SYBASE®

An SAP® Company

System Administration Guide: Volume 2

Sybase IQ 15.3

Contents

Audience

This guide is for developers of applications that access data in Sybase® IQ databases.

Familiarity with relational database systems and introductory user-level experience with Sybase IQ is assumed. Use this guide with other manuals in the documentation set.

Using Procedures and Batches

Create procedures and batches for use with Sybase IQ.

Procedures store procedural SQL statements in the database for use by all applications. They enhance the security, efficiency, and standardization of databases. User-defined functions are one kind of procedure that return a value to the calling environment for use in queries and other SQL statements.

For many purposes, server-side JDBC provides a more flexible way to build logic into the database than SQL stored procedures. See *SQL Anywhere 11.0.1 > SQL Anywhere Server - Programming > SQL Anywhere Data Access APIs > SQL Anywhere JDBC driver > Introduction to JDBC*.

Batches are sets of SQL statements submitted to the database server as a group. Many features available in procedures, such as control statements, are also available in batches.

Overview of Procedures

Procedures store procedural SQL statements in a database for use by all applications. They can include control statements that allow repetition (**LOOP** statements) and conditional execution (**IF** and **CASE** statements) of SQL statements.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Procedure and trigger overview.*

Note: Sybase IQ does not support triggers. Information on triggers in the SQL Anywhere documentation can be ignored.

Benefits of Procedures

Definitions for procedures appear in the database separately from any one database application. This separation provides a number of advantages.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Benefits of procedures and triggers*.

Introduction to Procedures

This section discusses the available procedures and their functions.

Two system stored procedures that are useful when working with stored procedures are **sp_iqprocedure** and **sp_iqprocparm**. The **sp_iqprocedure** stored procedure displays information about system and user-defined procedures in a database. The **sp_iqprocparm** stored procedure displays information about stored procedure parameters, including these columns:

- proc_name
- proc_owner
- parm_name
- parm_type
- parm_mode
- domain_name
- width, scale
- default

See also

- *Procedure Results* on page 11

Creating Procedures

Procedures are created using the **CREATE PROCEDURE** statement. You must have RESOURCE authority to create a procedure.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Introduction to procedures > Creating procedures*.

Sybase IQ Example

Note: For examples, use the Sybase IQ demo database `iqdemo.db`.

```
CREATE PROCEDURE new_dept(IN id INT,
             IN name CHAR(35),
                IN head_id INT)
BEGIN
        INSERT
                INTO GROUPO.departments(DepartmentID,
                DepartmentName,
                DepartmentHeadID)
                values (id, name, head_id);
END
```

Note: To create a remote procedure in IQ, you must use the `AT` *`location-string`* SQL syntax of **CREATE PROCEDURE** to create a proxy stored procedure. This capability is currently certified on only Windows and Sun Solaris. The Create Remote Procedure Wizard in Sybase Central is available only for remote servers.

Altering Procedures

You can modify an existing procedure using either Sybase Central or Interactive SQL. You must have DBA authority or be the owner of the procedure.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Introduction to procedures > Altering procedures*.

For information on altering database object properties, see *Introduction to Sybase IQ > Managing Databases > Managing Procedures*.

For information on granting or revoking permissions for procedures, see *System Administration Guide: Volume 1 > Managing User IDs and Permissions > Managing Individual User IDs and Permissions > Granting Permissions on Procedures in Interactive SQL* and *System Administration Guide: Volume 1 > Managing User IDs and Permissions > Managing Individual User IDs and Permissions > Revoking User Permissions in Interactive SQL*.

You can also modify procedures using the **ALTER PROCEDURE** statement.

Calling Procedures

CALL statements invoke procedures. Procedures can be called by an application program or by other procedures.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Introduction to procedures > Calling procedures*.

See also

- *Permissions to Execute Procedures* on page 6

Copying Procedures in Sybase Central

You can copy procedure codes from one database to another connected database.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Introduction to procedures > Copying procedures in Sybase Central*.

Deleting Procedures

Once you create a procedure, it remains in the database until someone explicitly removes it. Only the owner of the procedure or a user with DBA authority can drop the procedure from the database.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Introduction to procedures > Deleting procedures*.

Permissions to Execute Procedures

A procedure is owned by the user who created it and that user can execute it without permission.

Permission to execute the procedure can be granted to other users using the **GRANT EXECUTE** command. For example, the owner of the procedure **new_dept** allows **another_user** to execute **new_dept** with the statement:

```
GRANT EXECUTE ON new_dept TO another_user
```

The following statement revokes permission to execute the procedure:

```
REVOKE EXECUTE ON new_dept FROM another_user
```

See *System Administration Guide: Volume 1 > Managing User IDs and Permissions > Managing Individual User IDs and Permissions > Granting Permissions on Procedures in Interactive SQL*.

See also

- *Calling Procedures* on page 5

Returning Procedure Results in Parameters

Procedures return results to the calling environment.

Procedures return results in one of the following ways:

- Individual values are returned as OUT or INOUT parameters.
- Result sets can be returned.
- A single result can be returned using a **RETURN** statement.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Introduction to procedures > Returning procedure results in parameters*.

Sybase IQ example

Note: For examples, use the Sybase IQ demo database `iqdemo.db`.

```
CREATE PROCEDURE SalaryList (IN department_id INT)
RESULT ( "Employee ID" INT, "Salary" NUMERIC(20,3) )
```

```
BEGIN
            SELECT EmployeeID, Salary
            FROM Employees
            WHERE Employees.DepartmentID = department_id;
END
```

Returning Procedure Results in Result Sets

In addition to returning results to the calling environment in individual parameters, procedures can return information in result sets. A result set is typically the result of a query.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Introduction to procedures > Returning procedure results in result sets*.

Creating and Selecting from Temporary Tables

If a procedure dynamically creates and then selects the same temporary table within a stored procedure, you must use the **EXECUTE IMMEDIATE WITH RESULT SET ON** syntax to avoid `Column not found` errors.

For example:

```
CREATE PROCEDURE p1 (IN @t varchar(30))
   BEGIN
      EXECUTE IMMEDIATE
      'SELECT * INTO #resultSet FROM ' || @t;
      EXECUTE IMMEDIATE WITH RESULT SET ON
      'SELECT * FROM #resultSet';   END
```

Introduction to User-Defined Functions

User-defined functions are a class of procedures that return a single value to the calling environment. This section introduces creating, using, and dropping user-defined functions.

Creating User-Defined Functions

You use the **CREATE FUNCTION** statement to create user-defined functions. However, you must have RESOURCE authority.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Introduction to user-defined functions > Creating user-defined functions*.

For a complete description of the **CREATE FUNCTION** syntax, including performance considerations and differences between SQL Anywhere and IQ, see *Reference: Statements and Options > SQL Statements > CREATE FUNCTION Statement*.

Calling User-Defined Functions

A user-defined function can be used, subject to permissions, in any place you would use a built-in nonaggregate function.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Introduction to user-defined functions > Calling user-defined functions*.

Sybase IQ example

Note: For examples, use the Sybase IQ demo database `iqdemo.db`.

```
SELECT fullname (GivenName, SurName)FROM Employees;
```

fullname (GivenName, SurName)

Fran Whitney Matthew Cobb Philip Chin...

Dropping User-Defined Functions

Once a user-defined function is created, it remains in the database until it is explicitly removed.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Introduction to user-defined functions > Dropping user-defined functions*.

Permissions to Execute User-Defined Functions

A user-defined function is owned by the user who created it, and that user can execute it without permission.

The owner of a user-defined function can grant permissions to other users with the **GRANT EXECUTE** command.

For example, the creator of the function `fullname` allows `another_user` to use `fullname` with the statement:

```
GRANT EXECUTE ON fullname TO another_user
```

The following statement revokes permission to use the function:

```
REVOKE EXECUTE ON fullname FROM another_user
```

See *System Administration Guide: Volume 1 > Managing User IDs and Permissions > Managing Individual User IDs and Permissions > Granting Permissions on Procedures in Interactive SQL*.

Introduction to Batches

A simple batch consists of a set of SQL statements, separated by semicolons.

For example, the following statements form a batch that creates an Eastern Sales department and transfers all sales representatives from Massachusetts (MA) to that department.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Introduction to batches*.

Sybase IQ example

Note: For examples, use the Sybase IQ demo database `iqdemo.db`.

```
INSERT
INTO Departments ( DepartmentID, DepartmentName )
VALUES ( 220, 'Eastern Sales' ) ;
UPDATE Employees

SET DepartmentID = 220
WHERE DepartmentID = 200
AND state = 'GA' ;

COMMIT ;
```

Control Statements

There are a number of control statements for logical flow and decision making in the body of the procedure or in a batch.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Control statements*.

For complete descriptions of each, see the entries in *Reference: Statements and Options >SQL Statements*.

Using Compound Statements

Compound statements can be nested, and combined with other control statements to define execution flow in procedures or in batches.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Control statements > Using compound statements*.

See also

- *SQL Statements Allowed in Procedures* on page 10
- *Structure of Procedures* on page 10

- *Transactions and Savepoints in Procedures* on page 15

Declarations In Compound Statements

Local declarations in a compound statement immediately follow the **BEGIN** keyword. These local declarations exist only within the compound statement.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Control statements > Declarations in compound statements*.

Atomic Compound Statements

An *atomic* statement is a statement that is executed completely or not at all.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Control statements > Atomic compound statements*.

Structure of Procedures

The body of a procedure consists of a compound statement.

A compound statement consists of a **BEGIN** and an **END**, enclosing a set of SQL statements. Semicolons delimit each statement.

See also

- *SQL Statements Allowed in Procedures* on page 10
- *Transactions and Savepoints in Procedures* on page 15
- *Using Compound Statements* on page 9

SQL Statements Allowed in Procedures

You can use almost all SQL statements within procedures, including the following:

- **SELECT**, **UPDATE**, **DELETE**, **INSERT**, and **SET VARIABLE**
- The **CALL** statement to execute other procedures
- Control statements
- Cursor statements
- Exception handling statements
- The **EXECUTE IMMEDIATE** statement

Some SQL statements you cannot use within procedures include:

- **CONNECT** statement
- **DISCONNECT** statement

You can use **COMMIT**, **ROLLBACK**, and **SAVEPOINT** statements within procedures with certain restrictions.

See the *Usage* section for each statement in *Reference: Statements and Options > SQL Statements*.

See also

- *Structure of Procedures* on page 10
- *Transactions and Savepoints in Procedures* on page 15
- *Using Compound Statements* on page 9

Declaring Parameters for Procedures

Procedure parameters appear as a list in the **CREATE PROCEDURE** statement.

Parameter names must conform to the rules for other database identifiers such as column names. They must have valid data types , and must be prefixed with one of the keywords `IN`, `OUT` or `INOUT`.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > The structure of procedures and triggers > Declaring parameters for procedures*.

Passing Parameters to Procedures

You can take advantage of default values of stored procedure parameters with either of two forms of the **CALL** statement.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > The structure of procedures and triggers > Passing parameters to procedures*.

Passing Parameters to Functions

UDFs are not invoked with the **CALL** statement, but are used in the same manner that built-in functions are used.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > The structure of procedures and triggers > Passing parameters to functions*.

Procedure Results

Procedures can return results of either single or multiple rows of data.

Results consisting of a single row of data can be passed back as arguments to the procedure. Results consisting of multiple rows of data are passed back as result sets. Procedures can also return a single value given in the **RETURN** statement.

For simple examples of how to return results from procedures, see . For more detailed information, see the following sections.

See also

- *Introduction to Procedures* on page 4

Returning a Value Using the RETURN Statement

The **RETURN** statement returns a single integer value to the calling environment, causing an immediate exit from the procedure.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Returning results from procedures > Returning a value using the RETURN statement*.

Returning Results as Procedure Parameters

Procedures can return results to the calling environment in the procedure's parameters.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Returning results from procedures > Returning results as procedure parameters*.

Returning Result Sets from Procedures

Result sets allow a procedure to return more than one row of results to the calling environment.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Returning results from procedures > Returning result sets from procedures*.

Returning Multiple Result Sets from Procedures

A procedure can return more than one result set to the calling environment.

The method for returning multiple result sets differs for **dbisql** and **dbisqlc**.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Returning results from procedures > Returning multiple result sets from procedures*.

Returning Variable Result Sets from Procedures

The `RESULT` clause is optional in procedures. Omitting the result clause allows you to write procedures that return different result sets, with different numbers or types of columns, depending on how they are executed.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Returning results from procedures > Returning variable result sets from procedures*.

Cursors in Procedures

Cursors retrieve rows one at a time from a query or stored procedure with multiple rows in its result set.

A cursor is a handle or an identifier for the query or procedure, and for a current position within the result set.

Cursor Management Overview

Managing a cursor is similar to managing a file in a programming language.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Using cursors in procedures and triggers > Cursor management overview.*

The **sp_iqcursorinfo** stored procedure displays information about cursors currently open on the server. For more information, see *Reference: Building Blocks, Tables, and Procedures > System Procedures> sp_iqcursorinfo procedure.*

Cursor Positioning

Cursor positioning is extremely flexible.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - Programming > Introduction to Programming with SQL Anywhere > Using SQL in applications > Working with cursors > Cursor positioning.*

Note: Sybase IQ treats the `FIRST`, `LAST`, and `ABSOLUTE` options as starting from the beginning of the result set. It treats `RELATIVE` with a negative row count as starting from the current position.

Cursors and SELECT Statements in Procedures

The `TopCustomerValue` procedure uses a cursor on a **SELECT** statement and is based on the same query used in the `ListCustomerValue` procedure .

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Using cursors in procedures and triggers > Using cursors on SELECT statements in procedures.*

Errors and Warnings in Procedures

After an application program executes a SQL statement, it can examine a *return code* (or status code) for errors.

The return code indicates whether the statement executed successfully or failed and gives the reason for the failure.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Errors and warnings in procedures and triggers*.

Note: Sybase IQ does not support triggers. Information on triggers in the SQL Anywhere documentation can be ignored.

Default Error Handling in Procedures

Sybase IQ handles errors that occur during a procedure execution.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Errors and warnings in procedures and triggers > Default error handling in procedures and triggers*.

Note: Sybase IQ does not support triggers. Information on triggers in the SQL Anywhere documentation can be ignored.

Error Handling With ON EXCEPTION RESUME

The ON EXCEPTION RESUME clause is included in the **CREATE PROCEDURE** statement.

The procedure checks the statement when an error occurs. If the statement handles the error, then the procedure does not return control to the calling environment when an error occurs. Instead, it continues executing, resuming at the statement after the one causing the error.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Errors and warnings in procedures and triggers > Error handling with ON EXCEPTION RESUME*.

Default Handling of Errors and Warnings in Procedures

Errors and warnings are handled differently in procedures.

The default action for errors is to set a value for the *SQLSTATE* and *SQLCODE* variables, and return control to the calling environment in the event of an error, the default action for warnings is to set the *SQLSTATE* and *SQLCODE* values and continue execution of the procedure.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Errors and warnings in procedures and triggers > Default error handling of warnings in procedures and triggers*.

Note: Sybase IQ does not support triggers. You can ignore information about triggers in the SQL Anywhere documentation.

Using Exception Handlers in Procedures

Certain types of errors can be intercepted and handled within a procedure, rather than passing the error back to the calling environment. This is done through the use of an *exception handler*.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Errors and warnings in procedures and triggers > Using exception handlers in procedures and triggers*.

Note: Sybase IQ does not support triggers. Information on triggers in the SQL Anywhere documentation can be ignored.

Nested Compound Statements and Exception Handlers

Nested compound statements can be use to give users more control over which statements execute following an error and which do not.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Errors and warnings in procedures and triggers > Nested compound statements and exception handlers*.

Using the EXECUTE IMMEDIATE Statement In Procedures

The **EXECUTE IMMEDIATE** statement allows statements to be compiled inside procedures using a combination of literal strings (in quotes) and variables.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Using the EXECUTE IMMEDIATE statement in procedures*.

Transactions and Savepoints in Procedures

SQL statements in a procedure or trigger are part of the current transaction.

You can call several procedures within one transaction or have several transactions in one procedure.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Transactions and savepoints in procedures and triggers*.

Note: Sybase IQ does not support triggers. Information on triggers in the SQL Anywhere documentation can be ignored.

For more information, see *System Administration Guide: Volume 1 > Transactions and Versioning > Savepoints within transactions*.

See also

- *SQL Statements Allowed in Procedures* on page 10
- *Structure of Procedures* on page 10
- *Using Compound Statements* on page 9

Hiding the Contents of Procedures, Functions, and Views

In some cases, you may want to distribute an application and a database without disclosing the logic contained in the procedures, functions, triggers and views.

As an added security measure, you can obscure the contents of these objects using the SET HIDDEN clause of the **ALTER PROCEDURE**, **ALTER FUNCTION**, and **ALTER VIEW** statements.

See *"SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Hiding the contents of procedures, functions, triggers and views*.

Note: Sybase IQ does not support triggers. Information on triggers in the SQL Anywhere documentation can be ignored.

For more information, see the `ALTER FUNCTION` statement, `ALTER PROCEDURE` statement, and `ALTER VIEW` statement in *Reference: Statements and Options*.

Statements Allowed In Batches

Most SQL statements are acceptable in batches, with some exceptions.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Statements allowed in procedures, triggers, events, and batches.*

Note: Sybase IQ does not support triggers. Information on triggers in the SQL Anywhere documentation can be ignored.

Using SELECT Statements in Batches

You can include one or more **SELECT** statements in a batch.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Statements allowed in procedures, triggers, events, and batches > Using SELECT statements in batches*.

Sybase IQ Example

Note: For examples, use the Sybase IQ demo database `iqdemo.db`.

```
IF EXISTS(
            SELECT * FROM SYSTAB
                        WHERE table_name='Employees' )
THEN
            SELECT Surname AS LastName,
                        GivenName AS FirstName
            FROM Employees;
            SELECT Surname, GivenName
            FROM Customers;
            SELECT Surname, GivenName
            FROM Contacts;
ELSE
            MESSAGE 'The Employees table does not exist'
            TO CLIENT;
END IF
```

Using IQ UTILITIES to Create Your Own Stored Procedures

The system stored procedures provided in Sybase IQ are implemented in SQL, using the methods described in the rest of this chapter.

You must use the local temporary table and **IQ UTILITIES** statement in exactly the same way as system stored procedures:

Warning! Violating these rules can cause serious problems for your IQ server or database.

All SQL code for procedures is encrypted and compiled into the shared library `libiqscripts15_r.so` file on Unix and `iqscripts15.dll` file on Windows.

You can view the stored procedures code by using Sybase Central or by entering **sp_helptext 'owner.procname'** in Interactive SQL.

The syntax for **IQ UTILITIES** is:

IQ UTILITIES MAIN INTO *`local-temp-table-name arguments`*

The **IQ UTILITIES** command is only documented in *Reference: Statements and Options* to the IQ monitor, because of the strict requirements for its use and the risk to system operations if it is used incorrectly.

You may want to create your own variants of some of these procedures. Among the ways you might do this are:

1. Create a procedure that calls a system stored procedure.
2. Create a procedure that is independent of the system stored procedures but performs a similar function.
3. Create a procedure that uses the same structure as the system stored procedures but provides additional functionality. For example, you might want to display procedure results in graphical form in a front-end tool or browser rather than as text.
4. If you choose the second or third option, you need to understand the **IQ UTILITIES** statement and the strict requirements for using it.

How IQ Uses the IQ UTILITIES Command

IQ UTILITIES is the underlying statement that executes whenever you run most IQ system procedures. In most cases, users are unaware that **IQ UTILITIES** is executing. The only time **IQ UTILITIES** is issued directly by users is to run the IQ buffer cache monitor.

IQ UTILITIES provides a systematic way to collect and report on information maintained in the IQ system tables. There is no general user interface; you can only use **IQ UTILITIES** in the ways that existing system procedures do.

System procedures declare local temporary tables in which they store information. They execute **IQ UTILITIES** to get the information from the system tables and store this information in the local temporary table. The system procedures may simply report the information from the local temporary table or perform additional processing.

In some system procedures, the **IQ UTILITIES** statement includes a predefined number as one of its arguments. This number performs a specific function, for example, deriving a value from information in the system tables. See for a list of the numbers used as **IQ UTILITIES** arguments.

Choosing Procedures to Call

You can safely use **IQ UTILITIES** to create your own versions of documented system procedures that report on information in the database.

For example, **sp_iqspaceused** displays information about used and available space available in the IQ main and IQ temporary stores. Check the owner of the procedure you create from a system stored procedure to be sure your version of the procedure has the correct owner.

Do not create your own versions of system procedures that control IQ operations. Modifying procedures that control IQ operations can lead to serious problems.

Numbers Used by IQ UTILITIES

The following table lists the numbers used as arguments in the **IQ UTILITIES** command and the system procedure where each number is used.

For information on the function of these procedures, see *Reference: Building Blocks, Tables, and Procedures > System Procedures*.

Table 1. IQ UTILITIES values used in system procedures

Number	Procedure	Comments
10000	sp_iqtransaction	
20000	sp_iqconnection and sp_iqmpxcountdbremote	
30000	sp_iqspaceused	
40000	sp_iqspaceinfo	
50000	sp_iqlocks	
60000	sp_iqmpxversionfetch	Do Not Use
70000	sp_iqmpxdumptlvlog	
80000	sp_iqcontext	
100000	sp_iqindexfragmentation	
110000	sp_iqrowdensity	

Procedure Testing

Always test your procedures in a development environment first. Test procedures before you run them in a production environment to help maintain the stability of your IQ server and database.

Using OLAP

OLAP (online analytical processing) is an efficient method of data analysis of information stored in a relational database.

Using OLAP you can analyze data on different dimensions, acquire result sets with subtotaled rows, and organize data into multidimensional cubes, all in a single SQL query. You can also use filters to drill down into the data, returning result sets quickly. This chapter describes the SQL/OLAP functionality that Sybase IQ supports.

Note: The tables shown in OLAP examples are available in the `iqdemo` database.

About OLAP

The analytic functions, which offer the ability to perform complex data analysis within a single SQL statement, are facilitated by a category of software technology named online analytical processing (OLAP). Its functions are shown in the following list:

- **GROUP BY** clause extensions – **CUBE** and **ROLLUP**
- Analytical functions:
  - Simple aggregates – **AVG**, **COUNT**, **MAX**, **MIN**, and **SUM**, **STDDEV** and **VARIANCE**

    Note: You can use simple aggregate functions, except **Grouping()**, with an OLAP windowed function.
  - Window functions:
    - Windowing aggregates – **AVG**, **COUNT**, **MAX**, **MIN**, and **SUM**
    - Ranking functions – **RANK**, **DENSE_RANK**, **PERCENT_RANK**, and **NTILE**
    - Statistical functions – **STDDEV**, **STDDEV_SAMP**, **STDDEV_POP**, **VARIANCE**, **VAR_POP**, **VAR_SAMP**, **REGR_AVGX**, **REGR_AVGY**, **REGR_COUNT**, **REGR_INTERCEPT**, **REGR_R2**, **REGR_SLOPE**, **REGR_SXX**, **REGR_SXY**, **REGR_SYY**, **CORR**, **COVAR_POP**, **COVAR_SAMP**, **CUME_DIST**, **EXP_WEIGHTED_AVG**, and **WEIGHTED_AVG**.
    - Distribution functions – **PERCENTILE_CONT** and **PERCENTILE_DISC**
  - Numeric functions – **WIDTH_BUCKET**, **CEIL**, and **LN**, **EXP**, **POWER**, **SQRT**, and **FLOOR**

Extensions to the ANSI SQL standard to include complex data analysis were introduced as an amendment to the 1999 SQL standard. Sybase IQ added portions of these SQL enhancements provides additional comprehensive support for the extensions.

Some database products provide a separate OLAP module that requires you to move data from the database into the OLAP module before analyzing it. By contrast, Sybase IQ builds OLAP features into the database itself, making deployment and integration with other database features, such as stored procedures, easy and seamless.

OLAP Benefits

OLAP functions, when combined with the **GROUPING**, **CUBE**, and **ROLLUP** extensions, provide two primary benefits.

First, they let you perform multidimensional data analysis, data mining, time series analyses, trend analysis, cost allocations, goal seeking, ad hoc multidimensional structural changes, nonprocedural modeling, and exception alerting, often with a single SQL statement. Second, the window and reporting aggregate functions use a relational operator, called a *window* that can be executed more efficiently than semantically equivalent queries that use self-joins or correlated subqueries. The result sets you obtain using OLAP can have subtotal rows and can be organized into multidimensional cubes. See .

Moving averages and moving sums can be calculated over various intervals; aggregations and ranks can be reset as selected column values change; and complex ratios can be expressed in simple terms. Within the scope of a single query expression, you can define several different OLAP functions, each with its own partitioning rules.

See also

- *Distribution Functions* on page 59
- *OLAP Evaluation* on page 22
- *Ranking Functions* on page 48
- *Statistical Aggregate Functions* on page 54
- *Windowing* on page 37
- *Windowing Aggregate Functions* on page 52
- *BNF Grammar for OLAP Functions* on page 73

OLAP Evaluation

OLAP evaluation can be conceptualized as several phases of query execution that contribute to the final result.

You can identify OLAP phases of execution by the relevant clause in the query. For example, if a SQL query specification contains window functions, the **WHERE**, **JOIN**, **GROUP BY**, and **HAVING** clauses are processed first. Partitions are created after the groups defined in the GROUP BY clause and before the evaluation of the final **SELECT** list in the query's **ORDER BY** clause.

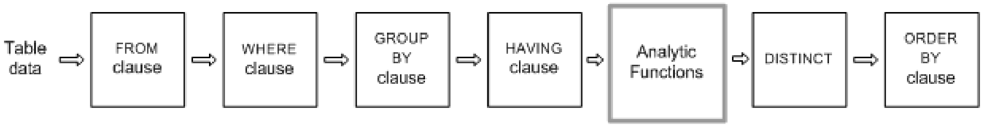
For the purpose of grouping, all NULL values are considered to be in the same group, even though NULL values are not equal to one another.

The **HAVING** clause acts as a filter, much like the **WHERE** clause, on the results of the GROUP BY clause.

Consider the semantics of a simple query specification involving the SQL statements and clauses, **SELECT**, **FROM**, **WHERE**, **GROUP BY**, and **HAVING** from the ANSI SQL standard:

1. The query produces a set of rows that satisfy the table expressions present in the **FROM** clause.
2. Predicates from the **WHERE** clause are applied to rows from the table. Rows that fail to satisfy the **WHERE** clause conditions (do not equal true) are rejected.
3. Except for aggregate functions, expressions from the **SELECT** list and in the list and GROUP BY clause are evaluated for every remaining row.
4. The resulting rows are grouped together based on distinct values of the expressions in the GROUP BY clause, treating NULL as a special value in each domain. The expressions in the GROUP BY clause serve as partition keys if a **PARTITION BY** clause is present.
5. For each partition, the aggregate functions present in the **SELECT** list or **HAVING** clause are evaluated. Once aggregated, individual table rows are no longer present in the intermediate result set. The new result set consists of the **GROUP BY** expressions and the values of the aggregate functions computed for each partition.
6. Conditions from the **HAVING** clause are applied to result groups. Groups are eliminated that do not satisfy the **HAVING** clause.
7. Results are partitioned on boundaries defined in the **PARTITION BY** clause. OLAP windows functions (rank and aggregates) are computed for result windows.

Figure 1: SQL processing for OLAP



See . See also .

See also

- *Distribution Functions* on page 59
- *OLAP Benefits* on page 22
- *Ranking Functions* on page 48
- *Statistical Aggregate Functions* on page 54
- *Windowing* on page 37
- *Windowing Aggregate Functions* on page 52
- *BNF Grammar for OLAP Functions* on page 73

GROUP BY Clause Extensions

Extensions to the **GROUP BY** clause let application developers write complex SQL statements that:

- Partition the input rows in multiple dimensions and combine multiple subsets of result groups.
- Create a “data cube,” providing a sparse, multi dimensional result set for data mining analyses.
- Create a result set that includes the original groups, and optionally includes a subtotal and grand-total row.

OLAP Grouping() operations, such as **ROLLUP** and **CUBE**, can be conceptualized as prefixes and subtotal rows.

Prefixes

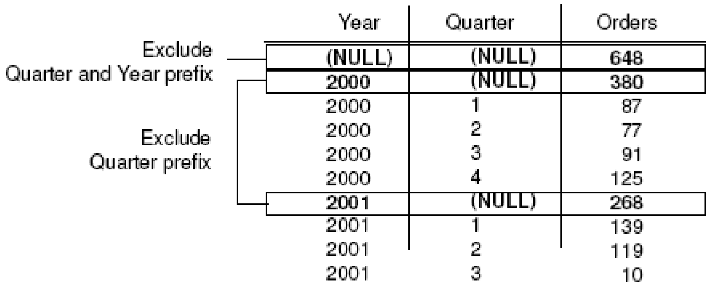
A list of *prefixes* is constructed for any query that contains a **GROUP BY** clause. A prefix is a subset of the items in the **GROUP BY** clause and is constructed by excluding one or more of the rightmost items from those in the query’s **GROUP BY** clause. The remaining columns are called the *prefix columns*.

ROLLUP example 1—In the following **ROLLUP** example query, the **GROUP BY** list includes two variables, *Year* and *Quarter*:

```
SELECT year (OrderDate) AS Year, quarter(OrderDate)
  AS Quarter, COUNT(*) Orders
FROM SalesOrders
GROUP BY ROLLUP(Year, Quarter)
ORDER BY Year, Quarter
```

The query’s two prefixes are:

- Exclude Quarter – the set of prefix columns contains the single column `Year`.
- Exclude both Quarter and Year – there are no prefix columns.


Exclude
Quarter and Year prefix
Exclude
Quarter prefix

Year	Quarter	Orders
(NULL)	(NULL)	648
2000	(NULL)	380
2000	1	87
2000	2	77
2000	3	91
2000	4	125
2001	(NULL)	268
2001	1	139
2001	2	119
2001	3	10

Note: The **GROUP BY** list contains the same number of prefixes as items.

Group by ROLLUP and CUBE

ROLLUP and **CUBE** are syntactic shortcuts that specify common grouping prefixes.

Group by ROLLUP

The **ROLLUP** operator requires an ordered list of grouping expressions to be supplied as arguments.

