



System Administration Guide: Volume 2

Sybase IQ

15.1

DOCUMENT ID: DC00800-01-1510-02

LAST REVISED: November 2009

Copyright © 2009 by Sybase, Inc. All rights reserved.

This publication pertains to Sybase software and to any subsequent release until otherwise indicated in new editions or technical notes. Information in this document is subject to change without notice. The software described herein is furnished under a license agreement, and it may be used or copied only in accordance with the terms of that agreement.

To order additional documents, U.S. and Canadian customers should call Customer Fulfillment at (800) 685-8225, fax (617) 229-9845.

Customers in other countries with a U.S. license agreement may contact Customer Fulfillment via the above fax number. All other international customers should contact their Sybase subsidiary or local distributor. Upgrades are provided only at regularly scheduled software release dates. No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of Sybase, Inc.

Sybase trademarks can be viewed at the Sybase trademarks page at <http://www.sybase.com/detail?id=1011207>. Sybase and the marks listed are trademarks of Sybase, Inc. ® indicates registration in the United States of America.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Unicode and the Unicode Logo are registered trademarks of Unicode, Inc.

All other company and product names mentioned may be trademarks of the respective companies with which they are associated.

Use, duplication, or disclosure by the government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of DFARS 52.227-7013 for the DOD and as set forth in FAR 52.227-19(a)-(d) for civilian agencies.

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

Contents

About This Book	ix
CHAPTER 1 Using Procedures and Batches	1
Overview of procedures	2
Benefits of procedures	2
Introduction to procedures	2
Creating procedures	3
Altering procedures	4
Calling procedures	4
Copying procedures in Sybase Central	5
Deleting procedures	5
Permissions to execute procedures	5
Returning procedure results in parameters	6
Returning procedure results in result sets	6
Introduction to user-defined functions	7
Creating user-defined functions	7
Calling user-defined functions	7
Dropping user-defined functions	8
Permissions to execute user-defined functions	8
Introduction to batches	9
Control statements	9
Using compound statements	10
Declarations in compound statements	10
Atomic compound statements	10
Structure of procedures	10
SQL statements allowed in procedures	11
Declaring parameters for procedures	11
Passing parameters to procedures	12
Passing parameters to functions	12
Returning results from procedures	12
Returning a value using the RETURN statement	12
Returning results as procedure parameters	13
Returning result sets from procedures	13
Returning multiple result sets from procedures	13

Returning variable result sets from procedures.....	14
Using cursors in procedures	14
Cursor management overview	14
Cursor positioning	14
Using cursors on SELECT statements in procedures	15
Errors and warnings in procedures	15
Default error handling in procedures	16
Error handling with ON EXCEPTION RESUME	16
Default handling of warnings in procedures	16
Using exception handlers in procedures	17
Nested compound statements and exception handlers	17
Using the EXECUTE IMMEDIATE statement in procedures	18
Transactions and savepoints in procedures.....	18
Tips for writing procedures.....	18
Hiding the contents of procedures, functions, and views	19
Statements allowed in batches	20
Using SELECT statements in batches	20
Using IQ UTILITIES to create your own stored procedures.....	21
How IQ uses the IQ UTILITIES command	21
Requirements for using IQ UTILITIES.....	22
Choosing procedures to call.....	23
Numbers used by IQ UTILITIES.....	23
Testing your procedures.....	24

CHAPTER 2	Using OLAP.....	25
	About OLAP	26
	OLAP benefits	27
	Understanding OLAP evaluation	27
	GROUP BY clause extensions.....	28
	Group by ROLLUP and CUBE	29
	Analytical functions	43
	Simple aggregate functions.....	44
	Windowing.....	44
	Numeric functions.....	70
	OLAP rules and restrictions	73
	Additional OLAP examples	75
	Example: Window functions in queries.....	75
	Example: Window with multiple functions	77
	Example: Calculate cumulative sum	77
	Example: Calculate moving average.....	78
	Example: ORDER BY results	79
	Example: Multiple aggregate functions in a query.....	80
	Example: Window frame comparing ROWS and RANGE.....	80
	Example: Window frame excludes current row	81

Example: Window frame for RANGE	82
Example: Unbounded preceding and unbounded following	83
Example: Default window frame for RANGE	84
BNF grammar for OLAP functions	85

CHAPTER 3	Sybase IQ as a Data Server	91
	Client/server interfaces to Sybase IQ.....	91
	Configuring IQ Servers with iqdsedit	92
	Sybase applications and Sybase IQ.....	95
	Open Client applications and Sybase IQ.....	95
	Setting up Sybase IQ as an Open Server	96
	System requirements	96
	Starting the database server as an Open Server	96
	Configuring your database for use with Open Client.....	97
	Characteristics of Open Client and jConnect connections	97
	Servers with multiple databases.....	98

CHAPTER 4	Accessing Remote Data	101
	Sybase IQ and remote data	102
	Requirements for accessing remote data.....	102
	Working with remote servers.....	103
	Working with external logins.....	108
	Working with proxy tables	109
	Example: A join between two remote tables	111
	Accessing multiple local databases.....	111
	Sending native statements to remote servers	112
	Using remote procedure calls (RPCs).....	112
	Transaction management and remote data	112
	Remote transaction management overview	113
	Restrictions on transaction management	113
	Internal operations	113
	Query parsing	114
	Query normalization	114
	Query preprocessing.....	114
	Server capabilities	114
	Complete passthrough of the statement	115
	Partial passthrough of the statement.....	115
	Troubleshooting remote data access	115
	Features not supported for remote data.....	115
	Case-sensitivity	116
	Connectivity problems	116
	General problems with queries.....	116
	Queries blocked on themselves	116

	Managing remote data access connections	117
CHAPTER 5	Server Classes for Remote Data Access.....	119
	Server classes overview	119
	JDBC-based server classes	119
	Configuration notes for JDBC classes.....	120
	Server class sajdbc	120
	Server class asejdbc	120
	ODBC-based server classes	121
	Defining ODBC external servers	121
	Server class saodbc	122
	Server class aseodbc	122
	Server class db2odbc	123
	Server class oraodbc.....	123
	Server class mssodbc	124
	Server class odbcc.....	124
CHAPTER 6	Automating Tasks Using Schedules and Events	127
	Introduction to scheduling and event handling.....	128
	Understanding schedules.....	128
	Defining schedules	129
	Understanding events	129
	Choosing a system event.....	129
	Defining trigger conditions for events	130
	Understanding event handlers	131
	Developing event handlers.....	131
	Schedule and event internals.....	132
	How the database server checks for system events	132
	How the database server checks for scheduled times.....	132
	How event handlers are executed.....	132
	Scheduling and event handling tasks.....	133
	Adding a schedule or event to a database	133
	Adding a manually-triggered event to a database.....	133
	Triggering an event handler	133
	Debugging an event handler	134
	Retrieving information about an event or schedule	134
APPENDIX A	Debugging Logic in the Database.....	135
	Introduction to debugging in the database	135
	Debugger features.....	135
	Requirements for using the debugger	136
	Tutorial 1: Getting started with the debugger	136

Lesson 1: Connect to a database and start the debugger	136
Tutorial 2: Debugging a stored procedure.....	137
Tutorial 3: Debugging a Java class	137
Preparing the database	137
Displaying Java source code into the debugger.....	138
Set a breakpoint	138
Run the method	139
Stepping through source code	139
Inspecting and modifying variables	140
Working with breakpoints	141
Working with variables	141
Writing debugger scripts	142
sybase.asa.procdebug.DebugScript class	142
sybase.asa.procdebug.IDebugAPI interface	143
sybase.asa.procdebug.IDebugWindow interface	147
Index	149

About This Book

Subject

Sybase® IQ is a high-performance decision support server designed specifically for data warehouses and data marts. This book, *System Administration Guide: Volume 2*, presents concepts and procedures necessary for programming with Sybase IQ.

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.

Related Sybase IQ documents

The Sybase IQ 15.1 documentation set includes:

- *Release Bulletin* provides information about last-minute changes to the product and documentation.
- *Installation and Configuration Guide* provides platform-specific instructions on installing, migrating to a new version, and configuring Sybase IQ for a particular platform.
- *Advanced Security in Sybase IQ* covers the use of user encrypted columns within the Sybase IQ data repository. You need a separate license to install this product option.
- *Error Messages* lists Sybase IQ error messages referenced by Sybase error code, SQLCode, and SQLState, and SQL preprocessor errors and warnings.
- *IMSL Numerical Library Users Guide: Volume 2 of 2 C Stat Library* contains a concise description of the IMSL C Stat Library time series C functions. This book is only available to RAP – The Trading Edition™ Enterprise users.
- *Introduction to Sybase IQ* includes hands-on exercises for those unfamiliar with Sybase IQ or with the Sybase Central™ database management tool.
- *Large Objects Management in Sybase IQ* explains storage and retrieval of binary large objects (BLOBs) and character large objects (CLOBs) within the Sybase IQ data repository. You need a separate license to install this product option.

-
- *New Features in Sybase IQ 15.0* documents new features and behavior changes for version 15.0.
 - *New Features Summary Sybase IQ 15.1* summarizes new features and behavior changes for the current version.
 - *Performance and Tuning Guide* describes query optimization, design, and tuning issues for very large databases.
 - *Quick Start* lists steps to build and query the demo database provided with Sybase IQ for validating the Sybase IQ software installation. Includes information on converting the demo database to multiplex.
 - *Reference Manual* – Includes two reference guides to Sybase IQ:
 - *Reference: Building Blocks, Tables, and Procedures* describes SQL, stored procedures, data types, and system tables that Sybase IQ supports.
 - *Reference: Statements and Options* describes the SQL statements and options that Sybase IQ supports.
 - *System Administration Guide* – Includes two volumes:
 - *System Administration Guide: Volume 1* describes startup, connections, database creation, population and indexing, versioning, collations, system backup and recovery, troubleshooting, and database repair.
 - *System Administration Guide: Volume 2* describes writing and running procedures and batches, programming with OLAP, accessing remote data, setting up IQ as an Open Server, scheduling and event handling, programming with XML, and debugging.
 - *User-Defined Functions Guide* provides information about the user-defined functions, their parameters, and possible usage scenarios.
 - *Using Sybase IQ Multiplex* tells how to use multiplex capability, designed to manage large query loads across multiple nodes.
 - *Utility Guide* provides Sybase IQ utility program reference material, such as available syntax, parameters, and options.

**Related SQL
Anywhere
documentation**

Because Sybase IQ shares many components with SQL Anywhere Server, a component of the SQL Anywhere® package, Sybase IQ supports many of the same features as SQL Anywhere Server. The IQ documentation set refers you to SQL Anywhere documentation, where appropriate.

Documentation for SQL Anywhere includes:

- *SQL Anywhere Server – Database Administration* describes how to run, manage, and configure SQL Anywhere databases. It describes database connections, the database server, database files, backup procedures, security, high availability, and replication with Replication Server®, as well as administration utilities and options.
- *SQL Anywhere Server – Programming* describes how to build and deploy database applications using the C, C++, Java, PHP, Perl, Python, and .NET programming languages such as Visual Basic and Visual C#. This book also describes a variety of programming interfaces such as ADO.NET and ODBC.
- *SQL Anywhere Server – SQL Reference* provides reference information for system procedures, and the catalog (system tables and views). It also provides an explanation of the SQL Anywhere implementation of the SQL language (search conditions, syntax, data types, and functions).
- *SQL Anywhere Server – SQL Usage* describes how to design and create databases; how to import, export, and modify data; how to retrieve data; and how to build stored procedures and triggers.

