SYBASE®

User-Defined Functions Guide

Sybase IQ 15.2

DOCUMENT ID: DC01034-01-1520-01
LAST REVISED: April 2010
Copyright © 2010 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

Audience 1
Related Documents 3
Understanding user-defined functions 5
Sybase IQ overview 5
User-Defined Functions Compliance with Sybase IQ databases 7
Practices to avoid 8
Types of user-defined functions 8
Naming conventions for user-defined functions 9
Design basics of user-defined functions 10
Creating and executing user-defined functions 11
Creating a user-defined function 11
Creating a user-defined function using SQL Anywhere dialects 12
Declaring a scalar user-defined function in Sybase Central 12
Declaring a user-defined aggregate function in Sybase Central 13
User-defined function restrictions 14
Calling user-defined functions 14
Setting the dynamic library interface 15
Dropping user-defined functions 15
Granting and revoking permissions 15
Maintenance of user-defined functions 16
Compiling and linking source code to build dynamically linkable libraries 16
AIX switches 17
HP-UX switches 18
Linux switches 18
Solaris switches 19
Windows switches 20

Using Microsoft Visual Studio debugger for user-defined functions 21
SQL data types 21
Testing user-defined functions 25
Enabling and disabling user-defined functions 25
Initially executing a user-defined function 26
Managing external libraries 26
Controlling error checking and call tracing 27
Enabling full tracing in a debug environment 27
Viewing Sybase IQ log files 28
Scalar User-Defined Functions 29
Declaring a Scalar UDF 29
UDF example: my_plus declaration 30
UDF example: my_plus_counter declaration 32
Defining a scalar UDF 33
Scalar UDF descriptor structure 33
Scalar UDF context structure 34
UDF example: my_plus definition 36
UDF example: my_plus _counter definition 37
User-defined aggregate functions 41
Declaring a UDAF 41
UDAF example: my_sum declaration 44
UDAF example: my_bit_xor declaration 45
UDAF example: my_bit_or declaration 45
UDAF example: my_interpolate declaration 45
Defining an aggregate UDF 47
Aggregate UDF descriptor structure 49
Calculation context 53
UDAF context structure 54
UDAF example: my_sum definition 57
UDAF example: my_bit_xor definition 61
UDAF example: my_bit_or definition 65
UDAF example: my_interpolate definition 67
Context storage of aggregate user-defined functions 74

UDF callback functions and calling patterns 75
UDF and UDAF callback functions 75
Scalar UDF calling pattern 76
Aggregate UDF calling patterns 76
Simple aggregate ungrouped 77
Simple aggregate grouped 77
OLAP-style aggregate calling pattern with unbounded window 78
OLAP-style unoptimized cumulative window aggregate 79
OLAP-style optimized cumulative window aggregate 79
OLAP-style unoptimized moving window aggregate 80
OLAP-style optimized moving window aggregate 81
OLAP-style unoptimized moving window following aggregate 82
OLAP-style optimized moving window following aggregate 83
OLAP-style unoptimized moving window without current 83
OLAP-style optimized moving window without current 85
External function prototypes 85
Index 89

Audience

This book, *User-Defined Functions Guide* is intended for the users who wants to extend the functionality of Sybase® IQ. Use this book for building and incorporating very complicated logic into the SQL queries or statements and for concepts about procedures for programming scalar and aggregate user-defined functions with Sybase IQ.

Related Documents

Additional information is available in Sybase IQ and SQL Anywhere documents.

Related Sybase IQ Documents

The Sybase IQ documentation set includes:

- *Release Bulletin* for your platform – contains last-minute information that was too late to be included in the books.
  A more recent version of the release bulletin may be available. To check for critical product or document information that was added after the release of the product CD, use the Sybase Product Manuals Web site.
- *Installation and Configuration Guide* for your platform – describes installation, upgrading, and some configuration procedures for Sybase IQ.
- *New Features Summary Sybase IQ* – summarizes new features and behavior changes for the current version.
- *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 User's 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 available only to RAP – The Trading Edition™ Enterprise users.
- *Introduction to Sybase IQ* – includes exercises for those unfamiliar with Sybase IQ or with the Sybase Central™ database management tool.
- *Performance and Tuning Guide* – describes query optimization, design, and tuning issues for very large databases.
- *Quick Start* – discusses 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* – 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:
  - *System Administration Guide: Volume 1* – describes start-up, connections, database creation, population and indexing, versioning, collations, system backup and recovery, troubleshooting, and database repair.
  - *System Administration Guide: Volume 2* – describes how to write and run procedures and batches, program with OLAP, access remote data, and set up IQ as an Open Server.

This book also discusses scheduling and event handling, XML programming, and debugging.

- *Time Series Guide* describes SQL functions used for time series forecasting and analysis. You need RAP – The Trading Edition™ Enterprise to use this product option.
- *Unstructured Data Analytics in Sybase IQ* explains how to store and retrieve unstructured data within the Sybase IQ data repository. You need a separate license to install this product option.
- *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, which manages 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 Documents

Note: Because Sybase IQ shares many components with SQL Anywhere®, a component of SQL Anywhere Studio®, Sybase IQ supports many of the same features as SQL Anywhere. The IQ documentation set refers you to SQL Anywhere Studio 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 Studio 11.0 collection at *Product Manuals* and in *DocCommentXchange*.

Understanding user-defined functions

Sybase IQ overview

Learn how user-defined functions are used within Sybase IQ.

Sybase IQ allows user defined functions (UDFs), which execute within the database container. The UDF execution feature is available as an optional component for use within Sybase IQ or within the RAPStore component of RAP - The Trading Edition™ R3.

The use of these external C/C++ UDFs interfaces requires the IQ_UDF license.

These external C/C++ UDFs differ from the Interactive SQL UDFs available in earlier versions of Sybase IQ. Interactive SQL UDFs are unchanged and do not require a special license.

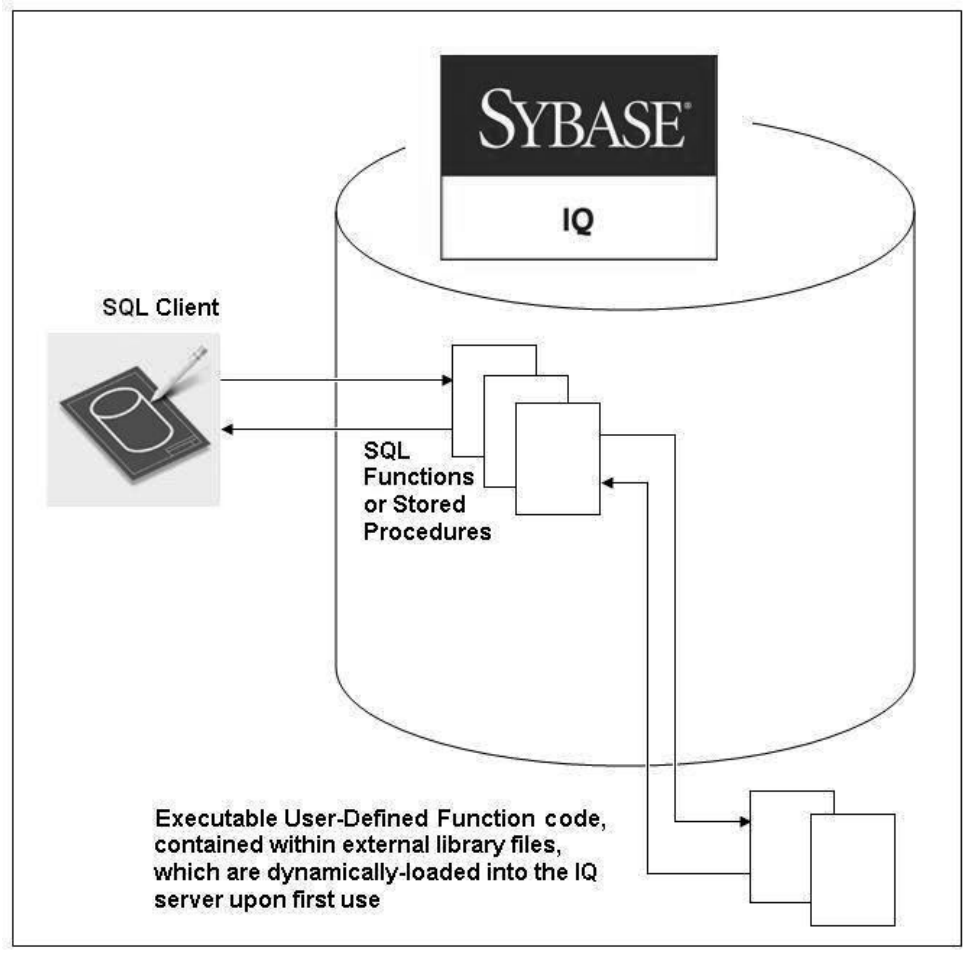
UDFs that execute within Sybase IQ take advantage of the extreme performance of the IQ server, while also providing users the flexibility of analyzing their data with the flexibility of a programmatic solution. User-Defined Functions consist of two components:

- UDF declaration, and
- UDF executable code

A UDF is declared in the SQL environment through a SQL function or stored procedure which describes the parameters and provides a reference to the external library.

The actual executable portion of the UDF is contained within an external (shared object or dynamic load) library file, which is automatically loaded by the IQ server upon the first invocation of a UDF Declaration function or stored procedure associated with that library. Once loaded, the library remains resident in the IQ server for rapid access through subsequent invocations of SQL functions or stored procedures that reference the library.

The Sybase IQ user-defined function architecture is represented in the diagram below.



Sybase IQ supports high-performance in-process external C/C++ user-defined functions. This style of UDF supports functions written in C or C++ code that adhere to the interfaces described in this guide.

The C/C++ source code for the UDFs is compiled into one or more external libraries that are subsequently loaded into the IQ server's process space when needed. The UDF calling mechanism is defined to the Sybase IQ server through a SQL function. When the SQL function is invoked from a SQL query, the IQ server loads the corresponding library if it has not already been loaded.

For simplicity of managing the UDF installation, Sybase recommends that UDF developers package many UDF functions within a single library.

To facilitate the construction of UDFs, Sybase IQ includes a C-based API. The API comprises a set of predefined entry points for the UDFs, a well-defined context data structure, and a series

of SQL callback functions that provide a communication mechanism from the UDF back to the Sybase IQ server. The Sybase IQ UDF API allows software vendors and expert end-users to develop, package, and sell their own UDFs.

User-Defined Functions Compliance with Sybase IQ databases

Developing user-defined functions to work with Sybase IQ databases.

Seamless Execution

UDFs must run seamlessly within the database container. Although Sybase IQ is a complex product consisting of many files, the main user interaction is through an IQ server process (iqsrv15), using industry-standard Structured Query Language (SQL). Execution of UDFs should be accomplished entirely through SQL commands; the user not need understand the underlying implementation method to use the UDFs.

UDFs run under the cover of Sybase IQ, so do not write console messages. Present any feedback to the user through predefined exception messages.

UDFs should manage memory and temporary results as defined by the Sybase IQ UDF API.

Sybase IQ manages disk I/O in a reliable manner to guarantee data availability and integrity. UDFs should generally not write to or read from the file system.

Sybase IQ is a multiuser application. Many users can simultaneously execute the same UDF. Certain OLAP queries cause a UDF to be executed multiple times within the same query, sometimes in parallel. For additional details on setting UDFs to run in parallel, see *Aggregate UDF calling patterns* on page 76.

Internationalization

Sybase IQ has been internationalized, so that it can be sold in different countries around the world, to users who speak many different languages. Error messages have been extracted from the code and put into external files. This lets you localize error messages to new languages, without having to make extensive code changes.