ROLLUP syntax.

```
SELECT … [ GROUPING (column-name) … ] …
GROUP BY [ expression [, …]
| ROLLUP ( expression [, …] ) ]
```

GROUPING takes a column name as a parameter and returns a Boolean value as listed in the following table:

Table 2. Values returned by GROUPING with the ROLLUP operator

If the value of the result is	GROUPING returns
NULL created by a **ROLLUP** operation	1 (TRUE)
NULL indicating the row is a subtotal	1 (TRUE)
Not created by a **ROLLUP** operation	0 (FALSE)
A stored NULL	0 (FALSE)

ROLLUP first calculates the standard aggregate values specified in the **GROUP BY** clause. Then **ROLLUP** moves from right to left through the list of grouping columns and creates progressively higher-level subtotals. A grand total is created at the end. If *n* is the number of grouping columns, then **ROLLUP** creates *n*+1 levels of subtotals.

This SQL syntax...	Defines the following sets...
`GROUP BY ROLLUP (A, B, C);`	(A, B, C) (A, B) (A) ( )

ROLLUP and subtotal rows

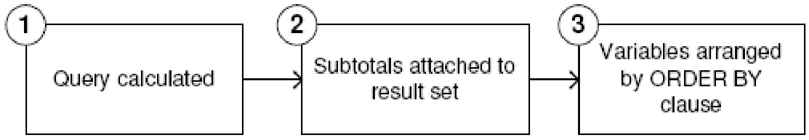
ROLLUP is equivalent to a **UNION** of a set of **GROUP BY** queries. The result sets of the following queries are identical. The result set of **GROUP BY** (A, B) consists of subtotals over all those rows in which A and B are held constant. To make a union possible, column C is assigned NULL.

This ROLLUP query...	Is equivalent to this query without ROLLUP...
`select year(order-date) as year, quar-ter(orderdate) as Quarter, count(*) Or-dersfrom SalesOr-dersgroup by Rollup (year, quarter)order by year, quarter`	`Select null,null, count(*) Orders from SalesOrdersunion allSELECT year(orderdate) AS YEAR, NULL, count(*) Orders from SalesOr-dersGROUP BY year(orderdate) union allSELECT year(orderdate) as YEAR, quarter(orderdate) as QUATER, count(*) Orders from SalesOr-dersGROUP BY year(orderdate), quar-ter(orderdate)`

Subtotal rows can help you analyze data, especially if there are large amounts of data, different dimensions to the data, data contained in different tables, or even different databases altogether. For example, a sales manager might find reports on sales figures broken down by sales representative, region, and quarter to be useful in understanding patterns in sales. Subtotals for the data give the sales manager a picture of overall sales from different perspectives. Analyzing this data is easier when summary information is provided based on the criteria that the sales manager wants to compare.

With OLAP, the procedure for analyzing and computing row and column subtotals is invisible to users.

Figure 2: Subtotals


1
Query calculated
2
Subtotals attached to result set
3
Variables arranged by ORDER BY clause

1. This step yields an intermediate result set that has not yet considered the **ROLLUP**.
2. Subtotals are evaluated and attached to the result set.
3. The rows are arranged according to the **ORDER BY** clause in the query.

NULL values and subtotal rows

When rows in the input to a **GROUP BY** operation contain NULL, there is the possibility of confusion between subtotal rows added by the **ROLLUP** or **CUBE** operations and rows that contain NULL values that are part of the original input data.

The Grouping() function distinguishes subtotal rows from others by taking a column in the **GROUP BY** list as its argument, and returning 1 if the column is NULL because the row is a subtotal row, and 0 otherwise.

The following example includes Grouping() columns in the result set. Rows are highlighted that contain NULL as a result of the input data, not because they are subtotal rows. The

Grouping() columns are highlighted. The query is an outer join between the `Employees` table and the `SalesOrders` table. The query selects female employees who live in Texas, New York, or California. NULL appears in the columns corresponding to those female employees who are not sales representatives (and therefore have no sales).

Note: For examples, use the Sybase IQ demo database `iqdemo.db`.

```
SELECT Employees.EmployeeID as EMP, year(OrderDate) as
  YEAR, count(*) as ORDERS, grouping(EMP) as
  GE, grouping(YEAR) as GY
  FROM Employees LEFT OUTER JOIN SalesOrders on
  Employees.EmployeeID = SalesOrders.SalesRepresentative
  WHERE Employees.Sex IN ('F') AND Employees.State
  IN ('TX', 'CA', 'NY')
GROUP BY ROLLUP (YEAR, EMP)
ORDER BY YEAR, EMP
```

The preceding query returns:

```
EMP         YEAR      ORDERS     GE      GY
------      ----      ------     --      --
NULL        NULL           5      1       0
NULL        NULL         169      1       1
 102        NULL           1      0       0
 309        NULL           1      0       0
1062        NULL           1      0       0
1090        NULL           1      0       0
1507        NULL           1      0       0
NULL        2000          98      1       0
 667        2000          34      0       0
 949        2000          31      0       0
1142        2000          33      0       0
NULL        2001          66      1       0
 667        2001          20      0       0
949         2001          22      0       0
1142        2001          24      0       0
```

For each prefix, a *subtotal row* is constructed that corresponds to all rows in which the prefix columns have the same value.

To demonstrate **ROLLUP** results, examine the example query again:

```
SELECT year (OrderDate) AS Year, quarter
  (OrderDate) AS Quarter, COUNT (*) Orders
FROM SalesOrders
  GROUP BY ROLLUP (Year, Quarter)
  ORDER BY Year, Quarter
```

In this query, the prefix containing the `Year` column leads to a summary row for Year=2000 and a summary row for Year=2001. A single summary row for the prefix has no columns, which is a subtotal over all rows in the intermediate result set.

The value of each column in a subtotal row is as follows:

- Column included in the prefix – the value of the column. For example, in the preceding query, the value of the `Year` column for the subtotal over rows with Year=2000 is 2000.
- Column excluded from the prefix – NULL. For example, the `Quarter` column has a value of NULL for the subtotal rows generated by the prefix consisting of the Year column.
- Aggregate function – an aggregate over the values of the excluded columns.

  Subtotal values are computed over the rows in the underlying data, not over the aggregated rows. In many cases, such as **SUM** or **COUNT**, the result is the same, but the distinction is important in the case of statistical functions such as **AVG**, **STDDEV**, and **VARIANCE**, for which the result differs.

Restrictions on the **ROLLUP** operator are:

- The **ROLLUP** operator supports all of the aggregate functions available to the **GROUP BY** clause except **COUNT DISTINCT** and **SUM DISTINCT**.
- **ROLLUP** can only be used in the **SELECT** statement; you cannot use **ROLLUP** in a subquery.
- A grouping specification that combines multiple **ROLLUP**, **CUBE**, and **GROUP BY** columns in the same **GROUP BY** clause is not currently supported.
- Constant expressions as **GROUP BY** keys are not supported.

For the general format of an expression, see *Reference: Building Blocks, Tables, and Procedures > Expressions* and *Reference: Building Blocks, Tables, and Procedures > SQL Language Elements*.

ROLLUP example 2—The following example illustrates the use of **ROLLUP** and **GROUPING** and displays a set of mask columns created by **GROUPING**. The digits 0 and 1 displayed in columns S, N, and C are the values returned by **GROUPING** to represent the value of the **ROLLUP** result. A program can analyze the results of this query by using a mask of “011” to identify subtotal rows and “111” to identify the row of overall totals.

```
SELECT size, name, color, SUM(quantity),
  GROUPING(size) AS S,
  GROUPING(name) AS N,
  GROUPING(color) AS C
FROM Products
GROUP BY ROLLUP(size, name, color) HAVING (S=1 or N=1 or C=1)
ORDER BY size, name, color;
```

The preceding query returns:

```
size               name           color     SUM  S  N  C
----               -----          ------     ---  -  -  -
(NULL)             (NULL)          (NULL)    496  1  1  1
Large              (NULL)          (NULL)     71  0  1  1
Large              Sweatshirt      (NULL)     71  0  0  1
Medium             (NULL)          (NULL)    134  0  1  1
Medium             Shorts          (NULL)     80  0  0  1
Medium             Tee Shirt       (NULL)     54  0  0  1
One size fits all  (NULL)          (NULL)    263  0  1  1
One size fits all  Baseball Cap    (NULL)    124  0  0  1
One size fits all  Tee Shirt       (NULL)     75  0  0  1
One size fits all  Visor           (NULL)     64  0  0  1
Small              (NULL)          (NULL)     28  0  1  1
Small              Tee Shirt       (NULL)     28  0  1  1
```

Note: In the Rollup Example 2 results, the SUM column displays as *SUM(products.quantity)*.

ROLLUP example 3—The following example illustrates the use of **GROUPING** to distinguish stored NULL values and "NULL" values created by the **ROLLUP** operation. Stored NULL values are then displayed as [NULL] in column `prod_id`, and "NULL" values created by **ROLLUP** are replaced with ALL in column `PROD_IDS`, as specified in the query.

```
SELECT year(ShipDate) AS Year,
                ProductID, SUM(quantity)AS OSum,
CASE
        WHEN GROUPING(Year) = 1
        THEN 'ALL'
        ELSE
        CAST(Year AS char(8))
END,
CASE
        WHEN GROUPING(ProductID) = 1
        THEN 'ALL'
        ELSE
        CAST(ProductIDas char(8))
END
FROM SalesOrderItems
GROUP BY ROLLUP(Year, ProductID) HAVING OSum > 36
ORDER BY Year, ProductID;
```

The preceding query returns:

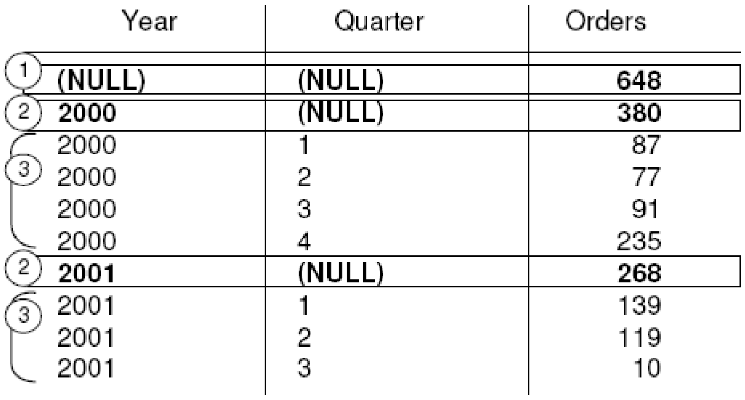
```
Year  ProductID  OSum  ...(Year)...  ...(ProductID)...
---------     -------   ---   ----------   --------
NULL           NULL   28359   ALL          ALL
2000           NULL   17642   2000         ALL
2000           300     1476   2000         300
2000           301     1440   2000         301
2000           302     1152   2000         302
2000           400     1946   2000         400
2000           401     1596   2000         401
2000           500     1704   2000         500
2000           501     1572   2000         501
2000           600     2124   2000         600
2000           601     1932   2000         601
2000           700     2700   2000         700
2001           NULL   10717   2001         ALL
2001           300      888   2001         300
2001           301      948   2001         301
2001           302      996   2001         302
2001           400     1332   2001         400
2001           401     1105   2001         401
2001           500      948   2001         500
2001           501      936   2001         501
2001           600      936   2001         600
```

```
2001            601       792   2001          601
2001            700      1836   2001          700
```

ROLLUP example 4—The next example query returns data that summarizes the number of sales orders by year and quarter.

```
SELECT year (OrderDate) AS Year,
quarter(OrderDate) AS Quarter, COUNT (*) Orders
FROM SalesOrders
GROUP BY ROLLUP (Year, Quarter)
ORDER BY Year, Quarter
```

The following figure illustrates the query results with subtotal rows highlighted in the result set. Each subtotal row contains a NULL value in the column or columns over which the subtotal is computed.



	Year	Quarter	Orders
1	**(NULL)**	**(NULL)**	**648**
2	**2000**	**(NULL)**	**380**
3	2000	1	87
	2000	2	77
	2000	3	91
	2000	4	235
2	**2001**	**(NULL)**	**268**
3	2001	1	139
	2001	2	119
	2001	3	10

Row [1] represents the total number of orders across both years (2000, 2001) and all quarters. This row contains NULL in both the `Year` and `Quarter` columns and is the row where all columns were excluded from the prefix.

Note: Every **ROLLUP** operation returns a result set with one row where NULL appears in each column except for the aggregate column. This row represents the summary of each column to the aggregate function. For example, if SUM were the aggregate function in question, this row would represent the grand total of all values.

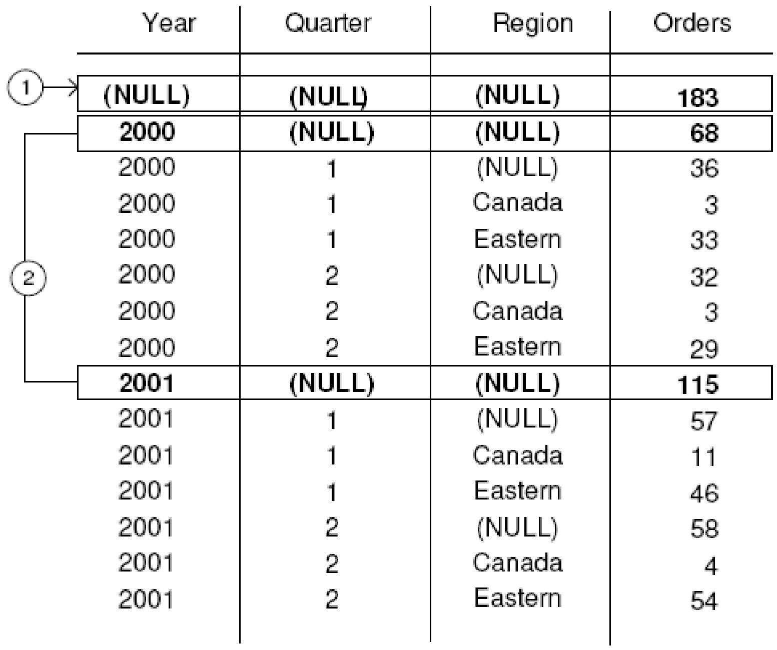
Row [2] represent the total number of orders in the years 2000 and 2001, respectively. Both rows contain NULL in the `Quarter` column because the values in that column are rolled up to give a subtotal for `Year`. The number of rows like this in your result set depends on the number of variables that appear in your **ROLLUP** query.

The remaining rows marked [3] provide summary information by giving the total number of orders for each quarter in both years.

ROLLUP example 5—This example of the **ROLLUP** operation returns a slightly more complicated result set, which summarizes the number of sales orders by year, quarter, and region. In this example, only the first and second quarters and two selected regions (Canada and the Eastern region) are examined.

```
SELECT year(OrderDate) AS Year, quarter(OrderDate)AS Quarter,
region, COUNT(*) AS OrdersFROM SalesOrders WHERE region IN
('Canada','Eastern') AND quarter IN (1, 2)GROUP BY ROLLUP (Year,
Quarter, Region)ORDER BY Year, Quarter, Region
```

The following figure illustrates the result set from the above query. Each subtotal row contains a NULL in the column or columns over which the subtotal is computed.

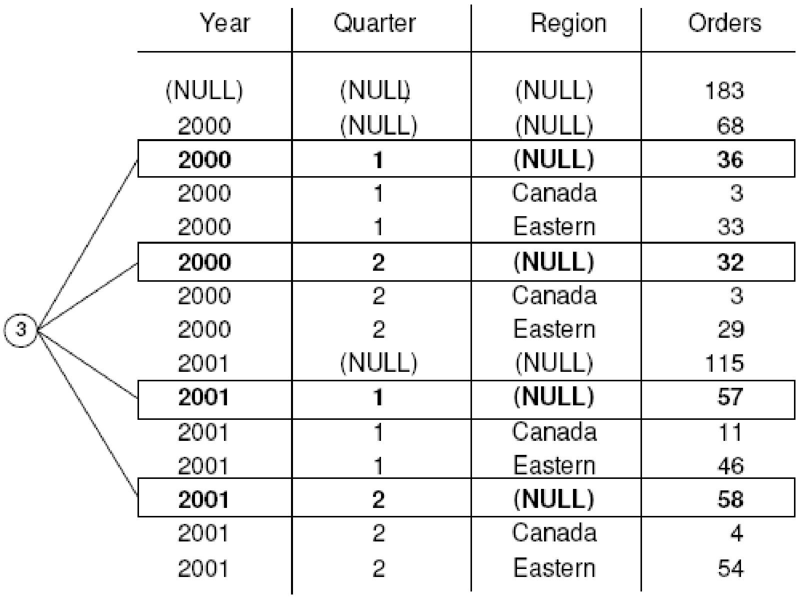


Year	Quarter	Region	Orders
(NULL)	(NULL)	(NULL)	183
2000	(NULL)	(NULL)	68
2000	1	(NULL)	36
2000	1	Canada	3
2000	1	Eastern	33
2000	2	(NULL)	32
2000	2	Canada	3
2000	2	Eastern	29
2001	(NULL)	(NULL)	115
2001	1	(NULL)	57
2001	1	Canada	11
2001	1	Eastern	46
2001	2	(NULL)	58
2001	2	Canada	4
2001	2	Eastern	54

Row [1] is an aggregate over all rows and contains NULL in the `Year`, `Quarter`, and `Region` columns. The value in the `Orders` column of this row represents the total number of orders in Canada and the Eastern region in quarters 1 and 2 in the years 2000 and 2001.

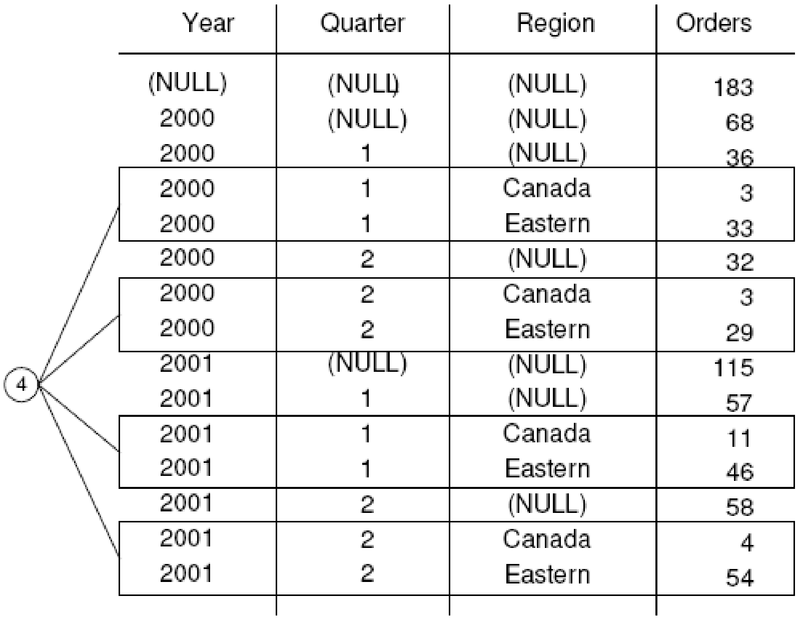
The rows marked [2] represent the total number of sales orders in each year (2000) and (2001) in quarters 1 and 2 in Canada and the Eastern region. The values of these rows [2] are equal to the grand total represented in row [1].

The rows marked [3] provide data about the total number of orders for the given year and quarter by region.



Year	Quarter	Region	Orders
(NULL)	(NULL)	(NULL)	183
2000	(NULL)	(NULL)	68
2000	**1**	**(NULL)**	**36**
2000	1	Canada	3
2000	1	Eastern	33
2000	**2**	**(NULL)**	**32**
2000	2	Canada	3
2000	2	Eastern	29
2001	(NULL)	(NULL)	115
2001	**1**	**(NULL)**	**57**
2001	1	Canada	11
2001	1	Eastern	46
2001	**2**	**(NULL)**	**58**
2001	2	Canada	4
2001	2	Eastern	54

The rows marked [4] provide data about the total number of orders for each year, each quarter, and each region in the result set.



Year	Quarter	Region	Orders
(NULL)	(NULL)	(NULL)	183
2000	(NULL)	(NULL)	68
2000	1	(NULL)	36
2000	1	Canada	3
2000	1	Eastern	33
2000	2	(NULL)	32
2000	2	Canada	3
2000	2	Eastern	29
2001	(NULL)	(NULL)	115
2001	1	(NULL)	57
2001	1	Canada	11
2001	1	Eastern	46
2001	2	(NULL)	58
2001	2	Canada	4
2001	2	Eastern	54

Group by CUBE

The **CUBE** operator in the **GROUP BY** clause analyzes data by forming the data into groups in more than one dimension (grouping expression).

CUBE requires an ordered list of dimensions as arguments and enables the **SELECT** statement to calculate subtotals for all possible combinations of the group of dimensions that you specify in the query and generates a result set that shows aggregates for all combinations of values in selected columns.

CUBE syntax:

```
SELECT … [ GROUPING (column-name) … ] …
GROUP BY [ expression [,…]
| CUBE ( expression [,…] ) ]
```

GROUPING takes a column name as a parameter, and returns a Boolean value as listed in the following table:

Table 3. Values returned by GROUPING with the CUBE operator

If the value of the result is	GROUPING returns
NULL created by a **CUBE** operation	1 (TRUE)
NULL indicating the row is a subtotal	1 (TRUE)
Not created by a **CUBE** operation	0 (FALSE)
A stored NULL	0 (FALSE)

CUBE is particularly useful when your dimensions are not a part of the same hierarchy.

This SQL syntax...	Defines the following sets...
`GROUP BY CUBE (A, B, C);`	(A, B, C) (A, B) (A, C) (A) (B, C) (B) (C) ( )

Restrictions on the **CUBE** operator are:

- The **CUBE** operator supports all of the aggregate functions available to the **GROUP BY** clause, but **CUBE** is currently not supported with **COUNT DISTINCT** or **SUM DISTINCT**.

- **CUBE** is currently not supported with the inverse distribution analytical functions, **PERCENTILE_CONT** and **PERCENTILE_DISC**.
- **CUBE** can only be used in the **SELECT** statement; you cannot use **CUBE** in a **SELECT** subquery.
- A **GROUPING** specification that combines **ROLLUP**, **CUBE**, and **GROUP BY** columns in the same GROUP BY clause is not currently supported.
- Constant expressions as **GROUP BY** keys are not supported.

Note: CUBE performance diminishes if the size of the cube exceeds the size of the temp cache.

GROUPING can be used with the **CUBE** operator to distinguish between stored NULL values and NULL values in query results created by **CUBE**.

See the examples in the description of the **ROLLUP** operator for illustrations of the use of the **GROUPING** function to interpret results.

All **CUBE** operations return result sets with at least one row where NULL appears in each column except for the aggregate columns. This row represents the summary of each column to the aggregate function.

CUBE example 1—The following queries use data from a census, including the state (geographic location), gender, education level, and income of people. The first query contains a **GROUP BY** clause that organizes the results of the query into groups of rows, according to the values of the columns `state`, `gender`, and `education` in the table `census` and computes the average income and the total counts of each group. This query uses only the **GROUP BY** clause without the **CUBE** operator to group the rows.

```
SELECT State, Sex as gender, DepartmentID,
COUNT(*),CAST(ROUND(AVG(Salary),2) AS NUMERIC(18,2))AS AVERAGEFROM
employees WHERE state IN ('MA' , 'CA')GROUP BY State, Sex,
DepartmentIDORDER BY 1,2;
```

The results from the above query:

```
state  gender  DepartmentID  COUNT()     AVERAGE
-----  ------  -------   --------    --------
CA       F       200       2         58650.00
CA       M       200           1         39300.00
```

Use the **CUBE** extension of the **GROUP BY** clause, if you want to compute the average income in the entire census of state, gender, and education and compute the average income in all possible combinations of the columns `state`, `gender`, and `education`, while making only a single pass through the census data. For example, use the **CUBE** operator if you want to compute the average income of all females in all states, or compute the average income of all people in the census according to their education and geographic location.

When **CUBE** calculates a group, a NULL value is generated for the columns whose group is calculated. The **GROUPING** function must be used to distinguish whether a NULL is a NULL stored in the database or a NULL resulting from **CUBE**. The **GROUPING** function returns 1 if the designated column has been merged to a higher level group.

CUBE example 2—The following query illustrates the use of the **GROUPING** function with **GROUP BY CUBE**.

```
SELECT case grouping(State) WHEN 1 THEN 'ALL' ELSE StateEND AS
c_state, case grouping(sex) WHEN 1 THEN 'ALL'ELSE Sex end AS
c_gender, case grouping(DepartmentID)WHEN 1 THEN 'ALL' ELSE
cast(DepartmentID as char(4)) endAS c_dept, COUNT(*),
CAST(ROUND(AVG(salary),2) ASNUMERIC(18,2))AS AVERAGEFROM employees
WHERE state IN ('MA' , 'CA')GROUP BY CUBE(state, sex,
DepartmentID)ORDER BY 1,2,3;
```

The results of this query are shown below. The NULLs generated by **CUBE** to indicate a subtotal row are replaced with ALL in the subtotal rows, as specified in the query.

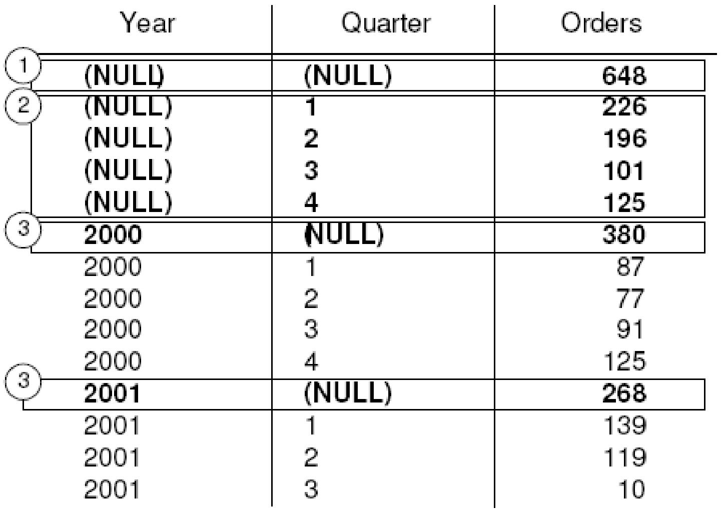
```
c_state   c_gender  c_dept    COUNT()   AVERAGE
-------   --------  -------    -----     --------
ALL       ALL        200         3       52200.00
ALL       ALL        ALL         3       52200.00
ALL       F          200         2       58650.00
ALL       F          ALL         2       58650.00
ALL       M         200          1       39300.00
ALL       M         ALL          1       39300.00
CA        ALL       200          3       52200.00
CA        ALL       ALL          3       52200.00
CA        F         200          2       58650.00
CA        F         ALL          2       58650.00
CA        M         200          1       39300.00
CA        M         ALL          1       39300.00
```

CUBE example 3—In this example, the query returns a result set that summarizes the total number of orders and then calculates subtotals for the number of orders by year and quarter.

Note: As the number of variables that you want to compare increases, the cost of computing the cube increases exponentially.

```
SELECT year (OrderDate) AS Year, quarter(OrderDate) AS Quarter, COUNT
(*) OrdersFROM SalesOrdersGROUP BY CUBE (Year, Quarter)ORDER BY Year,
Quarter
```

The figure that follows represents the result set from the query. The subtotal rows are highlighted in the result set. Each subtotal row has a NULL in the column or columns over which the subtotal is computed.



Year	Quarter	Orders
(NULL)	**(NULL)**	**648**
(NULL)	**1**	**226**
(NULL)	**2**	**196**
(NULL)	**3**	**101**
(NULL)	**4**	**125**
2000	**(NULL)**	**380**
2000	1	87
2000	2	77
2000	3	91
2000	4	125
2001	**(NULL)**	**268**
2001	1	139
2001	2	119
2001	3	10

The first highlighted row [1] represents the total number of orders across both years and all quarters. The value in the `Orders` column is the sum of the values in each of the rows marked [3]. It is also the sum of the four values in the rows marked [2].

The next set of highlighted rows [2] represents the total number of orders by quarter across both years. The two rows marked by [3] represent the total number of orders across all quarters for the years 2000 and 2001, respectively.

Analytical Functions

Sybase IQ offers both simple and windowed aggregation functions that offer the ability to perform complex data analysis within a single SQL statement.

You can use these functions to compute results for queries such as "What is the quarterly moving average of the Dow Jones Industrial average," or "List all employees and their cumulative salaries for each department." Moving averages and cumulative sums can be calculated over various intervals, and aggregations and ranks can be partitioned, so aggregate calculation is reset when partition values change. Within the scope of a single query expression, you can define several different OLAP functions, each with its own arbitrary partitioning rules. Analytical functions can be broken into two categories:

- Simple aggregate functions, such as **AVG**, **COUNT**, **MAX**, **MIN**, and **SUM** summarize data over a group of rows from the database. The groups are formed using the **GROUP BY** clause of the **SELECT** statement.
- Unary statistical aggregate functions that take one argument include **STDDEV**, **STDDEV_SAMP**, **STDDEV_POP**, **VARIANCE**, **VAR_SAMP**, and **VAR_POP.**

Both the simple and unary categories of aggregates summarize data over a group of rows from the database and can be used with a window specification to compute a moving window over a result set as it is processed.

Note: The aggregate functions **AVG**, **SUM**, **STDDEV**, **STDDEV_POP**, **STDDEV_SAMP**, **VAR_POP**, **VAR_SAMP**, and **VARIANCE** do not support binary data types `BINARY` and `VARBINARY`.

Simple Aggregate Functions

Simple aggregate functions, such as **AVG**, **COUNT**, **MAX**, **MIN**, and **SUM** summarize data over a group of rows from the database.

The groups are formed using the **GROUP BY** clause of the **SELECT** statement. These aggregates are allowed only in the select list and in the **HAVING** and **ORDER BY** clauses of a **SELECT** statement.

Note: With the exception of Grouping() functions, both the simple and unary aggregates can be used in a windowing function that incorporates a <window clause> in a SQL query specification (a *window*) that conceptually creates a moving window over a result set as it is processed.

See *Reference: Building Blocks, Tables, and Procedures > SQL Functions > Aggregate functions*.

Windowing

A major feature of the ANSI SQL extensions for OLAP is a construct called a *window*. This windowing extension lets users divide result sets of a query (or a logical partition of a query) into groups of rows called partitions and determine subsets of rows to aggregate with respect to the current row.

You can use three classes of window functions with a window: ranking functions, the row numbering function, and window aggregate functions.

```
<WINDOWED TABLE FUNCTION TYPE> ::=
  <RANK FUNCTION TYPE> <LEFT PAREN> <RIGHT PAREN>
  | ROW_NUMBER <LEFT PAREN> <RIGHT PAREN>
  | <WINDOW AGGREGATE FUNCTION>
```

Windowing extensions specify a window function type over a window name or specification and are applied to partitioned result sets within the scope of a single query expression. A window partition is a subset of rows returned by a query, as defined by one or more columns in a special **OVER** clause:

```
olap_function() OVER (PARTITION BY col1, col2...)
```

Windowing operations let you establish information such as the ranking of each row within its partition, the distribution of values in rows within a partition, and similar operations.