You can also refer to the SQL Anywhere documentation in the SQL Anywhere 11.01 collection at Product Manuals at <http://sybooks.sybase.com> and in DocCommentXchange at http://dcx.sybase.com/dcx_home.php.

Related SySAM documents

Documentation for Sybase Software Asset Management (SySAM) includes:

- *Sybase Software Asset Management (SySAM) 2* introduces asset management concepts and provides instructions for establishing and administering SySAM 2 licenses.
- *SySAM 2 Quick Start Guide* tells you how to get your SySAM-enabled Sybase product up and running.
- *FLEXnet Licensing End User Guide* explains FLEXnet Licensing for administrators and end users and describes how to use the tools that are part of the standard FLEXnet Licensing distribution kit from Sybase.

Other sources of information

Use the Sybase Getting Started CD, the SyBooks™ CD, and the Sybase Product Manuals Web site to learn more about your product:

- The Getting Started CD contains release bulletins and installation guides in PDF format, and may also contain other documents or updated information not included on the SyBooks CD. It is included with your software. To read or print documents on the Getting Started CD, you need Adobe Acrobat Reader, which you can download at no charge from the Adobe Web site using a link provided on the CD.
- The SyBooks CD contains product manuals and is included with your software. The Eclipse-based SyBooks browser allows you to access the manuals in an easy-to-use, HTML-based format.

Some documentation may be provided in PDF format, which you can access through the PDF directory on the SyBooks CD. To read or print the PDF files, you need Adobe Acrobat Reader.

Refer to the *SyBooks Installation Guide* on the Getting Started CD, or the *README.txt* file on the SyBooks CD for instructions on installing and starting SyBooks.

- The Sybase Product Manuals Web site is an online version of the SyBooks CD that you can access using a standard Web browser. In addition to product manuals, you will find links to EBFs/Maintenance, Technical Documents, Case Management, Solved Cases, newsgroups, and the Sybase Developer Network.

To access the Sybase Product Manuals Web site, go to Product Manuals at <http://www.sybase.com/support/manuals/>.

Sybase certifications on the Web

Technical documentation at the Sybase Web site is updated frequently.

❖ Finding the latest information on product certifications

- 1 Point your Web browser to Technical Documents at <http://certification.sybase.com/ucr/search.do>.
- 2 Either select the product family and product under Search by Base Product; or select the platform and product under Search by Platform.
- 3 Select Search to display the availability and certification report for the selection.

❖ Finding the latest information on component certifications

- 1 Point your Web browser to Availability and Certification Reports at <http://certification.sybase.com/>.
- 2 Either select the product family and product under Search by Base Product; or select the platform and product under Search by Platform.

- 3 Select Search to display the availability and certification report for the selection.

❖ **Creating a personalized view of the Sybase Web site (including support pages)**

Set up a MySybase profile. MySybase is a free service that allows you to create a personalized view of Sybase Web pages.

- 1 Point your Web browser to Technical Documents at <http://www.sybase.com/support/techdocs/>.
- 2 Click MySybase and create a MySybase profile.

Sybase EBFs and software maintenance

❖ **Finding the latest information on EBFs and software maintenance**

- 1 Point your Web browser to the Sybase Support Page at <http://www.sybase.com/support>.
- 2 Select EBFs/Maintenance. If prompted, enter your MySybase user name and password.
- 3 Select a product.
- 4 Specify a time frame and click Go. A list of EBF/Maintenance releases is displayed.

Padlock icons indicate that you do not have download authorization for certain EBF/Maintenance releases because you are not registered as a Technical Support Contact. If you have not registered, but have valid information provided by your Sybase representative or through your support contract, click Edit Roles to add the “Technical Support Contact” role to your MySybase profile.

- 5 Click the Info icon to display the EBF/Maintenance report, or click the product description to download the software.

Syntax conventions

This documentation uses these conventions in syntax descriptions:

- **Keywords** SQL keywords are shown in UPPERCASE. However, SQL keywords are case-insensitive, so you can enter keywords in any case; SELECT, Select, and select are equivalent.
- **Placeholders** Items that must be replaced with appropriate identifiers or expressions are shown in *italics*.

- **Continuation** Lines beginning with an ellipsis (...) are a continuation of the statements from the previous line.
- **Repeating items** Lists of repeating items are shown with an element of the list followed by an ellipsis (...). One or more list elements are allowed. If multiple elements are specified, they must be separated by commas.
- **Optional portions** Optional portions of a statement are enclosed by square brackets. For example:

```
RELEASE SAVEPOINT [ savepoint-name ]
```

The square brackets indicate that the *savepoint-name* is optional. Do not type the brackets.

- **Options** When none or only one of a list of items must be chosen, the items are separated by vertical bars and the list enclosed in square brackets. For example:

```
[ ASC | DESC ]
```

The square brackets indicate that you can choose ASC, DESC, or neither. Do not type the brackets.

- **Alternatives** When precisely one of the options must be chosen, the alternatives are enclosed in curly braces. For example:

```
QUOTES { ON | OFF }
```

The curly braces indicate that you must include either ON or OFF. Do not type the brackets.

Typographic conventions

Table 1 lists the typographic conventions used in this documentation.

Table 1: Typographic conventions

Item	Description
Code	SQL and program code appears in a monospaced (fixed-width) font.
User entry	Text entered by the user is shown in a monospaced (fixed-width) font.
<i>file names</i>	File names are shown in italic.
database objects	Names of database objects, such as tables and procedures, are shown in bold, sans serif type in print, and in italic online.

Accessibility features

This document is available in an HTML version that is specialized for accessibility. You can navigate the HTML with an adaptive technology such as a screen reader, or view it with a screen enlarger.

Sybase IQ 15.1 and the HTML documentation have been tested for compliance with U.S. government Section 508 Accessibility requirements. Documents that comply with Section 508 generally also meet non-U.S. accessibility guidelines, such as the World Wide Web Consortium (W3C) guidelines for Web sites.

Configuring your accessibility tool

You might need to configure your accessibility tool for optimal use. Some screen readers pronounce text based on its case; for example, they pronounce ALL UPPER CASE TEXT as initials, and Mixed Case Text as words. You might find it helpful to configure your tool to announce syntax conventions. Consult the documentation for your tool for information on using screen readers.

For information about how Sybase supports accessibility, see Sybase Accessibility at <http://www.sybase.com/accessibility>. The Sybase Accessibility site includes links to information on Section 508 and W3C standards.

For a Section 508 compliance statement for Sybase IQ, go to Sybase Accessibility at <http://www.sybase.com/products/accessibility>.

The demo database

Sybase IQ includes scripts that create a demo database (*iqdemo.db*). Many of the queries and code samples in this document use the demo database as a data source.

The demo database contains internal information about a small company (employees, departments, and financial data), as well as product and sales information (products, sales orders, customers, and contacts).

See the Sybase IQ installation guide for your platform or talk to your system administrator for more information about the demo database.

If you need help

Each Sybase installation that has purchased a support contract has one or more designated people who are authorized to contact Sybase Technical Support. If you cannot resolve a problem using the manuals or online help, please have the designated person contact Sybase Technical Support or the Sybase subsidiary in your area.

Using Procedures and Batches

About this chapter

This chapter explains how you 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. 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.

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

Contents

Topic	Page
Overview of procedures	2
Benefits of procedures	2
Introduction to procedures	2
Introduction to user-defined functions	7
Introduction to batches	9
Control statements	9
Structure of procedures	10
Returning results from procedures	12
Using cursors in procedures	14
Errors and warnings in procedures	15
Using the EXECUTE IMMEDIATE statement in procedures	18
Transactions and savepoints in procedures	18
Tips for writing procedures	18
Statements allowed in batches	20
Using IQ UTILITIES to create your own stored procedures	21

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 statement) and conditional execution (IF statement and CASE statement) of SQL statements.

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

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

Introduction to procedures

To use procedures, know how to:

- Create procedures
- Call procedures from a database application
- Drop or remove procedures
- Control who has permission to use procedures

This section discusses these aspects of using procedures, and also describes some of the different uses of procedures.

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`

Creating procedures

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

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

Sybase IQ example

Note For examples, use the Sybase IQ demo database `iqdemo.db`. For installation information, see “The demo database” on page xv.

```
create procedure new_dept(IN id INT, IN name CHAR(35),
IN head_id INT)
BEGIN
INSERT
INTO GROUP0.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. See “Using remote procedure calls (RPCs)” on page 112. 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 “Altering procedures” in SQL Anywhere documentation in SQL Anywhere 11.0.1 > SQL Anywhere Server - *SQL Usage* > *Stored Procedures and Triggers* > *Using procedures, triggers, and batches* > *Introduction to procedures*.

For information on altering database object properties, see Chapter 4, “Managing Databases,” in *Introduction to Sybase IQ*.

For information on granting or revoking permissions for procedures, see “Granting permissions on procedures” and “Revoking user permissions” in Chapter 8, “Managing User IDs and Permissions,” in the *System Administration Guide: Volume 1*.

See also `ALTER PROCEDURE` statement and `CREATE PROCEDURE` statement in *Reference: Statements and Options*.

Calling procedures

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

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

Also see:

- CALL statement in Chapter 1, “SQL Statements,” in *Reference: Statements and Options*.
- “Permissions to execute procedures” on page 5.

Copying procedures in Sybase Central

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

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

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 “Deleting procedures” in SQL Anywhere documentation in SQL Anywhere 11.0.1 > SQL Anywhere Server - *SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Introduction to 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 “Granting permissions on procedures” in *System Administration Guide: Volume 1*.

Returning procedure results in parameters

Procedures return results to the calling environment 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 “Returning procedure results in parameters” in SQL Anywhere documentation in SQL Anywhere 11.0.1 > SQL Anywhere Server - *SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Introduction to procedures*.

Sybase IQ example

Note For examples, use the Sybase IQ demo database iqdemo.db. For installation information, see “The demo database” on page xv.

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

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

For a complete description of the CREATE FUNCTION syntax, including performance considerations and differences between SQL Anywhere and IQ, see Chapter 1, “SQL Statements,” in *Reference: Statements and Options*.

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

Sybase IQ example

Note For examples, use the Sybase IQ demo database iqdemo.db. For installation information, see “The demo database” on page xv.

```
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 “Dropping user-defined functions” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches > Introduction to 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 “Granting permissions on procedures” in Chapter 8, “Managing User IDs and Permissions,” *System Administration Guide: Volume 1*.

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 “Introduction to batches” in SQL Anywhere documentation in SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches.