To support multiple languages, UDFs should also be internationalized. In general, most UDFs will operate on numeric data. In some cases, a UDF may accept string keywords as one or more of the parameters. Place these keywords in external files, in addition to any exception text and log messages used by the UDF.

Sybase IQ has also been localized to a few non-English foreign languages. To support localization to the same languages as Sybase IQ supports, Sybase recommends that you internationalize UDFs allowing them to be localized at a later date, by an independent organization.

For details about international language support in Sybase IQ, see the *Sybase IQ System Administration Guide, > International Languages and Character Sets.*

See also *Debugging Using Cross-Character-Set Maps*at www.Sybase.com. This paper discusses how to with multi byte data, as opposed to input keywords, exception messages, and log entries.

Platform Differences

Develop UDFs to run on a variety of platforms supported by Sybase IQ. The Sybase IQ 15.x server runs on 64-bit architectures, and is supported under several platforms of the MS Windows (64-bit) family of operating systems. It is also supported on various types and versions of UNIX (64-bit), including Solaris, HP-UX, AIX, and Linux.

Practices to avoid

Learn good practices for creating user-defined functions.

- Do not hard-code library paths in SQL registration scripts. This practice makes it difficult to provide flexibility to the user to install the UDFs into the same directory as Sybase IQ.
- Do not write output files. Sybase IQ version 15.1 includes an architectural limitation on UDF results within Sybase IQ. Due to this limitation, some UDFs have been developed to write to temporary results files outside of the Sybase IQ container. The first ESD release of version 15.1 will expand, this architectural limit to a usable size. The limit is scheduled to be removed completely from a future version of Sybase IQ.
- Do not write ambiguous code, or constructs that can unexpectedly loop forever, without providing a mechanism for the user to cancel the UDF invocation (see the function **'get_is_cancelled()'** in (*UDF and UDAF callback functions* on page 75).
- Do not perform complex, or memory-intensive operations that are repeated every invocation. When a UDF call is made against a table that contains many thousands of rows, efficient execution becomes paramount. Sybase recommends that you allocate blocks of memory for a thousand to several thousand rows at a time, rather than on a row-by-row basis.
- Do not open a database connection, or perform database operations from within a UDF. All parameters and data required for UDF execution must be passed as parameters to the UDF.
- Do not use reserved words when naming UDFs.

Types of user-defined functions

There are several types of user-defined functions.

- Scalar or aggregate – the UDF operates either on a single value (scalar) or multiple values (aggregate). Aggregate UDFs are also sometimes known as UDAs or UDAFs. The context

structure for coding aggregate UDFs is slightly different than the context structure used for coding scalar UDFs.

- Deterministic or non deterministic – the result of a function can be determined either solely by the input parameters and data (deterministic), or by some random behavior (non-deterministic). Parameters of non-deterministic UDFs typically need a random seed as one of the input parameters.
- Aggregate UDFs only) single output or Multiple outputs – an aggregate function can produce either a single result, or a set of results. The number of data points in the output result set may not necessarily match the number of data points in the input set. In Sybase IQ 15.1, multiple-output aggregate UDFs must use a temporary output file to hold the results. In future versions, you may be able to implement aggregate functions that return multiple results packaged in a large binary object (BLOB).

Naming conventions for user-defined functions

UDF names must follow the same restrictions as other identifiers in Sybase IQ.

Sybase IQ identifiers have a maximum length of 128 bytes. For simplicity of use, UDF names should start with an alphabetic character. Alphabetic characters as defined by Sybase IQ include the letters of the alphabet, plus underscore (_), at sign (@), number or pound sign (#) and dollar sign ($). UDF names should consist entirely of these alphabetic characters as well as digits (the numbers 0 through 9). UDF names should not conflict with SQL reserved words. There is a list of SQL reserved words in the Sybase IQ 15.1 *Reference: Building Blocks, Tables, and Procedures > SQL Language Elements > section Reserved words*

Although UDF names (as other identifiers) may also contain reserved words, spaces, characters other than those listed above, and may start with a non-alphabetic character, this is not recommended. If UDF names have any of these characteristics, you must enclose them in quotes or square brackets, which makes it more difficult to use them.

The UDFs reside in the same name space as other SQL functions and stored procedures. To avoid conflicts with existing stored procedures and functions, preface UDFs with a unique short (2-letter to 5-letter) acronym and underscore. Choose UDF names that do not conflict with other SQL functions or stored procedures already defined in the local environment.

These are some of the prefixes that are already in use:

- **debugger_tutorial** – a stored procedure delivered with the native Sybase IQ installation.
- **ManageContacts** – a stored procedure delivered with the Sybase IQ demo database.
- **Show** – stored procedures used to display data from the Sybase IQ demo database.
- **sp_Detect_MPX_DDL_conflicts** – a stored procedure delivered with the native Sybase IQ installation.
- **sp_iqevbegintxn** – a stored procedure delivered with the native Sybase IQ installation.
- **sp_iqmpx** – functions and stored procedures provided by Sybase IQ to assist in multiplex administration.

- **ts_** – optional financial time series and forecasting functions.

Design basics of user-defined functions

There are some basic considerations to keep in mind while developing UDFs.

This document assumes that the UDF developer is familiar with the basics of developing software, including good program design and development, independent testing, and so on.

In addition to standard software development practices, UDF developers should remember that they are developing code to be executed within the Sybase IQ database container, and to understand the limitations imposed by the database container.

Developers of aggregate UDFs should also be familiar with OLAP queries, and how they translate into UDF calling patterns.

Because the UDFs may be invoked by several threads simultaneously, they must be constructed to be thread-safe.

Sample code

Starting with Sybase IQ 15.1, sample UDF source code is delivered along with the product. The newest version of the sample code is always delivered with the most current version of Sybase IQ.

To see if there were last-minute changes to the sample UDF source code, see to the Sybase IQ Release Bulletin for the relevant release and operating system platform.

The sample UDF code documented in the User-Defined Functions Guide may not be the latest version as delivered with the Sybase IQ product. On UNIX platforms, the sample UDF code is in:

```
$SYBASE/IQ-15_1/samples/udf
(where $SYBASE is the installation root)
```

On Windows platforms, the sample UDF code is in:

```
C:\Documents and Settings\All Users\SybaseIQ\samples\udf
```

Creating and executing user-defined functions

User-defined functions (UDFs) return a single value to the calling environment.

Note: User-defined functions are a licensable option, and require the IQ_UDF license. Installing the license enables user-defined functions.

You can install Sybase IQ in a wide variety of configurations. UDFs must be easily installed within this environment, and must be able to run within all supported configurations. The Sybase IQ installer provides a default installation directory, but allows users to select a different installation directory. UDF developers should consider providing the same flexibility with the installation of the UDF libraries and associated SQL function definition scripts.

Creating a user-defined function

Learn how to create and configure external C/C++ user-defined functions (UDFs).

For instructions on creating UDFs using Interactive SQL, see *System Administration Guide: Volume 2 > Using Procedures and Batches.*

1. Declare the UDF to the server by using the **CREATE FUNCTION** or **CREATE AGGREGATE FUNCTION** statements. Write and execute these statements as commands, or use the appropriate CREATE statement using the Sybase Central New Function wizard.

   The external C/C++ form of the CREATE FUNCTION statement requires DBA or RESOURCE authority, so standard users do not have the authority to declare any UDFs of this type.
2. *Write the UDF library identification function.* on page 15.
3. Define the UDF as a set of C or C++ functions. See *Defining a scalar UDF* on page 33 or *Defining an aggregate UDF* on page 47.
4. Implement the function entry points in C/C++.
5. *Compile the UDF functions and the library identification functions.* on page 16.
6. Link the compiled file into a dynamically linkable library.

Any reference to a UDF in a SQL statement first, if necessary, links the dynamically linkable library. The *calling patterns* on page 75 are then called.

Because these high-performance external C/C++ user-defined functions involve the loading of non-server library code into the process space of the server, there are potential risks to data integrity, data security, and server robustness from poorly or maliciously written functions. To

manage these risks, each IQ server can explicitly *enable or disable this functionality* on page 25.

Creating a user-defined function using SQL Anywhere dialects

Watcom-SQL and Transact-SQL are SQL dialects supported by SQL Anywhere, and can be used when creating user-defined functions.

1. In Sybase Central, connect to the database as a user with DBA or Resource authority.
2. **Select View > Folders.** > > >
3. In the left pane, right-click **Procedures & Functions** and select New > Function.
4. Enter a name for the function and select the user who will own the function.
5. Select the SQL dialect or language for the function. Click **Next.**
6. Select the type of value to be returned in the function, and specify the size, units, and scale for the value.
7. Type a name for the return value and click **Next.**
8. Add a comment describing the purpose of the new function. Click **Finish.**
9. In the right pane, click the **SQL tab** to complete the procedure code.

Declaring a scalar user-defined function in Sybase Central

Sybase IQ supports simple scalar UDFs that can be used anywhere the SQRT function can be used. These scalar UDFs can be deterministic, which means that for a given set of argument values, the function always returns the same result value. Sybase IQ also supports nondeterministic scalar functions, which means that the same arguments can return different results.

1. In Sybase Central, connect to the database as a user with DBA or Resource authority.
2. In the left pane, right-click **Procedures & Functions** and **select New > Function**.
3. In the **Welcome dialog**, type a name for the function and select the user to be the owner of the function.
4. To create a user-defined function, select **External C/C++**. Click **Next**.
5. In the **External Function Attributes dialog**, select **Scalar**.
6. Type the name of the dynamically linkable library file, omitting the .so or .dll extension.
7. Type a name for the descriptor function. Click **Next**.
8. Select the type of value to be returned in the function, and specify the size, units, and scale for the value. Click **Next**.
9. Select whether or not the function is deterministic.
10. Specify if the function respects or ignores NULL values.
11. Select whether the privileges used for running the function are from the defining user (definer) or the calling user (invoker).

12. Add a comment describing the purpose of the new function. Click **Finish**.
13. In the right pane, click the **SQL tab** to complete the procedure code.

Declaring a user-defined aggregate function in Sybase Central

Sybase IQ supports user-defined aggregate functions (UDAFs). The SUM function is an example of a built-in aggregate function. A simple aggregate function takes a set of argument values and produces a single result value from that set of inputs. User-defined aggregate functions can be written that can be used anywhere the SUM aggregate can be used.

1. In Sybase Central, connect to the database as a user with DBA or Resource authority.
2. In the left pane, right-click **Procedures & Functions** and **select New > Function**.
3. In the **Welcome dialog**, type a name for the function and select which user will be the owner of the function.
4. To create a user-defined function, select **External C/C++**. Click **Next**.
5. Select **Aggregate**.
6. Type the name of the dynamically linkable library file, omitting the .so or .dll extension.
7. Type a name for the descriptor function. Click **Next**.
8. Select the type of value to be returned in the function, and specify the size, units, and scale for the value. Click **Next**.
9. Select whether the privileges used for running the function are from the defining user (definer) or the calling user (invoker).
10. Specify whether the function is allowed to be, required to be, or not allowed to be, used in an **OVER** clause. Click **Next**.

    If the function is not allowed to be used in an **OVER** clause, proceed with step 14.
11. Specify if the function requires the user of an **ORDER BY** clause when it is used to define a window. Click **Next**.
12. Specify if the function is allowed to be used in a **WINDOW FRAME** clause, is required to be used in an **WINDOW FRAME** clause, or is not allowed to be used in a **WINDOW FRAME** clause. Click **Next**.