Windowing also lets you compute moving averages and sums on your data, enhancing the ability to evaluate your data and its impact on your operations.

An OLAP window's three essential parts

The OLAP windows comprise three essential aspects: window partitioning, window ordering, and window framing. Each has a significant impact on the specific rows of data visible in a window at any point in time. Meanwhile, the OLAP **OVER** clause differentiates OLAP functions from other analytic or reporting functions with three distinct capabilities:

- Defining window partitions (**PARTITION BY** clause).
- Ordering rows within partitions (**ORDER BY** clause).
- Defining window frames (ROWS/RANGE specification).

To specify multiple windows functions, and to avoid redundant window definitions, you can specify a name for an OLAP window specifications. In this usage, the keyword, **WINDOW**, is followed by at least one window definition, separated by commas. A window definition includes the name by which the window is known in the query and the details from the windows specification, which lets you to define window partitioning, ordering, and framing:

```
<WINDOW CLAUSE> ::= <WINDOW DEFINITION LIST>
```

```
<WINDOW DEFINITION LIST> ::=
  <WINDOW DEFINITION> [ { <COMMA> <WINDOW DEFINITION>
  } . . . ]
```

```
<WINDOW DEFINITION> ::=
  <NEW WINDOW NAME> AS <WINDOW SPECIFICATION>
```

```
<WINDOW SPECIFICATION DETAILS> ::=
  [ <EXISTING WINDOW NAME> ]
  [ <WINDOW PARTITION CLAUSE> ]
  [ <WINDOW ORDER CLAUSE> ]
  [ <WINDOW FRAME CLAUSE> ]
```

For each row in a window partition, users can define a window frame, which may vary the specific range of rows used to perform any computation on the current row of the partition. The current row provides the reference point for determining the start and end points of the window frame.

Window specifications can be based on either a physical number of rows using a window specification that defines a window frame unit of ROWS or a logical interval of a numeric value, using a window specification that defines a window frame unit of RANGE.

Within OLAP windowing operations, you can use the following functional categories:

- Ranking functions
- Windowing aggregate functions
- Statistical aggregate functions
- Distribution functions

See also

- *Distribution Functions* on page 59
- *OLAP Benefits* on page 22
- *OLAP Evaluation* on page 22
- *Ranking Functions* on page 48
- *Statistical Aggregate Functions* on page 54
- *Windowing Aggregate Functions* on page 52
- *BNF Grammar for OLAP Functions* on page 73

Window Partitioning

Window partitioning is the division of user-specified result sets (input rows) using a **PARTITION BY** clause.

A partition is defined by one or more value expressions separated by commas. Partitioned data is also implicitly sorted and the default sort order is ascending (ASC).

```
<WINDOW PARTITION CLAUSE> ::=
  PARTITION BY <WINDOW PARTITION EXPRESSION LIST>
```

If a window partition clause is not specified, then the input is treated as single partition.

Note: The term *partition* as used with analytic functions, refers only to dividing the set of result rows using a **PARTITION BY** clause.

A window partition can be defined based on an arbitrary expression. Also, because window partitioning occurs after GROUPING (if a **GROUP BY** clause is specified), the result of any aggregate function, such as **SUM**, **AVG**, and **VARIANCE**, can be used in a partitioning expression. Therefore, partitions provide another opportunity to perform grouping and ordering operations *in addition to* the **GROUP BY** and **ORDER BY** clauses; for example, you can construct queries that compute aggregate functions over aggregate functions, such as the maximum **SUM** of a particular quantity.

You can specify a **PARTITION BY** clause, even if there is no **GROUP BY** clause.

See also

- *Window Framing* on page 40
- *Window Ordering* on page 39

Window Ordering

Window ordering is the arrangement of results (rows) within each window partition using a window order clause, which contains one or more value expressions separated by commas.

If a window order clause is not specified, the input rows could be processed in an arbitrary order.

```
<WINDOW ORDER CLAUSE> ::= <ORDER SPECIFICATION>
```

The OLAP window order clause is different from the **ORDER BY** clause that can be appended to a nonwindowed query expression.

The **ORDER BY** clause in an OLAP function, for example, typically defines the expressions for sorting rows within window partitions; however, you can use the **ORDER BY** clause without a **PARTITION BY** clause, in which case the sort specification ensures that the OLAP function is applied to a meaningful (and intended) ordering of the intermediate result set.

An order specification is a prerequisite for the ranking family of OLAP functions; it is the **ORDER BY** clause, not an argument to the function itself, that identifies the measures for the ranking values. In the case of OLAP aggregates, the **ORDER BY** clause is not required in general, but it is a prerequisite to defining a window frame. This is because the partitioned rows must be sorted before the appropriate aggregate values can be computed for each frame.

The **ORDER BY** clause includes semantics for defining ascending and descending sorts, as well as rules for the treatment of NULL values. By default, OLAP functions assume an ascending order, where the lowest measured value is ranked 1.

Although this behavior is consistent with the default behavior of the **ORDER BY** clause that ends a **SELECT** statement, it is counterintuitive for most sequential calculations. OLAP calculations often require a descending order, where the highest measured value is ranked 1; this requirement must be explicitly stated in the **ORDER BY** clause with the DESC keyword.

Note: Ranking functions require a <window order clause> because they are defined only over sorted input. As with an <order by clause> in a <query specification>, the default sort sequence is ascending.

The use of a <window frame unit> of RANGE also requires the existence of a <window order clause>. In the case of RANGE, the <window order clause> may only consist of a single expression.

See also

- *Window Framing* on page 40
- *Window Partitioning* on page 39

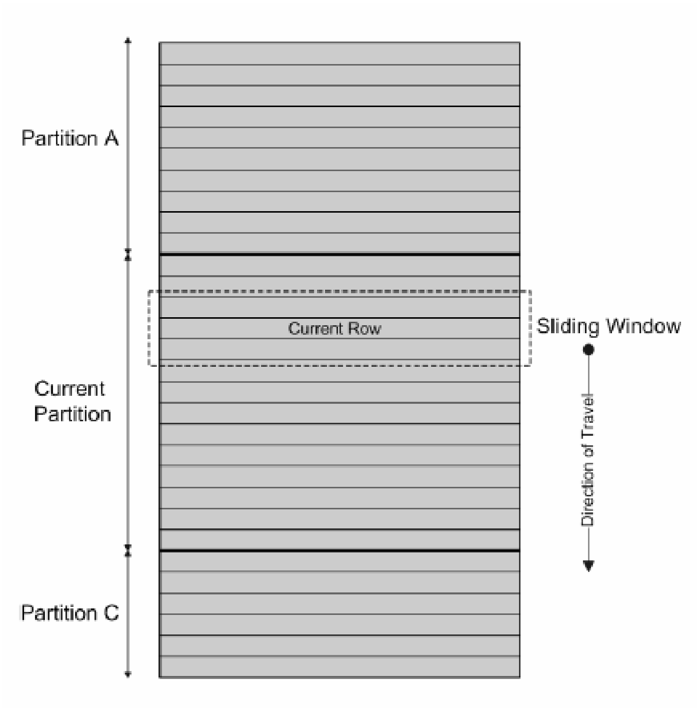
Window Framing

For nonranking aggregate OLAP functions, you can define a window frame with a window frame clause, which specifies the beginning and end of the window relative to the current row.

```
<WINDOW FRAME CLAUSE> ::=
  <WINDOW FRAME UNIT>
  <WINDOW FRAME EXTENT>
```

This OLAP function is computed with respect to the contents of a moving frame rather than the fixed contents of the whole partition. Depending on its definition, the partition has a start row and an end row, and the window frame slides from the starting point to the end of the partition.

Figure 3: Three-row moving window with partitioned input



UNBOUNDED PRECEDING and FOLLOWING

Window frames can be defined by an unbounded aggregation group that either extends back to the beginning of the partition (UNBOUNDED PRECEDING) or extends to the end of the partition (UNBOUNDED FOLLOWING), or both.

UNBOUNDED PRECEDING includes all rows within the partition preceding the current row, which can be specified with either ROWS or RANGE. UNBOUNDED FOLLOWING includes all rows within the partition following the current row, which can be specified with either ROWS or RANGE.

The value FOLLOWING specifies either the range or number of rows following the current row. If ROWS is specified, then the value is a positive integer indicating a number of rows. If RANGE is specified, the window includes any rows that are less than the current row plus the specified numeric value. For the RANGE case, the data type of the windowed value must be comparable to the type of the sort key expression of the **ORDER BY** clause. There can be only one sort key expression, and the data type of the sort key expression must allow addition.

The value PRECEDING specifies either the range or number of rows preceding the current row. If ROWS is specified, then the value is a positive integer indicating a number of rows. If RANGE is specified, the window includes any rows that are less than the current row minus the specified numeric value. For the RANGE case, the data type of the windowed value must be comparable to the type of the sort key expression of the **ORDER BY** clause. There can be only one sort key expression, and the data type of the sort key expression must allow subtraction. This clause cannot be specified in second bound group if the first bound group is CURRENT ROW or value FOLLOWING.

The combination BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING provides an aggregate over an entire partition, without the need to construct a join to a grouped query. An aggregate over an entire partition is also known as a reporting aggregate.

CURRENT ROW concept

In physical aggregation groups, rows are included or excluded based on their position relative to the current row, by counting adjacent rows. The current row is simply a reference to the next row in a query's intermediate results. As the current row advances, the window is reevaluated based on the new set of rows that lie within the window. There is no requirement that the current row be included in a window.

If a window frame clause is not specified, the default window frame depends on whether or not a window order clause is specified:

- If the window specification contains a window order clause, the window's start point is **UNBOUNDED PRECEDING**, and the end point is **CURRENT ROW**, thus defining a varying-size window suitable for computing cumulative values.
- If the window specification does not contain a window order clause, the window's start point is **UNBOUNDED PRECEDING**, and the end point is **UNBOUNDED FOLLOWING**, thus defining a window of fixed size, regardless of the current row.

Note: A window frame clause cannot be used with a ranking function.

You can also define a window by specifying a window frame unit that is row-based (rows specification) or value-based (range specification).

```
<WINDOW FRAME UNIT> ::= ROWS | RANGE
```

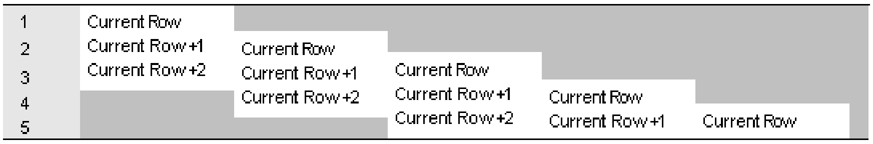
```
<WINDOW FRAME EXTENT> ::= <WINDOW FRAME START> | <WINDOW FRAME
BETWEEN>
```

When a window frame extent specifies **BETWEEN**, it explicitly provides the beginning and end of a window frame.

If the window frame extent specifies only one of these two values then the other value defaults to **CURRENT ROW**.

Row-based window frames—In the example rows [1] through [5] represent a partition; each row becomes the current row as the OLAP window frame slides forward. The frame is defined as Between Current Row And 2 Following, so each frame includes a maximum of three rows and a minimum of one row. When the frame reaches the end of the partition, only the current row is included. The shaded areas indicate which rows are excluded from the frame at each step.

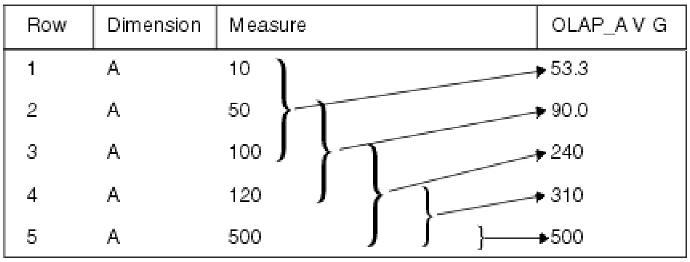
Figure 4: Row-based window frames



The window frame imposes the following rules:

- When row [1] is the current row, rows [4] and [5] are excluded.
- When row [2] is the current row, rows [5] and [1] are excluded.
- When row [3] is the current row, rows [1] and [2] are excluded.
- When row [4] is the current row, rows [1], [2], and [3] are excluded.
- When row [5] is the current row, rows [1], [2], [3], and [4] are excluded.

The following diagram applies these rules to a specific set of values, showing the OLAP **AVG** function that would be calculated for each row. The sliding calculations produce a moving average with an interval of three rows or fewer, depending on which row is the current row:



Row	Dimension	Measure	OLAP_AVG
1	A	10	53.3
2	A	50	90.0
3	A	100	240
4	A	120	310
5	A	500	500

The following example demonstrates a sliding window:

```
SELECT dimension, measure,
  AVG(measure) OVER(partition BY dimension
    ORDER BY measure
    ROWS BETWEEN CURRENT ROW and 2 FOLLOWING)
    AS olap_avg
FROM ...
```

The averages are computed as follows:

- Row [1] = (10 + 50 + 100)/3
- Row [2] = (50+ 100 + 120)/3
- Row [3] = (100 + 120 + 500)/3
- Row [4] = (120 + 500 + NULL)/3
- Row [5] = (500 + NULL + NULL)/3

Similar calculations would be computed for all subsequent partitions in the result set (such as, B, C, and so on).

If there are no rows in the current window, the result is NULL, except for **COUNT**.

See also

- *Window Ordering* on page 39
- *Window Partitioning* on page 39

ROWS

The window frame unit ROWS defines a window in the specified number of rows before or after the current row, which serves as the reference point that determines the start and end of a window.

Each analytical calculation is based on the current row within a partition. To produce determinative results for a window expressed in rows, the ordering expression should be unique.

The reference point for all window frames is the current row. The SQL/OLAP syntax provides mechanisms for defining a row-based window frame as any number of rows preceding or following the current row or preceding and following the current row.

The following list illustrates common examples of a window frame unit:

- Rows between unbounded preceding and current row – specifies a window whose start point is the beginning of each partition and the end point is the current row and is often used to construct windows that compute cumulative results, such as cumulative sums.
- Rows between unbounded preceding and unbounded following – specifies a fixed window, regardless of the current row, over the entire partition. The value of a window aggregate function is, therefore, identical in each row of the partition.
- Rows between 1 preceding and 1 following – specifies a fixed-sized moving window over three adjacent rows, one each before and after the current row. You can use this window frame unit to compute, for example, a 3-day or 3-month moving average.

  Be aware of meaningless results that may be generated by gaps in the windowed values when using ROWS. If the set of values is not continuous, consider using RANGE instead of ROWS, because a window definition based on RANGE automatically handles adjacent rows with duplicate values and does not include other rows when there are gaps in the range.

  Note: In the case of a moving window, it is assumed that rows containing NULL values exist before the first row, and after the last row, in the input. This means that in a 3-row moving window, the computation for the last row in the input—the current row— includes the immediately preceding row and a NULL value.
- Rows between current row and current row – restricts the window to the current row only.
- Rows between 1 preceding and 1 preceding – specifies a single row window consisting only of the preceding row, with respect to the current row. In combination with another window function that computes a value based on the current row only, this construction makes it possible to easily compute deltas, or differences in value, between adjacent rows.

See also

- *RANGE* on page 45

RANGE

Range-based window frames—The SQL/OLAP syntax supports another kind of window frame whose limits are defined in terms of a value-based—or range-based—set of rows, rather than a specific sequence of rows.

Value-based window frames define rows within a window partition that contain a specific range of numeric values. The OLAP function's **ORDER BY** clause defines the numeric column to which the range specification is applied, relative to the current row's value for that column. The range specification uses the same syntax as the rows specification, but the syntax is interpreted in a different way.

The window frame unit, RANGE, defines a window frame whose contents are determined by finding rows in which the ordering column has values within the specified range of value relative to the current row. This is called a logical offset of a window frame, which you can specify with constants, such as "3 preceding," or any expression that can be evaluated to a numeric constant. When using a window defined with RANGE, there can be only a single numeric expression in the **ORDER BY** clause.

Note: ORDER BY key must be a numeric data in RANGE window frame

For example, a frame can be defined as the set of rows with *year* values some number of years preceding or following the current row's year:

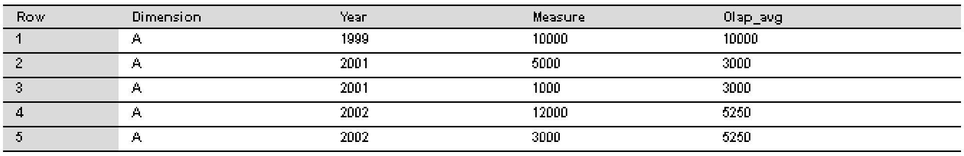
```
ORDER BY year ASC range BETWEEN CURRENT ROW and 1 PRECEDING
```

In the above example query, 1 preceding means the current row's *year* value minus 1.

This kind of range specification is inclusive. If the current row's *year* value is 2000, all rows in the window partition with year values 2000 and 1999 qualify for the frame, regardless of the physical position of those rows in the partition. The rules for including and excluding value-based rows are quite different from the rules applied to row-based frames, which depend entirely on the physical sequence of rows.

Put in the context of an OLAP **AVG()** calculation, the following partial result set further demonstrates the concept of a value-based window frame. Again, the frame consists of rows that:

- Have the same year as the current row
- Have the same year as the current row minus 1



Row	Dimension	Year	Measure	Olap_avg
1	A	1999	10000	10000
2	A	2001	5000	3000
3	A	2001	1000	3000
4	A	2002	12000	5250
5	A	2002	3000	5250

The following query demonstrates a range-based window definition:

```
SELECT dimension, year, measure,
  AVG(measure) OVER(PARTITION BY dimension
    ORDER BY year ASC
    range BETWEEN CURRENT ROW and 1 PRECEDING)
    as olap_avg
FROM ...
```

The averages are computed as follows:

- Row [1] = 1999; rows [2] through [5] are excluded; AVG = 10,000/1
- Row [2] = 2001; rows [1], [4], and [5] are excluded; AVG = 6,000/2
- Row [3] = 2001; rows [1], [4], and [5] are excluded; AVG = 6,000/2
- Row [4] = 2002; row [1] is excluded; AVG = 21,000/4
- Row [5] = 2002; row [1] is excluded; AVG = 21,000/4

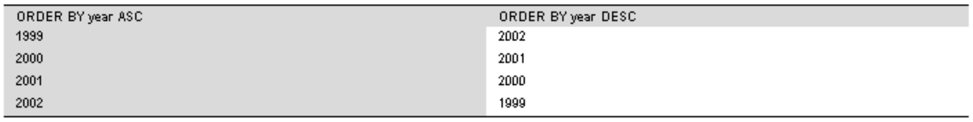
Ascending and descending order for value -based frames—The **ORDER BY** clause for an OLAP function with a value-based window frame not only identifies the numeric column on which the range specification is based; it also declares the sort order for the **ORDER BY** values. The following specification is subject to the sort order that precedes it (ASC or DESC):

```
RANGE BETWEEN CURRENT ROW AND n FOLLOWING
```

The specification *n* FOLLOWING means:

- Plus *n* if the partition is sorted in default ascending order (ASC)
- Minus *n* if the partition is sorted in descending order (DESC)

For example, assume that the `year` column contains four distinct values, from 1999 to 2002. The following table shows the default ascending order of these values on the left and the descending order on the right:



ORDER BY year ASC	ORDER BY year DESC
1999	2002
2000	2001
2001	2000
2002	1999

If the current row is 1999 and the frame is specified as follows, rows that contain the values 1999 and 1998 (which does not exist in the table) are included in the frame:

```
ORDER BY year DESC range BETWEEN CURRENT ROW and 1 FOLLOWING
```

Note: The sort order of the **ORDER BY** values is a critical part of the test for qualifying rows in a value-based frame; the numeric values alone do not determine exclusion or inclusion.

Using an unbounded window—The following query produces a result set consisting of all of the products accompanied by the total quantity of all products:

```
SELECT id, description, quantity,
  SUM(quantity) OVER () AS total
FROM products;
```

Computing deltas between adjacent rows—Using two windows—one over the current row and the other over the previous row—provides a direct way of computing deltas, or changes, between adjacent rows.

```
SELECT EmployeeID, Surname, SUM(salary)
OVER(ORDER BY BirthDate rows between current row and current row)
AS curr, SUM(Salary)
OVER(ORDER BY BirthDate rows between 1 preceding and 1 preceding)
AS prev, (curr-prev) as delta
FROM Employees
WHERE State IN ('MA', 'AZ', 'CA', 'CO') AND DepartmentID>10
ORDER BY EmployeeID, Surname;
```

The results from the query:

```
EmployeeID  Surname    curr      prev       delta
------            ---------    --------  ---------  ----------
148           Jordan                51432.000191
Bertrand              29800.000            39300.000
-9500.000278         Melkisetian          48500.000
42300.000           6200.000299          Overbey
39300.000            41700.750           -2400.750318
Crow                41700.750            45000.000
-3299.250586           Coleman              42300.000
46200.000           -3900.000690          Poitras
46200.000            29800.000           16400.000703
Martinez              55500.800             51432.000
4068.800949          Savarino              72300.000
55500.800           16799.2001101          Preston
37803.000           48500.000          -10697.0001142
Clark                45000.000            72300.000
-27300.000
```

Although the window function **SUM()** is used, the sum contains only the salary value of either the current or previous row because of the way the window is specified. Also, the `prev` value of the first row in the result is NULL because it has no predecessor; therefore, the `delta` is NULL as well.

In each of the examples above, the function used with the **OVER()** clause is the **SUM()** aggregate function.

See also

- *ROWS* on page 44

Explicit and Inline Window Clauses

SQL OLAP provides two ways of specifying a window in a query:

- The explicit window clause lets you define a window that follows a **HAVING** clause. You reference windows defined with those window clauses by specifying their names when you invoke an OLAP function, such as:

  ```
  SUM ( ...) OVER w2
  ```

- The inline window specification lets you define a window in the **SELECT** list of a query expression. This capability lets you define your windows in a window clause that follows the **HAVING** clause and then reference them by name from your window function invocations, or to define them along with the function invocations.

  Note: If you use an inline window specification, you cannot name the window. Two or more window function invocations in a single **SELECT** list that use identical windows must either reference a named window defined in a window clause or they must define their inline windows redundantly.

Window function example—The following example shows a window function. The query returns a result set that partitions the data by department and then provides a cumulative summary of employees' salaries, starting with the employee who has been at the company the longest. The result set includes only those employees who reside in Massachusetts. The column `Sum_Salary` provides the cumulative total of employees' salaries.

```
SELECT DepartmentID, Surname, StartDate, Salary,SUM(Salary) OVER
(PARTITION BY DepartmentID ORDER BYstartdate rows between unbounded
preceding andcurrent row) AS sum_salaryFROM EmployeesWHERE State IN
('CA') AND DepartmentID IN (100, 200)ORDER BY DepartmentID;
```

The following result set is partitioned by department.

```
DepartmentID  Surname   start_date      salary    sum_salary
------------              -----------  ------------
----------     -----------
200                       Overbey               1987-02-19
39300.000       39300.000
200                      Savarino              1989-11-07
72300.000      111600.000
200                      Clark                 1990-07-21
45000.000      156600.000
```

Ranking Functions

Ranking functions let you compile a list of values from the data set in ranked order, as well as compose single-statement SQL queries that fulfil requests such as, "Name the top 10 products shipped this year by total sales," or "Give the top 5% of salespersons who sold orders to at least 15 different companies."

SQL/OLAP defines five functions that are categorized as ranking functions:

```
<RANK FUNCTION TYPE> ::=
  RANK | DENSE_RANK | PERCENT_RANK | ROW_NUMBER | NTILE
```

Ranking functions let you compute a rank value for each row in a result set based on the order specified in the query. For example, a sales manager might need to identify the top or bottom sales people in the company, the highest- or lowest-performing sales region, or the best- or worst-selling products. Ranking functions can provide this information.

See also

- *Distribution Functions* on page 59

- *OLAP Benefits* on page 22
- *OLAP Evaluation* on page 22
- *Statistical Aggregate Functions* on page 54
- *Windowing* on page 37
- *Windowing Aggregate Functions* on page 52
- *BNF Grammar for OLAP Functions* on page 73

RANK() Function

The **RANK** function returns a number that indicates the rank of the current row among the rows in the row's partition, as defined by the **ORDER BY** clause.

The first row in a partition has a rank of 1, and the last rank in a partition containing 25 rows is 25. **RANK** is specified as a syntax transformation, which means that an implementation can choose to actually transform **RANK** into its equivalent, or it can merely return a result equivalent to the result that transformation would return.

In the following example, ws1 indicates the window specification that defines the window named w1.

```
RANK() OVER ws
```

is equivalent to:

```
( COUNT (*) OVER ( ws RANGE UNBOUNDED PRECEDING )
- COUNT (*) OVER ( ws RANGE CURRENT ROW ) + 1 )
```

The transformation of the **RANK** function uses logical aggregation (RANGE). As a result, two or more records that are tied—or have equal values in the ordering column—have the same rank.The next group in the partition that has a different value has a rank that is more than one greater than the rank of the tied rows. For example, if there are rows whose ordering column values are 10, 20, 20, 20, 30, the rank of the first row is 1 and the rank of the second row is 2. The rank of the third and fourth row is also 2, but the rank of the fifth row is 5. There are no rows whose rank is 3 or 4. This algorithm is sometimes known as sparse ranking.

See also *Reference: Building Blocks, Tables, and Procedures > SQL Functions > RANK function [Analytical].*

DENSE_RANK() Function

DENSE_RANK DENSE_RANK returns ranking values without gaps.

The values for rows with ties are still equal, but the ranking of the rows represents the positions of the clusters of rows having equal values in the ordering column, rather than the positions of the individual rows. As in the **RANK** example, where rows ordering column values are 10, 20, 20, 20, 30, the rank of the first row is still 1 and the rank of the second row is still 2, as are the ranks of the third and fourth rows. The last row, however, is 3, not 5.

DENSE_RANK is computed through a syntax transformation, as well.

```
DENSE_RANK() OVER ws
```

is equivalent to:

```
COUNT ( DISTINCT ROW ( expr_1, . . ., expr_n ) )
 OVER ( ws RANGE UNBOUNDED PRECEDING )
```

In the above example, *expr_1* through *expr_n* represent the list of value expressions in the sort specification list of window w1.

See also *Reference: Building Blocks, Tables, and Procedures > SQL Functions > DENSE_RANK function [Analytical]*.

PERCENT_RANK() Function

The **PERCENT_RANK** function calculates a percentage for the rank, rather than a fractional amount, and returns a decimal value between 0 and 1.

PERCENT_RANK returns the relative rank of a row, which is a number that indicates the relative position of the current row within the window partition in which it appears. For example, in a partition that contains 10 rows having different values in the ordering columns, the third row is given a **PERCENT_RANK** value of 0.222 …, because you have covered 2/9 (22.222...%) of rows following the first row of the partition. **PERCENT_RANK** of a row is defined as one less than the **RANK** of the row divided by one less than the number of rows in the partition, as seen in the following example (where "ANT" stands for an approximate numeric type, such as `REAL` or `DOUBLE PRECISION`).

```
PERCENT_RANK() OVER ws
```

is equivalent to:

```
CASE
  WHEN COUNT (*) OVER ( ws RANGE BETWEEN UNBOUNDED
  PRECEDING AND UNBOUNDED FOLLOWING ) = 1
  THEN CAST (0 AS ANT)
  ELSE
   ( CAST ( RANK () OVER ( ws ) AS ANT ) -1 /
   ( COUNT (*) OVER ( ws RANGE BETWEEN UNBOUNDED
   PRECEDING AND UNBOUNDED FOLLOWING ) - 1 )
END
```

See also *Reference: Building Blocks, Tables, and Procedures > SQL Functions > PERCENT_RANK function [Analytical]*.

ROW_NUMBER() Function

The **ROW_NUMBER** function returns a unique row number for each row.

If you define window partitions, **ROW_NUMBER** starts the row numbering in each partition at 1, and increments each row by 1. If you do not specify a window partition, **ROW_NUMBER** numbers the complete result set from 1 to the total cardinality of the table.

The **ROW_NUMBER** function syntax is:

```
ROW_NUMBER() OVER ([PARTITION BY window partition] ORDER BY window
ordering)
```

ROW_NUMBER does not require an argument, but you must specify the parentheses.

The **PARTITION BY** clause is optional. The **OVER** (**ORDER_BY**) clause cannot contain a window frame **ROWS**/**RANGE** specification.

Ranking Examples

These are some of the ranking functions examples:

Ranking example 1—The SQL query that follows finds the male and female employees from California, and ranks them in descending order according to salary.

```
SELECT Surname, Sex, Salary, RANK() OVER (
ORDER BY Salary DESC) as RANK FROM Employees
WHERE State IN ('CA') AND DepartmentID =200
ORDER BY Salary DESC;
```

The results from the above query:

```
Surname      Sex        Salary    RANK
-------      ---        ------    ----
Savarino      F          72300.000  1
Clark         F          45000.000  2
Overbey       M          39300.000  3
```

Ranking example 2—Using the query from the previous example, you can change the data by partitioning it by gender. The following example ranks employees in descending order by salary and partitions by gender:

```
SELECT Surname, Sex, Salary, RANK() OVER (PARTITION BY Sex
ORDER BY Salary DESC) AS RANK FROM Employees
WHERE State IN ('CA', 'AZ') AND DepartmentID IN (200, 300)
ORDER BY Sex, Salary DESC;
```

The results from the above query:

```
Surname            Sex      Salary     RANK
-------            ---      ---------  ----
Savarino            F        72300.000  1
Jordan              F        51432.000  2
Clark               F        45000.000  3
Coleman             M        42300.000  1
Overbey             M        39300.000  2
```

Ranking example 3—This example ranks a list of female employees in California and Texas in descending order according to salary. The **PERCENT_RANK** function provides the cumulative total in descending order.

```
SELECT Surname, Salary, Sex, CAST(PERCENT_RANK() OVER
(ORDER BY Salary DESC) AS numeric (4, 2)) AS RANK
FROM Employees WHERE State IN ('CA', 'TX') AND Sex ='F'
ORDER BY Salary DESC;
```

The results from the above query:

```
Surname          salary   sex           RANK
---------       ---------   ---   ----------
Savarino        72300.000   F           0.00
Smith           51411.000   F           0.33
Clark           45000.000   F           0.66
Garcia          39800.000   F           1.00
```

Ranking example 4—You can use the **PERCENT_RANK** function to find the top or bottom percentiles in the data set. This query returns male employees whose salary is in the top five percent of the data set.

```
SELECT * FROM (SELECT Surname, Salary, Sex,
CAST(PERCENT_RANK() OVER (ORDER BY salary DESC) as
numeric (4, 2)) AS percent
FROM Employees WHERE State IN ('CA') AND sex ='F' ) AS
DT where percent > 0.5
ORDER BY Salary DESC;
```

The results from the above query:

```
Surname           salary    sex     percent
---------       ----------    ---   ---------
Clark                          45000.000     F      1.00
```

Ranking example 5—This example uses the **ROW_NUMBER** function to return row numbers for each row in all window partitions. The query partitions the `Employees` table by department ID, and orders the rows in each partition by start date.

```
SELECT DepartmentID dID, StartDate, Salary ,
ROW_NUMBER()OVER(PARTITION BY dID ORDER BY StartDate)
FROM  Employees ORDER BY 1,2;
```

The results from the above query are:

```
dID        StartDate     Salary      Row_number()
========   ===========   ==========  =============
  100      1984-08-28    47500.000      1
  100      1985-01-01    62000.500      2
  100      1985-06-17    57490.000      3
  100      1986-06-07    72995.000      4
  100      1986-07-01    48023.690      5
  ...      ...           ...            ...
  200      1985-02-03    38500.000      1
  200      1985-12-06    54800.000      2
  200      1987-02-19    39300.000      3
  200      1987-07-10    49500.000      4
  ...      ...           ...            ...
  500      1994-02-27     24903.000     9
```

Windowing Aggregate Functions

Windowing aggregate functions let you manipulate multiple levels of aggregation in the same query.