Sybase IQ example

Note For examples, use the Sybase IQ demo database iqdemo.db. For installation information, see “The demo database” on page xv.

```
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 “Control statements” in SQL Anywhere documentation in SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches.

For complete descriptions of each, see the entries in Chapter 1, “SQL Statements,” in *Reference: Statements and Options*.

Using compound statements

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

See “Using compound statements” in SQL Anywhere documentation in 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

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

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

Atomic compound statements

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

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

Structure of procedures

The body of a procedure consists of a compound statement as discussed in “Using compound statements” on page 10. A compound statement consists of a **BEGIN** and an **END**, enclosing a set of SQL statements. Semicolons delimit each statement.

The SQL statements that can occur in procedures are described in “SQL statements allowed in procedures” on page 11.

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 (see “Control statements” on page 9)
- Cursor statements (see “Using cursors in procedures” on page 14)
- Exception handling statements (see “Using exception handlers in procedures” on page 17)
- 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 “Transactions and savepoints in procedures” on page 18).

For details, see the Usage section for each SQL statement in Chapter 1, “SQL Statements,” in *Reference: Statements and Options*.

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 (see Chapter 3, “SQL Data Types,” in *Reference: Building Blocks, Tables, and Procedures*), and must be prefixed with one of the keywords IN, OUT or INOUT.

See “Declaring parameters for procedures” in SQL Anywhere documentation in 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

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

See “Passing parameters to procedures” in SQL Anywhere documentation in *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

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

See “Passing parameters to functions” in SQL Anywhere documentation in *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*.

Returning results from procedures

Procedures can return results that are a single row of data, or multiple rows. 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 “Introduction to procedures” on page 2. For more detailed information, see the following sections.

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 “Returning a value using the RETURN statement” in SQL Anywhere documentation in *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

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

See “Returning results as procedure parameters” in SQL Anywhere documentation in *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

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

See “Returning result sets from procedures” in SQL Anywhere documentation in *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

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 “Returning multiple result sets from procedures” in SQL Anywhere documentation in *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

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

For information about the **DESCRIBE** statement, see Chapter 1, “SQL Statements,” in *Reference: Statements and Options*.

Using 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 “Cursor management overview” in SQL Anywhere documentation in *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*.

The `sp_iqcursorinfo` stored procedure displays information about cursors currently open on the server. For more information, see “`sp_iqcursorinfo` procedure” in Chapter 7, “System Procedures,” in *Reference: Building Blocks, Tables, and Procedures*.

Cursor positioning

A cursor can be positioned in a variety of places.

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

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.

Using cursors on SELECT statements in procedures

The following procedure uses a cursor on a SELECT statement. It illustrates several features of the stored procedure language, and is based on the same query used in the ListCustomerValue procedure described in “Returning result sets from procedures”.

See “Using cursors on SELECT statements in procedures” in SQL Anywhere documentation in *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*.

Errors and warnings in procedures

After an application program executes a SQL statement, it can examine a **return code** (or status code). This return code indicates whether the statement executed successfully or failed and gives the reason for the failure.

See “Errors and warnings in procedures and triggers” in *SQL Anywhere Server - SQL Usage*.

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

Default error handling in procedures

This section describes how Sybase IQ handles errors that occur during a procedure execution, if you have no error handling built in to the procedure.

See “Default error handling in procedures and triggers” in SQL Anywhere documentation in *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.

Error handling with ON EXCEPTION RESUME

If the ON EXCEPTION RESUME clause is included in the CREATE PROCEDURE statement, the procedure checks the following 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 “Error handling with ON EXCEPTION RESUME” in SQL Anywhere documentation in *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 handling of warnings in procedures

Errors and warnings are handled differently. While 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 “Default error handling of warnings in procedures and triggers” in SQL Anywhere documentation in *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. You can ignore information about triggers in the SQL Anywhere documentation.

Using exception handlers in procedures

You may want to intercept certain types of errors and handle them within a procedure, rather than pass the error back to the calling environment. This is done through the use of an **exception handler**.

See “Using exception handlers in procedures and triggers” in SQL Anywhere documentation in *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.

Nested compound statements and exception handlers

You can use nested compound statements to give you more control over which statements execute following an error and which do not.

See “Nested compound statements and exception handlers” in SQL Anywhere documentation in *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 the EXECUTE IMMEDIATE statement in procedures

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

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

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

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

For more information, see:

- “Savepoints within transactions” in Chapter 10, “Transactions and Versioning,” in *System Administration Guide: Volume 1*
- Chapter 10, “Transactions and Versioning,” in *System Administration Guide: Volume 1*

Tips for writing procedures

This section provides some pointers for developing procedures. The following subjects are discussed in this topic:

- Checking if you need to change the command delimiter
- Remembering to delimit statements within your procedure
- Using fully-qualified names for tables in procedures
- Specifying dates and times in procedures

For more information on dates and times, see “Date and time data types” in Chapter 3, “SQL Data Types,” in *Reference: Building Blocks, Tables, and Procedures*.
- Verifying procedure input arguments are passed correctly

For information on determining the destination of the MESSAGE statement output, see MESSAGE statement in Chapter 1, “SQL Statements,” in *Reference: Statements and Options*.

See “Tips for writing procedures” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches*.

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

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 within 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 “Hiding the contents of procedures, functions, triggers and views” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches*.

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

For more information, see 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 “Statements allowed in procedures, triggers, events, and batches” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, 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 “Using SELECT statements in batches” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Using procedures, triggers, and batches*.

Sybase IQ example

Note For examples, use the Sybase IQ demo database iqdemo.db. For installation information, see “The demo database” on page xv.

```
EIF EXISTS(SELECT *
FROM isystab
WHERE table_name='Employees' )
THEN
SELECT Surname AS LastName,
emp_fname AS FirstName
FROM Employees;
SELECT Surname, GivenName
```

```
FROM Customers;  
  
SELECT Surname, GivenName  
  
FROM Contacts;  
  
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 may want to create your own variants of some of these procedures. Among the ways you might do this are:

- Create a procedure that calls a system stored procedure.
- Create a procedure that is independent of the system stored procedures but performs a similar function.
- 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.

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 Table 1-1 on page 24 for a list of the numbers used as IQ UTILITIES arguments.

Requirements for using IQ UTILITIES

The requirements discussed throughout this chapter also apply when writing stored procedures using IQ UTILITIES; there are also several crucial requirements.

If you use IQ UTILITIES in your procedure, you must use this statement in exactly the same way that existing procedures use it. In particular, you must:

- Declare a local temporary table in which you store results from the procedure. This table must have exactly the same schema as in the system stored procedures, including column names, column width, column order, data types, precision, and so on.
- Issue an EXECUTE IMMEDIATE command to execute IQ UTILITIES and store its results in the temporary table.
- Where the IQ UTILITIES statement includes a number, you must use exactly the same number as in the system stored procedures, for exactly the same purpose. You cannot create your own numbers or change the way in which existing numbers are used.

In other words, you must use the local temporary table and IQ UTILITIES statement in exactly the same way as system stored procedures:

- Do not eliminate columns or add extra columns.
- Do not alter the contents of the table used in the system procedures. Users who call your procedure may also call other procedures that use the same table.

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

IQ system procedures are in the file *iqprocs.sql* in the *scripts* directory of your IQ installation directory.

The syntax for IQ UTILITIES is:

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

For examples of how this command is used, refer to the *iqprocs.sql* file.

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.

The numbers in IQ system procedures are fixed. They do not change from release to release, although new numbers may be added in future releases.

Give your procedures a different name from the system procedures.

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 Chapter 7, “System Procedures,” in *Reference: Building Blocks, Tables, and Procedures*.

Table 1-1: IQ UTILITIES values used in system procedures

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

Testing your procedures

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

Using OLAP

About this chapter

OLAP (online analytical processing) is an efficient method of data analysis on 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.

Contents

Topic	Page
About OLAP	26
GROUP BY clause extensions	28
Analytical functions	43
Simple aggregate functions	44
Windowing	44
Ranking functions	58
Windowing aggregate functions	62
Statistical aggregate functions	64
Distribution functions	67
Numeric functions	70
OLAP rules and restrictions	73
Additional OLAP examples	75
BNF grammar for OLAP functions	85

About OLAP

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.

These 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

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 “Windowing” on page 44.

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.

Understanding 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.

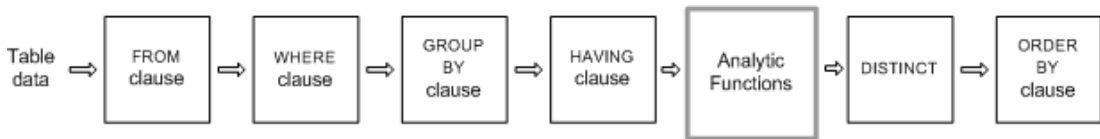
The HAVING clause acts as a filter, much like the WHERE clause, on the results of the GROUP BY clause.

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 window functions (rank and aggregates) are computed for result windows.

Figure 2-1: SQL processing for OLAP



See “Grammar rule 2” on page 85. See also “BNF grammar for OLAP functions” on page 85.

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.

Prefixes

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

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.

	Year	Quarter	Orders
Exclude Quarter and Year prefix	(NULL)	(NULL)	648
	2000	(NULL)	380
	2000	1	87
	2000	2	77
Exclude Quarter prefix	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

Two important syntactic shortcuts exist to concisely specify common grouping for prefixes. The first of these patterns is called ROLLUP, and the second is called CUBE.

Group by ROLLUP

The ROLLUP operator requires an ordered list of grouping expressions to be supplied as arguments, as in the following 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 Table 2-1.

Table 2-1: 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

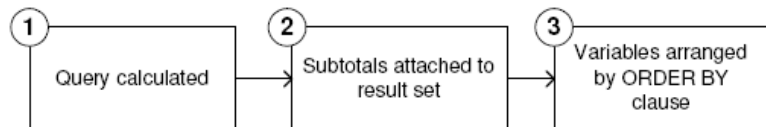
ROLLUP is equivalent to a UNION of a set of GROUP BY queries. The result sets of the following queries are identical. The result set of GROUP BY (A, B) consists of subtotals over all those rows in which A and B are held constant. To make a union possible, column C is assigned NULL.

This ROLLUP query...	Is equivalent to this query without ROLLUP...
<pre>SELECT A, B, C, SUM(D) FROM T1 GROUP BY ROLLUP (A, B, C);</pre>	<pre>SELECT * FROM (T1 (SELECT A, B, C, SUM(D) GROUP BY A, B, C) UNION ALL (SELECT A, B, NULL, SUM(D) GROUP BY A, B) UNION ALL (SELECT A, NULL, NULL, SUM(D) GROUP BY A) UNION ALL (SELECT NULL, NULL, NULL, SUM(D))))</pre>

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 2-2 shows conceptually how Sybase IQ creates subtotals:

Figure 2-2: Subtotals



- 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 employee table and the sales_order table. The query selects female employees who live in either 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. For installation information, see “The demo database” on page xv.

```
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 following result set is from the query.

EmployeeID	YEAR	Orders	GE	GY
-----	----	-----	--	--
NULL	NULL	1	1	0
NULL	NULL	165	1	1
1090	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 “Expressions,” Chapter 2, “SQL Language Elements,” in *Reference: Building Blocks, Tables, and Procedures*.

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 results from the above query:

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(ProductID
as char(8)) END
FROM SalesOrderItems
GROUP BY ROLLUP(Year, ProductID) HAVING OSum > 36
ORDER BY Year, ProductID;

```

The results from the above query:

Year	ProductID	OSum	PROD_IDS	
-----	-----	---	-----	-----
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 Orders
```

```

FROM SalesOrders WHERE region IN ('Canada',
'Eastern') AND quarter IN (1, 2)
GROUP BY ROLLUP (Year, Quarter, Region)
ORDER BY Year, Quarter, Region

```

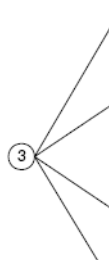
The following figure illustrates the result set from the above query. Each subtotal row contains a NULL in the column or columns over which the subtotal is computed.

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.

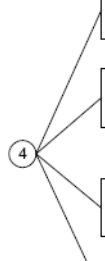
The rows marked [2] represent the total number of sales orders in each year (2000) and (2001) in quarters 1 and 2 in Canada and the Eastern region. The values of these rows [2] are equal to the grand total represented in row [1].

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 Table 2-2.

Table 2-2: 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 AVERAGE
FROM employees WHERE state IN ('MA' , 'CA')
GROUP BY State, Sex, DepartmentID
ORDER 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 State
      END 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)) end
      AS c_dept, COUNT(*), CAST(ROUND(AVG(salary),2) AS
      NUMERIC(18,2))AS AVERAGE
FROM 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 (*) Orders
FROM SalesOrders
GROUP 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
1	(NULL)	(NULL)	648
2	(NULL)	1	226
	(NULL)	2	196
	(NULL)	3	101
	(NULL)	4	125
3	2000	(NULL)	380
	2000	1	87
	2000	2	77
	2000	3	91
	2000	4	125
3	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 “Windowing” on page 44.

See “Aggregate functions,” Chapter 4, “SQL Functions,” in *Reference: Building Blocks, Tables, and Procedures*.

Windowing

A major feature of the ANSI SQL extensions for OLAP is a construct called a **window**. This windowing extension let 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>
```

See also “Grammar rule 6” on page 85.