    If the function is not allowed to be used in a **WINDOW FRAME** clause, skip to step 14.
13. Identify the constraints on the **WINDOW FRAME** clause. Click **Next**.
14. Specify if duplicate input values need to be filtered out by the database server prior to calling the function.
15. Identify if the return value of the function is **NULL** or a fixed value when it is called with no data. Click **Next**.
16. Add a comment describing the purpose of the new function. Click **Finish**.
17. In the right pane, click the **SQL tab** to complete the procedure code.

    The new function appears in **Procedures & Functions**.

User-defined function restrictions

External C/C++ user-defined functions have some restrictions.

- Write all UDFs in a manner that allows them to be called simultaneously by different users while receiving different context functions.
- If a UDF accesses a global or shared data structure, the UDF definition must implement the appropriate locking around its accesses to that data, including the releasing of that locking under all normal code paths and all error handling situations.
- UDFs implemented in C++ may provide overloaded "new" operators for their classes, but they should never overload the global "new" operator. On some platforms, the effect of doing so is not limited to the code defined within that specific library.
- Write all aggregate UDFs and all deterministic scalar UDFs such that the receipt of the same input values always produces the same output values. Any scalar function for which this is not true must be declared as NONDETERMINISTIC to avoid the potential for incorrect answers.
- Users can create a standard SQL function without a DBA authority, but they cannot create a function which will invoke an external library without having DBA permissions. Attempting to do this results in an error message `"You do not have permission to use the create function statement."`

Calling user-defined functions

You can use a user-defined function, subject to permissions, any place you use a built-in nonaggregate function.

This Interactive SQL statement returns a full name from two columns containing a first and last name:

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

fullname (Employees.GivenName,Employees.SurName)
Fran Whitney
Matthew Cobb
Philip Chin
...

The following statement returns a full name from a supplied first and last name:

```
SELECT fullname ('Jane', 'Smith');
```

fullname ('Jane','Smith')
Jane Smith

Any user who has been granted Execute permissions for the function can use the *fullname* function.

Setting the dynamic library interface

Specify the interface style to be used in the dynamically linkable library.

Each dynamically loaded library must contain exactly one copy of this definition:

```
extern "C" a_sql_uint32 extfn_use_new_api(void )
{
return EXTFN_V3_API;
}
```

This definition informs the server of which interface style is being used, and therefore how to access the UDFs defined in this dynamically linkable library. For high-performance IQ UDFs, only new interface style (EXTFN_V3_API) is supported.

Dropping user-defined functions

Once you create a user-defined function, it remains in the database until it is explicitly removed. Only the owner of the function or a user with DBA authority can drop a function from the database.

For example, to remove the function *fullname* from the database, enter:

```
DROP FUNCTION fullname
```

Granting and revoking permissions

A user-defined function is owned by the user who created it, and only that user can execute it without permission. The owner can grant permissions to other users using the **GRANT EXECUTE** command.

For example, the creator of the function *fullname* can allow *another_user* to use *fullname* by issuing:

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

Or can revoke permissions by issuing:

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

See *System Administration Guide: Volume 1 > Managing User IDs and Permissions > Granting permissions on procedures*.

Maintenance of user-defined functions

Many Sybase IQ installations are in mission-critical environments, where customers require an extremely high level of availability. System Administrators must be able to install and upgrade UDFs with little or no impact to the Sybase IQ server.

An application must not attempt to access an external library while the associated library file is being moved, overwritten, or deleted. Since libraries are automatically loaded whenever an associated SQL function is invoked, it is important to follow these steps in the exact order whenever performing any type of maintenance on existing UDF libraries:

1. Ensure all users who invoke UDFs do not have any pending queries in progress
2. Revoke the execute permission from users, and drop the SQL functions and stored procedures which reference external UDF code modules
3. Unload the library from the IQ server, using the **call sa_external_library_unload** command (shutting down the IQ server also automatically unloads the library).
4. Perform the desired maintenance on the external library files (copy, move, update, delete).
5. Edit SQL function and stored procedure definitions in the registration scripts to reflect external library locations, if the libraries were moved.
6. Grant the execute permission to users, and run registration scripts to re-create the SQL functions and stored procedures which reference external UDF code modules.
7. Invoke a SQL function or stored procedure that references the external UDF code to ensure the IQ server can dynamically load the external library.

Compiling and linking source code to build dynamically linkable libraries

Use compile and link switches when building dynamically linkable libraries for any user-defined function.

1. A UDF dynamically linkable library must include an implementation of the function **extfn_use_new_api()**. The source code for this function is in *Setting the dynamic library interface* on page 15. This function informs the server of the API style that all functions in the library adhere to. The sample source file `my_main.cxx` contains this function; you can use it without modification.
2. A UDF dynamically linkable library must also contain object code for at least one UDF function. A UDF dynamically linkable library may optionally contain multiple UDFs.

3. Link together the object code for each UDF as well as the **extfn_use_new_api()** to form a single library.

For example, to build the library "libudfex:"

- Compile each source file to produce an object file:

```
my_main.cxx
            my_bit_or.cxx
            my_bit_xor.cxx
            my_interpolate.cxx
            my_plus.cxx
            my_plus_counter.cxx
            my_sum.cxx
```

- Link together each object produced into a single library.

After the dynamically linkable library has been compiled and linked, complete one of these tasks:

- (Recommended) update the CREATE FUNCTION ... EXTERNAL NAME to include an explicit path name for the UDF library.
- Place the UDF library file into the directory where all the IQ libraries are stored.
- Start the IQ server with a library load path that includes the location of the UDF library.
  On UNIX modify the LD_LIBRARY_PATH within the `start_iq startup` script. While LD_LIBRARY_PATH is universal to all UNIX variants, SHLIB_PATH is preferred on HP, and LIB_PATH is preferred on AIX.
  On UNIX platforms, the external name specification can contain a fully qualified name, in which case the LD_LIBRARY_PATH is not used. On the Windows platform, a fully qualified name cannot be used and the library search path is defined by the PATH environment variable.

AIX switches

Use the following compile and link switches when building shared libraries on AIX.

xlC 8.0 on a PowerPC

Important: Include the code for extfn_use_new_api() in each UDF library.

Note: To compile on AIX 6.1 systems, the minimum level of the xlC compiler is 8.0.0.24.

compile switches

```
-q64 -qarch=ppc64 -qtbtable=full -qsrcmsg -qalign=natural -
qnoansialias
-qmaxmem=-1 -qenum=int -qhalt=e -qflag=w -qthreaded -
qxflags=NLOOPING
-qtmplinst=none -qthreaded
```

link switches

```
-brtl -G -lg  -lpthreads_compat -lpthreads -lm_r -ldl -bnolibpath -
v
```

HP-UX switches

Use the following compile and link switches when building shared libraries on HP-UX.

aCC 6.17 on Itanium

Important: Include the code for extfn_use_new_api() in each UDF library.

compile switches

```
+noeh -ext +W740,749,829 +W1031 +DD64 +DSblended +FPD -Aa +ub
-U_HP_INSTANTIATE_T_IN_LIB -Wc,-ansi_for_scope,on -mt -z
```

link switches

```
 -b -Wl,+s
```

Linux switches

Use the following compile and link switches when building shared libraries on Linux.

g++ 4.1.1 on x86

Important: Include the code for extfn_use_new_api() in each UDF library.

compile switches

```
-fPIC -fsigned-char -fno-exceptions -pthread -fno-omit-frame-
pointer
-Wno-deprecated -Wno-ctor-dtor-privacy
```

link switches

```
-ldl -lnsl -lm -lpthread -shared -Wl,-Bsymbolic -Wl,-shared
```

Note: You can use gcc on Linux as well. While linking with **gcc**, link in the C++ run time library by adding `-lstdc++` to the link switches.

Examples

- *Example 1*

  ```
  g++ -c my_interpolate.cxx -fPIC -fsigned-char -fno-exceptions -
  pthread
        -fno-omit-frame-pointer -Wno-deprecated -Wno-ctor-dtor-
  privacy
        -I${IQDIR15}/sdk/include/
  ```

- *Example 2*

  ```
  g++ -c my_main.cxx -fPIC -fsigned-char -fno-exceptions -pthread
        -fno-omit-frame-pointer -Wno-deprecated -Wno-ctor-dtor-
  privacy
        -I${IQDIR15}/sdk/include/
  ```

- *Example 3*

```
ld -G my_main.o my_interpolate.o -ldl -lnsl -lm -lpthread -shared
      -o my_udf_library.so
```

xlC 8.0 on a PowerPC

compile switches

```
 -q64 -qarch=ppc64 -qcheck=nullptr -qinfo=gen -qtbtable=full -
qsrcmsg
-qnoansialias -qminimaltoc -qmaxmem=-1 -qenum=int -qhalt=e -qflag=w
-qthreaded
-qxflags=NLOOPING -qtmplinst=none
```

link switches

```
-qmkshrobj -ldl -lg  -qthreaded -lnsl -lm
```

Solaris switches

Use the following compile and link switches when building shared libraries on Solaris.

Sun Studio 12 on SPARC

Important: Include the code for extfn_use_new_api() in each UDF library.

compile switches

```
-mt -noex +w -KPIC -i -instances=explicit -V -xtarget=ultra3cu -m64
-xlibmopt
-xlibmil -features=no%conststrings
-erroff=truncwarn,nokeyworddefine,diffenumtype
```

link switches

```
-z defs -G -ldl -lnsl -lsocket -ladm -lposix4 -lCrun -lCstd -lc -lm
-lefi
-liostream -lkstat
```

Sun Studio 12 on x86

compile switches

```
+w2 -m64 -features=no%conststrings
-erroff=truncwarn,nokeyworddefine,diffenumtype,doubunder -errtags -
mt  -noex
-KPIC -instances=explicit -xlibmopt -xlibmil
```

link switches

```
-z defs -G -ldl -lnsl -lsocket -ladm -lposix4 -lCrun -lCstd -lc -lm
-lefi
-liostream -lkstat -m64
```

Windows switches

Use the following compile and link switches when building shared libraries on Windows.

Visual Studio 2008 on x86

Important: Include the code for extfn_use_new_api() in each UDF library.

compile and link switches

This example is for a DLL containing the my_plus function. You must include an EXPORT switch for the descriptor function for each UDF contained in the DLL.

```
cl /Zi /LD   /I includefilepath     my_main.cxx my_plus.cxx  /link /
map
/INCREMENTAL:NO -EXPORT:extfn_use_new_api -EXPORT:my_plus  /
out:libiqudfex.dll
```

Example

Environment setup

```
set VCBASE=c:\dev\vc9
      set MSSDK=C:\dev\mssdk6.0a
      set IQINSTALLDIR=C:\Sybase\IQ
      set OBJ_DIR=%IQINSTALLDIR%\IQ-15_1\samples\udf\objs
      set SRC_DIR=%IQINSTALLDIR%\IQ-15_1\samples\udf\src
      call %VCBASE%\VC\bin\vcvars32.bat
```

- *Example 1*

```
%VCBASE%\VC\bin\amd64\cl -c -nologo -DNDEBUG -DWINNT -D_USRDLL
      -D_WINDLL -D_WIN64 -DWIN64 -
D_WIN32_WINNT=_WIN32_WINNT_WINXP
     -DWINVER=_WIN32_WINNT_WINXP -D_MBCS -GS -W3 -Zi -favor:AMD64
      -DSYB_LP64 -D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=64 -
DHMSWNT
      -D_CRT_SECURE_NO_DEPRECATE -D_CRT_NONSTDC_NO_DEPRECATE
      -DPOINTERS_ARE_64BITS -DLONG_IS_64BITS -
D_RWSTD_NO_EXCEPTIONS
      -I"%VCBASE%\VC\include" -I"%MSSDK%\include "-I"%MSSDK%\Lib
\AMD64"
      -I"%VCBASE%\VC\lib\amd64" -DMSDCXX -DINT64_WORKAROUND
      -DSUPPORTS_UDAF -Od -Zi -MD -I"%IQINSTALLDIR%\IQ-15_1\sdk
\include"
     -Fo"%OBJ_DIR%\my_interpolate.o" %SRC_DIR%\my_interpolate.cxx
```