For example, you can list all quarters during which expenses are less than the average. You can use aggregate functions, including the simple aggregate functions **AVG**, **COUNT**, **MAX**, **MIN**, and **SUM**, to place results—possibly computed at different levels in the statement—on the same row. This placement provides a means to compare aggregate values with detail rows within a group, avoiding the need for a join or a correlated subquery.

These functions also let you compare nonaggregate values to aggregate values. For example, a salesperson might need to compile a list of all customers who ordered more than the average number of a product in a specified year, or a manager might want to compare an employee's salary against the average salary of the department.

If a query specifies **DISTINCT** in the **SELECT** statement, then the **DISTINCT** operation is applied after the window operator. A window operator is computed after processing the **GROUP BY** clause and before the evaluation of the **SELECT** list items and a query's **ORDER BY** clause.

Windowing aggregate example 1—This query returns a result set, partitioned by year, that shows a list of the products that sold higher-than-average sales.

```
SELECT * FROM (SELECT Surname AS E_name, DepartmentID ASDept,
CAST(Salary AS numeric(10,2) ) AS Sal,CAST(AVG(Sal) OVER(PARTITION
BY DepartmentID) ASnumeric(10, 2)) AS Average,
CAST(STDDEV_POP(Sal)OVER(PARTITION BY DepartmentID) AS
numeric(10,2)) ASSTD_DEVFROM EmployeesGROUP BY Dept, E_name, Sal) AS
derived_table WHERESal> (Average+STD_DEV )ORDER BY Dept, Sal,
E_name;
```

The results from the query:

```
 E_name    Dept                          Sal    Average   STD_DEV
--------  -----          --------  -------         --------
Lull              100       87900.00           58736.28
16829.59Sheffield         100           87900.00
58736.28           16829.59Scott           100
96300.00          58736.28          16829.59Sterling
200        64900.00           48390.94
13869.59Savarino            200        72300.00
48390.94           13869.59Kelly           200
87500.00           48390.94          13869.59Shea
300        138948.00           59500.00
30752.39Blaikie            400        54900.00
43640.67           11194.02Morris           400
61300.00           43640.67          11194.02Evans
400        68940.00           43640.67
11194.02Martinez            500        55500.80
33752.20            9084.49
```

For the year 2000, the average number of orders was 1,787. Four products (700, 601, 600, and 400) sold higher than that amount. In 2001, the average number of orders was 1,048 and 3 products exceeded that amount.

Windowing aggregate example 2—This query returns a result set that shows the employees whose salary is one standard deviation greater than the average salary of their department. Standard deviation is a measure of how much the data varies from the mean.

```
SELECT * FROM (SELECT Surname AS E_name, DepartmentID AS
  Dept, CAST(Salary AS numeric(10,2) ) AS Sal,
  CAST(AVG(Sal) OVER(PARTITION BY dept) AS
  numeric(10, 2)) AS Average, CAST(STDDEV_POP(Sal)
  OVER(PARTITION BY dept) AS numeric(10,2)) AS
  STD_DEV
FROM Employees
GROUP BY Dept, E_name, Sal) AS derived_table WHERE
  Sal> (Average+STD_DEV )
ORDER BY Dept, Sal, E_name;
```

Every department has at least one employee whose salary significantly deviates from the mean, as shown in these results:

```
E_name        Dept        Sal    Average      STD_DEV
--------      ----   --------   --------     --------
Lull           100   87900.00   58736.28     16829.59
Sheffield      100   87900.00   58736.28     16829.59
Scott          100   96300.00   58736.28     16829.59
Sterling       200   64900.00   48390.94     13869.59
Savarino       200   72300.00   48390.94     13869.59
Kelly          200   87500.00   48390.94     13869.59
Shea           300  138948.00   59500.00     30752.39
Blaikie        400   54900.00   43640.67     11194.02
Morris         400   61300.00   43640.67     11194.02
Evans          400   68940.00   43640.67     11194.02
Martinez       500   55500.80   33752.20      9084.49
```

Employee Scott earns $96,300.00, while the average salary for department 100 is $58,736.28. The standard deviation for department 100 is 16,829.00, which means that salaries less than $75,565.88 (58736.28 + 16829.60 = 75565.88) fall within one standard deviation of the mean.

See also

- *Distribution Functions* on page 59
- *OLAP Benefits* on page 22
- *OLAP Evaluation* on page 22
- *Ranking Functions* on page 48
- *Statistical Aggregate Functions* on page 54
- *Windowing* on page 37
- *BNF Grammar for OLAP Functions* on page 73

Statistical Aggregate Functions

The ANSI SQL/OLAP extensions provide a number of additional aggregate functions that permit statistical analysis of numeric data. This support includes functions to compute variance, standard deviation, correlation, and linear regression.

Standard deviation and variance

The SQL/OLAP general set functions that take one argument include those appearing in bold in this syntax statement:

```
<SIMPLE WINDOW AGGREGATE FUNCTION TYPE> ::=
  <BASIC AGGREGATE FUNCTION TYPE>
  | STDDEV | STDDEV_POP | STDDEV_SAMP
  | VARIANCE | VARIANCE_POP | VARIANCE_SAMP
```

- **STDDEV_POP** – computes the population standard deviation of the provided value expression evaluated for each row of the group or partition (if **DISTINCT** is specified, each row that remains after duplicates are eliminated), defined as the square root of the population variance.
- **STDDEV_SAMP** – computes the population standard deviation of the provided value expression evaluated for each row of the group or partition (if **DISTINCT** is specified, each row that remains after duplicates are eliminated), defined as the square root of the sample variance.
- **VAR_POP** – computes the population variance of value expression evaluated for each row of the group or partition (if **DISTINCT** is specified, each row that remains after duplicates are eliminated), defined as the sum of squares of the difference of value expression from the mean of value expression, divided by the number of rows (remaining) in the group or partition.
- **VAR_SAMP** – computes the sample variance of value expression evaluated for each row of the group or partition (if **DISTINCT** is specified, each row that remains after duplicates are eliminated), defined as the sum of squares of the difference of value expression, divided by one less than the number of rows (remaining) in the group or partition.

These functions, including **STDDEV** and **VARIANCE**, are true aggregate functions in that they can compute values for a partition of rows as determined by the query's **ORDER BY** clause. As with other basic aggregate functions such as **MAX** or **MIN**, their computation ignores NULL values in the input. Also, regardless of the domain of the expression being analyzed, all variance and standard deviation computation uses IEEE double-precision floating point. If the input to any variance or standard deviation function is the empty set, then each function returns NULL as its result. If **VAR_SAMP** is computed for a single row, it returns NULL, while **VAR_POP** returns the value 0.

Correlation

The SQL/OLAP function that computes a correlation coefficient is:

- **CORR** – returns the correlation coefficient of a set of number pairs.

You can use the CORR function either as a windowing aggregate function (where you specify a window function type over a window name or specification) or as a simple aggregate function with no OVER clause.

Covariance

The SQL/OLAP functions that compute covariances include:

- **COVAR_POP** – returns the population covariance of a set of number pairs.
- **COVAR_SAMP** – returns the sample covariance of a set of number pairs.

The covariance functions eliminate all pairs where expression1 or expression2 has a null value.

You can use the covariance functions either as windowing aggregate functions (where you specify a window function type over a window name or specification) or as simple aggregate functions with no OVER clause.

Cumulative distribution

The SQL/OLAP function that calculates the relative position of a single value among a group of rows is **CUME_DIST**.

The window specification must contain an ORDER_BY clause.

Composite sort keys are not allowed in the CUME_DIST function.

Regression analysis

The regression analysis functions calculate the relationship between an independent variable and a dependent variable using a linear regression equation. The SQL/OLAP linear regression functions include:

- **REGR_AVGX** – computes the average of the independent variable of the regression line.
- **REGR_AVGY** – computes the average of the dependent variable of the regression line.
- **REGR_COUNT** – returns an integer representing the number of nonnull number pairs used to fit the regression line.
- **REGR_INTERCEPT** – computes the y-intercept of the regression line that best fits the dependent and independent variables.
- **REGR_R2** – computes the coefficient of determination (the goodness-of-fir statistic) for the regression line.
- **REGR_SLOPE** – computes the slope of the linear regression line fitted to nonnull pairs.
- **REGR_SXX** – returns the sum of squares of the independent expressions used in a linear regression model. Use this function to evaluate the statistical validity of the regression model.
- **REGR_SXY** – returns the sum of products of the dependent and independent variables. Use this function to evaluate the statistical validity of the regression model.
- **REGR_SYY** – returns values that can evaluate the statistical validity of a regression model.

You can use the regression analysis functions either as windowing aggregate functions (where you specify a window function type over a window name or specification) or as simple aggregate functions with no OVER clause.

Weighted OLAP aggregates

The weighted OLAP aggregate functions calculate weighted moving averages:

- **EXP_WEIGHTED_AVG** – calculates an exponentially weighted moving average. Weightings determine the relative importance of each quantity comprising the average. Weights in **EXP_WEIGHTED_AVG** decrease exponentially. Exponential weighting applies more weight to the most recent values and decreases the weight for older values, while still applying some weight

- **WEIGHTED_AVG** – calculates a linearly weighted moving average where weights decrease arithmetically over time. Weights decrease from the highest weight for the most recent data points, down to zero for the oldest data point.

The window specification must contain an **ORDER_BY** clause.

Nonstandard database industry extensions
Non-ANSI SQL/OLAP aggregate function extensions used in the database industry include **FIRST_VALUE**, **MEDIAN**, and **LAST_VALUE**.

- **FIRST_VALUE** – returns the first value from a set of values.
- **MEDIAN** – returns the median from an expression.
- **LAST_VALUE** – returns the last value from a set of values.

The **FIRST_VALUE** and **LAST_VALUE** functions require a window specification. You can use the **MEDIAN** function either as windowing aggregate function (where you specify a window function type over a window name or specification) or as a simple aggregate function with no OVER clause.

See also

- *Distribution Functions* on page 59
- *OLAP Benefits* on page 22
- *OLAP Evaluation* on page 22
- *Ranking Functions* on page 48
- *Windowing* on page 37
- *Windowing Aggregate Functions* on page 52
- *BNF Grammar for OLAP Functions* on page 73

Interrow Functions

The interrow functions, **LAG** and **LEAD**, provide access to previous or subsequent values in a data series, or to multiple rows in a table.

Interrow functions also partition simultaneously without a self-join. LAG provides access to a row at a given physical offset prior to the **CURRENT ROW** in the table or partition. LEAD provides access to a row at a given physical offset after the **CURRENT ROW** in the table or partition.

LAG and **LEAD** syntax is identical. Both functions require an **OVER** (**ORDER_BY**) window specification. For example:

```
LAG (value_expr) [, offset [, default]]) OVER ([PARTITION BY window
partition] ORDER BY window ordering)
```

and:

```
LEAD (value_expr) [, offset [, default]]) OVER ([PARTITION BY window
partition] ORDER BY window ordering)
```

The **PARTITION BY** clause in the **OVER** (**ORDER_BY**) clause is optional. The **OVER** (**ORDER_BY**) clause cannot contain a window frame **ROWS**/**RANGE** specification.

value_expr is a table column or expression that defines the offset data to return from the table. You can define other functions in the *value_expr*, with the exception of analytic functions.

For both functions, specify the target row by entering a physical offset. The *offset* value is the number of rows above or below the current row. Enter a nonnegative numeric data type (entering a negative value generates an error). If you enter 0, Sybase IQ returns the current row.

The optional *default* value defines the value to return if the *offset* value goes beyond the scope of the table. The default value of *default* is **NULL**. The data type of *default* must be implicitly convertible to the data type of the *value_expr* value, or Sybase IQ generates a conversion error.

LAG example 1—The inter-row functions are useful in financial services applications that perform calculations on data streams, such as stock transactions. This example uses the **LAG** function to calculate the percentage change in the trading price of a particular stock. Consider the following trading data from a fictional table called `stock_trades`:

```
traded at            symbol   price
-------------------  ------   ------
2009-07-13 06:07:12  SQL      15.84
2009-07-13 06:07:13  TST       5.75
2009-07-13 06:07:14  TST       5.80
2009-07-13 06:07:15  SQL      15.86
2009-07-13 06:07:16  TST       5.90
2009-07-13 06:07:17  SQL      15.86
```

Note: The fictional `stock_trades` table is not available in the `iqdemo` database.

The query partitions the trades by stock symbol, orders them by time of trade, and uses the **LAG** function to calculate the percentage increase or decrease in trade price between the current trade and the previous trade:

```
select  stock_symbol as 'Stock',
    traded_at    as 'Date/Time of Trade',
    trade_price  as 'Price/Share',
    cast ( ( ( (trade_price
        - (lag(trade_price, 1)
        over (partition by stock_symbol
            order by traded_at)))
        / trade_price)
    * 100.0) as numeric(5, 2) )
        as '% Price Change vs Previous Price'
from stock_trades
order by 1, 2
```

The query returns these results:

```
Stock   Date/Time of Trade   Price/  % Price Change_vs
symbol                       Share   Previous Price
------  -------------------  -----   -----------------
SQL     2009-07-13 06:07:12  15.84   NULL
SQL     2009-07-13 06:07:15  15.86   0.13
SQL     2009-07-13 06:07:17  15.86   0.00
TST     2009-07-13 06:07:13   5.75   NULL
TST     2009-07-13 06:07:14   5.80   0.87
TST     2009-07-13 06:07:16   5.90   1.72
```

The `NULL` result in the first and fourth output rows indicates that the **LAG** function is out of scope for the first row in each of the two partitions. Since there is no previous row to compare to, Sybase IQ returns `NULL` as specified by the *default* variable.

Distribution Functions

SQL/OLAP defines several functions that deal with ordered sets.

The two inverse distribution functions are **PERCENTILE_CONT** and **PERCENTILE_DISC**. These analytical functions take a percentile value as the function argument and operate on a group of data specified in the **WITHIN GROUP** clause or operate on the entire data set.

These functions return one value per group. For **PERCENTILE_DISC** (discrete), the data type of the results is the same as the data type of its **ORDER BY** item specified in the **WITHIN GROUP** clause. For **PERCENTILE_CONT** (continuous), the data type of the results is either numeric, if the **ORDER BY** item in the **WITHIN GROUP** clause is a numeric, or double, if the **ORDER BY** item is an integer or floating point.

The inverse distribution analytical functions require a **WITHIN GROUP** (**ORDER BY**) clause. For example:

```
PERCENTILE_CONT ( expression1 )
WITHIN GROUP ( ORDER BY expression2 [ ASC | DESC ] )
```

The value of *expression1* must be a constant of numeric data type and range from 0 to 1 (inclusive). If the argument is NULL, then a “wrong argument for percentile” error is returned. If the argument value is less than 0, or greater than 1, then a “data value out of range” error is returned.

The **ORDER BY** clause, which must be present, specifies the expression on which the percentile function is performed and the order in which the rows are sorted in each group. This **ORDER BY** clause is used only within the **WITHIN GROUP** clause and is *not* an **ORDER BY** for the **SELECT** statement.

The **WITHIN GROUP** clause distributes the query result into an ordered data set from which the function calculates a result.

The value *expression2* is a sort specification that must be a single expression involving a column reference. Multiple expressions are not allowed and no rank analytical functions, set functions, or subqueries are allowed in this sort expression.

The ASC or DESC parameter specifies the ordering sequence as ascending or descending. Ascending order is the default.

Inverse distribution analytical functions are allowed in a subquery, a **HAVING** clause, a view, or a union. The inverse distribution functions can be used anywhere the simple nonanalytical aggregate functions are used. The inverse distribution functions ignore the NULL value in the data set.

PERCENTILE_CONT example—This example uses the **PERCENTILE_CONT** function to determine the 10th percentile value for car sales in a region using the following data set:

```
sales       region        dealer_name
-----       ---------     -----------
900         Northeast     Boston
800         Northeast     Worcester
800         Northeast     Providence
700         Northeast     Lowell
540         Northeast     Natick
500         Northeast     New Haven
450         Northeast     Hartford
800         Northwest     SF
600         Northwest     Seattle
500         Northwest     Portland
400         Northwest     Dublin
500         South         Houston
400         South         Austin
300         South         Dallas
200         South         Dover
```

In the following example query, the **SELECT** statement contains the **PERCENTILE_CONT** function:

```
SELECT region, PERCENTILE_CONT(0.1)
WITHIN GROUP ( ORDER BY ProductID DESC )
FROM ViewSalesOrdersSales GROUP BY region;
```

The result of the **SELECT** statement lists the 10th percentile value for car sales in a region:

```
region           percentile_cont
---------        ---------------
Canada                        601.0
Central                         700.0
Eastern                         700.0
```

```
South                        700.0
Western                        700.0
```

PERCENTILE_DISC example—This example uses the **PERCENTILE_DISC** function to determine the 10th percentile value for car sales in a region, using the following data set:

```
sales       region       dealer_name
-----       ---------    -----------
900         Northeast    Boston
800         Northeast    Worcester
800         Northeast    Providence
700         Northeast    Lowell
540         Northeast    Natick
500         Northeast    New Haven
450         Northeast    Hartford
800         Northwest    SF
600         Northwest    Seattle
500         Northwest    Portland
400         Northwest    Dublin
500         South        Houston
400         South        Austin
300         South        Dallas
200         South        Dover
```

In the following query, the **SELECT** statement contains the **PERCENTILE_DISC** function:

```
SELECT region, PERCENTILE_DISC(0.1) WITHIN GROUP
  (ORDER BY sales DESC )
FROM carSales GROUP BY region;
```

The result of the **SELECT** statement lists the 10th percentile value for car sales in each region:

```
region           percentile_cont
---------        ---------------
Northeast        900
Northwest        800
South            500
```

For more information about the distribution functions, see *Reference: Building Blocks, Tables, and Procedures > SQL Functions > PERCENTILE_CONT function [Analytical]* and *PERCENTILE_DISC function [Analytical]*.

See also

- *OLAP Benefits* on page 22
- *OLAP Evaluation* on page 22
- *Ranking Functions* on page 48
- *Statistical Aggregate Functions* on page 54
- *Windowing* on page 37
- *Windowing Aggregate Functions* on page 52
- *BNF Grammar for OLAP Functions* on page 73

Numeric Functions

OLAP numeric functions supported by Sybase IQ include **CEILING** (**CEIL** is an alias), **EXP** (**EXPONENTIAL** is an alias), **FLOOR**, **LN** (**LOG** is an alias), **SQRT**, and **WIDTH_BUCKET**.

```
<numeric value function> :: =
 <natural logarithm>
| <exponential function>
| <power function>
| <square root>
| <floor function>
| <ceiling function>
| <width bucket function>
```

Table 4. Numeric value functions and syntax

Numeric value function	Syntax
Natural logarithm	**LN** (*numeric-expression*)
Exponential function	**EXP** (*numeric-expression*)
Power function	**POWER** (*numeric-expression1*, *numeric-expression2*)
Square root	**SQRT** (*numeric-expression*)
Floor function	**FLOOR** (*numeric-expression*)
Ceiling function	**CEILING** (*numeric-expression*)
Width bucket function	**WIDTH_BUCKET** (*expression*, *min_value*, *max_value*, *num_buckets*)

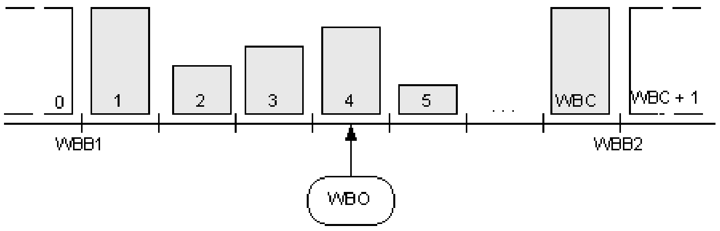
The semantics of the numeric value functions are:

- **LN** – returns the natural logarithm of the argument value. Raises an error condition if the argument value is 0 or negative. LN is a synonym for **LOG**.
- **EXP** – returns the value computed by raising the value of *e* (the base of natural logarithms) to the power specified by the value of the argument.
- **POWER** – returns the value computed by raising the value of the first argument to the power specified by the value of the second argument. If the first argument is 0 and the second is 0, returns one. If the first argument is 0 and the second is positive, returns 0. If the first argument is 0 and the second argument is negative, raises an exception. If the first argument is negative and the second is not an integer, raises an exception.
- **SQRT** – returns the square root of the argument value, defined by syntax transformation to "**POWER** (*expression*, *0.5*)."
- **FLOOR** – returns the integer value nearest to positive infinity that is not greater than the value of the argument.
- **CEILING** – returns the integer value nearest to negative infinity that is not less than the value of the argument. CEIL is a synonym for CEILING.

WIDTH_BUCKET function

The **WIDTH_BUCKET** function is somewhat more complicated than the other numeric value functions. It accepts four arguments: "live value," two range boundaries, and the number of equal-sized (or as nearly so as possible) partitions into which the range indicated by the boundaries is to be divided. WIDTH_BUCKET returns a number indicating the partition into which the live value should be placed, based on its value as a percentage of the difference between the higher range boundary and the lower boundary. The first partition is partition number one.

To avoid errors when the live value is outside the range of boundaries, live values that are less than the smaller range boundary are placed into an additional first bucket, bucket zero, and live values that are greater than the larger range boundary are placed into an additional last bucket, bucket N+1.



For example, **WIDTH_BUCKET** (*14, 5, 30, 5*) returns 2 because:

- (30-5)/5 is 5, so the range is divided into 5 partitions, each 5 units wide.
- The first bucket represents values from 0.00% to 19.999 …%; the second represents values from 20.00% to 39.999 …%; and the fifth bucket represents values from 80.00% to 100.00%.
- The bucket chosen is determined by computing (5*(14-5)/(30-5)) + 1 — one more than the number of buckets times the ratio of the offset of the specified value from the lower value to the range of possible values, which is (5*0/25) + 1, which is 2.8. This value is the range of values for bucket number 2 (2.0 through 2.999 …), so bucket number 2 is chosen.

WIDTH_BUCKET example

The following example creates a ten-bucket histogram on the `credit_limit` column for customers in Massachusetts in the sample table and returns the bucket number (“Credit Group”) for each customer. Customers with credit limits greater than the maximum value are assigned to the overflow bucket, 11:

Note: This example is for illustration purposes only and was not generated using the iqdemo database.

```
SELECT customer_id, cust_last_name, credit_limit,
  WIDTH_BUCKET(credit_limit, 100, 5000, 10) "Credit
  Group"
  FROM customers WHERE territory = 'MA'
  ORDER BY "Credit Group";
```

```
CUSTOMER_ID  CUST_LAST_NAME  CREDIT_LIMIT  Credit Group
-----------  --------------  ------------  ------------
825          Dreyfuss                 500             1
826          Barkin                   500             1
853          Palin                    400             1
827          Siegel                   500             1
843          Oates                    700             2
```

```
844          Julius                   700           2
835          Eastwood                1200           3
840          Elliott                 1400           3
842          Stern                   1400           3
841          Boyer                   1400           3
837          Stanton                 1200           3
836          Berenger                1200           3
848          Olmos                   1800           4
847           Streep                     5000              11
```

When the bounds are reversed, the buckets are open-closed intervals. For example: **WIDTH_BUCKET** (*credit_limit*, *5000*, *0*, *5*). In this example, bucket number 1 is (4000, 5000], bucket number 2 is (3000, 4000], and bucket number 5 is (0, 1000]. The overflow bucket is numbered 0 (5000, +infinity), and the underflow bucket is numbered 6 (-infinity, 0].

See also
Reference: Building Blocks, Tables, and Procedures > SQL Functions > BIT_LENGTH function [String], *EXP function [Numeric]*, *FLOOR function [Numeric]*, *POWER function [Numeric]*, *SQRT function [Numeric]*, and *WIDTH_BUCKET function [Numerical]*.

OLAP Rules and Restrictions

The following provides an overview for the rules and restrictions that govern OLAP functionality.

OLAP functions can be used
Sybase IQ provides SQL OLAP functions with rules, restrictions and limitations.

- In the **SELECT** list
- In expressions
- As arguments of scalar functions
- In the final **ORDER BY** clause (by using aliases or positional references to OLAP functions elsewhere in the query)

OLAP functions cannot be used
OLAP functions *cannot* be used under these conditions:

- In subqueries.
- In the search condition of a **WHERE** clause.
- As arguments for **SET** (aggregate) functions. For example, the following expression is invalid:

  ```
  SUM(RANK() OVER(ORDER BY dollars))
  ```
- A windowed aggregate cannot be an argument to argument to another unless the inner one was generated within a view or derived table. The same applies to ranking functions.
- Window aggregate and **RANK** functions are not allowed in a **HAVING** clause.

- Window aggregate functions should not specify **DISTINCT**.
- Window function cannot be nested inside of other window functions.
- Inverse distribution functions are not supported with the **OVER** clause.
- Outer references are not allowed in a window definition clause.
- Correlation references are allowed within OLAP functions, but correlated column aliases are not allowed.

Columns referenced by an OLAP function must be grouping columns or aggregate functions from the same query block in which the OLAP function and the **GROUP BY** clause appear. OLAP processing occurs after the grouping and aggregation operations and before the final **ORDER BY** clause is applied; therefore, it must be possible to derive the OLAP expressions from those intermediate results. If there is no **GROUP BY** clause in a query block, OLAP functions can reference other columns in the select list.

Sybase IQ limitations

The Sybase IQ limitations with SQL OLAP functions are:

- User-defined functions in a window frame definition are not supported.
- The constants used in a window frame definition must be unsigned numeric value and should not exceed the value of maximum `BIG INT` 2^{63-1}.
- Window aggregate functions and **RANK** functions cannot be used in **DELETE** and **UPDATE** statements.
- Window aggregate and **RANK** functions are not allowed in subqueries.
- CUME_DIST is currently not supported.
- Grouping sets are currently not supported.
- Correlation and linear regression functions are currently not supported.

Additional OLAP Examples

This section provides additional examples using the OLAP functions.

Both start and end points of a window may vary as intermediate result rows are processed. For example, computing a cumulative sum involves a window with the start point fixed at the first row of each partition and an end point that slides along the rows of the partition to include the current row.

As another example, both the start and end points of the window can be variable yet define a constant number of rows for the entire partition. Such a construction lets users compose queries that compute moving averages; for example, a SQL query that returns a moving three-day average stock price.

Example: Window Functions in Queries

This query lists all products shipped in July and August 2005 and the cumulative shipped quantity by shipping date:

```
SELECT p.id, p.description, s.quantity, s.shipdate,
```

```
SUM(s.quantity) OVER (PARTITION BY productid ORDER BYs.shipdate rows
between unbounded preceding andcurrent row)FROM SalesOrderItems s
JOIN Products p on(s.ProductID =p.id) WHERE s.ShipDate BETWEEN
'2001-05-01' and'2001-08-31' AND s.quantity > 40ORDER BY p.id;
```

The results from the above query:

```
ID   description   quantity   ship_date   sum quantity
---  -----------   --------   ---------   ------------
302  Crew Neck           60  2001-07-02             60
400  Cotton Cap          60  2001-05-26             60
400  Cotton Cap          48  2001-07-05            108
401  Wool cap            48  2001-06-02             48
401  Wool cap            60  2001-06-30            108
401  Wool cap            48  2001-07-09            156
500  Cloth Visor         48  2001-06-21             48
501  Plastic Visor       60  2001-05-03             60
501  Plastic Visor       48  2001-05-18            108
501  Plastic Visor       48  2001-05-25            156
501  Plastic Visor       60  2001-07-07            216
601  Zipped Sweatshirt   60  2001-07-19             60
700  Cotton Shorts       72  2001-05-18             72
700  Cotton Shorts       48  2001-05-31            120
```

In this example, the computation of the **SUM** window function occurs after the join of the two tables and the application of the query's **WHERE** clause. The query uses an inline window specification that specifies that the input rows from the join is processed as follows:

1. Partition (group) the input rows based on the value of the `prod_id` attribute.
2. Within each partition, sort the rows by the `ship_date` attribute.
3. For each row in the partition, evaluate the **SUM()** function over the quantity attribute, using a sliding window consisting of the first (sorted) row of each partition, up to and including the current row.