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...)
```

An OLAP window's three essential parts

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.

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). See “Window partitioning” on page 46.
- Ordering rows within partitions (ORDER BY clause). See “Window ordering” on page 47.
- Defining window frames (ROWS/RANGE specification). See “Window framing” on page 48.

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. See “Window framing” on page 48.

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

- “Ranking functions” on page 58
- “Windowing aggregate functions” on page 62
- “Statistical aggregate functions” on page 64
- “Distribution functions” on page 67

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.

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. See “Grammar rule 31” on page 87.

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. (See “Window framing” on page 48.) 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 “Window framing.”

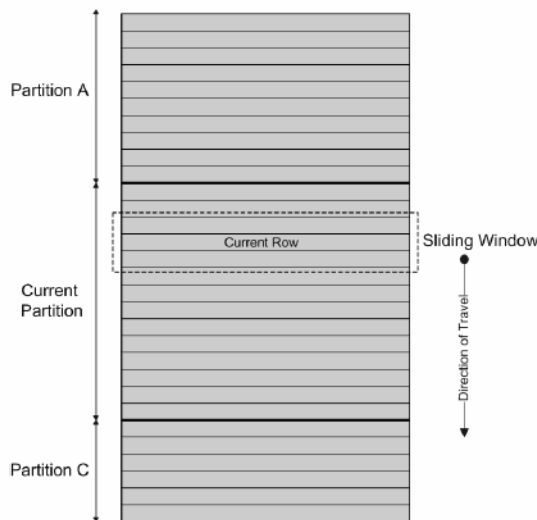
Window framing

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 2-3: Three-row moving window with partitioned input



UNBOUNDED
PRECEEDING 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. See “ROWS” on page 50 and “RANGE” on page 53.

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 **PREDCEEDING** 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**.

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. See Figure 2-3 on page 48.

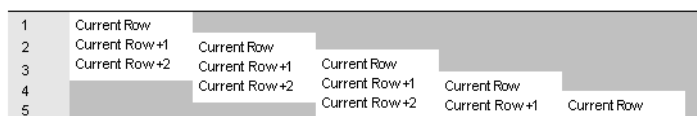
Be aware of meaningless results that may be generated by gaps in the windowed values when using ROWS. If the set of values is not continuous, consider using RANGE instead of ROWS, because a window definition based on RANGE automatically handles adjacent rows with duplicate values and does not include 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 “Computing deltas between adjacent rows” on page 55.

Row-based window frames In the example in Figure 2-4, 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 in Figure 2-4.

Figure 2-4: Row-based window frames



The window frame in Figure 2-4 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_A V G
1	A	10	53.3
2	A	50	
3	A	100	
4	A	120	90.0
5	A	500	
			240
			310
			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] = $(120 + 500 + \text{NULL})/3$
- Row [5] = $(500 + \text{NULL} + \text{NULL})/3$

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.

RANGE

Range-based window frames The previous example, Row-based window frames, demonstrates one among many row-based window frame definitions. The SQL/OLAP syntax also 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	6000	3000
3	A	2001	1000	3000
4	A	2002	12000	6250
5	A	2002	3000	6250

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	51432.000		
191	Bertrand	29800.000	39300.000	-9500.000
278	Melkisetian	48500.000	42300.000	6200.000
299	Overbey	39300.000	41700.750	-2400.750
318	Crow	41700.750	45000.000	-3299.250

586	Coleman	42300.000	46200.000	-3900.000
690	Poitras	46200.000	29800.000	16400.000
703	Martinez	55500.800	51432.000	4068.800
949	Savarino	72300.000	55500.800	16799.200
1101	Preston	37803.000	48500.000	-10697.000
1142	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.

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 BY
startdate rows between unbounded preceding and
current row) AS sum_salary
FROM Employees
WHERE 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 answer questions 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.” The functions include `RANK()`, `DENSE_RANK()`, `PERCENT_RANK()`, and `NTILE()` with a `PARTITION BY` clause. See “Ranking functions” on page 58.

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

```
<RANK FUNCTION TYPE> ::=  
RANK | DENSE_RANK | PERCENT_RANK | 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.

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 “RANK function [Analytical],” Chapter 4, “SQL Functions,” in *Reference: Building Blocks, Tables, and Procedures*.

DENSE_RANK() function

Although RANK returns duplicate values in the ranking sequence when there are ties between values, 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 “DENSE_RANK function [Analytical],” Chapter 4, “SQL Functions,” in *Reference: Building Blocks, Tables, and Procedures*.

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. In other words, 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 PERCENT_RANK function [Analytical] in Chapter 4, “SQL Functions,” in *Reference: Building Blocks, Tables, and Procedures*.

Ranking examples

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

```
SELECT Surname, Salary, Sex, 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	salary	sex	rank
-----	-----	---	----

Scott	96300.000	M	1
Lull	87900.000	M	2
Pastor	74500.000	F	3
Shishov	72995.000	F	4
Samuels	37400.000	M	19

Ranking example 2 Using the query from Ranking example 1, 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, Salary, Sex, 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	salary	sex	rank
-----	-----	---	----
Savarino	72300.000	F	1
Clark	45000.000	F	2
Overbey	39300.000	M	3

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	percent
-----	-----	---	-----
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

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 AS
```

```

Dept, CAST(Salary AS numeric(10,2) ) AS Sal,
CAST(AVG(Sal) OVER(PARTITION BY DepartmentID) AS
numeric(10, 2)) AS Average, CAST(STDDEV_POP(Sal)
OVER(PARTITION BY DepartmentID) 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;

```

The results from the query:

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

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.

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.

	<p>The covariance functions eliminate all pairs where expression1 or expression2 has a null value.</p> <p>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.</p>
Cumulative distribution	<p>The SQL/OLAP function that calculates the relative position of a single value among a group of rows is CUME_DIST.</p> <p>The window specification must contain an ORDER_BY clause.</p> <p>Composite sort keys are not allowed in the CUME_DIST function.</p>
Regression analysis	<p>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:</p> <ul style="list-style-type: none">• 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-fit 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. <p>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.</p>

**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.

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 “PERCENTILE_CONT function [Analytical]” and “PERCENTILE_DISC function [Analytical],” in Chapter 4, “SQL Functions,” in *Reference: Building Blocks, Tables, and Procedures*.

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>
```

The syntax for each supported numeric value function is shown in Table 2-3.

Table 2-3: Numeric value functions and syntax

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

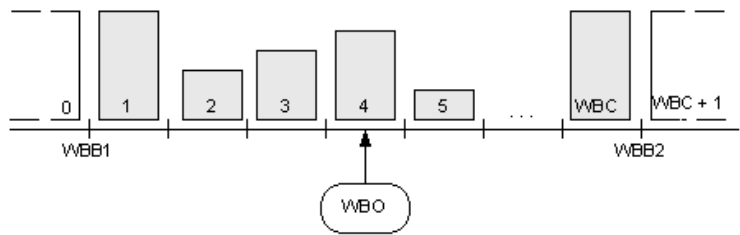
The semantics of the numeric value functions are:

- 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 $(5*9/25) + 1$, 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

“`BIT_LENGTH` function [String],” “`EXP` function [Numeric],” “`FLOOR` function [Numeric],” “`POWER` function [Numeric],” “`SQRT` function [Numeric],” and “`WIDTH_BUCKET` function [Numerical],” Chapter 4, “SQL Functions,” in *Reference: Building Blocks, Tables, and Procedures*.

OLAP rules and restrictions

OLAP functions can be used

Within SQL queries, OLAP functions can be used:

- 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. See Figure 2-3 on page 48.

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

Consider the following query, which 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 BY
                             s.shipdate rows between unbounded preceding and
                             current 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 > 40
ORDER 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. See Figure 2-3 on page 48.

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. From “Grammar rule 32” on page 87:

<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. From “Grammar rule 31” on page 87:

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

Example: Window with multiple functions

To define 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 ws1
FROM 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 rows
between unbounded preceding and current row)
ORDER BY p.id;
```

The results from the above query:

ID	description	quantity	ship_date	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 DepartmentID, start_date, name, salary,
SUM(salary) OVER (PARTITION BY DepartmentID ORDER BY
start_date ROWS BETWEEN UNBOUNDED PRECEDING AND
```

```
        CURRENT ROW)
FROM emp1
ORDER BY DepartmentID, 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	max(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	max(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 following 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	<pre> <SELECT LIST EXPRESSION> ::= <EXPRESSION> <GROUP BY EXPRESSION> <AGGREGATE FUNCTION> <GROUPING FUNCTION> <TABLE COLUMN> <WINDOWED TABLE FUNCTION> </pre>
Grammar rule 2	<pre> <QUERY SPECIFICATION> ::= <FROM CLAUSE> [<WHERE CLAUSE>] [<GROUP BY CLAUSE>] [<HAVING CLAUSE>] [<WINDOW CLAUSE>] [<ORDER BY CLAUSE>] </pre>
Grammar rule 3	<pre> <ORDER BY CLAUSE> ::= <ORDER SPECIFICATION> </pre>
Grammar rule 4	<pre> <GROUPING FUNCTION> ::= GROUPING <LEFT PAREN> <GROUP BY EXPRESSION> <RIGHT PAREN> </pre>
Grammar rule 5	<pre> <WINDOWED TABLE FUNCTION> ::= <WINDOWED TABLE FUNCTION TYPE> OVER <WINDOW NAME OR SPECIFICATION> </pre>
Grammar rule 6	<pre> <WINDOWED TABLE FUNCTION TYPE> ::= <RANK FUNCTION TYPE> <LEFT PAREN> <RIGHT PAREN> ROW_NUMBER <LEFT PAREN> <RIGHT PAREN> <WINDOW AGGREGATE FUNCTION> </pre>
Grammar rule 7	<pre> <RANK FUNCTION TYPE> ::= RANK DENSE RANK PERCENT RANK CUME_DIST </pre>
Grammar rule 8	<pre> <WINDOW AGGREGATE FUNCTION> ::= <SIMPLE WINDOW AGGREGATE FUNCTION> <STATISTICAL AGGREGATE FUNCTION> </pre>
Grammar rule 9	<pre> <AGGREGATE FUNCTION> ::= <DISTINCT AGGREGATE FUNCTION> <SIMPLE AGGREGATE FUNCTION> <STATISTICAL AGGREGATE FUNCTION> </pre>
Grammar rule 10	<pre> <DISTINCT AGGREGATE FUNCTION> ::= <BASIC AGGREGATE FUNCTION TYPE> <LEFT PAREN> </pre>