- *Example 2*

```
%VCBASE%\VC\bin\amd64\cl -c -nologo -DNDEBUG -DWINNT -D_USRDLL
      -D_WINDLL -D_WIN64 -DWIN64 -
D_WIN32_WINNT=_WIN32_WINNT_WINXP
     -DWINVER=_WIN32_WINNT_WINXP -D_MBCS -GS -W3 -Zi -favor:AMD64
      -DSYB_LP64 -D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=64 -
DHMSWNT
      -D_CRT_SECURE_NO_DEPRECATE -D_CRT_NONSTDC_NO_DEPRECATE
      -DPOINTERS_ARE_64BITS -DLONG_IS_64BITS -
```

```
D_RWSTD_NO_EXCEPTIONS
      -I"%VCBASE%\VC\include" -I"%MSSDK%\include "-I"%MSSDK%\Lib
\AMD64"
      -I"%VCBASE%\VC\lib\amd64" -DMSDCXX -DINT64_WORKAROUND
      -DSUPPORTS_UDAF -Od -Zi -MD -I"%IQINSTALLDIR%\IQ-15_1\sdk
\include"
      -Fo"%OBJ_DIR%\my_main.o" %SRC_DIR%\my_main.cxx
```

- *Example 3*

```
%VCBASE%\VC\bin\amd64\link /LIBPATH:%VCBASE%\VC\lib\amd64
      /LIBPATH:%MSSDK%\lib\bin64 kernel32.lib -manifest -DLL -
nologo
      -MAP:"%OBJ_DIR%\libudfex.map_deco" /OUT:"%OBJ_DIR%
\libudfex.dll"
      "%OBJ_DIR%\my_interpolate.o" "%OBJ_DIR%\my_main.o" /DLL
      -EXPORT:extfn_use_new_api -EXPORT:my_interpolate
```

- *Example 4*

```
%MSSDK%\bin\mt -nologo -manifest "%OBJ_DIR%
\libudfex.dll.manifest"
      -outputresource:"%OBJ_DIR%\libudfex.dll;2"
```

Using Microsoft Visual Studio debugger for user-defined functions

These steps will give the Microsoft Visual Studio 2008 developers the ability to step through the user-defined functions code.

1. Attach the debugger to a running server:

   ```
   devenv /debugexe "%IQDIR15%\bin32\iqsrv15.exe"
   ```

2. Goto Debug | Attach to Process
3. To start the server and debugger together:

   ```
   devenv /debugexe "%IQDIR15%\bin32\iqsrv15.exe" [commandline
   options for your server]
   ```

   Each platform will have a debugger and each will have their own command line syntax. Sybase IQ source code is not required. The msvs debugger will recognize when the user-defined functions source is executed and break at the set breakpoints. When control returns from the user-defined functions to the server, you will only see machine code.

SQL data types

UDF declarations support only certain SQL data types.

You can use the following SQL data types in a UDF declaration, either as data types for arguments to a UDF, or as return-value data types:

- UNSIGNED BIGINT – an unsigned 64-bit integer, requiring 8 bytes of storage. The data type identifier to be used within UDF code is DT_UNSBIGINT, and the C/C++ data type typedef to be used for such values within a UDF is “a_sql_uint64”. Several C/C++ typedefs are included with Sybase IQ to make it easier for application developers to write portable UDF implementations.
- BIGINT – a signed 64-bit integer, requiring 8 bytes of storage. The data type identifier is DT_BIGINT, and the C/C++ data type typedef to be used for such values is “a_sql_int64.”
- UNSIGNED INT – an unsigned 32-bit integer, requiring 4 bytes of storage. The data type identifier is DT_UNSINT, and the C/C++ data type typedef to be used for such values is “a_sql_uint32.”
- INT – a signed 32-bit integer, requiring 4 bytes of storage. The data type identifier is DT_INT, and the C/C++ data type typedef to be used for such values is “a_sql_int32.”
- SMALLINT – a signed 16-bit integer, requiring 2 bytes of storage. The data type identifier is DT_SMALLINT, and the C/C++ data type to be used for such values is “short.”
- TINYINT – An unsigned 8-bit integer, requiring 1 byte of storage. The data type identifier is DT_TINYINT, and the C/C++ data type to be used for such values is “unsigned char.”
- DOUBLE – a signed 64-bit double-precision floating point number, requiring 8 bytes of storage. The data type identifier is DT_DOUBLE, and the C/C++ data type to be used for such values is “double.”
- REAL – a signed 32-bit floating point number, requiring 4 bytes of storage. The data type identifier is DT_FLOAT, and the C/C++ data type to be used for such values is “float.”
- FLOAT – in SQL, depending on the associated precision, a FLOAT is either a signed 32-bit floating point number requiring 4 bytes of storage, or a signed 64-bit double-precision floating point number requiring 8 bytes of storage. You can use the SQL data type FLOAT only in a UDF declaration if the optional precision for FLOAT data types is not supplied. Without a precision, FLOAT is a synonym for REAL, for which the data type identifier is DT_FLOAT, and the C/C++ data type to be used for such values is “float.”
- CHAR(<n>) – a fixed-length blank-padded character string, in the database default character set. The maximum possible length, “<n>”, is 32767. The data is not null-byte terminated. The data type identifier is DT_FIXCHAR, and the C/C++ data type to be used for such values is “char *.”
- VARCHAR(<n>) – a varying-length character string, in the database default character set. The maximum possible length, “<n>”, is 32767. The data is not null-byte terminated. For UDF input arguments, the actual length, when the value is not NULL, must be retrieved from the total_length field within the an_extfn_value structure. Similarly, for a UDF result of this type, the actual length must be set in the total_length field. The data type identifier is DT_VARCHAR, and the C/C++ data type to be used for such values is “char *.”
- BINARY(<n>) – a fixed-length null-byte padded binary, value with a maximum possible binary length, “<n>”, of 32767. The data is not null-byte terminated. The data type identifier is DT_BINARY, and the C/C++ data type usually used for such values is “unsigned char *.”
- VARBINARY(<n>) – a varying-length binary value, for which the maximum possible length, “<n>”, is 32767. The data is not null-byte terminated. For UDF input arguments, the actual length, when the value is not NULL, must be retrieved from the total_length

field within the an_extfn_value structure. Similarly, for a UDF result of this type, you must set the actual length in the total_length field. The data is not null-byte terminated. The data type identifier is DT_BINARY, and the C/C++ data type usually used for such values is "unsigned char *."

- DATE – a calendar date value, which is passed to or from a UDF as an unsigned integer. The value given to the UDF is guaranteed to be usable in comparison and sorting operations. A larger value indicates a later date. If the actual date components are required, the UDF must invoke the convert_value api in order to convert to the type DT_TIMESTAMP_STRUCT. This date type represents date and time with this structure:

```
typedef struct sqldatetime {
    unsigned short    year;        /* e.g. 1992           */
    unsigned char    month;        /* 0-11               */
   unsigned char   day_of_week;   /* 0-6  0=Sunday, 1=Monday, ...
*/
    unsigned short    day_of_year;    /* 0-365           */
    unsigned char    day;        /* 1-31               */
    unsigned char    hour;        /* 0-23               */
    unsigned char    minute;    /* 0-59                */
    unsigned char    second;    /* 0-59                */
    a_sql_uint32     microsecond;    /* 0-999999           */
} SQLDATETIME;
```

- TIME – a value that precisely describes a moment within a given day. The value is passed to the UDF as an UNSIGNED BIGINT. The value given to the UDF is guaranteed to be usable in comparison and sorting operations. A larger value indicates a later time. If the actual time components are required, the UDF must invoke the convert_value to convert to the type DT_TIMESTAMP_STRUCT.
- DATETIME, SMALLDATETIME, or TIMESTAMP – a calendar date and time value, which is passed to or from a UDF as an UNSIGNED BIGINT. The value given to the UDF is guaranteed to be usable in comparison and sorting operations. A larger value indicates a later datetime. If the actual time components are required, the UDF must invoke the convert_value to convert to the type DT_TIMESTAMP_STRUCT.

Unsupported data types

You cannot use the following SQL data types in a UDF declaration, either as data types for arguments to a UDF, or as return-value data types:

- BIT – Should typically be handled in the UDF declaration as a TINYINT data type, and then the implicit data type conversion from BIT automatically handles the value translation.
- DECIMAL(<precision>, <scale>) or NUMERIC(<precision>, <scale>) – Depending on the usage, DECIMAL is typically handled as a DOUBLE data type, but various conventions may be imposed to enable the use of INT or BIGINT data types.
- LONG VARCHAR – Not currently supported.
- LONG BINARY – Not currently supported.
- TEXT – Not currently supported.

Testing user-defined functions

After UDF external code has been coded, compiled and linked, and the corresponding SQL functions and stored procedures have been defined, the UDFs are ready to be tested.

The reliability required by a database is extremely high. UDFs running within a database environment must maintain this high level of reliability. With the first implementation of the UDF API, UDFs run within the Sybase IQ server. If a UDF aborts prematurely or unexpectedly, the Sybase IQ server may abort. Ensure via thorough testing in a development or test environment, that UDFs do not terminate prematurely or abort unexpectedly under any circumstances.

Enabling and disabling user-defined functions

Sybase IQ includes a security feature, external_procedure_v3, which enables or disables the ability of a server to make use of high performance in-process UDFs.

A database should maintain data integrity. Under no circumstances should data be lost, modified, augmented, or corrupted. Since UDF execution happens within the Sybase IQ server, there is a risk of corrupting data; practice caution with memory management and any other use of pointers. Sybase strongly recommends that you to install and execute UDFs within a read-only multiplex node. For added protection, use a startup option in each IQ server to enable or disable the execution of UDF.

Note: By default, UDF execution on a multiplex writer and coordinator nodes are disabled.. All other nodes are enabled by default.

Administrators can enable version 3 UDFs for any server by specifying this in the server startup command or in the configuration file:

```
-sf -external_procedure_v3
```

Administrators can disable version 3 UDFs for any server by specifying this in the server startup command or in the configuration file:

```
-sf external_procedure_v3
```

Additional information on the -sf flag is available in the *SQL Anywhere Server - Database Administration* guide. Do not use the values listed in the SQL Anywhere document which are applicable to Sybase IQ.

Initially executing a user-defined function

To ensure the safest environment possible, Sybase strongly recommends that you install and invoke UDFs from a read-only IQ server node in a multiplex installation.

The Sybase IQ server does not load the library containing the UDF code until the first time the UDF is invoked. The first execution of a UDF residing in a library that has not yet been loaded may be unusually slow. After the library is loaded, the subsequent invocation of the same UDF or another UDF contained in the same library have the expected performance.

Libraries using the stored procedure SA_EXTERNAL_LIBRARY_UNLOAD. These libraries are not reloaded when IQ server is stopped and restarted.

In environments where after-hours maintenance operations require a shutdown and restart of the IQ server, run some test queries after the Sybase IQ server has been restarted. This ensures that the appropriate libraries are loaded in memory for optimal query performance during business hours.

Managing external libraries

Each external library is loaded the first time a UDF that requires it is invoked. A loaded library remains loaded for the life of the server. It is not loaded when a CREATE FUNCTION call is made, nor is it automatically unloaded when a DROP FUNCTION call is made.

