           | Guernsey    | 2       | Jersey
(1 row)
```

Notes for the on Condition

You often use the on clause with join conditions, but the on clause allows non-join predicates also. To ensure that you receive the results you want, take note of the order of evaluation of predicates.

- Place all join conditions within the on clause.
- Place all search condition predicates for the inner table within the on clause. The inner table is the right table in a left outer join. For example:

```
SELECT * FROM cows_one LEFT OUTER JOIN cows_two ON (cows_
one.cnumber = cows_two.cnumber AND cows_two.cnumber < 3);
```

- Place all search condition predicates for the OUTER table within the WHERE clause. An outer table is the left table in a left outer join. For example:

```
SELECT * FROM cows_one LEFT OUTER JOIN cows_two ON (cows_
one.cnumber = cows_two.cnumber) WHERE cows_two.cnumber < 3;
```

APPENDIX D

nzsql Command Line Options

What's in this appendix

The **nzsql** command has command line options and internal slash commands that you can use to affect input and output.

Command Line Options

Table D-1 describes the nzsql command line options.

Table D-1: nzsql Command Line Options

Option	Description
-a	Echoes all input from script
-A	Unaligned table output mode (-P format=unaligned)
-c *query*	Runs only single query (or slash command) and exits
-d *dbname*	Specifies the database name to connect to
-D *dbname*	Specifies the database name to connect to
-e	Echoes queries sent to backend
-E	Displays queries that internal commands generate
-f *filename*	Executes queries from file, then exits
-F *string*	Sets the field separator (default: "|") (-P fieldsep=)
-host *host*	Specifies the database server host (default: domain socket)
-H	HTML table output mode (-P format=html)
-l	Lists the available databases, then exits
-n	Disables readline
-o *filename*	Sends the query's output to a file name (or |pipe)

Table D-1: nzsql Command Line Options (continued)

Option	Description
-port *port*	Specifies the database server port (default: hardwired).
-P *var*[=*arg*]	Sets the printing option '*var*' to '*arg*'
-q	Runs quietly (no messages, only query output)
-R *string*	Sets the record separator (default: newline) (-P recordsep=)
-Rev	Shows version information and exits
-rev	Shows version information and exits
-s	Single-step mode (confirm each query)
-S	Single-line mode (newline terminates query)
-t	Prints rows only (-P tuples_only)
-time	Print time taken by queries
-T text	Sets the HTML table tag options (width, border) (-P tableattr=)
-u username	Specifies the database username
-U username	Specifies the database username
-v *name=val*	Sets the nzsql variable '*name*' to '*value*'
-V	Shows the version information and exits
-W password	Specifies the database user password
-pw password	Specifies the database user password
-x	Enables expanded table output (-P expanded)
-X	Does not read the startup file (~/.nzsqlrc)
-h or -?	Displays this help

Internal Slash Options

Table D-2 describes the nzsql internal slash options.

Table D-2: nzsql Internal Slash Options

Option	Description
\a	Toggles between unaligned and aligned mode
\act	Shows the current active sessions
\c[onnect] [dbname [user] [password]]	Connects to a new database

Table D-2: nzsql Internal Slash Options (continued)

Option	Description
\C *title*	Table title
\copy ...	Performs a SQL COPY with data stream to the client machine
\d *table*	Describes the table (or view, index, sequence, synonym) **Note:** For the \d options, you can add a plus sign (+) for verbose output. For example, \d+ *table* or \dpu+ *name*.
\d{tlvlilslelx}	Lists tables/views/indices/sequences/temp tables/external tables
\d{mly}	Lists materialized views/synonyms
\dS{tlvlils}	Lists system tables/views/indexes/sequences
\dM{tlvlils}	Lists system management tables/views/indexes/sequences
\dp *name*	Lists user permissions
\dpu *name*	Lists permissions granted to a user
\dpg *name*	Lists permissions granted to a group
\dgp *name*	Lists grant permissions for a user
\dgpu *name*	Lists grant permissions granted to a user
\dgpg *name*	Lists grant permissions granted to a group
\d{ulU}	Lists users/User Groups
\d{glGlGr}	Lists groups/Group Users/Resource Group Users
\da[+] *name*	Lists aggregates, + for additional fields
\dd [object]	Lists comment for table, type, function, or operator
\df[+] *name*	Lists functions, + for additional fields
\dl[+] *name*	Lists libraries, + for additional fields
\do	Lists operators
\dT	Lists data types
\e [file]	Edits the current query buffer or [file] with external editor
\echo *text*	Writes the text to stdout
\f *sep*	Changes the field separator
\g [file]	Sends the query to the backend (and the results in [file] or lpipe)

Table D-2: nzsql Internal Slash Options (continued)

Option	Description
\h [cmd]	Provides help on syntax of sql commands, type * for all commands
\H	Toggles HTML mode (currently off)
\i *file*	Reads and executes queries from <file>
\l	Lists all databases
\o [file]	Sends all query results to [file], or |pipe
\p	Shows the content of the current query buffer
\pset *opt*	Set table output <opt> = {format|border|expanded|fieldsep|nulll|recordsep|tuples_only|title|tableattr|pager}
\q	Quits the **nzsql** command
\qecho *text*	Writes text to query output stream (see \o)
\r	Resets (clears) the query buffer
\s [file]	Prints the history or saves it in [file]
\set *var value*	Sets an internal variable. Note that \set specified without any variable or argument displays a list of the current session variables and their values.