An alternative construction for the query is to specify the window separate from the functions that use it. This is useful when more than one window function is specified that are based on the same window. In the case of the query using window functions, a construction that uses the window clause (declaring a window identified by cumulative) is as follows:

```
SELECT p.id, p.description, s.quantity, s.shipdate, SUM(s.quantity)
OVER(cumulative ROWS BETWEEN UNBOUNDED PRECEDING and CURRENT ROW )
cumulative FROM SalesOrderItems s JOIN Products p On (s.ProductID
=p.id)WHERE s.shipdate BETWEEN ‘2001-07-01’ and ‘2001-08-31’Window
cumulative as (PARTITION BY s.productid ORDER BY s.shipdate)ORDER BY
p.id;
```

The window clause appears before the **ORDER BY** clause in the query specification. When using a window clause, the following restrictions apply:

- The inline window specification cannot contain a **PARTITION BY** clause.
- The window specified within the window clause cannot contain a window frame clause.

  ```
  <WINDOW FRAME CLAUSE> ::=
    <WINDOW FRAME UNIT>
    <WINDOW FRAME EXTENT>
  ```
- Either the inline window specification, or the window specification specified in the window clause, can contain a window order clause, but not both.

  ```
  <WINDOW ORDER CLAUSE> ::= <ORDER SPECIFICATION>
  ```

Example: Window with multiple functions

This query defines a single (named) window and compute multiple function results over it:

```
SELECT p.ID, p.Description, s.quantity, s.ShipDate,SUM(s.Quantity)
OVER ws1, MIN(s.quantity) OVER ws1FROM SalesOrderItems s JOIN
Products p ON (s.ProductID =p.ID) WHERE s.ShipDate BETWEEN
'2000-01-09' AND'2000-01-17' AND s.Quantity > 40 window ws1
AS(PARTITION BY productid ORDER BY shipdate rowsbetween unbounded
preceding and current row)ORDER BY p.id;
```

The results from the above query:

```
ID   Description   quantity   shipDate     SUM   MIN
---  -----------   --------   -----------  ---   ---
400  Cotton Cap          48    2000-01-09   48    48
401  Wool cap            48    2000-01-09   48    48
500  Cloth Visor         60    2000-01-14   60    60
500  Cloth Visor         60    2000-01-15  120    60
501  Plastic Visor           60    2000-01-14   60    60
```

Example: Calculate cumulative sum

This query calculates a cumulative sum of salary per department and **ORDER BY** `start_date`.

```
SELECT dept_id, start_date, name, salary,
  SUM(salary) OVER (PARTITION BY dept_id ORDER BY
  start_date ROWS BETWEEN UNBOUNDED PRECEDING AND
  CURRENT ROW)
FROM emp1
ORDER BY dept_id, start_date;
```

The results from the above query:

```
DepartmentID  start_date    name     salary       sum

                                                    (salary)

-------                 ----------    ----   ------   ---------
100                  1996-01-01       Anna
18000            18000
100                   1997-01-01       Mike
```

```
28000    46000
100                     1998-01-01    Scott     29000    75000
100                     1998-02-01    Antonia   22000    97000
100                     1998-03-12    Adam      25000   122000
100                     1998-12-01    Amy       18000   140000
200                     1998-01-01    Jeff      18000    18000
200                     1998-01-20    Tim       29000    47000
200                     1998-02-01    Jim       22000    69000
200                     1999-01-10    Tom       28000    97000
300                     1998-03-12    Sandy     55000    55000
300                     1998-12-01    Lisa      38000    93000
300                     1999-01-10    Peter     48000    141000
```

Example: Calculate moving average

This query generates the moving average of sales in three consecutive months. The size of the window frame is three rows: two preceding rows plus the current row. The window slides from the beginning to the end of the partition.

```
SELECT prod_id, month_num, sales, AVG(sales) OVER
  (PARTITION BY prod_id ORDER BY month_num ROWS
  BETWEEN 2 PRECEDING AND CURRENT ROW)
FROM sale WHERE rep_id = 1
ORDER BY prod_id, month_num;
```

The results from the above query:

```
prod_id   month_num      sales       avg(sales)
-------   ---------      ------      ----------
10                1        100          100.00
10                2        120          110.00
10                3        100          106.66
10                4        130          116.66
10                5        120          116.66
10                6        110          120.00
20                1         20           20.00
20                2         30           25.00
20                3         25           25.00
20                4         30           28.33
20                5         31           28.66
20                6         20           27.00
30                1         10           10.00
30                2         11           10.50
30                3         12           11.00
30                4          1            8.00
```

Example: ORDER BY results

In this example, the top **ORDER BY** clause of a query is applied to the final results of a window function. The **ORDER BY** in a window clause is applied to the input data of a window function.

```
SELECT prod_id, month_num, sales, AVG(sales) OVER
  (PARTITION BY prod_id ORDER BY month_num ROWS
  BETWEEN 2 PRECEDING AND CURRENT ROW)
```

```
FROM sale WHERE rep_id = 1
ORDER BY prod_id desc, month_num;
```

The results from the above query:

```
prod_id  month_num       sales   avg(sales)
-------  ---------       -----   ----------
30               1          10       10.00
30               2          11       10.50
30               3          12       11.00
30               4           1        8.00
20               1          20       20.00
20               2          30       25.00
20               3          25       25.00
20               4          30       28.33
20               5          31       28.66
20               6          20       27.00
10               1         100      100.00
10               2         120      110.00
10               3         100      106.66
10               4         130      116.66
10               5         120      116.66
10                   6         110       120.00
```

Example: Multiple aggregate functions in a query

This example calculates aggregate values against different windows in a query.

```
SELECT prod_id, month_num, sales, AVG(sales) OVER
  (WS1 ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS
  CAvg, SUM(sales) OVER(WS1 ROWS BETWEEN UNBOUNDED
  PRECEDING AND CURRENT ROW) AS CSum
FROM sale WHERE rep_id = 1 WINDOW WS1 AS (PARTITION BY
  prod_id
ORDER BY month_num)
ORDER BY prod_id, month_num;
```

The results from the above query:

```
prod_id  month_num    sales     CAvg     CSum
-------  ---------    -----     ----     ----
10               1      100   110.00      100
10               2      120   106.66      220
10               3      100   116.66      320
10               4      130   116.66      450
10               5      120   120.00      570
10               6      110   115.00      680
20               1       20    25.00       20
20               2       30    25.00       50
20               3       25    28.33       75
20               4       30    28.66      105
20               5       31    27.00      136
20               6       20    25.50      156
30               1       10    10.50       10
30               2       11    11.00       21
```

```
30                3       12     8.00       33
30             4        1      6.50       34
```

Example: Window frame comparing ROWS and RANGE

This query compares ROWS and RANGE. The data contain duplicate ROWS per the **ORDER BY** clause.

```
SELECT prod_id, month_num, sales, SUM(sales) OVER
  (ws1 RANGE BETWEEN 2 PRECEDING AND CURRENT ROW) AS
    Range_sum, SUM(sales) OVER
  (ws1 ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS
    Row_sum
FROM sale window ws1 AS (PARTITION BY prod_id ORDER BY
  month_num)
ORDER BY prod_id, month_num;
```

The results from the above query:

```
prod_id  month_num   sales   Range_sum   Row_sum
-------  ---------   -----   ----------  -------
10               1     100         250       100
10               1     150         250       250
10               2     120         370       370
10               3     100         470       370
10               4     130         350       350
10               5     120         381       350
10               5      31         381       281
10               6     110         391       261
20               1      20          20        20
20               2      30          50        50
20               3      25          75        75
20               4      30          85        85
20               5      31          86        86
20               6      20          81        81
30               1      10          10        10
30               2      11          21        21
30               3      12          33        33
30               4       1          25        24
30               4       1          25        14
```

Example: Window frame excludes current row

In this example, you can define the window frame to exclude the current row. The query calculates the sum over four rows, excluding the current row.

```
SELECT prod_id, month_num, sales, sum(sales) OVER
  (PARTITION BY prod_id ORDER BY month_num RANGE
  BETWEEN 6 PRECEDING AND 2 PRECEDING)
FROM sale
ORDER BY prod_id, month_num;
```

The results from the above query:

```
prod_id   month_num       sales    sum(sales)
-------   ---------       -----    ----------
```

```
10                1          100        (NULL)
10                1          150        (NULL)
10                2          120        (NULL)
10                3          100          250
10                4          130          370
10                5          120          470
10                5           31          470
10                6          110          600
20                1           20        (NULL)
20                2           30        (NULL)
20                3           25           20
20                4           30           50
20                5           31           75
20                6           20          105
30                1           10        (NULL)
30                2           11        (NULL)
30                3           12           10
30                4            1           21
30                    4           1           21
```

Example: Window frame for RANGE

This query illustrates the RANGE window frame. The number of rows used in the summation is variable.

```
SELECT prod_id, month_num, sales, SUM(sales) OVER
  (PARTITION BY prod_id ORDER BY month_num RANGE
  BETWEEN 1 FOLLOWING AND 3 FOLLOWING)
FROM sale
ORDER BY prod_id, month_num;
```

The results from the above query:

```
prod_id   month_num       sales    sum(sales)
-------   ---------       -----    ----------
10                1         100           350
10                1         150           350
10                2         120           381
10                3         100           391
10                4         130           261
10                5         120           110
10                5          31           110
10                6         110        (NULL)
20                1          20            85
20                2          30            86
20                3          25            81
20                4          30            51
20                5          31            20
20                6          20        (NULL)
30                1          10            25
30                2          11            14
30                3          12             2
30                4           1        (NULL)
30                4           1        (NULL)
```

Example: Unbounded preceding and unbounded following

In this example, the window frame can include all rows in the partition. The query calculates max(sales) sale over the entire partition (no duplicate rows in a month).

```
SELECT prod_id, month_num, sales, SUM(sales) OVER
  (PARTITION BY prod_id ORDER BY month_num ROWS
  BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
FROM sale WHERE rep_id = 1
ORDER BY prod_id, month_num;
```

The results from the above query:

```
prod_id   month_num       sales    SUM(sales)
-------   ---------       -----    ----------
10                1         100           680
10                2         120           680
10                3         100           680
10                4         130           680
10                5         120           680
10                6         110           680
20                1          20           156
20                2          30           156
20                3          25           156
20                4          30           156
20                5          31           156
20                6          20           156
30                1          10            34
30                2          11            34
30                3          12            34
30                4           1            34
```

The query in this example is equivalent to:

```
SELECT prod_id, month_num, sales, SUM(sales) OVER
  (PARTITION BY prod_id )
FROM sale WHERE rep_id = 1
ORDER BY prod_id, month_num;
```

Example: Default window frame for RANGE

This query illustrates the default window frame for RANGE:

```
SELECT prod_id, month_num, sales, SUM(sales) OVER
  (PARTITION BY prod_id ORDER BY month_num)
FROM sale
ORDER BY prod_id, month_num;
```

The results from the above query:

```
prod_id   month_num       sales    SUM(sales)
-------   ---------       -----    ----------
10                1         100           250
10                1         150           250
10                2         120           370
10                3         100           470
```

```
10               4         130           600
10               5         120           751
10               5          31           751
10               6         110           861
20               1          20            20
20               2          30            50
20               3          25            75
20               4          30           105
20               5          31           136
20               6          20           156
30               1          10            10
30               2          11            21
30               3          12            33
30               4           1            35
30               4           1            35
```

The query in this example is equivalent to:

```
SELECT prod_id, month_num, sales, SUM(sales) OVER
  (PARTITION BY prod_id ORDER BY month_num RANGE
  BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
FROM sale
ORDER BY prod_id, month_num;
```

BNF Grammar for OLAP Functions

The Backus-Naur Form grammar outlines the specific syntactic support for the various ANSI SQL analytic functions, many of which are implemented in Sybase IQ.

Grammar rule 1

```
<SELECT LIST EXPRESSION> ::=
  <EXPRESSION>
  | <GROUP BY EXPRESSION>
  | <AGGREGATE FUNCTION>
  | <GROUPING FUNCTION>
  | <TABLE COLUMN>
  | <WINDOWED TABLE FUNCTION>
```

Grammar rule 2

```
<QUERY SPECIFICATION> ::=
  <FROM CLAUSE>
  [ <WHERE CLAUSE> ]
  [ <GROUP BY CLAUSE> ]
  [ <HAVING CLAUSE> ]
  [ <WINDOW CLAUSE> ]
[ <ORDER BY CLAUSE> ]
```

Grammar rule 3

```
<ORDER BY CLAUSE> ::= <ORDER SPECIFICATION>
```

Grammar rule 4

```
<GROUPING FUNCTION> ::=
  GROUPING <LEFT PAREN> <GROUP BY EXPRESSION>
  <RIGHT PAREN>
```

Grammar rule 5

```
<WINDOWED TABLE FUNCTION> ::=
  <WINDOWED TABLE FUNCTION TYPE> OVER <WINDOW NAME OR
  SPECIFICATION>
```

Grammar rule 6

```
<WINDOWED TABLE FUNCTION TYPE> ::=
  <RANK FUNCTION TYPE> <LEFT PAREN> <RIGHT PAREN>
  | ROW_NUMBER <LEFT PAREN> <RIGHT PAREN>
  | <WINDOW AGGREGATE FUNCTION>
```

Grammar rule 7

```
<RANK FUNCTION TYPE> ::=
  RANK | DENSE RANK | PERCENT RANK | CUME_DIST
```

Grammar rule 8

```
<WINDOW AGGREGATE FUNCTION> ::=
  <SIMPLE WINDOW AGGREGATE FUNCTION>
  | <STATISTICAL AGGREGATE FUNCTION>
```

Grammar rule 9

```
<AGGREGATE FUNCTION> ::=
  <DISTINCT AGGREGATE FUNCTION>
  | <SIMPLE AGGREGATE FUNCTION>
  | <STATISTICAL AGGREGATE FUNCTION>
```

Grammar rule 10

```
<DISTINCT AGGREGATE FUNCTION> ::=
  <BASIC AGGREGATE FUNCTION TYPE> <LEFT PAREN>
  <DISTINCT> <EXPRESSION> <RIGHT PAREN>
  | LIST <LEFT PAREN> DISTINCT <EXPRESSION>
  [ <COMMA> <DELIMITER> ]
  [ <ORDER SPECIFICATION> ] <RIGHT PAREN>
```

Grammar rule 11

```
<BASIC AGGREGATE FUNCTION TYPE> ::=
  SUM | MAX | MIN | AVG | COUNT
```

Grammar rule 12

```
<SIMPLE AGGREGATE FUNCTION> ::=
  <SIMPLE AGGREGATE FUNCTION TYPE> <LEFT PAREN>
  <EXPRESSION> <RIGHT PAREN>
  | LIST <LEFT PAREN> <EXPRESSION> [ <COMMA>
  <DELIMITER> ]
  [ <ORDER SPECIFICATION> ] <RIGHT PAREN>
```

Grammar rule 13

```
<SIMPLE AGGREGATE FUNCTION TYPE> ::= <SIMPLE WINDOW AGGREGATE
FUNCTION TYPE>
```

Grammar rule 14

```
<SIMPLE WINDOW AGGREGATE FUNCTION> ::=
  <SIMPLE WINDOW AGGREGATE FUNCTION TYPE> <LEFT PAREN>
  <EXPRESSION> <RIGHT PAREN>
| GROUPING FUNCTION
```

Grammar rule 15

```
<SIMPLE WINDOW AGGREGATE FUNCTION TYPE> ::=
  <BASIC AGGREGATE FUNCTION TYPE>
  | STDDEV | STDDEV_POP | STDDEV_SAMP
  | VARIANCE | VARIANCE_POP | VARIANCE_SAMP
```

Grammar rule 16

```
<STATISTICAL AGGREGATE FUNCTION> ::=
  <STATISTICAL AGGREGATE FUNCTION TYPE> <LEFT PAREN>
  <DEPENDENT EXPRESSION> <COMMA> <INDEPENDENT
  EXPRESSION> <RIGHT PAREN>
```

Grammar rule 17

```
<STATISTICAL AGGREGATE FUNCTION TYPE> ::=
  CORR | COVAR_POP | COVAR_SAMP | REGR_R2 |
  REGR_INTERCEPT | REGR_COUNT | REGR_SLOPE |
  REGR_SXX | REGR_SXY | REGR_SYY | REGR_AVGY |
  REGR_AVGX
```

Grammar rule 18

```
<WINDOW NAME OR SPECIFICATION> ::=
  <WINDOW NAME> | <IN-LINE WINDOW SPECIFICATION>
```

Grammar rule 19

```
<WINDOW NAME> ::= <IDENTIFIER>
```

Grammar rule 20

```
<IN-LINE WINDOW SPECIFICATION> ::= <WINDOW SPECIFICATION>
```

Grammar rule 21

```
<WINDOW CLAUSE> ::= <WINDOW WINDOW DEFINITION LIST>
```

Grammar rule 22

```
<WINDOW DEFINITION LIST> ::=
  <WINDOW DEFINITION> [ { <COMMA> <WINDOW DEFINITION>
  } . . . ]
```

Grammar rule 23

```
<WINDOW DEFINITION> ::=
  <NEW WINDOW NAME> AS <WINDOW SPECIFICATION>
```

Grammar rule 24

```
<NEW WINDOW NAME> ::= <WINDOW NAME>
```

Grammar rule 25

```
<WINDOW SPECIFICATION> ::=
  <LEFT PAREN> <WINDOW SPECIFICATION> <DETAILS> <RIGHT
  PAREN>
```

Grammar rule 26

```
<WINDOW SPECIFICATION DETAILS> ::=
  [ <EXISTING WINDOW NAME> ]
  [ <WINDOW PARTITION CLAUSE> ]
  [ <WINDOW ORDER CLAUSE> ]
  [ <WINDOW FRAME CLAUSE> ]
```

Grammar rule 27

```
<EXISTING WINDOW NAME> ::= <WINDOW NAME>
```

Grammar rule 28

```
<WINDOW PARTITION CLAUSE> ::=
  PARTITION BY <WINDOW PARTITION EXPRESSION LIST>
```

Grammar rule 29

```
<WINDOW PARTITION EXPRESSION LIST> ::=
  <WINDOW PARTITION EXPRESSION>
  [ { <COMMA> <WINDOW PARTITION EXPRESSION> } . . . ]
```

Grammar rule 30

```
<WINDOW PARTITION EXPRESSION> ::= <EXPRESSION>
```

Grammar rule 31

```
<WINDOW ORDER CLAUSE> ::= <ORDER SPECIFICATION>
```

Grammar rule 32

```
<WINDOW FRAME CLAUSE> ::=
  <WINDOW FRAME UNIT>
  <WINDOW FRAME EXTENT>
```

Grammar rule 33

```
<WINDOW FRAME UNIT> ::= ROWS | RANGE
```

Grammar rule 34

```
<WINDOW FRAME EXTENT> ::= <WINDOW FRAME START> | <WINDOW FRAME
BETWEEN>
```

Grammar rule 35

```
<WINDOW FRAME START> ::=
  UNBOUNDED PRECEDING
  | <WINDOW FRAME PRECEDING>
  | CURRENT ROW
```

Grammar rule 36

```
<WINDOW FRAME PRECEDING> ::= <UNSIGNED VALUE SPECIFICATION>
PRECEDING
```

Grammar rule 37

```
<WINDOW FRAME BETWEEN> ::=
  BETWEEN <WINDOW FRAME BOUND 1> AND <WINDOW FRAME
  BOUND 2>
```

Grammar rule 38

```
<WINDOW FRAME BOUND 1> ::= <WINDOW FRAME BOUND>
```

Grammar rule 39

```
<WINDOW FRAME BOUND 2> ::= <WINDOW FRAME BOUND>
```

Grammar rule 40

```
<WINDOW FRAME BOUND> ::=
  <WINDOW FRAME START>
  | UNBOUNDED FOLLOWING
  | <WINDOW FRAME FOLLOWING>
```

Grammar rule 41

```
<WINDOW FRAME FOLLOWING> ::= <UNSIGNED VALUE SPECIFICATION>
FOLLOWING
```

Grammar rule 42

```
<GROUP BY EXPRESSION> ::= <EXPRESSION>
```

Grammar rule 43

```
<SIMPLE GROUP BY TERM> ::=
  <GROUP BY EXPRESSION>
  | <LEFT PAREN> <GROUP BY EXPRESSION> <RIGHT PAREN>
  | <LEFT PAREN> <RIGHT PAREN>
```

Grammar rule 44

```
<SIMPLE GROUP BY TERM LIST> ::=
  <SIMPLE GROUP BY TERM> [ { <COMMA> <SIMPLE GROUP BY
  TERM> } . . . ]
```

Grammar rule 45

```
<COMPOSITE GROUP BY TERM> ::=
  <LEFT PAREN> <SIMPLE GROUP BY TERM>
  [ { <COMMA> <SIMPLE GROUP BY TERM> } . . . ]
  <RIGHT PAREN>
```

Grammar rule 46

```
<ROLLUP TERM> ::= ROLLUP <COMPOSITE GROUP BY TERM>
```

Grammar rule 47

```
<CUBE TERM> ::= CUBE <COMPOSITE GROUP BY TERM>
```

Grammar rule 48

```
<GROUP BY TERM> ::=
  <SIMPLE GROUP BY TERM>
  | <COMPOSITE GROUP BY TERM>
  | <ROLLUP TERM>
  | <CUBE TERM>
```

Grammar rule 49

```
<GROUP BY TERM LIST> ::=
  <GROUP BY TERM> [ { <COMMA> <GROUP BY TERM> } … ]
```

Grammar rule 50

```
<GROUP BY CLAUSE> ::= GROUP BY <GROUPING SPECIFICATION>
```

Grammar rule 51

```
<GROUPING SPECIFICATION> ::=
  <GROUP BY TERM LIST>
  | <SIMPLE GROUP BY TERM LIST> WITH ROLLUP
  | <SIMPLE GROUP BY TERM LIST> WITH CUBE
  | <GROUPING SETS SPECIFICATION>
```

Grammar rule 52

```
<GROUPING SETS SPECIFICATION> ::=
  GROUPING SETS <LEFT PAREN> <GROUP BY TERM LIST>
  <RIGHT PAREN>
```

Grammar rule 53

```
<ORDER SPECIFICATION> ::= ORDER BY <SORT SPECIFICATION LIST>
  <SORT SPECIFICATION LIST> ::= <SORT SPECIFICATION>
  [ { <COMMA> <SORT SPECIFICATION> } . . . ]
  <SORT SPECIFICATION> ::= <SORT KEY>
  [ <ORDERING SPECIFICATION> ] [ <NULL ORDERING> ]
  <SORT KEY> ::= <VALUE EXPRESSION>
  <ORDERING SPECIFICATION> ::= ASC | DESC
  <NULL ORDERING> := NULLS FIRST | NULLS LAST
```

See also

- *Distribution Functions* on page 59
- *OLAP Benefits* on page 22
- *OLAP Evaluation* on page 22
- *Ranking Functions* on page 48
- *Statistical Aggregate Functions* on page 54
- *Windowing* on page 37
- *Windowing Aggregate Functions* on page 52

Sybase IQ as a Data Server

Sybase IQ supports client application connections through either ODBC or JDBC. This chapter describes how to use Sybase IQ as a data server for client applications.

With certain limitations, Sybase IQ may also appear to certain client applications as an Open Server.This chapter also briefly describes the restrictions for creating and running these applications.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - Programming > SQL Anywhere Data Access APIs > Sybase Open Client API > Open Client architecture*.

The facilities described in this chapter do not provide remote data access for IQ users on Windows and Sun Solaris systems. Remote data access is provided by Component Integration Services (CIS), the core interoperability feature of OmniConnect™.

Client/Server Interfaces to Sybase IQ

To simplify, use a Sybase application or a third-party client application with Sybase IQ.

An understanding of how these pieces fit together may be helpful for configuring your database and setting up applications. This section explains how the pieces fit together. For more details about third-party client applications, see the *Installation and Configuration Guide*.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Replication > Using SQL Anywhere as an Open Server > Open Clients, Open Servers, and TDS*.

Configuring IQ Servers with iqdsedit

Sybase IQ can communicate with other Adaptive Servers, Open Server applications, and client software on the network.

Clients can talk to one or more servers, and servers can communicate with other servers via remote procedure calls. In order for products to interact with one another, each needs to know where the others reside on the network. This network service information is stored in the interfaces file.

Note: Sybase IQ provides versions of Open Client utilities that have limited functionality to enable INSERT...LOCATION, including:

- iqisql
- iqdsedit
- iqdscp (UNIX only)
- iqocscfg (Windows only)

The Interfaces File

When you use an Open Client™ program to connect to a database server, the program looks up the server name in the interfaces file and then connects to the server using the address.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Replication > Using SQL Anywhere as an Open Server > Configuring Open Servers > The interfaces file*.

iqdsedit Database Administration Utility

The **iqdsedit** utility allows you to configure the interfaces file (interfaces or SQL.ini).

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Replication > Using SQL Anywhere as an Open Server > Configuring Open Servers > Using the DSEdit utility*.

Starting iqdsedit

On Windows, the **iqdsedit** executable is in the `%SYBASE%\IQ-15_3\bin32` or `%SYBASE%\IQ-15_3\bin64` directories, which is automatically added to your path during installation.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Replication > Using SQL Anywhere as an Open Server > Configuring Open Servers > Starting DSEdit*.

Opening a Directory Services Session

You can add, modify, or delete entries for servers, including Sybase IQ servers in the Select Directory Service window.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Replication > Using SQL Anywhere as an Open Server > Configuring Open Servers > Opening a directory services session*.

Adding a Server Entry

The server entry appears in the Server field. To specify the attributes of the server, you must modify the entry.

The server name entered here does not need to match the name provided on the Sybase IQ command line. The server address, not the server name, is used to identify and locate the server.

The server name field is purely an identifier for Open Client. For Sybase IQ, if the server has more than one database loaded, the IQDSEDIT server name entry identifies which database to use.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Replication > Using SQL Anywhere as an Open Server > Configuring Open Servers > Adding a server entry*.

Adding or Changing a Server Address

Once you have entered a Server Name, you need to modify the Server Address to complete the interfaces file entry.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Replication > Using SQL Anywhere as an Open Server > Configuring Open Servers > Adding or changing the server address*.

Port number

The port number you enter must match the port specified on the Sybase IQ database server command line. The default port number for the Sybase IQ server is 2638.

The following are valid server address entries:

```
elora,2638
123.85.234.029,2638
```

See also

- *Starting the Database Server as an Open Server* on page 85

Verifying a Server Address

On Windows, you can verify your network connection by using the **Ping** server command from the Server Object menu.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Replication > Using SQL Anywhere as an Open Server > Configuring Open Servers > Verifying the server address*.

Renaming a Server Entry

You can rename server entries from the `dsedit` session window.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Replication > Using SQL Anywhere as an Open Server > Configuring Open Servers > Renaming a server entry*.

Deleting Server Entries

You can delete server entries from the `dsedit` session window.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Replication > Using SQL Anywhere as an Open Server > Configuring Open Servers > Deleting server entries*.

Sybase Applications and Sybase IQ

The ability of Sybase IQ to act as an Open Server enables Sybase applications such as OmniConnect to work with Sybase IQ.

To use the Open Client libraries, the client application must use only the supported system tables, views, and stored procedures.

OmniConnect support

Sybase OmniConnect provides a unified view of disparate data within an organization, allowing users to access multiple data sources without having to know what the data looks like or where it is located. In addition, OmniConnect performs heterogeneous joins of data across the enterprise, enabling cross-platform table joins of targets such as DB2, Sybase Adaptive Server® Enterprise, SQL Anywhere, Oracle, and VSAM.

Using the Open Server interface, Sybase IQ can act as a data source for OmniConnect.

Open Client Applications and Sybase IQ

You can build Open Client applications to access data in Sybase IQ base tables using the Open Client libraries directly from a C or C++ programming environment such as PowerSoft Power ++™. If such applications reference catalog tables, views, or system stored procedures, these objects must be supported by both Adaptive Server Enterprise (Transact-SQL™ syntax) and Sybase IQ.

See *Reference: Building Blocks, Tables, and Procedures > Appendix A, "Compatibility with Other Sybase Databases*.

Configuring Open Client

When connecting to Sybase IQ using Open Client or when using the **INSERT...LOCATION** syntax, you can set various Open Client configuration parameters in an Open Client runtime configuration (`.cfg`) file.

For example, you can change the maximum default number of connections, which is controlled by the value of the **CS_MAX_CONNECT** option.

The application name for **INSERT...LOCATION** is `Sybase IQ`. (The space between the words is required.) This application name is set at the Open Client connection level, not at the Open Client context level. For details about using an Open Client runtime configuration file and the options available, see the Open Client *Client-Library C Reference Manual*.

To have the `.cfg` take effect, stop and restart the Sybase IQ server. You may also specify certain configuration parameters in the **INSERT...LOCATION** command line. Parameters set in **INSERT...LOCATION** are superseded by parameters set in the configuration file.

When used as a remote server, Sybase IQ supports Tabular Data Steam (TDS) password encryption. The Sybase IQ server accepts a connection with an encrypted password sent by the client. For information on connection properties to set for password encryption, see *Open*

Server 15.5 > Open Client Client-Library/C Reference Manual > Client-Library Topics > Security features > Adaptive Server Enterprise security features > Security handshaking: encrypted password.

Note: Password encryption requires Open Client 15.0. TDS password encryption requires Open Client 15.0 ESD #7 or later.

To enable the Sybase IQ server to accept a jConnect connection with an encrypted password, set the jConnect **ENCRYPT_PASSWORD** connection property to true.

Sybase IQ as an Open Server

This section describes how to set up an Sybase IQ server to receive connections from Open Client applications.

System Requirements

There are separate requirements at the client and server for using Sybase IQ as an Open Server.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Replication > Using SQL Anywhere as an Open Server > Setting up SQL Anywhere as an Open Server > System requirements*.

Note: When connecting to a remote Sybase IQ from a local SQL Anywhere Enterprise server using OmniConnect, use these server classes:

- To connect to Sybase IQ 12 or later, use server classes `asaodbc` and `sajdbc`.
- To connect to Sybase IQ 11.x, use server class `asiq`.

Starting the Database Server as an Open Server

If you want to use Sybase IQ as an Open Server, you must ensure that it is started using the TCP/IP protocol.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Replication > Using SQL Anywhere as an Open Server > Setting up SQL Anywhere as an Open Server > Starting the database server as an Open Server*.

Every application using TCP/IP on a machine uses a distinct TCP/IP *port*, so that network packets are sent to the correct application. The default port for Sybase IQ is port 2638, which is used for shared memory communications. You can specify a different port number:

```
start_iq -x tcpip{port=2629} -n myserver iqdemo.db
```

See also

- *Adding or Changing a Server Address* on page 83

Configuring Your Database for Use with Open Client

Your database must be Sybase IQ 12.0 or later.

If you are using Sybase IQ together with Adaptive Server Enterprise, ensure that your database is created for maximum compatibility with Adaptive Server Enterprise.

When connecting to Sybase IQ as an Open Server, applications frequently assume services they expect under Adaptive Server Enterprise are provided. These services are not always present.

See *Reference: Building Blocks, Tables, and Procedures > Appendix A, "Compatibility with Other Sybase Databases"*.

Characteristics of Open Client and jConnect Connections

When Sybase IQ is serving applications over TDS, it automatically sets relevant database options to values that are compatible with SQL Anywhere Server default behavior. These options are set temporarily, for the duration of the connection only. The client application can override these options at any time.

Note: Sybase IQ does not support the `ANSI_BLANKS`, `FLOAT_AS_DOUBLE`, and `TSQL_HEX_CONSTANT` options.