If the library version must be updated, the **dbo.sa_external_library_unload** procedure forces the library to be unloaded without restarting the server. The call to unload the external library is successful only if the library in question is not currently in use. The procedure takes one optional parameter, a long varchar, that specifies the name of the library to be unloaded. If no parameter is specified, all external libraries not in use are unloaded.

Note: Unload existing libraries from a running Sybase IQ server before replacing the dynamically link library. Failure to unload the library can result in a server crash. Before replacing a dynamically linkable library, either shut down the Sybase IQ server or use the sa_external_library_unload function to unload the library.

For Windows, unload an external function library using:

```
call sa_external_library_unload('library')
```

For UNIX, unload an external function library using:

```
call sa_external_library_unload('library.so')
```

If a registered function uses a complete path, for example, `/abc/def/library`, first unregister the function.

In Windows, use

```
call sa_external_library_unload('/abc/def/library')
```

In UNIX, use

```
call sa_external_library_unload('\abc\def\library.so')
```

Note: The library path is required in the SQL function declaration only if the library is not already located within a directory in the library load path.

Controlling error checking and call tracing

The **external_UDF_execution_mode** option controls the amount of error checking and call tracing that is performed when statements involving external V3 user-defined functions are evaluated.

You can use **external_UDF_execution_mode** during development of a UDF to aid in debugging while you are developing UDFs.

Allowed values

0, 1, 2

Default value

0

Scope

Can be set as public, temporary, or user.

Description

When set to 0, the default, external UDFs are evaluated in a manner that optimizes the performance of statements using UDFs.

When set to 1, external UDFs are evaluated to validate the information passed back and forth to each UDF function.

When set to 2, external UDFs are evaluated to not only validate the information passed back and forth to the UDF, but also to log, in the `iqmsg` file, every call to the functions provided by the UDFs and every callback from those functions back into the server.

Enabling full tracing in a debug environment

Consider turning on full tracing within Sybase IQ in a debug environment. To enable IQ tracing, add the following flags to the Sybase IQ server startup command line or the Sybase IQ config file:

```
-zr all -zo filename
```

where *filename* is the complete path to the tracing output file.

Viewing Sybase IQ log files

Sybase IQ provides extensive logging and tracing capabilities. UDFs should provide the same or better level of detailed logging, in the event of problems in the UDF code.

Log files for the database are generally located with the database file and configuration file. On Unix platforms, there are two files named after the database instance, one with a .stderr extension and one with a .stdout extension. On Windows, by default, the stderr file is not generated. To capture the stderr messages along with the stdout messages under Windows, redirect the stdout and stderr:

```
iqsrv15.exe @iqdemo.cfg iqdemo.db 2>&1 > iqdemo.stdout
```

The Windows output messages are slightly different from what is generated on Unix platforms.

Scalar User-Defined Functions

Sybase IQ supports simple scalar user-defined functions (UDFs) that can be used anywhere the SQRT function can be used.

These scalar UDFs can be deterministic, which means that for a given set of argument values the function always returns the same result value, or they can be nondeterministic scalar functions, which means that the same arguments can return different results.

Note: The scalar UDF examples referenced in this chapter are installed with the IQ server, and can be found as .cxx files in `$IQDIR15/samples/udf`. You can also find them in the `$IQDIR15/lib64/libudfex` dynamically linkable library.

Declaring a Scalar UDF

Only a DBA, or someone with DBA authority can declare an in-process external UDF. There is also a server startup option that allows an administrator to enable or disable this style of user-defined function.

After the UDF code has been written and compiled, create a SQL function that invokes the UDF from the appropriate library file, sending the input data to the UDF.

Note: You can also *create the user-defined function declaration in Sybase Central* on page 12.

By default, all user-defined functions use the access permissions of the owner of the UDF.

Note: Users are required to have DBA authority in order to declare UDF functions.

The syntax for creating an IQ scalar UDF is:

```
scalar-udf-declaration:
CREATE FUNCTION [ owner.]function-name
     ( [ parameter , ... ] )
RETURNS data-type
     [ routine-characteristics ... ]
EXTERNAL NAME library-and-entry-point-name-string

parameter:
    param-name data-type [ DEFAULT value ]

routine-characteristics:
     [NOT] DETERMINISTIC
  | { IGNORE | RESPECT } NULL VALUES
     | SQL SECURITY { INVOKER | DEFINER }
```

The defaults for the characteristics in the above syntax are:

```
DETERMINISTIC
RESPECT NULL VALUES
SQL SECURITY DEFINER
```

To minimize potential security concerns, Sybase recommends that you use a fully qualified path name to a secure directory for the library name portion of the EXTERNAL NAME clause.

SQL Security

Defines whether the function is executed as the INVOKER, (the user who is calling the function), or as the DEFINER (the user who owns the function). The default is DEFINER.

SQL SECURITY INVOKERuses additional memory, because each user that calls the procedure requires annotation. Additionally, name resolution is performed on both the user name and the INVOKER. Qualify all object names (tables, procedures, and so on) with their appropriate owner.

External Name

A function using the **EXTERNAL NAME** clause is a wrapper around a call to a function in an external library. A function using **EXTERNAL NAME** can have no other clauses following the **RETURNS** clause. The library name may include the file extension, which is typically `.dll` on Windows and `.so` on UNIX. In the absence of the extension, the software appends the platform-specific default file extension for libraries.

The **EXTERNAL NAME** clause is not supported for temporary functions. See “Calling external libraries from procedures” in *SQL Anywhere Server – Programming*.

You can start the IQ server with a library load path that includes the location of the UDF library. On Unix variants, modify the LD_LIBRARY_PATH in the `start_iq startup` script. While LD_LIBRARY_PATH is universal to all UNIX variants, SHLIB_PATH is preferred on HP, and LIB_PATH is preferred on AIX.

On Unix platforms, the external name specification can contain a fully qualified name, in which case the LD_LIBRARY_PATH is not used. On the Windows platform, a fully qualified name cannot be used and the library search path is defined by the PATH environment variable.

Note: Scalar user-defined functions and user-defined aggregate functions are not supported in updatable cursors.

UDF example: my_plus declaration

The “my_plus” example is a simple scalar function that returns the result of adding its two integer argument values.

my_plus declaration

When my_plus resides within the dynamically linkable library my_shared_lib, the declaration for this example looks like this:

```
CREATE FUNCTION my_plus (IN arg1 INT, IN arg2 INT)
     RETURNS INT
     DETERMINISTIC
     IGNORE NULL VALUES
     EXTERNAL NAME 'my_plus@libudfex'
```

This declaration says that my_plus is a simple scalar UDF residing in my_shared_lib with a descriptor routine named describe_my_plus. Since the behavior of a UDF may require more than one actual C/C++ entry point for its implementation, this set of entry points is not directly part of the CREATE FUNCTION syntax. Instead, the CREATE FUNCTION statement EXTERNAL NAME clause identifies a descriptor function for this UDF. A descriptor function, when invoked, returns a descriptor structure that is defined in detail in the next section. That descriptor structure contains the required and optional function pointers that embody the implementation of this UDF.

This declaration says that my_plus accepts two INT arguments and returns an INT result value. If the function is invoked with an argument that is not an INT, and if the argument can be implicitly converted into an INT, the conversion happens before the function is called. If this function is invoked with an argument that cannot be implicitly converted into an INT, a conversion error is generated.

Further, the declaration states that this function is deterministic. A deterministic function always returns the identical result value when supplied the same input values. This means the result cannot depend on any external information beyond the supplied argument values, or on any side effects from previous invocations. By default, functions are assumed to be deterministic, so the results are the same if this characteristic is omitted from the CREATE statement.

The last piece of the above declaration is the IGNORE NULL VALUES characteristic. Nearly all built-in scalar functions return a NULL result value if any of the input arguments are NULL. The IGNORE NULL VALUES states that the my_plus function follows that convention, and therefore this UDF routine is not actually invoked when either of its input values are NULL. Since RESPECT NULL VALUES is the default for functions, this characteristic must be specified in the declaration for this UDF to get the performance benefits. All functions that may return a non-NULL result given a NULL input value must use the default RESPECT NULL VALUES characteristic.

In the following example query, my_plus appears in the SELECT list along with the equivalent arithmetic expression:

```
SELECT my_plus(t.x, t.y) AS x_plus_y_one, (t.x + t.y)AS x_plus_y_two
FROM t
WHERE t.z = 2
```

In the following example, my_plus is used in several different places and different ways within the same query:

```
SELECT my_plus(t.x, t.y), count(*)
FROM t
WHERE t.z = 2
```

```
AND my_plus(t.x, 5) > 10
AND my_plus(t.y, 5) > 10
GROUP BY my_plus(t.x, t.y)
```

UDF example: my_plus_counter declaration

The “my_plus_counter” example is a simple nondeterministic scalar UDF that takes a single integer argument, and returns the result of adding that argument value to an internal integer usage counter. If the input argument value is NULL, the result is the current value of the usage counter.

my_plus_counter declaration

Assuming that my_plus_counter also resides within the dynamically linkable library my_shared_lib, the declaration for this example is:

```
CREATE FUNCTION my_plus_counter (IN arg1 INT DEFAULT 0)
     RETURNS INT
     NOT DETERMINISTIC
     RESPECT NULL VALUES
     EXTERNAL NAME 'describe_my_plus_counter@my_shared_lib'
```

The RESPECT NULL VALUES characteristic means that this function is called even if the input argument value is NULL. This is necessary because the semantics of my_plus_counter includes:

- Internally keeping a usage count that increments even if the argument is NULL.
- A non-null value result when passed a NULL argument.

Because RESPECT NULL VALUES is the default, the results are the same if this clause is omitted from the declaration.

IQ restricts the usage of all nondeterministic functions. They are allowed only within the SELECT list of the top-level query block or in the SET clause of an UPDATE statement. They cannot be used within subqueries, or within a WHERE, ON, GROUP BY, or HAVING clause. This restriction applies to nondeterministic UDFs as well as to the nondeterministic built-in functions like GETUID and NUMBER.

The last detail in the above declaration is the DEFAULT qualifier on the input parameter. The qualifier tells the server that this function can be called with no arguments, and that when this happens the server automatically supplies a zero for the missing argument. If a DEFAULT value is specified, it must be implicitly convertible into the data type of that argument.

In the following example, the first SELECT list item adds the running counter to the value of t.x for each row. The second and third SELECT list items each return the same value for each row as the NUMBER function.

```
SELECT my_plus_counter(t.x),
     my_plus_counter(0),
     my_plus_counter(),
     NUMBER()
FROM t
```

Defining a scalar UDF

The C/C++ code for defining a scalar user-defined function includes four mandatory pieces.

- **extfnapiv3.h** – inclusion of the UDF interface definition header file.
- **_evaluate_extfn** – An evaluation function. All evaluation functions take two arguments:
  - an instance of the scalar UDF context structure that is unique to each usage of a UDF that contains a set of callback function pointers, and a pointer where a UDF can store UDF-specific data.
  - a pointer to a data structure that allows access to the argument values and to the result value through the supplied callbacks.
- **a_v3_extfn_scalar** – an instance of the scalar UDF descriptor structure that contains a pointer to the evaluation function.
- **Descriptor function** – returns a pointer to the scalar UDF descriptor structure.

These parts are optional:

- **_start_extfn** – an initialization function generally invoked once per SQL usage. If supplied, you must also place a pointer to this function into the scalar UDF descriptor structure. All initialization functions take one argument, a pointer to the scalar UDF context structure that is unique to each usage of a UDF. The context structure passed is the same one that is passed to the evaluation routine.
- **_finish_extfn** – a shutdown function generally invoked once per SQL usage. If supplied, a pointer to this function must also be placed into the scalar UDF descriptor structure. All shutdown functions take one argument, a pointer to the scalar UDF context structure that is unique to each usage of a UDF. The context structure passed is the same one that is passed to the evaluation routine.