	<pre><DISTINCT> <EXPRESSION> <RIGHT PAREN> LIST <LEFT PAREN> DISTINCT <EXPRESSION> [<COMMA> <DELIMITER>] [<ORDER SPECIFICATION>] <RIGHT PAREN></pre>
Grammar rule 11	<pre><BASIC AGGREGATE FUNCTION TYPE> ::= SUM MAX MIN AVG COUNT</pre>
Grammar rule 12	<pre><SIMPLE AGGREGATE FUNCTION> ::= <SIMPLE AGGREGATE FUNCTION TYPE> <LEFT PAREN> <EXPRESSION> <RIGHT PAREN> LIST <LEFT PAREN> <EXPRESSION> [<COMMA> <DELIMITER>] [<ORDER SPECIFICATION>] <RIGHT PAREN></pre>
Grammar rule 13	<pre><SIMPLE AGGREGATE FUNCTION TYPE> ::= <SIMPLE WINDOW AGGREGATE FUNCTION TYPE></pre>
Grammar rule 14	<pre><SIMPLE WINDOW AGGREGATE FUNCTION> ::= <SIMPLE WINDOW AGGREGATE FUNCTION TYPE> <LEFT PAREN> <EXPRESSION> <RIGHT PAREN> GROUPING FUNCTION</pre>
Grammar rule 15	<pre><SIMPLE WINDOW AGGREGATE FUNCTION TYPE> ::= <BASIC AGGREGATE FUNCTION TYPE> STDDEV STDDEV_POP STDDEV_SAMP VARIANCE VARIANCE_POP VARIANCE_SAMP</pre>
Grammar rule 16	<pre><STATISTICAL AGGREGATE FUNCTION> ::= <STATISTICAL AGGREGATE FUNCTION TYPE> <LEFT PAREN> <DEPENDENT EXPRESSION> <COMMA> <INDEPENDENT EXPRESSION> <RIGHT PAREN></pre>
Grammar rule 17	<pre><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</pre>
Grammar rule 18	<pre><WINDOW NAME OR SPECIFICATION> ::= <WINDOW NAME> <IN-LINE WINDOW SPECIFICATION></pre>
Grammar rule 19	<pre><WINDOW NAME> ::= <IDENTIFIER></pre>
Grammar rule 20	<pre><IN-LINE WINDOW SPECIFICATION> ::= <WINDOW SPECIFICATION></pre>
Grammar rule 21	<pre><WINDOW CLAUSE> ::= <WINDOW WINDOW DEFINITION LIST></pre>
Grammar rule 22	<pre><WINDOW DEFINITION LIST> ::= <WINDOW DEFINITION> [{ <COMMA> <WINDOW DEFINITION> } . . .]</pre>

Grammar rule 23	<pre> <WINDOW DEFINITION> ::= <NEW WINDOW NAME> AS <WINDOW SPECIFICATION> </pre>
Grammar rule 24	<pre> <NEW WINDOW NAME> ::= <WINDOW NAME> </pre>
Grammar rule 25	<pre> <WINDOW SPECIFICATION> ::= <LEFT PAREN> <WINDOW SPECIFICATION> <DETAILS> <RIGHT PAREN> </pre>
Grammar rule 26	<pre> <WINDOW SPECIFICATION DETAILS> ::= [<EXISTING WINDOW NAME>] [<WINDOW PARTITION CLAUSE>] [<WINDOW ORDER CLAUSE>] [<WINDOW FRAME CLAUSE>] </pre>
Grammar rule 27	<pre> <EXISTING WINDOW NAME> ::= <WINDOW NAME> </pre>
Grammar rule 28	<pre> <WINDOW PARTITION CLAUSE> ::= PARTITION BY <WINDOW PARTITION EXPRESSION LIST> </pre>
Grammar rule 29	<pre> <WINDOW PARTITION EXPRESSION LIST> ::= <WINDOW PARTITION EXPRESSION> [{ <COMMA> <WINDOW PARTITION EXPRESSION> } . . .] </pre>
Grammar rule 30	<pre> <WINDOW PARTITION EXPRESSION> ::= <EXPRESSION> </pre>
Grammar rule 31	<pre> <WINDOW ORDER CLAUSE> ::= <ORDER SPECIFICATION> </pre>
Grammar rule 32	<pre> <WINDOW FRAME CLAUSE> ::= <WINDOW FRAME UNIT> <WINDOW FRAME EXTENT> </pre>
Grammar rule 33	<pre> <WINDOW FRAME UNIT> ::= ROWS RANGE </pre>
Grammar rule 34	<pre> <WINDOW FRAME EXTENT> ::= <WINDOW FRAME START> <WINDOW FRAME BETWEEN> </pre>
Grammar rule 35	<pre> <WINDOW FRAME START> ::= UNBOUNDED PRECEDING <WINDOW FRAME PRECEDING> CURRENT ROW </pre>
Grammar rule 36	<pre> <WINDOW FRAME PRECEDING> ::= <UNSIGNED VALUE SPECIFICATION> PRECEDING </pre>
Grammar rule 37	<pre> <WINDOW FRAME BETWEEN> ::= BETWEEN <WINDOW FRAME BOUND 1> AND <WINDOW FRAME BOUND 2> </pre>
Grammar rule 38	<pre> <WINDOW FRAME BOUND 1> ::= <WINDOW FRAME BOUND> </pre>
Grammar rule 39	<pre> <WINDOW FRAME BOUND 2> ::= <WINDOW FRAME BOUND> </pre>
Grammar rule 40	<pre> <WINDOW FRAME BOUND> ::= </pre>

	<pre><WINDOW FRAME START> UNBOUNDED FOLLOWING <WINDOW FRAME FOLLOWING></pre>
Grammar rule 41	<pre><WINDOW FRAME FOLLOWING> ::= <UNSIGNED VALUE SPECIFICATION> FOLLOWING</pre>
Grammar rule 42	<pre><GROUP BY EXPRESSION> ::= <EXPRESSION></pre>
Grammar rule 43	<pre><SIMPLE GROUP BY TERM> ::= <GROUP BY EXPRESSION> <LEFT PAREN> <GROUP BY EXPRESSION> <RIGHT PAREN> <LEFT PAREN> <RIGHT PAREN></pre>
Grammar rule 44	<pre><SIMPLE GROUP BY TERM LIST> ::= <SIMPLE GROUP BY TERM> [{ <COMMA> <SIMPLE GROUP BY TERM> } . . .]</pre>
Grammar rule 45	<pre><COMPOSITE GROUP BY TERM> ::= <LEFT PAREN> <SIMPLE GROUP BY TERM> [{ <COMMA> <SIMPLE GROUP BY TERM> } . . .] <RIGHT PAREN></pre>
Grammar rule 46	<pre><ROLLUP TERM> ::= ROLLUP <COMPOSITE GROUP BY TERM></pre>
Grammar rule 47	<pre><CUBE TERM> ::= CUBE <COMPOSITE GROUP BY TERM></pre>
Grammar rule 48	<pre><GROUP BY TERM> ::= <SIMPLE GROUP BY TERM> <COMPOSITE GROUP BY TERM> <ROLLUP TERM> <CUBE TERM></pre>
Grammar rule 49	<pre><GROUP BY TERM LIST> ::= <GROUP BY TERM> [{ <COMMA> <GROUP BY TERM> } ...]</pre>
Grammar rule 50	<pre><GROUP BY CLAUSE> ::= GROUP BY <GROUPING SPECIFICATION></pre>
Grammar rule 51	<pre><GROUPING SPECIFICATION> ::= <GROUP BY TERM LIST> <SIMPLE GROUP BY TERM LIST> WITH ROLLUP <SIMPLE GROUP BY TERM LIST> WITH CUBE <GROUPING SETS SPECIFICATION></pre>
Grammar rule 52	<pre><GROUPING SETS SPECIFICATION> ::= GROUPING SETS <LEFT PAREN> <GROUP BY TERM LIST> <RIGHT PAREN></pre>
Grammar rule 53	<pre><ORDER SPECIFICATION> ::= ORDER BY <SORT SPECIFICATION LIST> <SORT SPECIFICATION LIST> ::= <SORT SPECIFICATION> [{ <COMMA> <SORT SPECIFICATION> } . . .]</pre>


```
<SORT SPECIFICATION> ::= <SORT KEY>  
[ <ORDERING SPECIFICATION> ] [ <NULL ORDERING> ]  
<SORT KEY> ::= <VALUE EXPRESSION>  
<ORDERING SPECIFICATION> ::= ASC | DESC  
<NULL ORDERING> := NULLS FIRST | NULLS LAST
```


Sybase IQ as a Data Server

About this chapter

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 “Open Client architecture” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - Programming > SQL Anywhere Data Access APIs > Sybase Open Client API*.

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™. For information on remote data access and proxy databases, see Chapter 4, “Accessing Remote Data” and Chapter 5, “Server Classes for Remote Data Access.”

Contents

Topic	Page
Client/server interfaces to Sybase IQ	91
Setting up Sybase IQ as an Open Server	96
Characteristics of Open Client and jConnect connections	97

Client/server interfaces to Sybase IQ

To simplify, use a Sybase application or a third-party client application with Sybase IQ, you need not know the details of connectivity interfaces or network protocols. However, 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 “Open Clients, Open Servers, and TDS” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Replication > Using SQL Anywhere as an Open Server*.

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

Using the iqdsedit utility

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

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

Starting iqdsedit

On Windows, the iqdsedit executable is in the *SYBASE\IQ-15_1\bin32* or *SYBASE\IQ-15_1\bin64* directories, which is automatically added to your path during installation.

See “Starting DSEdit” in SQL Anywhere documentation in *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

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

See “Opening a directory services session” in SQL Anywhere documentation in *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

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 “Adding a server entry” in SQL Anywhere documentation in *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

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

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

Port number

The port number you enter must match the port specified on the Sybase IQ database server command line, as described in “Starting the database server as an Open Server” on page 96. 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
```

Verifying the server address

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

See “Verifying the server address” in SQL Anywhere documentation in *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

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

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

Deleting server entries

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

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

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 Appendix A, “Compatibility with Other Sybase Databases” in *Reference: Building Blocks, Tables, and Procedures*.

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.

Setting up 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 “System requirements” in SQL Anywhere documentation in *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*.

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

See “Starting the database server as an Open Server” in SQL Anywhere documentation in *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*.

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
```

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 Appendix A, “Compatibility with Other Sybase Databases” in *Reference: Building Blocks, Tables, and Procedures*.

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.