Although Sybase IQ allows longer user names and passwords, TDS client user names and passwords cannot exceed 30 bytes. If your password or user ID is longer than 30 bytes, attempts to connect over TDS (for example, using jConnect) return an `Invalid user ID or password` error.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Replication > Using SQL Anywhere as an Open Server > Setting up SQL Anywhere as an Open Server > Characteristics of Open Client and jConnect connections*.

Note: ODBC applications, including Interactive SQL applications, automatically set certain database options to values mandated by the ODBC specification. This overwrites settings by the `LOGIN_PROCEDURE` database option. For details and a workaround, see *Reference: Statements and Options > LOGIN_PROCEDURE option*.

Servers with Multiple Databases

Using Open Client Library, you can connect to a specific database on a server containing multiple databases.

- Set up entries in the `interfaces` file for the server.
- Use the **-n** parameter on the **start_iq** command to set up a shortcut for the database name.

- Specify the **-S** *database_name* parameter with the database name on the **isql** command. This parameter is required whenever you connect.

You can run the same program against multiple databases without changing the program itself by putting the shortcut name into the program and merely changing the shortcut definition.

For example, the following `interfaces` file excerpt defines two servers, `live_sales` and `test_sales`:

```
live_sales
```

```
     query tcp ether myhostname 5555
    master tcp ether myhostname 5555
```

```
test_sales
```

```
     query tcp ether myhostname 7777
    master tcp ether myhostname 7777
```

Start the server and set up an alias for a particular database. The following command sets `live_sales` equivalent to `salesbase.db`:

```
start_iq -n sales_live <other parameters> -x \ ‘tcpip{port=5555}’
salesbase.db -n live_sales
```

To connect to the `live_sales` server:

```
isql -Udba -Psql -Slive_sales
```

A server name may only appear once in the `interfaces` file. Because the connection to Sybase IQ is now based on the database name, the database name must be unique. If all your scripts are set up to work on `salesbase` database, you will not have to modify them to work with `live_sales` or `test_sales`.

Accessing Remote Data

Sybase IQ can access data located on separate servers, both Sybase and non-Sybase, as if the data were stored on the local server.

Sybase IQ and Remote Data

SQL Anywhere remote data access gives you access to data in other data sources. You can use this feature to migrate data into a SQL Anywhere database. You can also use the feature to query data across databases.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Characteristics of Open Client and jConnect connections*.

Requirements for Accessing Remote Data

There are several basic elements required to access remote data.

Remote table mappings

Sybase IQ presents tables to a client application as if all the data in the tables were stored in the database to which the application is connected.

Internally, when Sybase IQ executes a query involving remote tables, it determines the storage location and accesses the remote location to retrieve data.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Remote table mappings*.

Server Classes

A *server class* is assigned to each remote server. The server class specifies the access method used to interact with the server. Different types of remote servers require different access methods.

The server classes provide Sybase IQ detailed server capability information. Sybase IQ adjusts its interaction with the remote server based on those capabilities.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Server classes*.

Note: OMNI JDBC classes are not supported with IPv6.

Remote Servers

Before remote objects can be mapped to a local proxy table, define the remote server where the remote object is located.

Create Remote Servers

Use the **CREATE SERVER** statement to set up remote server definitions.

For some systems, including Sybase IQ and SQL Anywhere, each data source describes a database, so a separate remote server definition is needed for each database.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Working with remote servers > Create remote servers using the CREATE SERVER statement*.

Loading Remote Data Without Native Classes

Load data by using DirectConnect.

Native classes use DirectConnect to access remote data sources:

- On 64-bit UNIX platforms
- On 32-bit platforms where no ODBC driver is available (for example, Microsoft SQL Server)

Loading MS SQL Server Data into an IQ Server on UNIX

This remote data example loads MS SQL Server data into an IQ server on UNIX.

For this example, assume that:

- An Enterprise Connect Data Access (ECDA) server named *mssql* exists on UNIX host *myhostname*, port 12530.
- The data is to be retrieved from an MS SQL server named *2000* on host *myhostname*, port 1433.

1. Using DirectConnect documentation, configure DirectConnect for your data source.
2. Make sure that ECDA server (*mssql*) is listed in the Sybase IQ interfaces file:

   ```
   mssql
   master tcp ether myhostname 12530
   query tcp ether myhostname 12530
   ```
3. Add a new user, using the user ID and password for server *mssql*:

   ```
   isql -Udba -Psql -Stst_iqdemo
   grant connect to chill identified by chill
   grant dba to chill
   ```
4. Log in as the new user to create a local table on Sybase IQ:

   ```
   isql -Uchill -Pchill -Stst_iqdemo
   create table billing(status char(1), name varchar(20), telno int)
   ```
5. Insert data:

   ```
   insert into billing location ‘mssql.pubs’ { select * from
   billing }
   ```

Querying Data Without Native Classes

Follow these guidelines to query data without native classes.

1. Configure ASE/CIS with a remote server and proxy to connect via DirectConnect. For example, use DirectConnect for Oracle to the Oracle server.
2. Configure Sybase IQ with a remote server using the ASEJDBC class to the ASE server. (The ASEODBC class is unavailable because there is no 64-bit Unix ODBC driver for ASE.)
3. Use the **CREATE EXISTING TABLE** statement to create proxy tables pointing to the proxy tables in ASE which in turn point to Oracle.

Querying Remote Data Using DirectConnect and Proxy Table from UNIX

Query data using DirectConnect.

This example shows how to access MS SQL Server data. For this example, assume the following:

- A Sybase IQ server on host *myhostname*, port 7594.
- An Adaptive Server Enterprise server on host *myhostname*, port 4101.
- An Enterprise Connect Data Access (ECDA) server exists named *mssql* on host *myhostname*, port 12530.
- The data is to be retrieved from an MS SQL server named *2000* on host *myhostname*, port 1433.

Setting Up Adaptive Server Enterprise to Query MS SQL Server

Set up Adaptive Server and Component Integration Services (CIS) to query MS SQL Server through DirectConnect.

For this example, assume that the server name is *jones_1207*.

1. Add an entry to the ASE interfaces file to connect to *mssql*:

   ```
   mssql
   ```

   ```
   master tcp ether hostname 12530
   ```

   ```
   query tcp ether hostname 12530
   ```

2. Enable CIS and remote procedure call handling from the ASE server. For example, if CIS is already enabled as the default:

   ```
   sp_configure 'enable cis'
   ```

   ```
   Parameter Name Default Memory Used Config Value Run Value
   ```

   ```
   enable cis                        1        0
   1            1
   ```

   ```
   (1 row affected)
   (return status=0)
   ```

```
sp_configure 'cis rpc handling', 1
```

```
Parameter Name Default Memory Used Config Value Run Value
```

```
enable cis                        0         0
0             1
```

```
(1 row affected)
Configuation option changed. The SQL Server need not be restarted
since the option is dynamic.
```

You may need to restart Adaptive Server Enterprise server after enabling CIS remote procedure call handling in older versions such as Sybase IQ 12.5.

3. Add the DirectConnect server to the ASE server's SYSSERVERS system table.

```
sp_addserver mssql, direct_connect, mssql
```

```
Adding server 'mssql', physical name 'mssql'
Server added.
(Return status=0)
```

4. Create the user in Adaptive Server Enterprise that will be used in Sybase IQ to connect to ASE.

```
sp_addlogin tst, tsttst
```

```
Password correctly set.
Account unlocked. New login created.
(return status = 0)
```

```
grant role sa_role to tst
use tst_db
sp_adduser tst
```

```
New user added.
(return status = 0)
```

5. Add an external login from the master database:

```
use master
sp_addexternlogin mssql, tst, chill, chill
```

```
User 'tst' will be known as 'chill' in remote server 'mssql'.
(return status = 0)
```

6. Create an ASE proxy table as the added user from the desired database:

```
isql -Utst -Ttsttst
use test_db
create proxy_table billing_tst at 'mssql.pubs..billing'
select * from billing_tst
```

```
status               name                telno
------               -----------               -----
D                BOTANICALLY              1
B                BOTANICALL              2
(2 rows affected)
```

Setting up Sybase IQ to Connect to the ASE Server
Follow these steps to query Adaptive Server Enterprise data.

1. Add an entry to the Sybase IQ interfaces file:

   ```
   jones_1207
   master tcp ether jones 4101
   query tcp ether jones 4101
   ```

2. Create the user to connect to ASE:

   ```
   grant connect to tst identified by tsttst
   grant dba to tst
   ```

3. Log in as the added user to create the 'asejdbc' server class and add external login:

   ```
   isql -Utst -Ptsttst -Stst_iqdemo
   create SERVER jones_1207 CLASS 'asejdbc' USING 'jones:4101/tst_db'
   create existing table billing_iq at
   'jones_1207.tst_db..billing_txt'
   select * from billing_iq
   ```

   ```
   status                name                telno
   ------                -----------               -----
   D                BOTANICALLY               1
   B                BOTANICALL               2
   (2 rows affected)
   ```

Delete Remote Servers

Use Sybase Central or a **DROP SERVER** statement to delete a remote server from the `ISYSSERVER` system table.

All remote tables defined on that server must already be dropped for this action to succeed.

See*SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Working with remote servers > Delete remote servers*.

Alter Remote Servers

Use the **ALTER SERVER** statement to modify the attributes of a server. These changes do not take effect until the next connection to the remote server.

See*SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Working with remote servers > Alter remote servers*.

List the Remote Tables On a Server

When configuring Sybase IQ, it is helpful to have access to a list of the remote tables available on a particular server.

See*SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Working with remote servers > List the remote tables on a server*.

See also *Reference: Building Blocks, Tables, and Procedures > sp_remote_tables system procedure* .

List Remote Server Capabilities

The **sp_servercaps** procedure displays information about a remote server's capabilities. Sybase IQ uses this capability information to determine how much of a SQL statement can be passed to a remote server.

See*SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Working with remote servers > List remote server capabilities*.

See also *Reference: Building Blocks, Tables, and Procedures > sp_servercaps system procedure*.

External Logins

Sybase IQ uses the names and passwords of its clients when it connects to a remote server on behalf of those clients. However, this behavior can be overridden by creating external logins.

External logins are alternate login names and passwords that are used when communicating with a remote server.

When Sybase IQ connects to the remote server, **INSERT...LOCATION** uses the remote login for the user ID of the current connection, if a remote login has been created with **CREATE EXTERNLOGIN** and the remote server has been defined with a **CREATE SERVER** statement. If the remote server is not defined, or a remote login has not been created for the user ID of the current connection, IQ connects using the user ID and password of the current connection. For more information and an example of **INSERT...LOCATION** using a remote login, see *Reference: Statements and Options > INSERT statement*.

If you are using an integrated login, the Sybase IQ name and password of the Sybase IQ client is the same as the database login ID and password that the Sybase IQ userid maps to in `syslogins`.

Create External Logins

Only the DBA account or an account with USER ADMIN authority can add or modify an external login.

See*SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Working with remote servers > Create external logins*.

For more information, see *Reference: Statements and Options > SQL Statements > CREATE EXTERNLOGIN Statement*.

Drop External Logins

Use the **DROP EXTERNLOGIN** statement to remove external logins from the Sybase IQ system tables.

See*SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Working with remote servers > Drop external logins*.

For more information, see *Reference: Statements and Options > SQL Statements > DROP EXTERNLOGIN Statement*.

Proxy Tables

Location transparency of remote data is enabled by creating a local proxy table that maps to the remote object.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Working with proxy tables*.

Specify Proxy Table Locations

The **AT** keyword is used with both **CREATE TABLE** and **CREATE EXISTING TABLE** to define the location of an existing object.

The location string has four components that are separated by either a period or a semicolon. Semicolons allow filenames and extensions to be used in the database and owner fields.

See*SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Working with proxy tables > Specify proxy table locations*.

Example

The following example illustrate the use of location strings:

- Sybase IQ:

```
'testiq..DBA.employee'
```

Create Proxy Tables

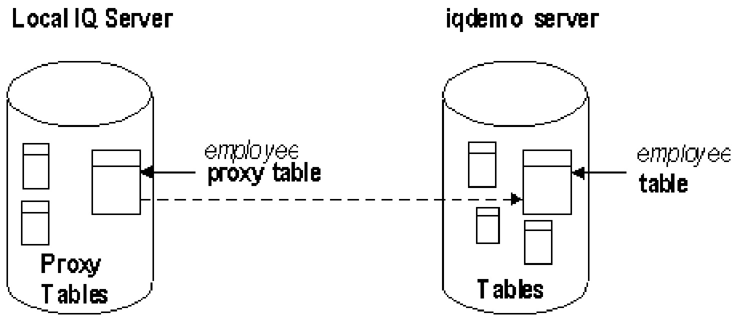
The **CREATE EXISTING TABLE** statement creates a proxy table that maps to an existing table on the remote server.

Sybase IQ derives the column attributes and index information from the object at the remote location.

Example

To create a proxy table named `p_employee` on the current server to a remote table named `employee` on the server named `iqdemo1`, use the following syntax:

```
CREATE EXISTING TABLE p_employee
AT 'iqdemo1..DBA.employee'
```



See *Reference: Statements and Options > CREATE EXISTING TABLE statement*.

CREATE TABLE Statement

The **CREATE TABLE** statement creates a new table on the remote server, and defines the proxy table for that table when you use the **AT** option.

Columns are defined using Sybase IQ data types. Sybase IQ automatically converts the data into the remote server's native types.

If you use the **CREATE TABLE** statement to create both a local and remote table, and then subsequently use the **DROP TABLE** statement to drop the proxy table, the remote table is also dropped. You can, however, use the **DROP TABLE** statement to drop a proxy table created using the **CREATE EXISTING TABLE** statement. In this case, the remote table is not dropped.

Example

The following statement creates a table named `Employees` on the remote server `iqdemo1`, and creates a proxy table named `members` that maps to the remote location:

```
CREATE TABLE members
( membership_id INTEGER NOT NULL,
member_name CHAR(30) NOT NULL,
office_held CHAR( 20 ) NULL)
AT 'iqdemo1..DBA.Employees'
```

For more information, see *Reference: Statements and Options > INSERT statement*.

List the Columns On a Remote Table

The **sp_remote_columns** system procedure produces a list of the columns on a remote table and a description of those data types.

If you are entering a **CREATE EXISTING TABLE** statement and you are specifying a column list, it may be helpful to get a list of the columns that are available on a remote table.

See*SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Working with proxy tables > List the columns on a remote table*.

For more information, see *Reference: Building Blocks, Tables, and Procedures > System Procedures > System Stored Procedures > Catalog Stored Procedures > sp_remote_columns System Procedure*.

Example: Join Between Two Remote Tables

The figure illustrates the remote Sybase IQ tables `employee` and `department` in the demo database, mapped to the local server named `testiq`.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Join remote tables*.

Multiple Local Databases

A Sybase IQ server may have several local databases running at one time. By defining tables in other local Sybase IQ databases as remote tables, you can perform cross-database joins.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Join tables from multiple local databases*.

Send Native Statements to Remote Servers

Use the **FORWARD TO** statement to send one or more statements to the remote server in its native syntax.

This statement can be used in two ways.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Send native statements to remote servers*.

Remote Procedure Calls (RPCs)

Sybase IQ users can issue procedure calls to remote servers that support the feature.

Sybase IQ, SQL Anywhere, and Adaptive Server Enterprise, as well as Oracle and DB2, support this feature. Issuing a remote procedure call is similar to using a local procedure call.

Create Remote Procedures

Use one of the procedures to issue a remote procedure call.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Using remote procedure calls (RPCs) > Create remote procedures*.

Transaction Management and Remote Data

Transactions provide a way to group SQL statements so that they are treated as a unit—either all work performed by the statements is committed to the database, or none of it is.

Transaction management with remote tables is handled somewhat differently than it is for local Sybase IQ tables. Transaction management for remote tables is handled for the most part as it is in SQL Anywhere, although there are some differences.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Creating Databases > Using transactions and isolation levels*.

For a general discussion of transactions in Sybase IQ, see *System Administration Guide: Volume 1 > Transactions and Versioning*.

Remote Transaction Management Overview

The method for managing transactions involving remote servers uses a *two-phase commit* protocol.

Sybase IQ implements a strategy that ensures transaction integrity for most scenarios.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Transaction management and remote data > Remote transaction management overview*.

Restrictions on Transaction Management

Transaction management has savepoints and nested statement restrictions.

Restrictions on transaction management are:

- Savepoints are not propagated to remote servers.
- If nested **BEGIN TRANSACTION** and **COMMIT TRANSACTION** statements are included in a transaction that involves remote servers, only the outermost set of statements is processed. The innermost set, containing the **BEGIN TRANSACTION** and **COMMIT TRANSACTION** statements, is not transmitted to remote servers.

Internal Operations

This section describes the underlying steps that SQL Anywhere performs on remote servers on behalf of client applications.

Query Parsing

When a statement is received from a client, the database server parses it. The database server raises an error if the statement is not a valid SQL Anywhere SQL statement.

Query Normalization

In query normalization, referenced objects are verified and data type compatibility is checked.

For example, consider this query:

```
SELECT *
FROM t1
WHERE c1 = 10
```

The query normalization stage verifies that table `t1` with a column `c1` exists in the system tables. It also verifies that the data type of column `c1` is compatible with the value 10. If the column's data type is `DATETIME`, for example, this statement is rejected.

Query Preprocessing

Query preprocessing prepares the query for optimization.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Internal operations > Query preprocessing*.

Server Capabilities

Each remote server defined to Sybase IQ has a set of capabilities associated with it. These capabilities are stored in the syscapabilities system table.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Internal operations > Server capabilities*.

Complete Passthrough of the Statement

The most efficient way to handle a statement is usually to pass as much of the original statement as possible to the remote server involved.

By default, Sybase IQ attempts to pass off as much of the statement as possible. In many cases, this is the complete statement as originally given to Sybase IQ.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Internal operations > Complete passthrough of the statement*.

Partial Passthrough of the Statement

If a statement contains references to multiple servers, or uses SQL features not supported by a remote server, the query is decomposed into simpler parts.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Internal operations > Partial passthrough of the statement*.

Remote Data Access Troubleshooting

This section provides some suggestions for troubleshooting access to remote servers.

Features Not Supported For Remote Data

Some features are never supported by Sybase IQ. Others are supported only for local data.

Sybase IQ has the following additions to the SQL Anywhere list:

- Java data types are not supported.
- When using Component Integration Services (CIS) in certain geographic regions, connection attempts return the error

  ```
  No Suitable Driver
  ```

  .

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Troubleshooting remote data access > Features not supported for remote data*.

Case Sensitivity

The case-sensitivity setting of your Sybase IQ database should match the settings used by any remote servers accessed.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Troubleshooting remote data access > Case sensitivity*.

Connectivity Problems

To verify you can connect to a remote server, perform a simple passthrough statement to a remote server to check your connectivity and remote login configuration.

For example:

```
FORWARD TO testiq {select @@version}
```

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Troubleshooting remote data access > Connectivity tests*.

General Problems with Queries

If you are faced with some type of problem with the way Sybase IQ is handling a query against a remote table, it is usually helpful to understand how Sybase IQ is executing that query.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Troubleshooting remote data access > General problems with queries*.

Managing Remote Data Access Connections

If you access remote databases via ODBC, the connection to the remote server is given a name.

The name can be used to drop the connection as one way to cancel a remote request.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Troubleshooting remote data access > Managing remote data access connections via ODBC*.

Server Classes for Remote Data Access

This chapter describes how Sybase IQ interfaces with various server classes.

Server Classes Overview

The behavior of a remote connection is determined by the server class in the **CREATE SERVER** statement. The server classes give Sybase IQ detailed server capability information.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Server classes for remote data access*.

JDBC-based Server Classes

JDBC-based server classes are used when Sybase IQ internally uses a Java virtual machine and jConnect™ for JDBC™ 5.5 to connect to the remote server.

The JDBC-based server classes are:

- Sybase IQ, and SQL Anywhere
- Sybase SQL Anywhere and Adaptive Server Enterprise (version 10 and later).

Configuration Notes for JDBC Classes

When you access remote servers defined with JDBC-based classes, consider the optimum performance, remote server access and remote server connection.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Server classes for remote data access > JDBC-based server classes > Configuration notes for JDBC classes*.

Server Class sajdbc

No special requirements exist for the configuration of Sybase IQ or SQL Anywhere data source.

Parameter Value in the CREATE SERVER Statement

The **USING** parameter in the **CREATE SERVER** statement takes the form *hostname:portnumber [/databasename]*.

Where:

- **hostname –** The machine that the remote server is running on

- **portnumber –** The TCP/IP port number that the remote server is listening on. The default port number that an Sybase IQ listens on is 2638.
- **databasename –** The Sybase IQ database that the connection will use. This is the name specified in the **-n** switch when the server was started, or in the **DBN** (**DatabaseName**) connection parameter.

Sybase IQ Example

To configure Sybase IQ server named `testiq` that is located on the machine apple and listening on port number 2638, use:

```
CREATE SERVER testiq
CLASS 'sajdbc'
USING 'apple:2638'
```

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Server classes for remote data access > JDBC-based server classes > Server class sajdbc > USING parameter in the CREATE SERVER statement*.

Server Class asejdbc

A server with server class asejdbc can be Adaptive Server Enterprise or SQL Anywhere Version 10 and later.

While Adaptive Server Enterprise data sources generally require no special configuration, ASE 15.5 requires the jConnect-6_0 metadata stored procedures and tables. See *Accessing Remote Data > Server Classes for Remote Data Access > JDBC-based Server Classes > Server Class aseodbc > Installing jConnect 6.0 Metadata*.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Server classes for remote data access > JDBC-based server classes > Server class asejdbc*.

Data Type Conversions

When you issue a **CREATE TABLE** statement to create a proxy table, Sybase IQ automatically converts the data types to the corresponding Adaptive Server Enterprise data types.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Server classes for remote data access > JDBC-based server classes > Server class asejdbc*.

Installing jConnect 6.0 Metadata

Proxy tables for Adaptive Server Enterprise 15.5 data require jConnect 6.0 metadata.

Without jConnect 6.0 metadata, a **CREATE EXISTING TABLE** statement may return the error `SQL Anywhere Error -667: Could not access column information for the table`.

Using **isql**:

1. Connect to your Adaptive Server Enterprise database.
2. Enter a command in this format:

   ```
   isql -I<path to interfaces>
   -Usa -P
   -S<ASE_server>
   -i$SYBASE/jConnect-6_0/sp/sql_server15.0.sql
   ```

ODBC-based Server Classes

Sybase IQ supports a variety of ODBC-based server classes.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Server classes for remote data access > ODBC-based server classes*.

ODBC External Servers

The most common method of defining an ODBC-based server is to base it on an ODBC data source.

To do this, you must create a data source name (DSN) in the ODBC Administrator.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Server classes for remote data access > ODBC-based server classes > Defining ODBC external servers*.

Sybase IQ example

A connection to Sybase IQ may be as follows:

```
CREATE SERVER testiq
CLASS 'asaodbc'
USING 'driver=adaptive server IQ 12.0;
eng=testasaiq;dbn=iqdemo;links=tcpip{}'
```

For more information on creating ODBC data sources for Sybase IQ, see *System Administration Guide: Volume 1 > Sybase IQ Connections > ODBC Data Sources*.

Server Class saodbc

To access SQL Anywhere database servers that support multiple databases, create an ODBC data source name defining a connection to each database. Issue a **CREATE SERVER** statement for each of these ODBC data source names.

A server with server class `saodbc` is one of:

- Sybase IQ version 12 or later
- SQL Anywhere

No special requirements exist for the configuration of a SQL Anywhere or Sybase IQ data source.

Server Class aseodbc

Sybase IQ requires the local installation of the Adaptive Server Enterprise ODBC driver and Open Client connectivity libraries to connect to a remote Adaptive Server with class `aseodbc`. However, the performance is better than with the `asejdbc` class.

A server with server class `aseodbc` is:

- Adaptive Server Enterprise
- SQL Anywhere (version 10 and later)

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Server classes for remote data access > ODBC-based server classes > Server class aseodbc*.

Server Class db2odbc

A server with server class db2odbc is IBM DB2.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Server classes for remote data access > ODBC-based server classes > Server class db2odbc*.

Server Class oraodbc

A server using server class oraodbc is Oracle version 10.0 or higher.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Server classes for remote data access > ODBC-based server classes > Server class oraodbc*.

Sybase IQ to Oracle Data Type Mappings

When you use a CREATE TABLE statement to create a remote table on an Oracle server, Sybase IQ converts the IQ data types to corresponding Oracle data types:

Table 5. Data mappings to a new remote Oracle table

Sybase IQ data type	Oracle data type
BIGINT	NUMBER(20,0)
BINARY(n)	if (n > 255) LONG RAW else RAW(n)
BIT	NUMBER(1,0)
CHAR(n)	If (n > 255) LONG else VARCHAR(n)
CHARACTER VARYING(n)	VARCHAR2(n)
CHARACTER(n)	VARCHAR2(n)

Sybase IQ data type	Oracle data type
DATE	DATE
DATETIME	DATE
DECIMAL(prec, scale)	NUMBER(prec, scale)
DOUBLE	FLOAT
FLOAT	FLOAT
INT	NUMBER(11,0)
LONG BINARY	LONG RAW
LONG VARCHAR	LONG or CLOB
MONEY	NUMBER(19,4)
NUMERIC(prec, scale)	NUMBER(prec, scale)
REAL	FLOAT
SMALLDATETIME	DATE
SMALLINT	NUMBER(5,0)
SMALLMONEY	NUMBER(10,4)
TIME	DATE
TIMESTAMP	DATE
TINYINT	NUMBER(3,0)
UNIQUEIDENTIFIERSTR	CHAR(36)
UNSIGNED BIGINT	NUMBER(20,0)
UNSIGNED INT	NUMBER(11,0)
UNSIGNED INTEGER	NUMBER(11,0)
VARBINARY(n)	if (n > 255) LONG RAW else RAW(n)
VARCHAR(n)	VARCHAR2(n)

Oracle to Sybase IQ Data Mappings

When you use a **CREATE EXISTING** statement to create a proxy table to an existing Oracle table, IQ converts the Oracle data types to corresponding IQ data types.

Table 6. Data mappings to existing Oracle tables

Oracle data type	IQ data type
BFILE	LONG BINARY
BLOB	LONG BINARY
CHAR(n)	CHAR(n)
CLOB	LONG VARCHAR
DATE	TIMESTAMP
DEC(prec, scale)	NUMERIC(prec, scale)
DECIMAL(prec, scale)	NUMERIC(prec, scale)
DOUBLE PRECISION	DOUBLE
FLOAT	DOUBLE
INT	NUMERIC(38,0)
INTEGER	NUMERIC(38,0)
NCHAR(n)	NCHAR(n)
NCLOB	LONG NVARCHAR
NUMBER(prec, scale)	NUMERIC(prec, scale)
NUMERIC(prec, scale)	NUMERIC(prec, scale)
NVARCHAR2(n)	VARCHAR(n)
RAW(n)	VARBINARY(n)
REAL	DOUBLE
SMALLINT	NUMERIC(38,0)
TIMESTAMP	TIMESTAMP
VARCHAR2(n)	VARCHAR(n)

Note:

- Sybase IQ lets you map proxy tables to Oracle views. Because Oracle identifiers always appear in upper case letters, you must use upper-case letters to create or refer to any proxy table that you map to an Oracle view.

- See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Server classes for remote data access > ODBC-based server classes > Server class oraodbc*.

Server Class mssodbc

A server with server class mssodbc is Microsoft SQL Anywhere version 6.5, Service Pack 4.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Server classes for remote data access > ODBC-based server classes > Server class mssodbc*.

Server Class odbc

ODBC data sources that do not have their own server class use server class odbc. You can use any ODBC driver.

The latest versions of Microsoft ODBC drivers can be obtained through the Microsoft Data Access Components (MDAC) distribution found at the Microsoft Download Center. The Microsoft driver versions listed are part of MDAC 2.0.

Microsoft Excel (Microsoft 3.51.171300)

Each Excel workbook is considered to be a database that holds several tables.

Tables are mapped to sheets in a workbook. When you configure an ODBC data source name in the ODBC driver manager, you specify a default workbook name associated with that data source, however, when you issue a **CREATE TABLE** statement, you can override the default and specify a workbook name in the location string.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Server classes for remote data access > ODBC-based server classes > Server class odbc > Microsoft Excel (Microsoft 3.51.171300)*.

Microsoft Foxpro (Microsoft 3.51.171300)

You can store Foxpro tables together inside a single Foxpro database file (`.dbc`), or you can store each table in its own separate `.dbf` file.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Server classes for remote data access > ODBC-based server classes > Server class odbc > Microsoft FoxPro (Microsoft 3.51.171300)*.

Lotus Notes SQL 2.0 (2.04.0203)

You can obtain Lotus Notes SQL 2.0 (2.04.0203) driver from the Lotus Web site.

Read the documentation that comes with it for an explanation of how Notes data maps to relational tables. You can easily map IQ tables to Notes forms.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Server classes for remote data access > ODBC-based server classes > Server class odbc > Lotus Notes SQL 2.0*.

Setting Up IQ to Access the Address Sample File

Set up IQ to access the address sample file.

1. Create an ODBC data source using the NotesSQL driver.

   The database will be the sample names file `c:\notes\data\names.nsf`. The Map Special Characters option should be turned on. For this example, the Data Source Name is *my_notes_dsn*.
2. Create an IQ server:

   ```
   CREATE SERVER names
   CLASS 'odbc'
   USING 'my_notes_dsn'
   ```
3. Map the Person form into an IQ table:

   ```
   CREATE EXISTING TABLE Person
   AT 'names...Person'
   ```
4. Query the table

   ```
   SELECT * FROM Person
   ```

Automating Tasks Using Schedules and Events

This chapter describes how to use scheduling and event handling features of Sybase IQ to automate database administration and other tasks.

Introduction to Scheduling and Event Handling

Many database administration tasks are best carried out systematically.

For example, a regular backup procedure is an important part of proper database administration procedures.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Maintaining Your Database > Automating tasks using schedules and events > Introduction to using schedules and events*.

Schedules

By scheduling activities you can ensure that a set of actions is executed at a set of preset times. The scheduling information and the event handler are both stored in the database itself.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Maintaining Your Database > Automating tasks using schedules and events > Understanding schedules*.

Sybase IQ Example

Note: For examples, use the Sybase IQ demo database `iqdemo.db`.

```
Create table OrderSummary(c1 date, c2 int);create event
Summarizeschedulestart time '6:00 pm'on ('Mon', 'Tue', 'Wed', 'Thu',
'Fri')handlerbegin   insert into DBA.OrderSummary   select
max(OrderDate), count(*)   from GROUPO.SalesOrders where OrderDate =
current dateend
```

Defining Schedules

For flexibility, schedule definitions are made up of several components.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Maintaining Your Database > Automating tasks using schedules and events > Understanding schedules > Defining schedules*.