Scalar UDF descriptor structure

The scalar UDF descriptor structure, a_v3_extfn_scalar, is defined as:

```
typedef struct a_v3_extfn_scalar {    //
    // Metadata descriptor for a scalar UDF
    // supplied by the UDF library to the server
    // An optional pointer to an initialize function
    void (*_start_extfn)(a_v3_extfn_scalar_context * cntxt);
    //
    // An optional pointer to a shutdown function
    void (*_finish_extfn)(a_v3_extfn_scalar_context *    cntxt);
    //
    // A required pointer to a function that will be
    // called for each invocation of the UDF on a
    // new set of argument values
    void (*_evaluate_extfn)(a_v3_extfn_scalar_context * cntxt, void
*args_handle);
    // RESERVED FIELDS MUST BE INITIALIZED TO NULL
 void  *reserved1_must_be_null;
```

```
 void  *reserved2_must_be_null;
 void  *reserved3_must_be_null;
 void  *reserved4_must_be_null;
 void  *reserved5_must_be_null;
                    …

} a_v3_extfn_scalar;
```

There should always be a single instance of **a_v3_extfn_scalar** for each defined scalar UDF. If the optional initialization function is not supplied, the corresponding value in the descriptor structure should be the null pointer. Similarly, if the shutdown function is not supplied, the corresponding value in the descriptor structure should be the null pointer.

The initialization function is called at least once before any calls to the evaluation routine, and the shutdown function is called at least once after the last evaluation call. The initialization and shutdown functions are normally called only once per usage.

Scalar UDF context structure

The scalar UDF context structure, **a_v3_extfn_scalar_context** that is passed to each of the functions specified within the scalar UDF descriptor structure, is defined as:

```
typedef struct a_v3_extfn_scalar_context {
//-------- Callbacks available via the context --------
//
  short (SQL_CALLBACK *get_value)(
    void           *arg_handle,
              a_sql_uint32    arg_num,
              an_extfn_value *value
              );
 short (SQL_CALLBACK *get_piece)(
              void *          arg_handle,
              a_sql_uint32    arg_num,
              an_extfn_value *value,
              a_sql_uint32    offset
              );
 short (SQL_CALLBACK *get_value_is_constant)(
              void *          arg_handle,
              a_sql_uint32    arg_num,
              a_sql_uint32 *  value_is_constant
              );
 short (SQL_CALLBACK *set_value)(
              void *          arg_handle,
              an_extfn_value *value,
              short              append
              );
 a_sql_uint32 (SQL_CALLBACK *get_is_cancelled)(
              a_v3_extfn_scalar_context * cntxt
              );
 short (SQL_CALLBACK *set_error)(
              a_v3_extfn_scalar_context * cntxt,
              a_sql_uint32    error_number,
              const char *    error_desc_string
              );
```

```
 void (SQL_CALLBACK *log_message)(
            const char *msg,
            short msg_length
               );
 short (SQL_CALLBACK *convert_value)(
               an_extfn_value   *input,
               an_extfn_value   *output
 //---------- Data available from the context ----------
 void * _user_data;   // read-write field
 //---------- For Server Internal Use Only -------------
 void  * _for_server_internal_use;
} a_v3_extfn_scalar_context;
```

The _user_data field within the scalar UDF context structure can be populated with data the UDF requires. Usually, it is filled in with a heap allocated structure by the _start_extfn function, and deallocated by the _finish_extfn function.

The rest of the scalar UDF context structure is filled with the set of callback functions, supplied by the engine, for use within each of the user's UDF functions. Most of these callback functions return a success status through a short result value; a true return indicates success. Well-written UDF implementations should never cause a failure status, but during development (and possibly in all debug builds of a given UDF library), Sybase recommends that you check that the return status values from the callbacks. Failures can come from coding errors within the UDF implementation, such as asking for more arguments than the UDF is defined to take.

The common set of arguments used by most of the callbacks includes:

- **arg_handle** – A pointer received by all forms of the evaluation methods, through which the values for input arguments passed to the UDF are available, and through which the UDF result value can be set.
- **arg_num** – An integer indicating which input argument is being accessed. Input arguments are numbered left to right in ascending order starting at one.
- **cntxt** – A pointer to the context structure that the server passes to all UDF entry points.
- **value** – A pointer to an instance of the an_extfn_value structure that is used to either get an input argument value from the server or to set the result value of the function. The an_extfn_value structure has this form:

  ```
  typedef struct an_extfn_value {
  void * data;
  a_SQL_uint32 piece_len;
  union {
          a_SQL_uint32 total_len;
          a_SQL_uint32 remain_len;
  } len;
  a_SQL_data_type type;
  } an_extfn_value;
  ```

UDF example: my_plus definition

The definition for the my_plus example.

my_plus definition

Because this UDF needs no initialization or shutdown function, those values within the descriptor structure are set to 0. The descriptor function name matches the EXTERNAL NAME used in the declaration. The evaluate method does not check the data type for arguments, because they are declared as INT.

```
#include "extfnapiv3.h"
#include <stdlib.h>


//  A simple deterministic scalar UDF that just adds
//  two integer arguments and then returns the result.
//
//  Corresponding SQL declaration:
//
//    CREATE FUNCTION my_plus(IN arg1 INT, IN arg2 INT)
//                                           RETURNS INT
//                                           DETERMINISTIC
//                                    IGNORE NULL VALUES
//                                           EXTERNAL NAME
'my_plus@libudfex'
//
#if defined __cplusplus
extern "C" {
#endif

static void my_plus_evaluate(a_v3_extfn_scalar_context *cntxt,
                      void *arg_handle)
{
  an_extfn_value  arg;
  an_extfn_value  outval;
  a_sql_int32 arg1, arg2, result;

  //  Get first argument
  (void) cntxt->get_value( arg_handle, 1, &arg );
  if (arg.data == NULL)
  {
      return;
  }
  arg1 = *((a_sql_int32 *)arg.data);

  //  Get second argument
  (void) cntxt->get_value( arg_handle, 2, &arg );
  if (arg.data == NULL)
  {
      return;
  }
  arg2 = *((a_sql_int32 *)arg.data);
```

```
  //  Set the result value
  outval.type = DT_INT;
  outval.piece_len = sizeof(a_sql_int32);
  result = arg1 + arg2;
  outval.data = &result;
  cntxt->set_value( arg_handle, &outval, 0 );
}


static a_v3_extfn_scalar my_plus_descriptor = {
    0,
    0,
    &my_plus_evaluate,
    0,            // Reserved - initialize to NULL
    0,            // Reserved - initialize to NULL
    0,            // Reserved - initialize to NULL
    0,            // Reserved - initialize to NULL
    0,            // Reserved - initialize to NULL
        NULL                    // _for_server_internal_use
};


a_v3_extfn_scalar *my_plus()
{
  return &my_plus_descriptor;
}

#if defined __cplusplus
}
#endif
```

UDF example: my_plus _counter definition

This example checks the argument value pointer data to see if the input argument value is NULL. It also has an initialization function and a shutdown function, each of which can tolerate multiple calls.

my_plus _counter definition

```
#include "extfnapiv3.h"
#include <stdlib.h>

//  A simple non-deterministic scalar UDF that adds
//  an internal integer usage counter to its integer
//  argument and then returns the resulting integer.
//
//  Here, the start function creates a little structure for
//  the counter, and then the finish function deallocates it.
//
//  Corresponding SQL declaration:
//
//          CREATE FUNCTION plus_counter(IN arg1 INT)
//                      RETURNS INT
//                      NOT DETERMINISTIC
//                      RESPECT NULL VALUES
//                      EXTERNAL NAME 'my_plus_counter@libudfex'
```

```
typedef struct my_counter {
  a_sql_int32 _counter;
} my_counter;

#if defined __cplusplus
extern "C" {
#endif

static void my_plus_counter_start(a_v3_extfn_scalar_context *cntxt)
{
  my_counter *cptr = (my_counter *)cntxt->_user_data;
  //  If we have not already allocated the
  //  counter structure, then do so now
  if (!cptr) {
    cptr = (my_counter *)malloc(sizeof(my_counter));
    cntxt->_user_data = cptr;
  }
  cptr->_counter = 0;
}


static void my_plus_counter_finish(a_v3_extfn_scalar_context *cntxt)
{
  //  If we still have an allocated the
  //  counter structure, then free it now
  if (cntxt->_user_data) {
    free(cntxt->_user_data);
    cntxt->_user_data = 0;
  }
}


static void my_plus_counter_evaluate(a_v3_extfn_scalar_context
*cntxt,
                                     void *arg_handle)
{
  an_extfn_value  arg;
  an_extfn_value  outval;
  a_sql_int32 arg1, result;

  //  Increment the usage counter
  my_counter *cptr = (my_counter *)cntxt->_user_data;
  cptr->_counter += 1;

  //  Get the one argument
  (void) cntxt->get_value( arg_handle, 1, &arg );
  if (!arg.data) {
    // argument value was NULL;
    arg1 = 0;
  } else {
    arg1 = *((a_sql_int32 *)arg.data);
  }

  outval.type = DT_INT;
```

```
  outval.piece_len = sizeof(a_sql_int32);
  result = arg1 + cptr->_counter;
  outval.data = &result;
  cntxt->set_value( arg_handle, &outval, 0 );
}


static a_v3_extfn_scalar my_plus_counter_descriptor =
    {   &my_plus_counter_start,
        &my_plus_counter_finish,
        &my_plus_counter_evaluate,
        NULL,                   // Reserved - initialize to NULL
        NULL,                   // Reserved - initialize to NULL
        NULL,                   // Reserved - initialize to NULL
        NULL,                   // Reserved - initialize to NULL
        NULL,                   // Reserved - initialize to NULL
        NULL,                   // _for_server_internal_use
};


a_v3_extfn_scalar *my_plus_counter()
{
  return &my_plus_counter_descriptor;
}

#if defined __cplusplus
}
#endif
```

User-defined aggregate functions

Sybase IQ supports user-defined aggregate functions (UDAFs). The SUM function is an example of a built-in aggregate function. A simple aggregate function produces a single result value from a set of argument values. You can write UDAFs that can be used anywhere the SUM aggregate can be used.

Note: The aggregate UDF examples referenced here are installed with the IQ server, and can be found as .cxx files in `$IQDIR15/samples/udf`. You can also find them in the `$IQDIR15/lib64/libudfex` dynamically linkable library.

Declaring a UDAF

Aggregate UDAFs are more powerful and more complex to create than scalar UDFs.

After the UDF code has been written and compiled, create a SQL function that invokes the UDF from the appropriate library file, sending the input data to the UDF.

Note: You can also *create the user-defined function declaration in Sybase Central* on page 12.

When implementing a UDAF, you must decide:

- Whether it will operate only across an entire data set or partition as an online analytical processing (OLAP) -style aggregate, like RANK.
- Whether it will operate as either a simple aggregate or an OLAP-style aggregate, like SUM.
- Whether it will operate only as a simple aggregate over an entire group.

The declaration and the definition of a UDAF reflects these usage decisions.