See “Characteristics of Open Client and jConnect connections” in SQL Anywhere documentation in *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*.

By default, the `sp_iq_process_login` system procedure performs Sybase IQ user administration, and then calls `sp_login_environment`, which in turn calls `sp_tsql_environment` for TDS 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 “`LOGIN_PROCEDURE` option” in *Reference: Statements and Options*.

Servers with multiple databases

Using Open Client Library, you can connect to a specific database on a server with 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*.

About this chapter

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

Contents

Topic	Page
Sybase IQ and remote data	102
Transaction management and remote data	112
Internal operations	113
Troubleshooting remote data access	115

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

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 “Remote table mappings” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data*.

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 “Server classes” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data*.

Note OMNI JDBC classes are not supported with IPv6.

Working with remote servers

Before you can map remote objects to a local proxy table, define the remote server where the remote object is located; this adds an entry to the ISYSSERVER system table for the remote server.

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

Loading remote data without native classes

You need to 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)

This section and the following one provide examples of loading and querying data by means of DirectConnect.

Non-Sybase remote data example

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.

❖ Loading MS SQL Server data into an IQ server on UNIX

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

Currently the best approach to accessing non-Sybase data on 64-bit systems is to do so indirectly, as follows:

- 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 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

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 for querying MS SQL Server**

- 1 Set up Adaptive Server and Component Integration Services (CIS) to MS SQL Server through DirectConnect. For example, assume that the server name is *jones_1207*.

- 2 Add an entry to the ASE interfaces file to connect to *mssql*:

```
mssql
master tcp ether hostname 12530
query tcp ether hostname 12530
```

- 3 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)
Configuration 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.

- 4 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)
```

- 5 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)
```

6 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)
```

7 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**

1 Add an entry to the 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 tttst
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 CLASEE '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)
```

Deleting remote servers

You can 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 “Delete remote servers” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Working with remote servers*.

Altering 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 “Alter remote servers” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Working with remote servers*.

Listing the remote tables on a server

When you are configuring Sybase IQ, you may find it helpful to have available a list of the remote tables available on a particular server.

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

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

Listing 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 “List remote server capabilities” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Working with remote servers*.

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

Working with 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 `INSERT` statement in *Reference: Statements and Options*.

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

Creating external logins

Only the login name and the DBA account can add or modify an external login.

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

For more information, see `CREATE EXTERNLOGIN` statement in *Reference: Statements and Options*.

Dropping external logins

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

For more information, see DROP EXTERNLOGIN statement in *Reference: Statements and Options*.

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

Working with proxy tables

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

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

Specifying proxy table locations

The AT keyword is used with both CREATE TABLE and CREATE EXISTING TABLE to define the location of an existing object. This 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 “Specify proxy table locations” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Working with proxy tables*.

Example

The following example illustrate the use of location strings:

- Sybase IQ:
`'testiq..DBA.employee'`

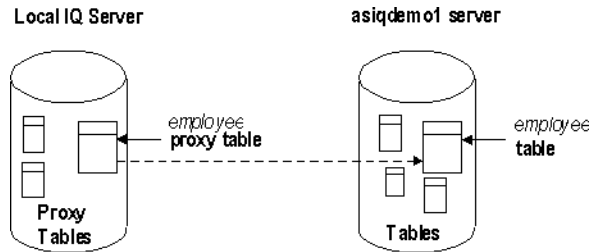
Creating proxy tables

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 CREATE EXISTING TABLE statement in *Reference: Statements and Options*.

Using the 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 employee 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 the INSERT statement in *Reference: Statements and Options*.

Listing the columns on a remote table

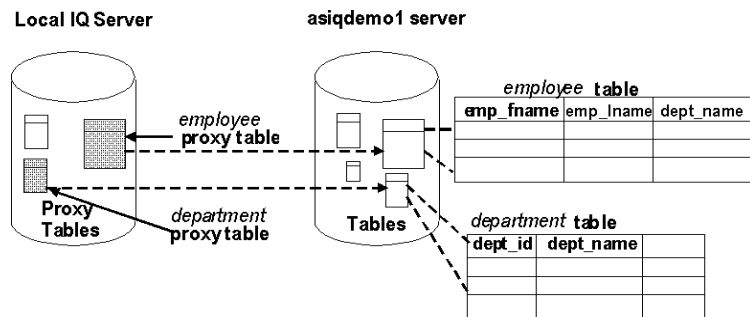
If you are entering a CREATE EXISTING TABLE statement and you are specifying a column list, it may be helpful to get a list of the columns that are available on a remote table. The `sp_remote_columns` system procedure produces a list of the columns on a remote table and a description of those data types.

See “List the columns on a remote table” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Working with proxy tables*.

For more information, see `sp_remote_columns` system procedure in the *Reference: Building Blocks, Tables, and Procedures*.

Example: A join between two remote tables

The following figure illustrates the remote Sybase IQ tables `employee` and `department` in the sample database, mapped to the local server named `testiq`.



See “Join remote tables” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data*.

Accessing 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 “Join tables from multiple local databases” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data*.

Sending 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 “Send native statements to remote servers” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data*.

Using 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.

Creating remote procedures

Use one of the following procedures to issue a remote procedure call.

See “Create remote procedures” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Using remote procedure calls (RPCs)*.

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 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 “Using transactions and isolation levels” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Creating Databases*.

For a general discussion of transactions in Sybase IQ, see Chapter 10, “Transactions and Versioning,” in *System Administration Guide: Volume 1*.

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 “Remote transaction management overview” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Accessing remote data > Transaction management and remote data*.

Restrictions on transaction management

Restrictions on transaction management are as follows:

- 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

The next step is called query normalization. During this step, referenced objects are verified and some data type compatibility is checked.

For example, consider the following 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 “Query preprocessing” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Internal operations*.

Server capabilities

The previous steps are performed on all queries, both local and remote.

The following steps depend on the type of SQL statement and the capabilities of the remote servers involved.

See “Server capabilities” in SQL Anywhere documentation in *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

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 “Complete passthrough of the statement” in SQL Anywhere documentation in *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

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 “Partial passthrough of the statement” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Internal operations*.

Troubleshooting remote data access

This section provides some hints for troubleshooting access to remote servers.

Features not supported for remote data

The following features are not supported on remote data. Some 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 “Features not supported for remote data” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Troubleshooting remote data access*.

Case-sensitivity

The case-sensitivity setting of your IQ database should match the settings used by any remote servers accessed.

See “Case sensitivity” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Troubleshooting remote data access*.

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 “Connectivity tests” in SQL Anywhere documentation in *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

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 “General problems with queries” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Troubleshooting remote data access*.

Queries blocked on themselves

If you access multiple databases on a single Sybase IQ or SQL Anywhere server, you may need to increase the number of threads used by the database server using the -gx command-line switch. By default, this switch is set to one more than the number of CPUs on the machine.

You must have enough threads available to support the individual tasks that are being run by a query. Failure to provide the number of required tasks can lead to a query becoming blocked on itself.

Note The `-gx` switch is not documented in the *Utility Guide*, as you do not normally need to set it for an IQ database. For any purpose other than the one described here, to increase the number of threads, set the `-iqmt` switch, which controls the number of threads for IQ store operations.

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 “Managing remote data access connections via ODBC” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Accessing remote data > Troubleshooting remote data access*.

Server Classes for Remote Data Access

About this chapter

This chapter describes how Sybase IQ interfaces with various server classes.

Contents

Topic	Page
Server classes overview	119
JDBC-based server classes	119
ODBC-based server classes	121

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 “Server classes for remote data access” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations*.

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:

- **sajdbc** Sybase IQ, and SQL Anywhere
- **asejdbc** 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 information in this topic:

See “Configuration notes for JDBC classes” in SQL Anywhere documentation in *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

No special requirements exist for the configuration of Sybase IQ or SQL Anywhere data source.

USING 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 “USING parameter in the CREATE SERVER statement” in SQL Anywhere documentation in *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*.

Server class asejdbc

A server with server class *asejdbc* can be:

- Adaptive Server Enterprise
- SQL Anywhere Version 10 and later

No special requirements exist for the configuration of an Adaptive Server Enterprise data source.

See “Server class asejdbc” in SQL Anywhere documentation in *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*.

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 “Server class asejdbc” in SQL Anywhere documentation in *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*.

ODBC-based server classes

Sybase IQ supports a variety of ODBC-based server classes.

See “ODBC-based server classes” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Remote Data and Bulk Operations > Server classes for remote data access*.

Defining ODBC external servers

The most common way 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 “Defining ODBC external servers” in SQL Anywhere documentation in *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*.

Sybase IQ example

A connection to an Sybase IQ may be as follows:

```
CREATE SERVER testiq  
CLASS 'asaodbc'  
USING 'driver=adaptive server IQ 12.0;  
eng=testasaiq;dbn=iqdemo;links=tcipip{ }'
```

For more information on creating ODBC data sources for Sybase IQ, see “Creating and editing ODBC data sources” in Chapter 3, “Sybase IQ Connections,” in the *System Administration Guide: Volume 1*.

Server class saodbc

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.

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.

Server class aseodbc

A server with server class aseodbc is:

- Adaptive Server Enterprise
- SQL Anywhere (version 10 and later)

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.

See “Server class aseodbc” in SQL Anywhere documentation in *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

A server with server class db2odbc is IBM DB2.

See “Server class db2odbc” in SQL Anywhere documentation in *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

A server with server class oraodbc is Oracle version 8.0 or higher.

See “Server class oraodbc” in SQL Anywhere documentation in *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*.

Use only Sybase IQ servers running on Windows to access Oracle data. There is no 64-bit driver manager for Sybase IQ servers running on 64-bit UNIX systems.

❖ Querying Oracle data using Sybase IQ servers on 64-bit UNIX

This process tells how to create the two sets of proxy tables needed to query Oracle data in this situation.

- 1 Configure DirectConnect for Oracle to connect to Oracle.
- 2 Configure proxy tables in DirectConnect for Oracle.
- 3 Create a remote server in Sybase IQ using the ASEJDBC class to the server and the port number for DirectConnect for Oracle.
- 4 Use a CREATE EXISTING TABLE statement to create a proxy table pointing to the ASE proxy tables in DirectConnect for Oracle.

❖ Loading Oracle data using Sybase IQ servers on 64-bit UNIX

For best performance when loading large data, access the remote database with a different method from that used for queries, as follows:

- 1 Create proxy tables in DirectConnect for Oracle.
- 2 Use the INSERT .. LOCATION statement to the proxy tables.

For details about INSERT .. LOCATION, see “Inserting from a different database” in Chapter 7, “Moving Data In and Out of Databases,” in *System Administration Guide: Volume 1*.

Server class mssodbc

A server with server class mssodbc is Microsoft SQL Anywhere version 6.5, Service Pack 4.

See “Server class mssodbc” in SQL Anywhere documentation in *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

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 logically 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 “Microsoft Excel (Microsoft 3.51.171300)” in SQL Anywhere documentation in *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)

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 “Microsoft FoxPro (Microsoft 3.51.171300)” in SQL Anywhere documentation in *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 (2.04.0203)

You can obtain this 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 “Lotus Notes SQL 2.0” in SQL Anywhere documentation in *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*.

To set up Sybase IQ to access the Address sample file, follow this procedure.