\t	Displays only rows (currently off)
\time	Prints the time taken by queries
\T *tags*	HTML table tags
\unset *var*	Unsets (deletes) the internal variable
\w *file*	Writes current query buffer to *file*
\x	Toggles expanded output (currently off)
\! [cmd]	Shell escape or command

APPENDIX E

Notices and Trademarks

What's in this appendix

- Notices
- Trademarks
- Electronic Emission Notices
- Regulatory and Compliance

This section describes some important notices, trademarks, and compliance information.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to: This information was developed for products and services offered in the U.S.A.

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785 U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing 2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE

IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Software Interoperability Coordinator, Department 49XA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. _enter the year or years_.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, ibm.com and Netezza are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at ibm.com/legal/copytrade.shtml.

Adobe is a registered trademark of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

NEC is a registered trademark of NEC Corporation.

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

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Red Hat is a trademark or registered trademark of Red Hat, Inc. in the United States and/or other countries.

D-CC, D-C++, Diab+, FastJ, pSOS+, SingleStep, Tornado, VxWorks, Wind River, and the Wind River logo are trademarks, registered trademarks, or service marks of Wind River Systems, Inc. Tornado patent pending.

APC and the APC logo are trademarks or registered trademarks of American Power Conversion Corporation.

Other company, product or service names may be trademarks or service marks of others.

Electronic Emission Notices

When you attach a monitor to the equipment, you must use the designated monitor cable and any interference suppression devices that are supplied with the monitor.

Federal Communications Commission (FCC) Statement

Note: This equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to Part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference, in which case the user will be required to correct the interference at his own expense.

Properly shielded and grounded cables and connectors must be used in order to meet FCC emission limits. IBM is not responsible for any radio or television interference caused by using other than recommended cables and connectors or by unauthorized changes or modifications to this equipment. Unauthorized changes or modifications could void the user's authority to operate the equipment.

This device complies with Part 15 of the FCC Rules. Operation is subject to the following two conditions: (1) this device may not cause harmful interference, and (2) this device must accept any interference received, including interference that might cause undesired operation.

Industry Canada Class A Emission Compliance Statement

This Class A digital apparatus complies with Canadian ICES-003.

Avis de conformité à la réglementation d'Industrie Canada

Cet appareil numérique de la classe A est conforme à la norme NMB-003 du Canada.

Australia and New Zealand Class A Statement

Attention: This is a Class A product. In a domestic environment this product may cause radio interference in which case the user may be required to take adequate measures.

European Union EMC Directive Conformance Statement

This product is in conformity with the protection requirements of EU Council Directive 2004/108/EC on the approximation of the laws of the Member States relating to electromagnetic compatibility. IBM cannot accept responsibility for any failure to satisfy the protection requirements resulting from a nonrecommended modification of the product, including the fitting of non-IBM option cards.

Attention: This is an EN 55022 Class A product. In a domestic environment this product may cause radio interference in which case the user may be required to take adequate measures.

Responsible manufacturer:

International Business Machines Corp.
New Orchard Road
Armonk, New York 10504
914-499-1900

European Community contact:

IBM Technical Regulations, Department M456
IBM-Allee 1, 71137 Ehningen, Germany
Telephone: +49 7032 15-2937
Email: tjahn@de.ibm.com

Germany Class A Statement

Deutschsprachiger EU Hinweis: Hinweis für Geräte der Klasse A EU-Richtlinie zur Elektromagnetischen Verträglichkeit

Dieses Produkt entspricht den Schutzanforderungen der EU-Richtlinie 2004/108/EG zur Angleichung der Rechtsvorschriften über die elektromagnetische Verträglichkeit in den EU-Mitgliedsstaaten und hält die Grenzwerte der EN 55022 Klasse A ein.

Um dieses sicherzustellen, sind die Geräte wie in den Handbüchern beschrieben zu installieren und zu betreiben. Des Weiteren dürfen auch nur von der IBM empfohlene Kabel angeschlossen werden. IBM übernimmt keine Verantwortung für die Einhaltung der Schutzanforderungen, wenn das Produkt ohne Zustimmung der IBM verändert bzw. wenn Erweiterungskomponenten von Fremdherstellern ohne Empfehlung der IBM gesteckt/eingebaut werden.