The syntax for creating IQ user-defined aggregate functions is:

```
aggregate-udf-declaration:
    CREATE AGGREGATE FUNCTION [ owner.]function-name
          ( [ parameter , ... ] )
    RETURNS data-type
        [ aggregate-routine-characteristics ... ]
    EXTERNAL NAME library-and-entry-point-name-string

parameter:
    param-name data-type [ DEFAULT value ]

aggregate-routine-characteristics:
      DUPLICATE { SENSITIVE | INSENSITIVE }
        -- is the server allowed to eliminate DISTINCT
    | SQL SECURITY {INVOKER | DEFINER}
    | OVER restrict
```

```
    | ORDER order-restrict
        -- Must the window-spec contain an ORDER BY?
    | WINDOW FRAME
        { { ALLOWED | REQUIRED }
            [ window-frame-constraints ... ]
          | NOT ALLOWED }
    | ON EMPTY INPUT RETURNS { NULL | VALUE }
  -- Call or skip function on NULL inputs

window-frame-constraints:
      VALUES { [ NOT ] ALLOWED }
    | CURRENT ROW { REQUIRED | ALLOWED }
    | [ UNBOUNDED ] { PRECEDING | FOLLOWING } restrict

restrict:   { [ NOT ] ALLOWED } | REQUIRED

order-restrict:
{ NOT ALLOWED | SENSITIVE | INSENSITIVE | REQUIRED
```

The handling of the return data type, arguments, data types, and default values are identical to that in the scalar UDF definition.

If a UDAF can be used as a simple aggregate, then it can potentially be used with the DISTINCT qualifier. The DUPLICATE clause in the UDAF declaration determines:

- Whether duplicate values can be considered for elimination before the UDAF is called because the results are sensitive to duplicates (such as for the built-in "COUNT(DISTINCT T.A)") or,
- Whether the results are insensitive to the presence of duplicates (such as for "MAX(DISTINCT T.A)").

The DUPLICATE INSENSITIVE option allows the optimizer to consider removing the duplicates without affecting the result, giving the optimizer the choice on how to execute the query. Write the UDAF to expect duplicates. If duplicate elimination is required, the server performs it before starting the set of _next_value_extfn calls.

Most of the remaining clauses that are not part of the scalar UDF syntax allow you to specify the usages for this function. By default, a UDAF is assumed to be usable as both a simple aggregate and as an OLAP-style aggregate with any kind of window frame.

For a UDAF to be used only as a simple aggregate function, declare it using:

```
OVER NOT ALLOWED
```

Any attempt to then use this aggregate as an OLAP-style aggregate generates an error.

For UDAFs that allow or require an OVER clause, the UDF definer can specify restrictions on the presence of the ORDER BY clause within the OVER clause by specifying "ORDER" followed by the restriction type. Window-ordering restriction types:

- **REQUIRED** – ORDER BY must be specified and cannot be eliminated.

- **SENSITIVE** – ORDER BY may or may not be specified, but cannot be eliminated when specified.
- **INSENSITIVE** – ORDER BY may or may not be specified, but the server can do ordering elimination for efficiency.
- **NOT ALLOWED** – ORDER BY cannot be specified.

Declare a UDAF that makes sense only as an OLAP-style aggregate over an entire set or partition that has been ordered, like the built-in RANK, with:

```
OVER REQUIRED
ORDER REQUIRED
WINDOW FRAME NOT ALLOWED
```

Declare a UDAF that makes sense only as an OLAP-style aggregate using the default window frame of UNBOUNDED PRECEDING to CURRENT ROW, with:

```
OVER REQUIRED
ORDER REQUIRED
WINDOW FRAME ALLOWED
    RANGE NOT ALLOWED
    UNBOUNDED PRECEDING REQUIRED
    CURRENT ROW REQUIRED
    FOLLOWING NOT ALLOWED
```

The defaults for the all various options and restriction sets are:

```
DUPLICATE SENSITIVE
SQL SECURITY DEFINER
OVER ALLOWED
ORDER SENSITIVE
WINDOW FRAME ALLOWED
CURRENT ROW ALLOWED
PRECEDING ALLOWED
UNBOUNDED PRECEDING ALLOWED
FOLLOWING ALLOWED
UNBOUNDED FOLLOWING ALLOWED
```

SQL Security

Defines whether the function is executed as the INVOKER, (the user who is calling the function), or as the DEFINER (the user who owns the function). The default is DEFINER.

When **SQL SECURITY INVOKER** is specified, more memory is used because each user that calls the procedure requires annotation. Also, when **SQL SECURITY INVOKER** is specified, name resolution is performed on both the user name and the INVOKER. Qualify all object names (tables, procedures, and so on) with their appropriate owner.

External Name

A function using the **EXTERNAL NAME** clause is a wrapper around a call to a function in an external library. A function using **EXTERNAL NAME** can have no other clauses following the **RETURNS** clause. The library name may include the file extension, which is typically `.dll` on Windows and `.so` on UNIX. In the absence of the extension, the software appends the platform-specific default file extension for libraries.

The **EXTERNAL NAME** clause is not supported for temporary functions. See “Calling external libraries from procedures” in *SQL Anywhere Server – Programming*.

The IQ server can be started with a library load path that includes the location of the UDF library. On Unix variants, this can be done by modifying the LD_LIBRARY_PATH within the start_iq startup script. While LD_LIBRARY_PATH is universal to all UNIX variants, SHLIB_PATH is preferred on HP, and LIB_PATH is preferred on AIX.

On Unix platforms, the external name specification can contain a fully qualified name, in which case the LD_LIBRARY_PATH is not used. On the Windows platform, a fully qualified name cannot be used and the library search path is defined by the PATH environment variable.

Note: Scalar user-defined functions and user-defined aggregate functions are not supported in updatable cursors.

UDAF example: my_sum declaration

The “my_sum” example is similar to the built-in SUM, except it only operates on integers.

my_sum declaration

Since my_sum, like SUM, can be used in any context, it has a relatively brief declaration:

```
CREATE AGGREGATE FUNCTION my_sum(IN arg1 INT)
    RETURNS BIGINT
    ON EMPTY INPUT RETURNS NULL
    EXTERNAL NAME 'describe_my_integer_sum@my_shared_lib'
```

The various usage restrictions all default to ALLOWED to specify that this function can be used anywhere in a SQL statement that any aggregate function is allowed.

Without any usage restrictions, my_sum is usable as a simple aggregate across an entire set of rows, as shown here:

```
SELECT MIN(t.x), COUNT (*), my_sum(t.y)
FROM t
```

Without usage restrictions, my_sum is also usable as a simple aggregate computed for each group as specified by a GROUP BY clause:

```
SELECT t.x, COUNT(*), my_sum(t.y)
FROM t
GROUP BY t.x
```

Because of the lack of usage restrictions, my_sum is usable as an OLAP-style aggregate with an OVER clause, as shown in this cumulative summation example:

```
SELECT t.x,
    my_sum(t.x)
        OVER (ORDER BY t.x ROWS BETWEEN UNBOUNDED PRECEDING AND
CURRENT ROW)
        AS cumulative_x,
    COUNT(*)
FROM t
```

```
GROUP BY t.x
ORDER BY t.x
```

UDAF example: my_bit_xor declaration

The “my_bit_xor” example is analogous to the SQL Anywhere (SA) built-in BIT_XOR, except it operates only on unsigned integers.

my_bit_xor declaration

The resulting declaration is:

```
CREATE AGGREGATE FUNCTION my_bit_xor(IN arg1 UNSIGNED    INT)
 RETURNS UNSIGNED INT
    ON EMPTY INPUT RETURNS NULL
    EXTERNAL NAME 'describe_my_bit_xor@my_shared_lib'
```

Like the my_sum example, my_bit_xor has no associated usage restrictions, and is therefore usable as a simple aggregate or as an OLAP-style aggregate with any kind of a window.

UDAF example: my_bit_or declaration

The “my_bit_or” example is similar to the SA built-in BIT_OR except it operates only on unsigned integers, and it can be used only as a simple aggregate.

my_bit_or declaration

The resulting declaration looks like:

```
CREATE AGGREGATE FUNCTION my_bit_or(IN arg1 UNSIGNED INT)
     RETURNS UNSIGNED INT
     ON EMPTY INPUT RETURNS NULL
     OVER NOT ALLOWED
     EXTERNAL NAME 'describe_my_bit_or@ my_shared_lib'
```

Unlike the my_bit_xor example, the OVER NOT ALLOWED phrase in the declaration restricts the use of this function to a simple aggregate. Because of that usage restriction, my_bit_or is only usable as a simple aggregate across an entire set of rows, or as a simple aggregate computed for each group as specified by a GROUP BY clause shown in the following example:

```
SELECT t.x, COUNT(*), my_bit_or(t.y)
FROM t
GROUP BY t.x
```

UDAF example: my_interpolate declaration

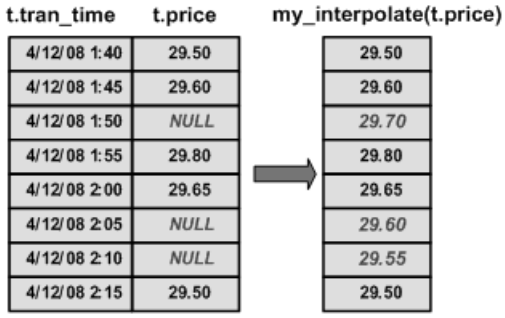
The “my_interpolate” example is an OLAP-style UDAF that attempts to fill in any missing values in a sequence (where missing values are denoted by NULLs) by performing linear

interpolation across any set of adjacent NULL values to the nearest non-NULL value in each direction.

my_interpolate declaration

If the input at a given row is not NULL, the result for that row is the same as the input value.

Figure 1: my_interpolate results



To operate at a sensible cost, my_interpolate must run using a fixed-width, row-based window, but the user can set the width of the window based on the maximum number of adjacent NULL values he or she expects to see. This function takes a set of double-precision floating point values and produces a resulting set of doubles.

The resulting UDAF declaration looks like this:

```
CREATE AGGREGATE FUNCTION my_interpolate (IN arg1 DOUBLE)
RETURNS DOUBLE
    OVER REQUIRED
    WINDOW FRAME REQUIRED
        RANGE NOT ALLOWED
        PRECEDING REQUIRED
        UNBOUNDED PRECEDING NOT ALLOWED
        FOLLOWING REQUIRED
        UNBOUNDED FOLLOWING NOT ALLOWED
    EXTERNAL NAME 'describe_my_interpolate@my_shared_lib'
```

OVER REQUIRED means that this function cannot be used as a simple aggregate (ON EMPTY INPUT, if used, is irrelevant).

WINDOW FRAME details specify that you must use a fixed-width, row-based window that extends both forward and backward from the current row when using this function. Because of these usage restrictions, my_interpolate is usable as an OLAP-style aggregate with an OVER clause similar to:

```
SELECT t.x,
    my_interpolate(t.x)
    OVER (ORDER BY t.x ROWS BETWEEN 5 PRECEDING AND 5 FOLLOWING)
        AS x_with_gaps_filled,
```

```
    COUNT(*)
FROM t
GROUP BY t.x
ORDER BY t.x
```

Within an OVER clause for my_interpolate, the precise number of preceding and following rows may vary, and optionally, you can use a PARTITION BY clause; otherwise, the rows must be similar to the example above given the usage restrictions in the declaration.

Defining an aggregate UDF

The C/C++ code for defining an aggregate user-defined function includes eight mandatory pieces.

- **extfnapiv3.h** – the UDF interface definition header file.
- **_start_extfn** – an initialization function invoked once per SQL usage. All initialization functions take one argument: a pointer to the aggregate UDF context structure that is unique to each usage of a UDAF. The context structure passed is the same one that is passed to all the supplied functions for that usage.
- **_finish_extfn** – a shutdown function invoked once per SQL usage. All shutdown functions take one argument: a pointer to the UDAF context structure that is unique to each usage of a UDAF.
- **_reset_extfn** – a reset function called once at the start of each new group, new partition, and if necessary, at the start of each window motion. All reset functions take one argument: a pointer to the UDAF context structure that is unique to each usage of a UDAF.
- **_next_value_extfn** – a function called for each new set of input arguments. _next_value_extfn takes two arguments:
  - A pointer to the UDAF context, and
  - An args_handle.