❖ Setting 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

About this chapter

This chapter describes how to use scheduling and event handling features of Sybase IQ to automate database administration and other tasks.

Contents

Topic	Page
Introduction to scheduling and event handling	128
Understanding schedules	128
Understanding events	129
Understanding event handlers	131
Schedule and event internals	132
Scheduling and event handling tasks	133

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 “Introduction to using schedules and events” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Maintaining Your Database > Automating tasks using schedules and events*.

Understanding 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 “Understanding schedules” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Maintaining Your Database > Automating tasks using schedules and events*.

Sybase IQ example

Note For examples, use the Sybase IQ demo database iqdemo.db. For installation information, see “The demo database” on page xv.

```
Create table OrderSummary(c1 date, c2 int);  
  
create event Summarize  
  
schedule  
  
start time '6:00 pm'  
  
on ('Mon', 'Tue', 'Wed', 'Thu', 'Fri')  
  
handler  
  
begin  
  
    insert into DBA.OrderSummary  
  
        select max(OrderDate), count(*)  
  
        from GROUPO.SalesOrders where OrderDate = current  
date  
  
end
```


Defining schedules

For flexibility, schedule definitions are made up of several components.

See “Defining schedules” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Maintaining Your Database > Automating tasks using schedules and events > Understanding schedules*.

Understanding 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 “Understanding system events” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Maintaining Your Database > Automating tasks using schedules and events*.

See also “Choosing a system event” on page 129, and “Defining trigger conditions for events” on page 130.

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 “Understanding system events” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Maintaining Your Database > Automating tasks using schedules and 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 “Defining trigger conditions for events” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Maintaining Your Database > Automating tasks using schedules and events > Understanding system events*.

Sybase IQ example

Note For examples, use the Sybase IQ demo database iqdemo.db. For installation information, see “The demo database” on page xv.

```
create event SecurityCheck
type ConnectFailed
handler
begin

declare num_failures int;
declare mins int;

insert into FailedConnections( log_time )
values ( current timestamp );

select count( * ) into num_failures
from FailedConnections
where 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 if
end if
end
```

Understanding 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 “Developing event handlers” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Maintaining Your Database > Automating tasks using schedules and events > Understanding system events*.

For more information, see “EVENT_PARAMETER function [System]” in Chapter 4, “SQL Functions,” in *Reference: Building Blocks, Tables, and Procedures*.

For an example on using event handling, see “Managing IQ user accounts and connections” in “Automating Tasks Using Schedules and Events,” in the *System Administration Guide: Volume 1*.

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 “How the database server checks for system events” in SQL Anywhere documentation in *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 times

The calculation of scheduled event times is done when the database server starts, and each time a scheduled event handler completes.

See “How the database server checks for scheduled events” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Maintaining Your Database > Automating tasks using schedules and events > Schedule and event intervals*.

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 “How event handlers are executed” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Maintaining Your Database > Automating tasks using schedules and events > Schedule and event intervals*.

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 “Adding an event to a database” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Maintaining Your Database > Automating tasks using schedules and events > Event handling tasks*.

For more information, see ALTER EVENT statement in Chapter 1, “SQL Statements,” in the *Reference: Statements and Options*.

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 “Adding a manually-triggered event to a database” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Maintaining Your Database > Automating tasks using schedules and events > Event handling tasks*.

For information on altering events, see ALTER EVENT statement in Chapter 1, “SQL Statements,” in *Reference: Statements and Options*.

Triggering an event handler

Any event handler can be manually triggered, as well as executed because 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 “Triggering an event handler” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Maintaining Your Database > Automating tasks using schedules and events > Event handling tasks*.

For more information, see TRIGGER EVENT statement in Chapter 1, “SQL Statements,” 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 “Debugging an event handler” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - Database Administration > Maintaining Your Database > Automating tasks using schedules and events > Event handling tasks*.

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.

Debugging Logic in the Database

About this appendix

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.

Contents

Topic	Page
Introduction to debugging in the database	135
Tutorial 1: Getting started with the debugger	136
Tutorial 2: Debugging a stored procedure	137
Tutorial 3: Debugging a Java class	137
Writing debugger scripts	142

Introduction to debugging in the database

You can use the debugger during the development of 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 “Introduction to the SQL Anywhere debugger” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Debugging procedures, functions, triggers, and events*.

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 *iqdemo.db*. For installation information, see “The demo database” on page xv.

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 sample database.

Start the debugger

See “Lesson 1: Connect to a database and start the debugger” in SQL Anywhere documentation in *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*.

Tutorial 2: Debugging a stored procedure

This tutorial describes a sample session for debugging a stored procedure. It is a continuation of “Tutorial 1: Getting started with the debugger” on page 136.

See “Lesson 2: Debug a stored procedure” in SQL Anywhere documentation in *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*.

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 sample 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 sample database. See “Preparing the database” on page 137.

Preparing the database

If you intend to run Java examples, install the Java example classes into the sample database.

See “Preparing for the examples” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - Programming > SQL Anywhere Data Access APIs > SQL Anywhere JDBC driver > Using JDBC to access data*.

Displaying Java source code into the debugger

❖ Displaying Java source code in the debugger

The debugger looks in a set of locations for source code files with *.java* extension.

- 1 In Sybase Central, select Mode > Debug.
- 2 When prompted to select the user to debug, specify * for all users and click OK.
- 3 From the debugger interface, select Debug > Set Java Source Code Path.
- 4 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

- 5 Click OK, and close the window.

Locating Java source code

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.

For example, on Windows, if you add the paths *%IQDIR15%\java* and *c:\Java\src* to the source path, and the debugger is trying to find a class called *iqdemo.Product*, it looks for the source code in *%IQDIR15%\asa\java\asademo\Product.Java* and *c:\Java\src\my\iqdemo\Product.Java*

Set a breakpoint

❖ Setting a breakpoint in a Java class

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.

Run the method

You can invoke the `Query()` method from Interactive SQL (DBISQL), and see its execution interrupted at the breakpoint.

❖ Invoking the method from Interactive SQL

- 1 Start Interactive SQL. Connect to the sample 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

Following the previous section, the debugger should have stopped executing `JDBCExamples.Query()` at the first statement in the method:

Examples

Here are some steps you can try:

- 1 **Step to the next line** Choose Run > Step Over, or press F7 to step to the next line in the current method. Try this two or three times.
- 2 **Run to a selected line** Select the following line using the mouse, and choose Run > Run To Selected, or press F6 to run to that line and break:

```
max_price = price;
```

The red arrow moves to the line.

- 3 **Set a breakpoint and execute to it** 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 **Experiment** 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 **Proceed to the next breakpoint** 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 sample database. See “Preparing the database” on page 137.

❖ Inspecting and modifying the value of a local variable

- 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.

Working with breakpoints

Breakpoints control when the debugger interrupts execution of your source code.

See “Working with breakpoints” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Debugging procedures, functions, triggers, and events*.

Working with variables

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 “Working with variables” in SQL Anywhere documentation in *SQL Anywhere 11.0.1 > SQL Anywhere Server - SQL Usage > Stored Procedures and Triggers > Debugging procedures, functions, triggers, and events.*

Writing 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.

A debugger window is represented by the “`sybase.asa.procdebug.IDebugWindow` interface” on page 147.

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

A

- accessibility
 - documentation xiv
- aggregate functions 44
 - statistical 64
 - STDDEV_POP 65
 - STDDEV_SAMP 65
 - VAR_POP 65
 - VAR_SAMP 65
- ALLOW_NULLS_BY_DEFAULT option
 - Open Client 97
- analytical functions 26
- asajdbc server class 120
- asaodbc server class 122
- ascending order 54
- asejdbc server class 120
- aseodbc server class 122
- AT clause
 - CREATE EXISTING TABLE statement 109
- atomic compound statements 10
- Automating 127

B

- batches
 - about 1, 9
 - SQL statements allowed 20
- BEGIN TRANSACTION statement
 - remote data access 113
- breakpoints
 - setting in a Java class 138

C

- CALL statement
 - about 2
 - examples 4

- parameters 12
 - syntax 9
- CASE statement
 - syntax 9
- case-sensitivity
 - remote access 116
- certifications
 - documentation
 - updated xii
- CHAINED option
 - Open Client 97
- CIS (Component Integration Services) 91
- Client-Library
 - about 91
- CLOSE statement
 - procedures 14
- command delimiter
 - setting 18
- COMMIT statement
 - compound statements 10
 - procedures 18
 - remote data access 113
- compliance
 - section 508 xiv
- components
 - certifications xii
- compound statements
 - atomic 10
 - declarations 10
 - using 10
- computing deltas between adjacent rows 55
- connections
 - debugging 136
 - remote 113
- CONTINUE_AFTER_RAISERROR option
 - Open Client 97
- control statements
 - list 9
- conventions
 - documentation xiii, xiv

- syntax xiii
- typographic xiv
- CREATE EXISTING TABLE statement
 - using 109
- CREATE PROCEDURE statement
 - examples 3
 - parameters 11
- CREATE TABLE statement
 - proxy tables 110
- CUBE operation 28, 29, 39
 - example 42
 - NULL 31
 - SELECT statement 39
- CURRENT ROW 48, 49
- current row 50
- cursors
 - and LOOP statement 15
 - in procedures 15
 - on SELECT statements 15
 - procedures 14

D

- data sources
 - external servers 121
- database options
 - Open Client 97
- databases
 - demo xv
 - multiple 111
 - multiple on server 98
 - proxy 91
- dates
 - procedures 18
- DB-Library
 - about 91
- debugger
 - about 135
 - connecting 136
 - features 135
 - getting started 136
 - requirements 136
 - starting 136
 - tutorial 136
- debugging

- breakpoints 138
- connection 136
- event handlers 134
- features 135
- introduction 135
- Java 137
- permissions 136
- requirements 136
- stored procedures 137
- DebugScript class 142
- DECLARE statement
 - compound statements 10
 - procedures 14
- deltas between adjacent rows, computing 55
- demo database xv
- descending order 54
- distribution functions 26, 46, 67
- documentation
 - accessibility features xiv
 - certifications xii
 - conventions xiii, xiv
 - on CD xi
 - online xi
 - SQL Anywhere x
 - Sybase IQ ix
- driver
 - missing 115
- DSEdit
 - entries 93
 - starting 93
 - using 92

E

- EBFs xiii
- encryption
 - hiding objects 19
- error handling
 - ON EXCEPTION RESUME 16
- errors
 - procedures 15
- event handlers 131
 - debugging 134
 - triggering 133
- events 127–134

- retrieving a schedule name 134
- retrieving an event name 134
- system 129
- trigger condition 129
- examples
 - OLAP 75
- exception handlers
 - procedures 17
- EXECUTE IMMEDIATE statement
 - procedures 18
- extensions to GROUP BY clause 26, 28
- external logins
 - about 108
 - creating 108
 - dropping 109

F

- Federal Rehabilitation Act
 - section 508 xiv
- FETCH statement
 - procedures 14
- FLOAT_AS_DOUBLE option
 - Open Client 97
- FOR statement
 - syntax 9
- FORWARD TO statement 112
- functions
 - aggregate 44
 - analytical 26, 43
 - correlation 65
 - covariance 65, 66
 - distribution 26, 67
 - inverse distribution 67
 - numeric 26
 - numerical 70
 - ordered sets 67
 - PERCENTILE_CONT function 67
 - PERCENTILE_DISC function 67
 - ranking 26, 58
 - reporting 62
 - simple aggregate 44
 - standard deviation 64
 - statistical 26
 - statistical aggregate 64