EN 55022 Klasse A Geräte müssen mit folgendem Warnhinweis versehen werden: "Warnung: Dieses ist eine Einrichtung der Klasse A. Diese Einrichtung kann im Wohnbereich Funk-Störungen verursachen; in diesem Fall kann vom Betreiber verlangt werden, angemessene Maßnahmen zu ergreifen und dafür aufzukommen."

Deutschland: Einhaltung des Gesetzes über die elektromagnetische Verträglichkeit von Geräten

Dieses Produkt entspricht dem "Gesetz über die elektromagnetische Verträglichkeit von Geräten (EMVG)". Dies ist die Umsetzung der EU-Richtlinie 2004/108/EG in der Bundesrepublik Deutschland.

Zulassungsbescheinigung laut dem Deutschen Gesetz über die elektromagnetische Verträglichkeit von Geräten (EMVG) (bzw. der EMC EG Richtlinie 2004/108/EG) für Geräte der Klasse A

Dieses Gerät ist berechtigt, in Übereinstimmung mit dem Deutschen EMVG das EG-Konformitätszeichen - CE - zu führen.

Verantwortlich für die Einhaltung der EMV Vorschriften ist der Hersteller:

International Business Machines Corp.
New Orchard Road
Armonk, New York 10504
914-499-1900

Der verantwortliche Ansprechpartner des Herstellers in der EU ist:

IBM Deutschland
Technical Regulations, Department M456
IBM-Allee 1, 71137 Ehningen, Germany
Telephone: +49 7032 15-2937
Email: tjahn@de.ibm.com

Generelle Informationen:

Das Gerät erfüllt die Schutzanforderungen nach EN 55024 und EN 55022 Klasse A.

Japan VCCI Class A Statement

この装置は、クラス A 情報技術装置です。この装置を家庭環境で使用すると電波妨害を引き起こすことがあります。この場合には使用者が適切な対策を講ずるよう要求されることがあります。 VCCI-A

This is a Class A product based on the standard of the Voluntary Control Council for Interference (VCCI). If this equipment is used in a domestic environment, radio interference may occur, in which case the user may be required to take corrective actions.

Japan Electronics and Information Technology Industries Association (JEITA) Statement

高調波ガイドライン適合品

Japan Electronics and Information Technology Industries Association (JEITA) Confirmed Harmonics Guidelines (products less than or equal to 20 A per phase)

Japan Electronics and Information Technology Industries Association (JEITA) Statement

高調波ガイドライン準用品

Japan Electronics and Information Technology Industries Association (JEITA) Confirmed Harmonics Guidelines (products greater than 20 A per phase)

Korea Communications Commission (KCC) Statement

이 기기는 업무용(A급)으로 전자파적합기기로서 판매자 또는 사용자는 이 점을 주의하시기 바라며, 가정외의 지역에서 사용하는 것을 목적으로 합니다.

This is electromagnetic wave compatibility equipment for business (Type A). Sellers and users need to pay attention to it. This is for any areas other than home.

Russia Electromagnetic Interference (EMI) Class A Statement

ВНИМАНИЕ! Настоящее изделие относится к классу А.
В жилых помещениях оно может создавать радиопомехи, для снижения которых необходимы дополнительные меры

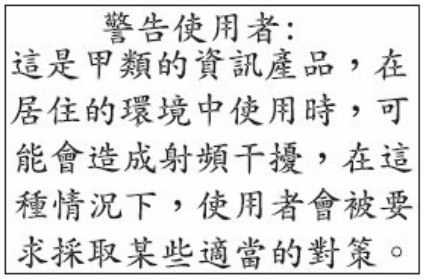
People's Republic of China Class A Electronic Emission Statement

中华人民共和国“A类”警告声明

声 明

此为A级产品，在生活环境中，该产品可能会造成无线电干扰。在这种情况下，可能需要用户对其干扰采取切实可行的措施。

Taiwan Class A Compliance Statement



Regulatory and Compliance

Regulatory Notices

Install the NPS system in a restricted-access location. Ensure that only those trained to operate or service the equipment have physical access to it. Install each AC power outlet near the NPS rack that plugs into it, and keep it freely accessible.

Provide approved circuit breakers on all power sources.

Product may be powered by redundant power sources. Disconnect ALL power sources before servicing.

High leakage current. Earth connection essential before connecting supply. Courant de fuite élevé. Raccordement à la terre indispensable avant le raccordement au réseau.

Homologation Statement

This product may not be certified in your country for connection by any means whatsoever to interfaces of public telecommunications networks.

Further certification may be required by law prior to making any such connection. Contact an IBM representative or reseller for any questions.

WEEE

Netezza Corporation is committed to meeting the requirements of the European Union (EU) Waste Electrical and Electronic Equipment (WEEE) Directive. This Directive requires producers of electrical and electronic equipment to finance the takeback, for reuse or recycling, of their products placed on the EU market after August 13, 2005.

Index

Symbols

Numerics

A

B

C

D

E

F

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X