Events

The database server tracks several kinds of system events. Event handlers are triggered when the system event is checked by the database server, and satisfies a provided *trigger condition*.

See*SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Maintaining Your Database > Automating tasks using schedules and events > Understanding system events*.

Choosing a System Event

Sybase IQ tracks several system events. Each system event provides a hook on which you can hang a set of actions.

The database server tracks the events for you, and executes the actions (as defined in the event handler) when needed.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Maintaining Your Database > Automating tasks using schedules and events > Understanding system events*.

Defining Trigger Conditions for Events

Each event definition has a system event associated with it. It also has one or more trigger conditions.

The event handler is triggered when the trigger conditions for the system event are satisfied.

Note: The trigger conditions associated with Sybase IQ events are not the same as SQL Anywhere or Adaptive Server Enterprise triggers, which execute automatically when a user attempts a specified data modification statement on a specified table.

See*SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Maintaining Your Database > Automating tasks using schedules and events > Understanding system events > Defining trigger conditions for events*.

Sybase IQ Example

Note: For examples, use the Sybase IQ demo database `iqdemo.db`.

```
create event SecurityCheck
type ConnectFailed
handler
begindeclare num_failures int;declare mins int;

insert into FailedConnections( log_time )values ( current
timestamp );
select count( * ) into num_failuresfrom FailedConnectionswhere
```

```
log_time >= dateadd( minute, -5,
    current timestamp );if( num_failures >= 3 ) then
    select datediff( minute, last_notification,        current
timestamp ) into mins    from Notification;
    if( mins > 30 ) then        update Notification        set
last_notification = current timestamp;        call
xp_sendmail( recipient='DBAdmin',               subject='Security
Check',"message"=        'over 3 failed connections in last 5
minutes' )    end ifend ifend
```

Event Handlers

Event handlers execute on a separate connection from the action that triggered the event, and so do not interact with client applications. They execute with the permissions of the creator of the event.

Developing Event Handlers

Event handlers, whether for scheduled events or for system event handling, contain compound statements, and are similar in many ways to stored procedures. You can add loops, conditional execution, and so on, and you can use the Sybase IQ debugger to debug event handlers.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Maintaining Your Database > Automating tasks using schedules and events > Understanding system events > Developing event handlers*.

The **EVENT_PARAMETER** function provides context information for event handlers. See *Reference: Building Blocks, Tables, and Procedures*.

For an example on using event handling, see *System Administration Guide: Volume 1 > Automating Tasks Using Schedules and Events > Managing user accounts and connections*.

Schedule and Event Internals

This section describes how the database server processes schedules and event definitions.

How the Database Server Checks for System Events

Events are classified according to their *event type*, as specified directly in the **CREATE EVENT** statement.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Maintaining Your Database > Automating tasks using schedules and events > Schedule and event intervals > How the database server checks for system events*.

How the Database Server Checks for Scheduled Times

The calculation of scheduled event times is done when the database server starts, and each time a scheduled event handler completes.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Maintaining Your Database > Automating tasks using schedules and events > Schedule and event intervals > How the database server checks for scheduled events*.

How Event Handlers are Executed

When an event handler is triggered, a temporary internal connection is made, on which the event handler is executed.

The handler is not executed on the connection that caused the handler to be triggered, and consequently statements such as **MESSAGE … TO CLIENT**, which interact with the client application, are not meaningful within event handlers.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Maintaining Your Database > Automating tasks using schedules and events > Event handling tasks > Adding an event to a database*.

Scheduling and Event Handling Tasks

This section collects together tasks related to automating schedules and events.

Adding a Schedule or Event to a Database

You can add schedules and events in Sybase Central and by using SQL.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Maintaining Your Database > Automating tasks using schedules and events > Event handling tasks > Adding an event to a database*.

Adding a Manually-triggered Event To a Database

If you create an event handler without a schedule or system event to trigger it, it is executed only when manually triggered.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Maintaining Your Database > Automating tasks using schedules and events > Event handling tasks > Adding a manually-triggered event to a database* .

Alter events using the **ALTER EVENT** statement. See *Reference: Statements and Options*.

Triggering an Event Handler

Any event handler can be manually triggered, as well as executed as a result of a schedule or system event. You may find it useful to manually trigger events during development of event handlers, and also, for certain events, in production environments.

For example, you may have a monthly sales report scheduled, but from time to time you may want to obtain a sales report for a reason other than the end of the month.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Maintaining Your Database > Automating tasks using schedules and events > Event handling tasks > Triggering an event handler*.

For more information on triggering, see the `TRIGGER EVENT` statement in *Reference: Statements and Options*.

Debugging an Event Handler

Debugging is a regular part of any software development. Event handlers can be debugged during the development process.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Maintaining Your Database > Automating tasks using schedules and events > Event handling tasks > Debugging an event handler*.

Retrieving Information about an Event or Schedule

Sybase IQ stores information about events, system events, and schedules in the system tables `SYSEVENT`, `SYSEVENTTYPE`, and `SYSSCHEDULE`.

When you alter an event using the **ALTER EVENT** statement, you specify the event name and, optionally, the schedule name. When you trigger an event using the **TRIGGER EVENT** statement, you specify the event name.

You can list event names by querying the system table `SYSEVENT`. For example:

```
SELECT event_id, event_name FROM SYSEVENT
```

You can list schedule names by querying the system table `SYSSCHEDULE`. For example:

```
SELECT event_id, sched_name FROM SYSSCHEDULE
```

Each event has a unique event ID. Use the event_id columns of `SYSEVENT` and `SYSSCHEDULE` to match the event to the associated schedule.

Data Access Using JDBC

This appendix describes how to use JDBC to access data.

JDBC can be used both from client applications and inside the database. Java classes using JDBC provide a more powerful alternative to SQL stored procedures for incorporating programming logic in the database.

JDBC Overview

JDBC provides a SQL interface for Java applications: if you want to access relational data from Java, you do so using JDBC calls.

Rather than a thorough guide to the JDBC database interface, this appendix provides some simple examples to introduce JDBC and illustrates how you can use it inside and outside the server. As well, this appendix provides more details on the server-side use of JDBC, running inside the database server.

The examples illustrate the distinctive features of using JDBC in Sybase IQ. For more information about JDBC programming, see any JDBC programming book.

JDBC and Sybase IQ

You can use JDBC with Sybase IQ in the following ways:

- Java client applications can make JDBC calls to Sybase IQ. The connection takes place through the Sybase jConnect JDBC driver or through the iAnywhere JDBC driver.

  In this appendix, the phrase *client application* applies both to applications running on a user's machine and to logic running on a middle-tier application server.
- JDBC in the server Java classes installed into a database can make JDBC calls to access and modify data in the database, using an internal JDBC driver.

The focus in this appendix is on server-side JDBC.

JDBC Resources

- Required Software

  You need TCP/IP to use the Sybase jConnect driver.

  The Sybase jConnect driver may already be available, depending on your installation of Sybase IQ.

See also

- *The jConnect Driver Files* on page 136

Choose JDBC Driver

Two JDBC drivers are provided for Sybase IQ:

Table 7.

Driver	Defintion
jConnect	This driver is a 100% pure Java driver. It communicates with Sybase IQ using the TDS client/ server protocol.
iAnywhere JDBC driver	This driver communicates with Sybase IQ using the Command Sequence client/server protocol. Its behavior is consistent with ODBC, embedded SQL, and OLE DB applications.

For jConnect documentation, see *jConnect for JDBC*.

When choosing which driver to use, you may want to consider the following factors:

- Features—Both drivers are JDK 2 compliant. The iAnywhere JDBC driver provides fully-scrollable cursors, which are not available in jConnect.
- Pure Java—The jConnect driver is a pure Java solution. The iAnywhere JDBC driver requires the Sybase IQ or Adaptive Server Anywhere ODBC driver and is not a pure Java solution.
- Performance—The iAnywhere JDBC driver provides better performance for most purposes than the jConnect driver.
- Compatibility—The TDS protocol used by the jConnect driver is shared with Adaptive Server Enterprise. Some aspects of the driver's behavior are governed by this protocol, and are configured to be compatible with Adaptive Server Enterprise.

Both drivers are available on Windows 95/98/Me and Windows NT/2000/2003/XP, as well as supported UNIX and Linux operating systems.

JDBC Considerations

Consider the following when running Java applications:

- An issue exists when connecting to a Sybase IQ 12.5 server through dbisql Java using the iAnywhere JDBC driver. For details, see *System Administration Guide: Volume 1 > Troubleshooting Hints > Data truncation or data conversion error*.
- Java applications running in Sybase IQ run slower than when run outside in a Sun Java Virtual Machine (JVM). Despite this limitation, Sybase recommends that you tune your applications by increasing the available memory for IQ JVM use with the database options `JAVA_HEAP_SIZE` and `JAVA_NAMESPACE_SIZE` in *Reference: Statements and Options > Database Options > JAVA_HEAP_SIZE*.

JDBC Program Structure

JDBC program structure has a sequenece of events.

The following sequence of events typically occur in JDBC applications:

- Create a Connection object—Calling a **getConnection** class method of the **DriverManager** class creates a **Connection** object, and establishes a connection with a database.
- Generate a Statement object—The **Connection** object generates a **Statement** object.
- Pass a SQL statement—A SQL statement that executed within the database environment passes to the **Statement** object. If the statement is a query, this action returns a **ResultSet** object.

  The **ResultSet** object contains the data returned from the SQL statement, but exposes it one row at a time (similar to the way a cursor works).
- Loop over the rows of the result set—The **next** method of the **ResultSet** object performs two actions:
  - The current row (the row in the result set exposed through the **ResultSet** object) advances one row.
  - A Boolean value (true/false) returns to indicate whether there is, in fact, a row to advance to.
- For each row, retrieve the values—Values are retrieved for each column in the **ResultSet** object by identifying either the name or position of the column. You can use the **getDate** method to get the value from a column on the current row.

Java objects can use JDBC objects to interact with a database and get data for their own use, for example to manipulate or for use in other queries.

Server-side JDBC Features

JDBC 1.2 is part of JDK 1.1. JDBC 2.0 is part of Java 2 (JDK 1.2).

Java in the database supplies a subset of the JDK version 1.1, so the internal JDBC driver supports JDBC version 1.2.

The internal JDBC driver (`asajdbc`) makes some features of JDBC 2.0 available from server-side Java applications, but does not provide full JDBC 2.0 support.

The JDBC classes in the **java.sql** package that is part of the Java in the database support are at level 1.2. Server-side features that are part of JDBC 2.0 are implemented in the **sybase.sql.ASA** package. To use JDBC 2.0 features you must cast your JDBC objects into the corresponding classes in the **sybase.sql.ASA** package, rather than the **java.sql** package. Classes that are declared as **java.sql** are restricted to JDBC 1.2 functionality only.

The classes in **sybase.sql.ASA** are as follows:

JDBC class	Sybase internal driver class
java.sql.Connection	sybase.sql.ASA.SAConnection
java.sql.Statement	sybase.sql.ASA.SAStatement
java.sql.PreparedStatement	sybase.sql.ASA.SAPreparedStatement
java.sql.CallableStatement	sybase.sql.ASA.SACallableStatement
java.sql.ResultSetMetaData	sybase.sql.ASA.SAResultSetMetaData
java.sql.ResultSet	sybase.sql.SAResultSet
java.sql.DatabaseMetaData	sybase.sql.SADatabaseMetaData

The following function provides a **ResultSetMetaData** object for a prepared statement without requiring a **ResultSet** or executing the statement. This function is not part of the JDBC standard.

```
ResultSetMetaData sybase.sql.ASA.SAPreparedStatement.describe()
```

JDBC 2.0 Restrictions

The following classes are part of the JDBC 2.0 core interface, but are not available in the **sybase.sql.ASA** package:

- java.sql.Blob
- java.sql.Clob
- java.sql.Ref
- java.sql.Struct
- java.sql.Array
- java.sql.Map

The following JDBC 2.0 core functions are not available in the **sybase.sql.ASA** package:

Class in sybase.sql.ASA	Missing functions
SAConnection	java.util.Map getTypeMap() void setTypeMap(java.util.Map map)
SAPreparedStatement	void setRef(int pidx, java.sql.Ref r) void setBlob(int pidx, java.sql.Blob b) void setClob(int pidx, java.sql.Clob c) void setArray(int pidx, java.sql.Array a)

Class in sybase.sql.ASA	Missing functions
SACallableStatement	Object getObject(pidx, java.util.Map map)
	java.sql.Ref getRef(int pidx)
	java.sql.Blob getBlob(int pidx)
	java.sql.Clob getClob(int pidx)
	java.sql.Array getArray(int pidx)
SAResultSet	Object getObject(int cidx, java.util.Map map)
	java.sql.Ref getRef(int cidx)
	java.sql.Blob getBlob(int cidx)
	java.sql.Clob getClob(int cidx)
	java.sql.Array getArray(int cidx)
	Object getObject(String cName, java.util.Map map)
	java.sql.Ref getRef(String cName)
	java.sql.Blob getBlob(String cName)
	java.sql.Clob getClob(String cName)
	java.sql.Array getArray(String cName)

Differences Between Client- and Server-side JDBC Connections

A difference between JDBC on the client and in the database server lies in establishing a connection with the database environment.

- Client side—In client-side JDBC, establishing a connection requires the Sybase jConnect JDBC driver. Passing arguments to the **DriverManager.getConnection** establishes the connection. The database environment is an external application from the perspective of the client application.
- Server-side—When using JDBC within the database server, a connection already exists. A value of **jdbc:default:connection** passes to **DriverManager.getConnection**, which provides the JDBC application with the ability to work within the current user connection. This is a quick, efficient and safe operation because the client application has already passed the database security to establish the connection. The user ID and password, having been provided once, do not need to be provided again. The asajdbc driver can only connect to the database of the current connection.

You can write JDBC classes in such a way that they can run both at the client and at the server by employing a single conditional statement for constructing the URL. An external connection requires the machine name and port number, while the internal connection requires **jdbc:default:connection**.

Establish JDBC Connections

This section presents classes that establish a JDBC database connection from a Java application.

Connect From a JDBC Client Application Using jConnect

If you wish to access database system tables (database metadata) from a JDBC application, you must add a set of jConnect system objects to your database.

If you wish to access database system tables (database metadata) from a JDBC application, you must add a set of jConnect system objects to your database. Asajdbc shares the same stored procedures for database metadata support with jConnect. These procedures are installed to all databases by default. The **iqinit** switch **-i** prevents this installation.

The **-i** switch is common to **iqinit** and the SQL Anywhere utility **dbinit**. For descriptions of the **-i** switch, see *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Administering Your Database > Database administration utilities > Initialization utility (dbinit)*.

The following complete Java application is a command-line application that connects to a running database, prints a set of information to your command line, and terminates.

Establishing a connection is the first step any JDBC application must take when working with database data.

See also

- *Establish Connection From a Server-side JDBC Class* on page 126
- *Sybase jConnect JDBC Driver* on page 135
- *Running the External Connection Example* on page 125
- *A Sample Distributed Application* on page 141

External Connection Example Code

The source code for the methods used to make a connection.

The source code can be found in the **main** method and the **ASAConnect** method of the file `JDBCExamples.java` in the `C:\Documents and Settings\All Users\SybaseIQ\samples\SQLAnywhere\JDBC` directory on Windows or `$SYBASE/IQ-15_3/samples/sqlanywhere/JDBC` on UNIX under your Sybase IQ installation directory:

```
// Import the necessary classes
import java.sql.*;              // JDBC
import com.sybase.jdbc.*;       // Sybase jConnect
import java.util.Properties;    // Properties
import sybase.sql.*;            // Sybase utilities
import asademo.*;               // Example classes
```

```
private static Connection conn;
public static void main(String args[])  {

    conn = null;
    String machineName;
    if ( args.length != 1 ) {
      machineName = "localhost";
    } else {
      machineName = new String( args[0] );
    }

    ASAConnect( "dba", "sql", machineName );
    if( conn!=null ) {
        System.out.println( "Connection successful" );
    }else{
        System.out.println( "Connection failed" );
    }

    try{
      serializeVariable();
      serializeColumn();
      serializeColumnCastClass();
    }
    catch( Exception e ) {
      System.out.println( "Error: " + e.getMessage() );
      e.printStackTrace();
    }
  }
}
private static void ASAConnect( String UserID,
                              String Password,
                              String Machinename ) {
    // uses global Connection variable

    String _coninfo = new String( Machinename );

    Properties _props = new Properties();
    _props.put("user", UserID );
    _props.put("password", Password );

    // Load the Sybase Driver
    try {
      Class.forName("com.sybase.jdbc.SybDriver").newInstance();

      StringBuffer temp = new StringBuffer();
      // Use the Sybase jConnect driver...
      temp.append("jdbc:sybase:Tds:");
      // to connect to the supplied machine name...
      temp.append(_coninfo);
      // on the default port number for ASA...
      temp.append(":2638");
      // and connect.
      System.out.println(temp.toString());
     conn = DriverManager.getConnection( temp.toString() , _props );
    }
    catch ( Exception e ) {
      System.out.println("Error: " + e.getMessage());
      e.printStackTrace();
    }
  }
```

How the External Connection Example Works

The external connection example is a Java command-line application.

Importing Packages

The application requires several libraries, which are imported in the first lines of `JDBCExamples.java`:

- The **java.sql** package contains the Sun Microsystems JDBC classes, which are required for all JDBC applications. You'll find it in the `classes.zip` file in your Java subdirectory.
- Imported from **com.sybase.jdbc**, the Sybase jConnect JDBC driver is required for all applications that connect using jConnect. You'll find it in the `jdbcdrv.zip` file in your Java subdirectory.
- The application uses a *property list*. The **java.util.Properties** class is required to handle property lists. You'll find it in the `classes.zip` file in your `Java` subdirectory.
- The **sybase.sql** package contains utilities used for serialization. You'll find it in the `asajdbc.zip` file in your `Java` subdirectory.
- The **asademo** package contains example classes used in some examples. You'll find it in the `asademo.jar` file in your `java` subdirectory.

The Main Method

Each Java application requires a class with a method named **main**, which is the method invoked when the program starts. In this simple example, **JDBCExamples.main** is the only method in the application.

The **JDBCExamples.main** method carries out the following tasks:

- Processes the command-line argument, using the machine name if supplied. By default, the machine name is *localhost*, which is appropriate for the personal database server.
- Calls the ASAConnect method to establish a connection.
- Executes several methods that scroll data to your command line.

The ASAConnect Method

The **JDBCExamples.ASAConnect** method carries out the following tasks:

- Connects to the default running database using Sybase jConnect.
  - **Class.forName** loads jConnect. Using the **newInstance** method works around issues in some browsers.

    - The **StringBuffer** statements build up a connection string from the literal strings and the supplied machine name provided on the command line.
    - **DriverManager.getConnection** establishes a connection using the connection string.
- Returns control to the calling method.

Running the External Connection Example

This section describes how to run the external connection example

1. From a system command prompt, change to the Sybase IQ installation directory.
2. Change to the `IQ-15_3/java` subdirectory
3. Ensure that your CLASSPATH environment variable includes the current directory (.) and the imported zip files. For example, from a command prompt (the following should be entered all on one line):

   ```
   set classpath=..\java\jdbcdrv.zip;.;..\java
   \asajdbc.zip;asademo.jar
   ```

   The default zip file name for Java is `classes.zip`. For classes in any file named `classes.zip`, you only need the directory name in the CLASSPATH variable, not the zip-file name itself. For classes in files with other names, you must supply the zip file name.

   You need the current directory in the CLASSPATH to run the example.
4. Ensure the database is loaded onto a database server running TCP/IP. You can start such a server on your local machine using the following command (from the `IQ-15_3/samples/sqlanywhere` subdirectory):

   On UNIX: `start_iq .../iqdemo`

   On Windows: `start_iq ...\iqdemo`
5. Enter the following at the command prompt to run the example:

   ```
   java JDBCExamples
   ```

   If you wish to try this against a server running on another machine, you must enter the correct name of that machine. The default is `localhost`, which is an alias for the current machine name.
6. Confirm that a list of people and products appears at your command prompt.

   If the attempt to connect fails, an error message appears instead. Confirm that you have executed all the steps as required. Check that your CLASSPATH is correct. An incorrect CLASSPATH results in a failure to locate a class.

See also

- *Establish Connection From a Server-side JDBC Class* on page 126
- *Connect From a JDBC Client Application Using jConnect* on page 122
- *Sybase jConnect JDBC Driver* on page 135

Establish Connection From a Server-side JDBC Class

SQL statements in JDBC are built using the **createStatement** method of a **Connection** object. Even classes running inside the server need to establish a connection to create a **Connection** object.

Establishing a connection from a server-side JDBC class is more straightforward than establishing an external connection. Because a user already connected executes the server-side class, the class simply uses the current connection.

Notes on JDBC Connections

- Autocommit behavior—The JDBC specification requires that, by default, a COMMIT is performed after each data modification statement. Currently, the server-side JDBC behavior is to commit. You can control this behavior using a statement such as the following:

  ```
  conn.setAutoCommit( false ) ;
  ```

  where **conn** is the current connection object.
- Connection defaults—From server-side JDBC, only the first call to **getConnection("jdbc:default:connection")** creates a new connection with the default values. Subsequent calls return a wrapper of the current connection with all connection properties unchanged. If you set AutoCommit to OFF in your initial connection, any subsequent **getConnection** calls within the same Java code return a connection with AutoCommit set to OFF.

  You may wish to ensure that closing a connection resets connection properties to their default values, so subsequent connections are obtained with standard JDBC values. The following type of code achieves this:

  ```
  Connection conn = DriverManager.getConnection("");
  boolean oldAutoCommit = conn.getAutoCommit();
  try {
       // do code here
  }
  finally {
      conn.setAutoCommit( oldAutoCommit );
  }
  ```

  This discussion applies not only to AutoCommit, but also to other connection properties such as TransactionIsolation and is ReadOnly.

See also

- *Connect From a JDBC Client Application Using jConnect* on page 122
- *Sybase jConnect JDBC Driver* on page 135
- *Running the External Connection Example* on page 125

Server-side Connection Example Code

This is an example source code for server-side connection.

You can find the source code in the **InternalConnect** method of `JDBCExamples.java` in the `C:\Documents and Settings\All Users\SybaseIQ\samples\SQLAnywhere\JDBC` directory under your Sybase IQ installation directory:

```
public static void InternalConnect() {
  try {
     conn = DriverManager.getConnection("jdbc:default:connection");
      System.out.println("Hello World");
    }
    catch ( Exception e ) {
      System.out.println("Error: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

How the Server-side Connection Example Works

In this simple example, **InternalConnect()** is the only method used in the application.

The application requires only one of the libraries (JDBC) imported in the first line of the `JDBCExamples.java` class. The others are for external connections. The package named **java.sql** contains the JDBC classes.

The **InternalConnect()** method carries out the following tasks:

1. Connects to the default running database using the current connection:
   - **DriverManager.getConnection** establishes a connection using a connection string of **jdbc:default:connection**.
2. Prints `Hello World` to the current standard output, which is the server window. **System.out.println** carries out the printing.
3. If there is an error in the attempt to connect, an error message appears in the server window, together with the place where the error occurred.

   The **try** and **catch** instructions provide the framework for the error handling.
4. The class terminates.

Running the Server-side Connection Example

This section describes how to run the server-side connection example.

1. If you have not already done so, compile the `JDBCExamples.java` file. If you are using the JDK, you can do the following in the `C:\Documents and Settings\All Users\SybaseIQ\samples\SQLAnywhere\JDBC` directory under your Sybase IQ installation directory from a command prompt:

   ```
   javac JDBCExamples.java
   ```

2. Start a database server using the demo database. You can start such a server on your local machine using the following command (from the `/ASIQ-12_7/java` subdirectory):

   On UNIX: `start_iq .../iqdemo`

   On Windows: `start_iq ...\iqdemo`

   The TCP/IP network protocol is not necessary in this case, since you are not using jConnect. However, you must have at least 8 Mb of cache available to use Java classes in the database.
3. Install the class into the demo database. Once connected to the demo database, you can do this from Interactive SQL using the following command:

   ```
   INSTALL JAVA NEW
   FROM FILE 'C:\Documents and Settings\All Users\SybaseIQ\samples
   \SQLAnywhere\JDBC\JDBCExamples.class'
   ```

   where *path* is the path to your installation directory.

   You can also install the class using Sybase Central. While connected to the demo database, open the Java Objects folder and double-click Add Class. Then follow the instructions in the wizard.
4. You can now call the **InternalConnect** method of this class just as you would a stored procedure:

   ```
   CALL JDBCExamples>>InternalConnect()
   ```

   The first time a Java class is called in a session, the internal Java virtual machine must be loaded. This can take a few seconds.
5. Confirm that the message `Hello World` prints on the server screen.

Use JDBC to Access Data

Java applications that hold some or all classes in the database have significant advantages over traditional SQL stored procedures. At an introductory level, however, it may be helpful to use the parallels with SQL stored procedures to demonstrate the capabilities of JDBC.

In the following examples, we write Java classes that insert a row into the `Department` table.

As with other interfaces, SQL statements in JDBC can be either *static* or *dynamic*. Static SQL statements are constructed in the Java application, and sent to the database. The database server parses the statement, and selects an execution plan, and executes the statement. Together, parsing and selecting an execution plan are referred to as *preparing* the statement.

If a similar statement has to be executed many times (many inserts into one table, for example), there can be significant overhead in static SQL because the preparation step has to be executed each time.

In contrast, a dynamic SQL statement contains placeholders. The statement, prepared once using these placeholders, can be executed many times without the additional expense of preparing.

In this section we use static SQL. Dynamic SQL is discussed in a later section.

Miscellaneous JDBC Notes

- Access permissions—Like all Java classes in the database, classes containing JDBC statements can be accessed by any user. There is no equivalent of the GRANT EXECUTE statement that grants permission to execute procedures, and there is no need to qualify the name of a class with the name of its owner.
- Execution permissions—Java classes are executed with the permissions of the connection executing them. This behavior is different to that of stored procedures, which execute with the permissions of the owner.

Installing the JDBCExamples Class

This section describes how to install the JDBCExamples.class and prepare for the JDBC examples in the remainder of this appendix.

Sample Code

The code fragments in this section are taken from the complete class `C:\Documents and Settings\All Users\SybaseIQ\samples\SQLAnywhere\JDBC\JDBCExamples.java`, under your installation directory.

Installing JDBCExamples Cass

1. If you have not already done so, install the `JDBCExamples.class` file into the demo database.
2. Once connected to the demo database from Interactive SQL, enter the following command in the SQL Statements pane:

   ```
   INSTALL JAVA NEW
   FROM FILE 'C:\Documents and Settings\All Users\SybaseIQ\samples
   \SQLAnywhere\JDBC\JDBCExamples.class'
   ```

   where *path* is the path to your installation directory.

   You can also install the class using Sybase Central. While connected to the demo database, open the Java Objects folder and double-click Add Java Class or JAR. Then follow the instructions in the wizard.

Using JDBC to Insert, Update, and Delete

The **Statement** object executes static SQL statements. You execute SQL statements such as INSERT, UPDATE, and DELETE, which do not return result sets, using the **executeUpdate** method of the **Statement** object. Statements such as CREATE TABLE and other data definition statements can also be executed using **executeUpdate**.

The following code fragment illustrates how JDBC carries out INSERT statements. It uses an internal connection held in the Connection object named **conn**. The code for inserting values from an external application using JDBC would need to use a different connection, but otherwise would be unchanged.