- STDDEV_POP function 65
- STDDEV_SAMP function 65
- user-defined 7
- VAR_POP function 65
- VAR_SAMP function 65
- variance 64
- window 27, 62
- windowing 44
- windowing aggregate 26, 62

G

- Getting Started CD xi
- GROUP BY
 - clause extensions 28
 - CUBE 29
 - ROLLUP 29
- GROUP BY clause extensions 28
- GROUPING function
 - NULL 31
 - ROLLUP operation 31
- gx option
 - threads 116

I

- IDebugAPI interface 143
- IDebugWindow 147
- IF statement
 - syntax 9
- Interactive SQL
 - command delimiter 18
- interfaces
 - IDebugAPI 143
 - IDebugWindow 147
- interfaces file
 - configuring 92
- inverse distribution functions 67
- IP address
 - about 94
- ISOLATION_LEVEL option
 - Open Client 97

J

- Java
 - about 135
 - about debugging 135
 - debugging 135, 137
- Java debugger
 - requirements 136
 - starting 136
 - tutorial 136

K

- keywords
 - remote servers 115

L

- LEAVE statement
 - syntax 9
- libctl.cfg file
 - DSEdit 93
- localhost
 - machine name 94
- logical offset of a window frame 53
- LOOP statement
 - in procedures 15
 - syntax 9

M

- maintenance
 - software xiii
- maintenance, product xiii
- managing
 - transactions 113
- multiple databases
 - DSEdit entries 93
 - joins 111
- MySybase
 - creating personalized view xii
 - EBFs xiii

N

- NULL
 - CUBE operation 31
 - ROLLUP operation 31
- NULL values
 - example 32
- NULL values and subtotal rows 31
- numeric functions 26

O

- objects
 - hiding 19
- ODBC
 - external servers 121
 - server classes 121
- OLAP 46
 - about 26
 - aggregate functions 44
 - analytical functions 26, 43
 - benefits 27
 - CUBE operation 39
 - current row 50
 - distribution functions 26, 46
 - extensions to GROUP BY clause 26
 - functionality 26
 - Grouping() 28
 - NULL values 31
 - numeric functions 26
 - ORDER BY clause 47
 - PARTITION BY clause 46
 - RANGE 46
 - range 53
 - ranking functions 26, 46
 - ROLLUP operator 30
 - ROWS 46
 - rows 50
 - semantic phases of execution 27
 - statistical aggregate functions 26
 - statistical functions 46
 - subtotal rows 30
 - using 27
 - window concept 45
 - window framing 45
 - window functions 27

- window ordering 45
 - window partitioning 45, 46
 - window sizes 46
 - windowing extensions 44
 - windows aggregate functions 26
- OLAP examples 75
 - ascending and descending order for value-based frames 54
 - calculate cumulative sum 77
 - calculate moving average 78
 - computing deltas between adjacent rows 55
 - default window frame for RANGE 84
 - default window frame for ROW 82
 - multiple aggregate functions in a query 80
 - ORDER BY results 79
 - range-based window frames 53
 - row-based window frames 51
 - unbounded preceding and unbounded following 83
 - unbounded window 55
 - using a window with multiple functions 77
 - window frame excludes current row 81
 - window frame with ROWS vs. RANGE 80
 - window functions 57
 - window functions in queries 75
- OLAP functions
 - distribution 67
 - numerical functions 70
 - ordered sets 67
 - ranking functions 58
 - statistical aggregate 64
 - windowing 44
 - aggregate functions 62
- OmniConnect 91
 - support 95
- ON EXCEPTION RESUME clause
 - about 16
- online analytical processing
 - CUBE operator 39
 - functionality 26
 - NULL values 31
 - ROLLUP operator 30
 - subtotal rows 30
- Open Client
 - configuring 92
 - interface 91

- Open Server
 - adding 92
 - addresses 94
 - architecture 91
 - starting 96
 - system requirements 96
- OPEN statement
 - procedures 14
- options
 - Open Client 97
- ORDER BY clause 47, 48
 - sort order 55
- ordered set functions 67
 - PERCENTILE_CONT 67
 - PERCENTILE_DISC 67
- OVER clause 45

P

- PARTITION BY clause 46
- PERCENTILE_CONT function 67
- PERCENTILE_DISC function 67
- permissions
 - debugging 136
 - procedures 5
 - user-defined functions 8
- phases of execution 27
- physical offset of a window frame 50
- ping
 - testing Open Client 94
- population variance function 65
- prefixes 29
 - ROLLUP operation 30
 - subtotal rows 30
- PREPARE statement
 - remote data access 113
- procedures
 - about 1
 - benefits of 2
 - calling 4
 - command delimiter 18
 - creating 3
 - cursors 14
 - cursors in 15
 - dates and times 18

- debugging 137
- default error handling 16
- displaying information about 3
- dropping 5
- error handling 15
- exception handlers 17
- EXECUTE IMMEDIATE statement 18
- execution permissions 5
- multiple result sets from 13
- owner 3
- parameters 3, 11, 12
- result sets 6, 13
- returning results 12, 13
- returning results from 6
- savepoints in 18
- SQL statements allowed in 11
- structure 10
- table names 18
- using 2
- variable result sets from 14
- warnings 16
- writing 18
- product manuals xi
- proxy databases 91
- proxy tables
 - about 102, 109
 - creating 102, 109, 110
 - location 109
 - properties 109

Q

- queries
 - prefixes 29
 - subtotal rows 30
- QUOTED_IDENTIFIER option
- Open Client 97

R

- RANGE 46
- range 53
 - logical offset of a window frame 53
 - window frame unit 47

- window order clause 47
- range specification 50, 53
- range-based window frames 53, 54
- rank functions
 - example 60, 61, 62
- ranking functions 26, 46
 - requirements with OLAP 47
 - window order clause 47
- remote data
 - location 109
- remote data access 91
 - case-sensitivity 116
 - internal operations 113
 - passthrough mode 112
 - remote servers 103
 - SQL Remote unsupported 115
 - troubleshooting 115
 - unsupported features 115
- remote procedure calls
 - about 112
- remote servers
 - about 103
 - altering 107
 - classes 119
 - creating 103
 - deleting 107
 - external logins 108
 - listing properties 108
 - transaction management 112
- remote tables
 - about 102
 - listing 107
 - listing columns 111
- Replication Server
 - support 95
- reporting functions 62
 - example 62, 63
- requirements for using the debugger 136
- reserved words
 - remote servers 115
- restrictions
 - remote data access 115
- result sets
 - multiple 13
 - procedures 6, 13
 - variable 14

- RETURN statement
 - about 12
 - ROLLBACK statement
 - compound statements 10
 - procedures 18
 - ROLLUP operation 28, 29
 - example 36
 - NULL 31
 - SELECT statement 30
 - subtotal rows 30
 - ROLLUP operator 30
 - row specification 50
 - row-based window frames 51
 - ROWS 46
 - rows 50
 - physical offset of a window frame 50
 - rows between 1 preceding and 1 following 51
 - rows between 1 preceding and 1 preceding 51
 - rows between current row and current row 51
 - rows between unbounded preceding and current row 50
 - rows between unbounded preceding and unbounded following 50
 - specification 53
 - subtotal rows 30
- S**
- SA_DEBUG group
 - debugger 136
 - sample variance function 65
 - savepoints
 - procedures 18
 - schedules 127–134
 - definition components 129
 - scripts
 - IDebugAPI interface 143
 - IDebugWindow interface 147
 - writing debugger 142
 - section 508
 - compliance xiv
 - security
 - hiding objects 19
 - semantic phases of execution 27
 - semicolon
 - command delimiter 18
 - server address
 - DSEEDIT 94
 - server classes
 - about 102
 - asajdbc 120
 - asaodbc 122
 - asejdbc 120
 - aseodbc 122
 - defining 102
 - ODBC 121
 - servers
 - multiple databases on 98
 - simple aggregate functions 44
 - sort order of ORDER BY in range-based frames 55
 - sp_iqprocedure
 - information about procedures 3
 - sp_iqprocparm
 - procedure parameters 3
 - SQL Remote
 - remote data access 115
 - sql.ini file
 - configuring 92
 - SQLCODE variable
 - introduction 15
 - SQLSTATE variable
 - introduction 15
 - standard deviation
 - functions 64
 - population function 65
 - sample function 65
 - standards
 - section 508 compliance xiv
 - statistical aggregate functions 64
 - statistical functions 46
 - aggregate 26
 - STDDEV_POP function 65
 - STDDEV_SAMP function 65
 - stored procedures
 - debugging 137
 - displaying information about 3
 - subtotal rows 30
 - construction 30
 - definition 30, 39
 - NULL values 31
 - ROLLUP operation 30

Index

- subtransactions
 - procedures 18
- summary information
 - CUBE operator 39
- summary rows
 - ROLLUP operation 30
- SYBASE environment variable
 - DSEdit 93
- SyBooks CD xi
- syntax
 - documentation conventions xiii
- syssservers system table
 - remote servers 103
- system events
 - trigger conditions 130

T

- table names
 - local 109
 - procedures 18
- tables
 - defining proxy 109, 110
 - listing remote 107
 - proxy 109
 - remote access 102
- Tabular Data Stream (TDS)
 - about 91
- TCP/IP
 - addresses 94
 - Open Server 96
- TDS. *See* Tabular Data Stream (TDS)
- times
 - procedures 18
- transaction management 112
- transactions
 - managing 113
 - procedures 18
 - remote data access 113
- trigger conditions
 - for system events 130
- triggering event handlers 133
- troubleshooting
 - remote data access 115
 - server address 94

- TSQL_HEX_CONSTANT option
 - Open Client 97
- TSQL_VARIABLES option
 - Open Client 97
- typographic
 - conventions xiv
- typography
 - documentation xiii

U

- UNBOUNDED FOLLOWING 48, 49
- UNBOUNDED PRECEDING 48
- UNBOUNDED PREDEDING 49
- unbounded window, using 55
- user-defined functions
 - calling 7
 - creating 7
 - dropping 8
 - execution permissions 8
 - parameters 12
 - using 7
- using unbounded windows 55

V

- value-based window frames 53
 - ascending and descending order 54
 - ORDER BY clause 54
- VAR_POP function 65
- VAR_SAMP function 65
- variance functions 64
- views
 - MySybase, creating personalized xii

W

- warnings
 - procedures 16
- WHILE statement
 - syntax 9
- window
 - frame clause 48

- operator 44
- order clause 47, 48
- ordering 45, 47
- window frame unit 47, 50, 53
 - range 53
 - rows 50
- window frames 45, 48
 - range based 53, 54
 - row based 51
- window functions
 - aggregate 26, 46
 - distribution 46
 - framing 48
 - ordering 47
 - OVER clause 45
 - partitioning 46
 - ranking 46
 - statistical 46
 - window function type 44
 - window name or specification 44
 - window partition 44
- window partitioning 45, 46
 - clause 46
 - GROUP BY operator 46
- window sizes
 - RANGE 46
 - ROWS 46
- windowing
 - aggregate functions 46, 62
 - extensions 44
 - functions 46
 - partitions 44