```
public static void InsertFixed()  {
    // returns current connection
    conn = DriverManager.getConnection("jdbc:default:connection");
    // Disable autocommit
    conn.setAutoCommit( false );

    Statement stmt = conn.createStatement();

    Integer IRows = new Integer( stmt.executeUpdate
       ("INSERT INTO Department (dept_id, dept_name )"
        + "VALUES (201, 'Eastern Sales')"
         ) );
    // Print the number of rows updated
    System.out.println(IRows.toString() + "row inserted" );
  }
```

Note: This code fragment is part of the **InsertFixed** method of the **JDBCExamples** class. On Windows you can build this class using `build.bat` in `C:\Documents and Settings\All Users\SybaseIQ\samples\SQLAnywhere\JDBC`.

- The **setAutoCommit** method turns off the AutoCommit behavior, so changes are only committed if you execute an explicit COMMIT instruction.
- The **executeUpdate** method returns an integer, which reflects the number of rows affected by the operation. In this case, a successful INSERT would return a value of one (1).
- The integer return type converts to an **Integer** object. The Integer class is a wrapper around the basic **int** data type, providing some useful methods such as **toString()**.
- The Integer **IRows** converts to a string to be printed. The output goes to the server window.

Running the JDBC Insert example

Create a very simple JDBC class.

1. Using Interactive SQL, connect to the demo database as user ID `dba`.
2. Ensure the JDBCExamples class has been installed. It is installed together with the other Java examples classes.
3. Call the method as follows:

   ```
   CALL JDBCExamples>>InsertFixed()
   ```
4. Confirm that a row has been added to the `department` table.

   ```
   SELECT *
   FROM department
   ```

The row with ID 201 is not committed. You can execute a **ROLLBACK** statement to remove the row.

Passing Arguments to Java Methods

The **InsertFixed** method can be expand to illustrate how arguments are passed to Java methods.

The following method uses arguments passed in the call to the method as the values to insert:

```
public static void InsertArguments(
                 String id, String name)  {
try {
      conn = DriverManager.getConnection(
                    "jdbc:default:connection" );

String sqlStr =  "INSERT INTO Department "
    + " ( dept_id, dept_name )"
    + " VALUES (" +  id +  ", '"  + name + "')";

      // Execute the statement
      Statement stmt = conn.createStatement();
      Integer IRows = new
Integer( stmt.executeUpdate( sqlStr.toString() ) );

      // Print the number of rows updated
      System.out.println(IRows.toString() + " row inserted" );
    }
    catch ( Exception e ) {
      System.out.println("Error: " + e.getMessage());
      e.printStackTrace();
    }
  }
```

Using a Java Method with Arguments

- The two arguments are the department id (an integer) and the department name (a string). Here, both arguments pass to the method as strings, because they are part of the SQL statement string.
- The INSERT is a static statement and takes no parameters other than the SQL itself.
- If you supply the wrong number or type of arguments, you receive the `Procedure Not Found` error.

1. If you have not already installed the `JDBCExamples.class` file into the demo database, do so.
2. Connect to the demo database from Interactive SQL, and enter the following command:

   ```
   call JDBCExamples>>InsertArguments( '203', 'Northern Sales' )
   ```

3. Verify that an additional row has been added to the Department table:

   ```
   SELECT *
   FROM Department
   ```

4. Roll back the changes to leave the database unchanged:

   ```
   ROLLBACK
   ```

Queries Using JDBC

The **Statement** object executes static queries, as well as statements that do not return result sets. For queries, you use the **executeQuery** method of the **Statement** object. This returns the result set in a **ResultSet** object.

The following code fragment illustrates how queries can be handled within JDBC. The code fragment places the total inventory value for a product into a variable named **inventory**. The product name is held in the **String** variable **prodname**. This example is available as the **Query** method of the **JDBCExamples** class.

The example assumes an internal or external connection has been obtained and is held in the Connection object named **conn**. It also assumes a variable

```
public static void Query () {
int max_price = 0;
    try{
      conn = DriverManager.getConnection(
                   "jdbc:default:connection" );

      // Build the query
      String sqlStr =  "SELECT id, unit_price "
    + "FROM product" ;

      // Execute the statement
      Statement stmt = conn.createStatement();
      ResultSet result = stmt.executeQuery( sqlStr );

      while( result.next() ) {
    int price = result.getInt(2);
    System.out.println( "Price is "  + price );
    if( price > max_price ) {
      max_price = price ;
    }
      }
    }
    catch( Exception e ) {
      System.out.println("Error: " + e.getMessage());
      e.printStackTrace();
    }
      return max_price;
  }
```

Running the Example

Once you have installed the **JDBCExamples** class into the demo database, you can execute this method using the following statement in Interactive SQL:

```
select JDBCExamples>>Query()
```

Notes

- The query selects the quantity and unit price for all products named **prodname**. These results are returned into the **ResultSet** object named **result**.
- There is a loop over each of the rows of the result set. The loop uses the **next** method.
- For each row, the value of each column is retrieved into an integer variable using the **getInt** method. **ResultSet** also has methods for other data types, such as **getString**, **getDate**, and **getBinaryString**.

  The argument for the **getInt** method is an index number for the column, starting from 1.

  Data type conversion from SQL to Java is carried out according to the information in "Java / SQL data type conversion" in the "SQL Data Types" chapter of the *Sybase IQ Reference Manual*.
- Sybase IQ supports bidirectional scrolling cursors. However, JDBC provides only the **next** method, which corresponds to scrolling forward through the result set.
- The method returns the value of **max_price** to the calling environment, and Interactive SQL displays it in the Results pane.

See also

- *Distributed Applications* on page 139
- *Insert and Retrieve Objects* on page 134

Using Prepared Statements for More Efficient Access

If you use the **Statement** interface, you parse each statement you send to the database, generate an access plan, and execute the statement. The steps prior to actual execution are called *preparing* the statement.

You can achieve performance benefits if you use the **PreparedStatement** interface. This allows you to prepare a statement using placeholders, and then assign values to the placeholders when executing the statement.

Using prepared statements is particularly useful when carrying out many similar actions, such as inserting many rows.

For more information on prepared statements, see *Reference: Statements and Options > SQL Statements > PREPARE statement [ESQL]*.

Example

The following example illustrates how to use the **PreparedStatement** interface, although inserting a single row is not a good use of prepared statements.

The following method of the **JDBCExamples** class carries out a prepared statement:

```
public static void JInsertPrepared(int id, String name) try {
      conn = DriverManager.getConnection(
                     "jdbc:default:connection");
      // Build the INSERT statement
      // ? is a placeholder character
      String sqlStr = "INSERT INTO Department "
    + "( dept_id, dept_name ) "
    + "VALUES ( ? , ? )" ;

      // Prepare the statement
      PreparedStatement stmt = conn.prepareStatement( sqlStr );

      stmt.setInt(1, id);
      stmt.setString(2, name );
      Integer IRows = new Integer(
                          stmt.executeUpdate() );

      // Print the number of rows updated
      System.out.println(IRows.toString() + " row inserted" );
    }
    catch ( Exception e ) {
      System.out.println("Error: " + e.getMessage());
      e.printStackTrace();
    }
  }
```

Running the example

Once you have installed the **JDBCExamples** class into the demo database, you can execute this example by entering the following statement:

```
call JDBCExamples>>InsertPrepared(
                           202, 'Eastern Sales' )
```

The string argument is enclosed in single quotes, which is appropriate for SQL. If you invoked this method from a Java application, use double quotes to delimit the string.

Insert and Retrieve Objects

As an interface to relational databases, JDBC is designed to retrieve and manipulate traditional SQL data types.

Sybase IQ also provides abstract data types in the form of Java classes. The way you access these Java classes using JDBC depends on whether you want to insert or retrieve the objects.

See also

- *Distributed Applications* on page 139
- *Queries Using JDBC* on page 132

Retrieve Objects

You can retrieve objects and their fields and methods by:

- Accessing methods and fields—Java methods and fields can be included in the select-list of a query. A method or field then appears as a column in the result set, and can be accessed using one of the standard **ResultSet** methods, such as **getInt**, or **getString**.

- Retrieving an objec—If you include a column with a Java class data type in a query select list, you can use the **ResultSet getObject** method to retrieve the object into a Java class. You can then access the methods and fields of that object within the Java class. Java objects can only be stored in the Catalog Store.

Insert Objects

From a server-side Java class, you can use the JDBC **setObject** method to insert an object into a column with Java class data type.

You can insert objects using a prepared statement. For example, the following code fragment inserts an object of type MyJavaClass into a column of table T:

```
java.sql.PreparedStatement ps =
    conn.prepareStatement("insert T values( ? )" );
ps.setObject( 1, new MyJavaClass() );
ps.executeUpdate();
```

An alternative is to set up a SQL variable that holds the object and then to insert the SQL variable into the table.

Sybase jConnect JDBC Driver

If you wish to use JDBC from a client application or applet, you must have Sybase jConnect to connect to Sybase IQ databases.

Depending on the installation package you received, Sybase IQ may or may not include Sybase jConnect. You must have jConnect in order to use JDBC from client applications. You can use server-side JDBC without jConnect.

For a full description of jConnect and its use with Sybase IQ, see the jConnect documentation available in the online books or from the *jConnect web site*

Note: Before you can use jConnect in your application, load the driver by entering the statement:

```
Class.forName("com.sybase.jdbc.SybDriver").newInstance();
```

Using the **newInstance** method works around issues in some browsers.

See also

- *Establish Connection From a Server-side JDBC Class* on page 126
- *Connect From a JDBC Client Application Using jConnect* on page 122
- *Running the External Connection Example* on page 125

Versions of jConnect Supplied with Sybase IQ

Sybase IQ provides two versions of the Sybase jConnect JDBC driver:

- Full version—If you choose to install jConnect, a jConnect subdirectory is added to your Sybase IQ installation. This holds a directory tree with all jConnect files.
- Zip file—The Remote Data Access features, and the Java debugger, both use jConnect to connect to the database. A zip file of the basic jConnect classes is provided to enable jConnect use even without the full development version of the driver.

The jConnect Driver Files

The Sybase jConnect driver is installed into a set of directories under the `jConnect` subdirectory of your Sybase IQ installation. If you have not installed jConnect, you can use the `jdbcdrv.zip` file installed into the Java subdirectory.

Classpath Setting for jConnect

For your application to use jConnect, the jConnect classes must be in your CLASSPATH environment variable at compile time and run time, so the Java compiler and Java runtime can locate the necessary files.

For example, the following command adds the jConnect driver class path to an existing CLASSPATH environment variable where *path* is your Sybase IQ installation directory.

```
set classpath=%classpath%;path\jConnect\classes
```

The following alternative command adds the `jdbcdrv.zip` file to your CLASSPATH.

```
set classpath=%classpath%;path\java\jdbcdrv.zip
```

Importing the jConnect Classes

The classes in jConnect are all in the **com.sybase** package. The client application needs to access classes in **com.sybase.jdbc**. For your application to use jConnect, you must import these classes at the beginning of each source file:

```
import com.sybase.jdbc.*
```

See also

- *JDBC Overview* on page 117

Installing jConnect System Objects Into a Database

If you wish to use jConnect to access system table information (database metadata), you must add the jConnect system objects to your database.

By default, the jConnect system objects are added to a database for any database created using version 12.7, and to any database upgraded to version 12.7.

You can choose to add the jConnect objects to the database either when creating or upgrading, or at a later time.

You can install the jConnect system objects from Interactive SQL.

Adding the jConnect System Objects To a Version 12.7 Database From Sybase Central:

1. Connect to the database from Sybase Central as a user with DBA authority.
2. In the left pane of the main Sybase Central viewer, right-click the database icon and choose Re-Install jConnect Meta-data Support from the popup menu.

Adding the jConnect System Objects To a Version 12.7 Database From Interactive SQL

Connect to the database from Interactive SQL as a user with DBA authority, and enter the following command in the SQL Statements pane:

```
read path\scripts\jcatalog.sql
```

where *path* is your Sybase IQ installation directory.

Note: You can also use a command prompt to add the jConnect system objects to a version 12.7 database. At the command prompt, type:

```
dbisql -c "uid=user;pwd=pwd" path\scripts\jcatalog.sql
```

where *user* and *pwd* identify a user with DBA authority, and *path* is your Sybase IQ installation directory.

Supply URL For the Server

To connect to a database via jConnect, you need to supply a Universal Resource Locator (URL) for the database.

An example given in the section is as follows:

```
StringBuffer temp = new StringBuffer();
// Use the Sybase jConnect driver...
temp.append("jdbc:sybase:Tds:");
// to connect to the supplied machine name...
temp.append(_coninfo);
// on the default port number for ASA...
temp.append(":2638");
// and connect.
System.out.println(temp.toString());
conn = DriverManager.getConnection(temp.toString() , _props );
```

The URL is put together in the following way:

```
jdbc:sybase:Tds:machine-name:port-number
```

The individual components are include:

- jdbc:sybase:Tds—The Sybase jConnect JDBC driver, using the TDS application protocol.

- machine-name—The IP address or name of the machine on which the server is running. If you are establishing a same-machine connection, you can use `localhost`, which means the current machine.
- port number—The port number on which the database server listens. The port number assigned to Sybase IQ is 2638. Use that number unless there are specific reasons not to do so.

The connection string must be less than 253 characters in length.

Specify Database On a Server

Each Sybase IQ server may have one or more databases loaded at a time. The URL as described above specifies a server, but does not specify a database. The connection attempt is made to the default database on the server.

You can specify a particular database by providing an extended form of the URL in one of the following ways.

Using the ServiceName Parameter

```
jdbc:sybase:Tds:machine-name:port-number?ServiceName=DBN
```

The question mark followed by a series of assignments is a standard way of providing arguments to a URL. The case of **ServiceName** is not significant, and there must be no spaces around the = sign. The *DBN* parameter is the database name.

Using the RemotePWD Parameter

A more general method allows you to provide additional connection parameters such as the database name, or a database file, using the **RemotePWD** field. You set **RemotePWD** as a Properties field using the **setRemotePassword()** method.

Here is sample code that illustrates how to use the field.

```
sybDrvr = (SybDriver)Class.forName(
     "com.sybase.jdbc2.jdbc.SybDriver" ).newInstance();
props = new Properties();
props.put( "User", "DBA" );
props.put( "Password", "SQL" );
sybDrvr.setRemotePassword(
    null, "dbf=asiqdemo.db", props );
Connection con = DriverManager.getConnection(
    "jdbc:sybase:Tds:localhost", props );
```

Using the database file parameter **DBF**, you can start a database on a server using jConnect. By default, the database is started with autostop=YES. If you specify a DBF or DBN of **utility_db**, then the utility database will automatically be started.

For information on the utility database, see *System Administration Guide: Volume 1 > Overview of Sybase IQ System Administration* and *Managing User IDs and Permissions*.

IQ specific connection parameters from TDS clients should be specified in RemotePWD.

This example shows how to specify IQ specific connection parameters, where myconnection becomes the IQ connection name:

```
p.put(“RemotePWD“,“,,CON=myconnection”);
```

where myconnection becomes the IQ connection name.

Distributed Applications

In a *distributed application*, parts of the application logic run on one machine, and parts run on another machine. With Sybase IQ, you can create distributed Java applications, where part of the logic runs in the database server, and part on the client machine.

Sybase IQ is capable of exchanging Java objects with an external, Java client.

Having the client application retrieve a Java object from a database is the key task in a distributed application This section describes how to accomplish that task.

Features of Distributed Applications

There are two other methods in `JDBCExamples.java` that use distributed computing:

- serializeVariable—This method creates a native Java object referenced by a SQL variable on the database server and passes it back to the client application.
- serializeColumnCastClass—This method is like the **serializeColumn** method, but demonstrates how to reconstruct subclasses. The column that is queried (**JProd** from the **product** table) is of data type **asademo.Product**. Some of the rows are **asademo.Hat**, which is a subclass of the Product class. The proper class is reconstructed on the client side.

Requirements for Distributed Applications

There are two tasks in building a distributed application.

- Any class running in the server must implement the Serializable interface. This is very simple.
- The client-side application must import the class, so the object can be reconstructed on the client side.

These tasks are described in the following sections.

See also

- *Insert and Retrieve Objects* on page 134
- *Queries Using JDBC* on page 132

Serializable Interfaces

Objects pass from the server to a client application in *serialized* form. For an object to be sent to a client application, it must implement the Serializable interface. Fortunately, this is a very simple task.

The Serializable interface contains no methods and no variables. Serializing an object converts it into a byte stream which allows it to be saved to disk or sent to another Java application where it can be reconstituted, or *deserialized*.

A serialized Java object in a database server, sent to a client application and deserialized, is identical in every way to its original state. Some variables in an object, however, either don't need to be or, for security reasons, should not be serialized. Those variables are declared using the keyword **transient**, as in the following variable declaration.

```
transient String password;
```

When an object with this variable is deserialized, the variable always contains its default value, null.

Custom serialization can be accomplished by adding **writeObject()** and **readObject()** methods to your class.

For more information about serialization, see Sun Microsystems' Java Development Kit (JDK).

Implementing the Serializable Interface

Implementing the Serializable interface amounts to simply declaring that your class can be serialized.
Add the words **implements java.io.Serializable** to your class definition.

For example, the Product class in the in `$SADIR/samples/asa/java/asademo` (UNIX) or `%SADIR%\samples\asa\java\asademo` (Windows) subdirectory implements the Serializable interface by virtue of the following declaration:

```
public class Product implements java.io.Serializable
```

Importing the Class On the Client Side

On the client side, any class that retrieves an object has to have access to the proper class definition to use the object.

To use the **Product** class, which is part of the **asademo** package, you must include the following line in your application:

```
import asademo.*
```

The `asademo.jar` file must be included in your CLASSPATH for this package to be located.

A Sample Distributed Application

The `JDBCExamples.java` class contains three methods that illustrate distributed Java computing. These are all called from the **main** method.

Here is the **getObjectColumn** method from the JDBCExamples class.

```
private static void getObjectColumn() throws Exception {
// Return a result set from a column containing
// Java objects
    asademo.ContactInfo ci;
    String name;
    String sComment ;

    if ( conn != null ) {
      Statement stmt = conn.createStatement();
      ResultSet rs = stmt.executeQuery(
           "SELECT JContactInfo FROM jdba.contact"
    );
    while ( rs.next() ) {
    ci = ( asademo.ContactInfo )rs.getObject(1);
  System.out.println( "\n\tStreet: " + ci.street +
      "City: " + ci.city +
      "\n\tState: " + ci.state +
      "Phone: " + ci.phone +
      "\n" );
      }
    }
  }
```

The getObject method is used in the same way as in the internal Java case.

See also

- *Connect From a JDBC Client Application Using jConnect* on page 122

Debugging Logic in the Database

This appendix describes how to use the Sybase debugger to assist in developing SQL stored procedures and event handlers, as well as Java stored procedures.

Introduction To Debugging In the Database

You can use the debugger during the development.

You can use the following objects:

- SQL stored procedures, event handlers, and user-defined functions.
- Java stored procedures in the database.

Debugger Features

You can use the debugger during the development of SQL stored procedures, triggers, event handlers, and user-defined functions.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Debugging procedures, functions, triggers, and events > Introduction to the SQL Anywhere debugger*.

Requirements for Using the Debugger

To use the debugger, you need:

- Permissions—You must either have DBA authority or be granted permissions in the SA_DEBUG group. This group is automatically added to all databases when they are created.
- Source code for Java classes—The source code for your application must be available to the debugger. For Java classes, the source code is held on a directory on your hard disk. For stored procedures, the source code is held in the database.
- Compilation options—To debug Java classes, they must be compiled so that they contain debugging information. For example, if you are using the Sun Microsystems JDK compiler `javac.exe`, the Java classes must be compiled using the `-g` command-line option.

Note: The Sybase IQ demo database is iqdemo.db.

Tutorial 1: Getting Started With the Debugger

These tutorials describe how to start the debugger, how to connect to a database, and how to debug a Java class.

Lesson 1: Connect To a Database and Start the Debugger

This tutorial shows you how to start the debugger, connect to a database, and attach to a connection for debugging. It uses the Sybase IQ demo database.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Debugging procedures, functions, triggers, and events > Tutorial: Getting started with the debugger > Lesson 1: Connect to a database and start the debugger*.

Tutorial 2: Debugging a stored procedure

This tutorial describes a sample session for debugging a stored procedure.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Debugging procedures, functions, triggers, and events > Tutorial: Getting started with the debugger > Lesson 2: Debug a stored procedure*.

Tutorial 3: Debugging a Java Class

In this tutorial, you call **JDBCExamples.Query()** from Interactive SQL (dbisql), interrupt the execution in the debugger, and trace through the source code for this method.

The **JDBCExamples.Query()** method executes the following query against the demo database:

```
SELECT ID, UnitPrice
FROM Products
```

It then loops through all the rows of the result set, and returns the one with the highest unit price.

You must compile classes with the `javac -g` option to debug them. The sample classes are compiled for debugging.

Note: To use the Java examples, you must have the Java example classes installed into the demo database.

Demo Database Java Example Classes

If you intend to run Java examples, install the Java example classes into the demo database.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - Programming > SQL Anywhere Data Access APIs > SQL Anywhere JDBC driver > Using JDBC to access data > Preparing for the examples*.

Displaying Java Source Code Into the Debugger

The debugger looks in a set of locations for source code files with `.java` extension.

The Java Source Code Path window holds a list of directories in which the debugger looks for Java source code. Java rules for finding packages apply. The debugger also searches the current CLASSPATH for source code.

1. Select Sybase IQ 15 in left folder view.
2. In Sybase Central, select Mode > Debug.
3. When prompted to select the user to debug, specify * for all users and click OK.
4. From the debugger interface, select Debug > Set Java Source Code Path.
5. Enter the path to the `java` subdirectory of your Sybase IQ installation directory. For example, if you installed Sybase IQ in `%IQDIR15%`, enter:

   ```
   %IQDIR15%\java
   ```
6. Click Browse Folder to select from a list of folders or individual files where the debugger looks for Java source.
7. Click Browse File to locate a file to add to the list.
8. Click OK, and close the window.

Setting a Breakpoint

You can set a breakpoint at the beginning of the Query() method. When the method is invoked, execution stops at the breakpoint.

1. In the Source Code window, scroll down until you see the beginning of the **Query()** method, near the end of the class, starting with:

   ```
   public static int Query() {
   ```
2. Click the green indicator to the left of the first line of the method, until it is red. The first line of the method is:

   ```
   int max_price = 0;
   ```

   Repeatedly clicking the indicator toggles its status. After setting the breakpoint, the Java class does not need to be recompiled.

Running the Method

You can invoke the **Query()** method from Interactive SQL (dbisql), and see its execution interrupted at the breakpoint.

1. Start Interactive SQL. Connect to the demo database as used ID `DBA` and password `sql`.

   The connection appears in the debugger Connections window list.
2. To invoke the method, enter the following command in Interactive SQL:

   ```
   SELECT JDBCExamples.Query()
   ```

   The query does not complete. Instead, execution is stopped in the debugger at the breakpoint. In Interactive SQL, the Stop button is active. In the debugger Source window, the red arrow indicates the current line.

You can now step through source code and carry out debugging activities in the debugger.

Stepping Through Source Code

From the previous section, the debugger should have stopped executing **JDBCExamples.Query()** at the first statement in the method:

1. Choose Debug > Step Over, or press F10 to step to the next line in the current method. Try this two or three times.
2. Click at the end of the following line using the mouse, and choose Debug > Run To Cursor, or press CTRL + F10 to run to that line and break:
3. Select the following line (line 292) and press F9 to set a breakpoint on that line:

   ```
   return max_price;
   ```

   An asterisk appears in the left column to mark the breakpoint. Press F5 to execute to that breakpoint.
4. Try different methods of stepping through the code. End with F5 to complete the execution.

   When you have completed the execution, the Interactive SQL data window displays the value 24.
5. To move to the next breakpoint, add an F5.

   When you have completed the execution, the Interactive SQL data window displays the value 24.

The complete set of options for stepping through source code appear on the **Run** menu. You can find more information in the debugger online Help.

Inspecting and Modifying Variables

You can inspect the values of both local variables (declared in a method) and class static variables in the debugger.

You can display class-level variables (static variables) in the Debugger window and inspect their values. For more information, see the debugger online Help.

You can inspect the values of local variables in a method as you step through the code, to better understand what is happening.

Note: To use the Java examples, you must have the Java example classes installed into the demo database.

1. Set a breakpoint at the first line of the **JDBCExamples.Query** method. This line is as follows:

   ```
   int max_price = 0
   ```
2. In Interactive SQL, execute the method again:

   ```
   SELECT JDBCExamples.Query()
   ```

   The query executes only as far as the breakpoint.
3. Press F7 to step to the next line. The **max_price** variable has now been declared and initialized to zero.
4. If the Locals window does not appear, choose Window > Locals to display it.

   The Locals window shows that there are several local variables. **max_price** has a value of zero. All other variables are listed as `variable not in scope`, which means they are not yet initialized.
5. In the Locals window, double-click the Value column entry for **max_price**, and change the value of **max_price** to 45.

   The value 45 is larger than any other price. Instead of returning 24, the query now returns 45 as the maximum price.
6. In the Source window, press F7 repeatedly to step through the code. The values of the variables appear in the Locals window. Step through until the **stmt** and **result** variables have values.
7. Expand the **result** object by clicking the icon next to it, or by setting the cursor on the line and pressing Enter. This displays the values of the fields in the object.
8. When you have experimented with inspecting and modifying variables, press F5 to complete the execution of the query and finish the tutorial.

Breakpoints

Breakpoints control when the debugger interrupts execution of your source code.

See*SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Debugging procedures, functions, triggers, and events > Working with breakpoints*.

View and Edit Variable Behavior

The debugger lets you view and edit the behavior of your variables while stepping through your code.

The debugger provides a Debugger Details pane, which displays the different kinds of variables used in stored procedures. The Debugger Details pane appears at the bottom of Sybase Central when Sybase Central is running in Debug mode.

See *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Debugging procedures, functions, triggers, and events > Working with variables*.

Write Debugger Scripts

The debugger allows you to write scripts in the Java programming language. A script is a Java class that extends the **sybase.asa.procdebug.DebugScript** class.

When the debugger runs a script, it loads the class and calls its **run** method. The first parameter of the **run** method is a pointer to an instance of the class. This interface lets you interact with and control the debugger.

You can compile scripts with a command such as the following:

```
javac -classpath %asany%/procdebug/ProcDebug.jar;%classpath%
myScript.Java.
```

sybase.asa.procdebug.DebugScript Class

The **DebugScript** class is as follows:

```
// All debug scripts must inherit from this class

package sybase.asa.procdebug;

abstract public class DebugScript
{
    abstract public void run( IDebugAPI db, String args[] );
    /*
```

```
        The run method is called by the debugger
        - args will contain command line arguments
    */

    public void OnEvent( int event ) throws DebugError {}
    /*
    - Override the following methods to process debug events
    - NOTE: this method will not be called unless you call
        DebugAPI.AddEventHandler( this );
    */

}
```

sybase.asa.procdebug.IDebugAPI Interface

The **IDebugAPI** interfaces is as follows:

```
package sybase.asa.procdebug;
import java.util.*;
public interface IDebugAPI

{
    // Simulate Menu Items

    IDebugWindow MenuOpenSourceWindow() throws DebugError;
    IDebugWindow MenuOpenCallsWindow() throws DebugError;
    IDebugWindow MenuOpenClassesWindow() throws DebugError;
    IDebugWindow MenuOpenClassListWindow() throws DebugError;
    IDebugWindow MenuOpenMethodsWindow() throws DebugError;
    IDebugWindow MenuOpenStaticsWindow() throws DebugError;
    IDebugWindow MenuOpenCatchWindow() throws DebugError;
    IDebugWindow MenuOpenProcWindow() throws DebugError;
    IDebugWindow MenuOpenOutputWindow() throws DebugError;
    IDebugWindow MenuOpenBreakWindow() throws DebugError;
    IDebugWindow MenuOpenLocalsWindow() throws DebugError;
    IDebugWindow MenuOpenInspectWindow() throws DebugError;
    IDebugWindow MenuOpenRowVarWindow() throws DebugError;
    IDebugWindow MenuOpenQueryWindow() throws DebugError;
    IDebugWindow MenuOpenEvaluateWindow() throws DebugError;
    IDebugWindow MenuOpenGlobalsWindow() throws DebugError;
    IDebugWindow MenuOpenConnectionWindow() throws DebugError;
    IDebugWindow MenuOpenThreadsWindow() throws DebugError;
    IDebugWindow GetWindow( String name ) throws DebugError;

    void MenuRunRestart() throws DebugError;
    void MenuRunHome() throws DebugError;
    void MenuRunGo() throws DebugError;
    void MenuRunToCursor() throws DebugError;
    void MenuRunInterrupt() throws DebugError;
    void MenuRunOver() throws DebugError;
    void MenuRunInto() throws DebugError;
    void MenuRunIntoSpecial() throws DebugError;
    void MenuRunOut() throws DebugError;
    void MenuStackUp() throws DebugError;
    void MenuStackDown() throws DebugError;
    void MenuStackBottom() throws DebugError;
    void MenuFileExit() throws DebugError;
    void MenuFileOpen( String name ) throws DebugError;
    void MenuFileAddSourcePath( String what ) throws DebugError;
    void MenuSettingsLoadState( String file ) throws DebugError;
    void MenuSettingsSaveState( String file ) throws DebugError;
    void MenuWindowTile() throws DebugError;
    void MenuWindowCascade() throws DebugError;
    void MenuWindowRefresh() throws DebugError;
    void MenuHelpWindow() throws DebugError;
    void MenuHelpContents() throws DebugError;
    void MenuHelpIndex() throws DebugError;
    void MenuHelpAbout() throws DebugError;
    void MenuBreakAtCursor() throws DebugError;
    void MenuBreakClearAll() throws DebugError;
    void MenuBreakEnableAll() throws DebugError;
    void MenuBreakDisableAll() throws DebugError;
    void MenuSearchFind( IDebugWindow w, String what ) throws
DebugError;
    void MenuSearchNext( IDebugWindow w ) throws DebugError;
    void MenuSearchPrev( IDebugWindow w ) throws DebugError;
    void MenuConnectionLogin() throws DebugError;
    void MenuConnectionReleaseSelected() throws DebugError;

    // output window
    void OutputClear();
    void OutputLine( String line );
    void OutputLineNoUpdate( String line );
    void OutputUpdate();

    // Java source search path

    void SetSourcePath( String path ) throws DebugError;
    String GetSourcePath() throws DebugError;

    // Catch java exceptions
    Vector GetCatching();
    void Catch( boolean on, String name ) throws DebugError;

    // Database connections
    int  ConnectionCount();
    void ConnectionRelease( int index );
    void ConnectionAttach( int index );
    String ConnectionName( int index );
    void ConnectionSelect( int index );

    // Login to database
    boolean LoggedIn();
    void Login( String url, String userId, String password, String
userToDebug ) throws DebugError;
    void Logout();

    // Simulate keyboard/mouse actions
    void DeleteItemAt( IDebugWindow w, int row ) throws DebugError;
    void DoubleClickOn( IDebugWindow w, int row ) throws DebugError;

    // Breakpoints
    Object BreakSet( String where ) throws DebugError;
    void  BreakClear( Object b ) throws DebugError;
    void BreakEnable( Object b, boolean enabled ) throws DebugError;
    void BreakSetCount( Object b, int count ) throws DebugError;
    int  BreakGetCount( Object b ) throws DebugError;
    void BreakSetCondition( Object b, String condition ) throws
DebugError;
    String BreakGetCondition( Object b ) throws DebugError;
    Vector GetBreaks() throws DebugError;

    // Scripting
    void RunScript( String args[] ) throws DebugError;
    void AddEventHandler( DebugScript s );
    void RemoveEventHandler( DebugScript s );

    // Miscellaneous
    void EvalRun( String expr ) throws DebugError;
    void QueryRun( String query ) throws DebugError;
    void QueryMoreRows() throws DebugError;
    Vector GetClassNames();
    Vector GetProcedureNames();
    Vector WindowContents( IDebugWindow window ) throws DebugError;
    boolean AtBreak();
    boolean IsRunning();
    boolean AtStackTop();
    boolean AtStackBottom();
    void SetStatusText( String msg );
    String GetStatusText();
    void WaitCursor();
    void OldCursor();
    void Error( Exception x );
    void Error( String msg );
    void Warning( String msg );
    String Ask( String title );
    boolean MenuIsChecked( String cmd );
    void MenuSetChecked( String cmd, boolean on );
    void AddInspectItem( String s ) throws DebugError;

    // Constants for DebugScript.OnEvent parameter
    public static final int EventBreak = 0;
    public static final int EventTerminate = 1;
    public static final int EventStep = 2;
    public static final int EventInterrupt = 3;
    public static final int EventException = 4;
    public static final int EventConnect = 5;
};
```

sybase.asa.procdebug.IDebugWindow Interface

The **IDebugWindow** interfaces is as follows:

```
// this interface represents a debugger window
package sybase.asa.procdebug;
public interface IDebugWindow
{
    public int GetSelected();
    /*
        get the currently selected row, or -1 if no selection
    */

    public boolean SetSelected( int i );
    /*
       set the currently selected row.  Ignored if i < 0 or i > #rows
    */

    public String StringAt( int row );
    /*
       get the String representation of the Nth row of the window.
Returns null if row > # rows
    */

    public java.awt.Rectangle GetPosition();
    public void SetPosition( java.awt.Rectangle r );
    /*
        get/set the windows position within the frame
    */

    public void Close();
    /*
        Close (destroy) the window
    */
}
```

Index