  As in scalar UDFs, the arg_handle is used with the supplied callback function pointers to access the actual argument values.
- **_evaluate_extfn** – an evaluation function similar to the scalar UDF evaluation function. All evaluation functions take two arguments:
  - A pointer to the UDAF context structure, and
  - An args_handle.
- **a_v3_extfn_aggregate** – an instance of the aggregate UDF descriptor structure that contains the pointers to all of the supplied functions for this UDF.
- **Descriptor function** – a descriptor function that returns a pointer to that aggregate UDF descriptor structure.

In addition to the mandatory pieces, there are several optional pieces that enable more optimized access for specific usage situations:

- **_drop_value_extfn** – an optional function pointer that is called for each input set of argument values that has fallen out of a moving window frame. This function should not

set the result of the aggregation. Use the get_value callback function to access the input argument values, and, if necessary, through repeated calls to the get_piece callback function.

Set the function pointer to the null pointer if:

- This aggregate cannot be used with a window frame,
- The aggregate is not reversible in some way, or
- The user is not interested in optimal performance.

If _drop_value_extfn is not supplied and the user has specified a moving window, each time the window frame moves, the reset function is called and each row within the window is included by a call to the next_value function, and finally the evaluate function is called.

If _drop_value_extfn is supplied, then each time the window frame moves, this drop value function is called for each row falling out of the window frame, then the next_value function is called for each row that has just been added into the window frame, and finally the evaluate function is called to produce the aggregate result.

- **_evaluate_cumulative_extfn** – an optional function pointer that may be called for each new input set of argument values. If this function is supplied, and the usage is in a row-based window frame that spans UNBOUNDED PRECEDING to CURRENT ROW, then this function is called instead of calling the next value function immediately followed by calling the evaluate function.

  _evalutate_cumulative_extfn must set the result of the aggregation through the set_value callback. Access to its set of input argument values is through the usual get_value callback function. This function pointer should be set to the null pointer if:

  - This aggregate will never be used is this manner, or
  - The user is not worried about optimal performance.
- **_next_subaggregate_extfn** – an optional callback function pointer that works together with an _evaluate_superaggregate_extfn to enable some usages of this aggregate to be optimized by running in parallel.

  Some aggregates, when used as simple aggregates (in other words, not OLAP-style aggregates with an OVER clause) can be partitioned by first producing a set of intermediate aggregate results where each intermediate result is computed from a disjointed subset of the input rows.

  Examples of such partitionable aggregates include:

  - SUM, where the final SUM can be computed by performing a SUM for each disjointed subset of the input rows and then performing a SUM over the sub-SUMs; and
  - COUNT(*), where the final COUNT can be computed by performing a COUNT for each disjoint subset of the input rows and then performing a SUM over the COUNTs from each partition.

  When an aggregate satisfies the above conditions, the server may choose to make the computation of that aggregate parallel. For aggregate UDFs, this parallel optimization can be applied only if both the _next_subaggregate_extfn function pointer and the _evaluate_superaggregate_extfn pointer are supplied.

The _reset_extfn function does not set the final result of the aggregation, and by definition, has exactly one input argument value that is the same data type as the defined return value of the aggregate UDF.

Access to the subaggregate input value is through the normal get_value callback function. Direct communication between subaggregates and the superaggregate is impossible; the server handles all such communication. The sub-aggregates and the super-aggregate do not share a context structure. Instead, individual sub-aggregates are treated exactly the same as nonpartitioned aggregates. The independent super-aggregate sees a calling pattern that looks like this:

```
_start_extfn
_reset_extfn
_next_subaggregate_extfn (repeated 0 to N times)
_evaluate_superaggregate_extfn
_finish_extfn
```

Or like this:

```
_start_extfn
_reset_extfn
_next_subaggregate_extfn (repeated 0 to N times)
_evaluate_superaggregate_extfn
_reset_extfn
_next_subaggregate_extfn (repeated 0 to N times)
_evaluate_superaggregate_extfn
_reset_extfn
_next_subaggregate_extfn (repeated 0 to N times)
_evaluate_superaggregate_extfn
_finish_extfn
```

If neither _evaluate_superaggregate_extfn or _next_subaggregate_extfn is supplied, then the UDAF is restricted, and not allowed as a simple aggregate within a query block containing GROUP BY CUBE or GROUP BY ROLLUP.

- **_evaluate_superaggregate_extfn** – the optional callback function pointer that works with the _next_subaggregate_extfn to enable some usages as a simple aggregate to be optimized through parallelization. _evaluate_superaggregate_extfn is called to return the result of a partitioned aggregate. The result value is sent to the server using the normal set_value callback function from the a_v3_extfn_aggregate_context structure.

Aggregate UDF descriptor structure

The aggregate UDF descriptor structure comprises serveral pieces:

- **typedef struct a_v3_extfn_aggregate** – the metadata descriptor for an aggregate UDF function supplied by the library.
- **_start_extfn** – required pointer to an initialization function for which the only argument is a pointer to a_v3_extfn_aggregate_context. Typically, used to allocate some structure and store its address in the _user_data field within the a_v3_extfn_aggregate_context. _start_extfn is only ever called once per a_v3_extfn_aggregate_context.

  ```
  void (*_start_extfn)(a_v3_extfn_aggregate_context *);
  ```

- **_finish_extfn** – required pointer to a shutdown function for which the only argument is a pointer to a_v3_extfn_aggregate_context. Typically, used to deallocate some structure with the address stored within the _user_data field in the a_v3_extfn_aggregate_context. _finish_extfn is only ever called once per a_v3_extfn_aggregate_context.

  ```
  void (*_finish_extfn)(a_v3_extfn_aggregate_context *);
  ```
- **_reset_extfn** – required pointer to a start-of-new-group function, for which the only argument is a pointer to a_v3_extfn_aggregate_context. Typically, used to reset some values in the structure for which the address was stashed within the _user_data field in the a_v3_extfn_aggregate_context. _reset_extfn is called repeatedly.

  ```
  void (*_reset_extfn)(a_v3_extfn_aggregate_context *);
  ```
- **_next_value_extfn** – required function pointer to be called for each new input set of argument values. The function does not set the result of the aggregation. Access to input argument values are through the get_value callback function and, if necessary, through repeated calls to the get_piece callback function, which is required only if piece_len is less than total_len.

  ```
  void (*_next_value_extfn)(a_v3_extfn_aggregate_context *cntxt,
  void *args_handle);
  ```
- **_evaluate_extfn** – required function pointer to be called to return the resulting aggregate result value. _evaluate_extfn is sent to the server using the set_value callback function.

  ```
  void (*_evaluate_extfn)(a_v3_extfn_aggregate_context *cntxt, void
  *args_handle);
  ```
- **_drop_value_extfn** – Optional function pointer that is called for each input set of argument values that has fallen out of a moving window frame. Do not use this function to set the result of the aggregation. Access to input argument values are through the get_value callback function and, if necessary, through repeated calls to the get_piece callback function; however, access is required only if piece_len is less than total_len. Set _drop_value_extfn to the null pointer if:
  - The aggregate cannot be used with a window frame.
  - The aggregate is not reversible in some way.
  - The user is not interested in optimal performance.

  If this function is not supplied, and the user has specified a moving window, then each time the window frame moves, the reset function is called and each row now within the window is included by a call to the next_value function. Finally, the evaluate function is called. However, if this function is supplied, each time the window frame moves, this drop_value function is called for each row falling out of the window frame, then the next_value function is called for each row that has just been added into the window frame. Finally, the evaluate function is called to produce the aggregate result.

  ```
  void (*_drop_value_extfn)(a_v3_extfn_aggregate_context    *cntxt,
  void *args_handle);
  ```
- **_evaluate_cumulative_extfn** – optional function pointer to be called for each new input set of argument values. If this function is supplied, and the usage is in a row-based window frame that spans UNBOUNDED PRECEDING to CURRENT ROW, then this function is called instead of next_value, immediately followed by calling evaluate. _evaluate_cumulative_extfn must set the result of the aggregation through the set_value

callback. Access to input argument values are through the get_value callback function and, if necessary, through repeated calls to the get_piece callback function, which is only required if piece_len is less than total_len.

```
void (*_evaluate_cumulative_extfn)(a_v3_extfn_aggregate_context
*cntxt, void *args_handle);
```

- **_next_subaggregate_extfn** – optional callback function pointer that, with the _evaluate_superaggregate_extfn function (and in some usages also with the _drop_subaggregate_extfn function), enables some usages of the aggregate to be optimized through parallel and partial results aggregation.

  Some aggregates, when used as simple aggregates (in other words, not OLAP-style aggregates with an OVER clause) can be partitioned by first producing a set of intermediate aggregate results where each of the intermediate results is computed from a disjoint subset of the input rows. Examples of such partitionable aggregates include:

  - SUM, where the final SUM can be computed by performing a SUM for each disjoint subset of the input rows and then performing a SUM over the sub-SUMs; and
  - COUNT(*), where the final COUNT can be computed by performing a COUNT for each disjoint subset of the input rows and then performing a SUM over the COUNTs from each partition.

  When an aggregate satisfies the above conditions, the server may choose to make the computation of that aggregate parallel. For aggregate UDFs, this optimization can be applied only if both the _next_subaggregate_extfn callback and the _evaluate_superaggregate_extfn callback are supplied. This usage pattern does not require _drop_subaggregate_extfn.

  Similarly, if an aggregate can be used with a RANGE-based OVER clause, an optimization can be applied if _next_subaggregate_extfn, _drop_subaggregate_extfn, and _evaluate_superaggregate_extfn) functions are all supplied by the UDAF implementation.

  _next_subaggregate_extfn does not set the final result of the aggregation, and by definition, has exactly one input argument value that is the same data type as the return value of the aggregate UDF. Access to the sub-aggregate input value is through the get_value callback function and, if necessary, through repeated calls to the get_piece callback function, which is required only if piece_len is less than total_len.

  Direct communication between sub-aggregates and the super-aggregate is impossible; the server handles all such communication. The sub-aggregates and the super-aggregate do not share the context structure. Individual sub-aggregates are treated exactly the same as nonpartitioned aggregates. The independent super-aggregate sees a calling pattern that looks like this:

```
      _start_extfn
 _reset_extfn
      _next_subaggregate_extfn (repeated 0 to N times)
      _evaluate_superaggregate_extfn
      _finish_extfn
```

```
void (*_next_subaggregate_extfn)(a_v3_extfn_aggregate_context
*cntxt, void *args_handle);
```

- **_drop_subaggregate_extfn** – optional callback function pointer that, together with _next_subaggregate_extfn and _evaluate_superaggregate_extfn, enables some usages involving RANGE-based OVER clauses to be optimized through a partial aggregation. _drop_subaggregate_extfn is called whenever a set of rows sharing a common ordering key value have collectively fallen out of a moving window. This optimization is applied only if all three functions are provided by the UDF.

  ```
  void (*_drop_subaggregate_extfn)(a_v3_extfn_aggregate_context
  *cntxt,    void *args_handle);
  ```

- **_evaluate_superaggregate_extfn** – optional callback function pointer that, together with _next_subaggregate_extfn (and in some cases also with _drop_subaggregate_extfn), enables some usages to be optimized by running in parallel.

  _evaluate_superaggregate_extfn is called, as described above, when it is time to return the result of a partitioned aggregate. The result value is sent to the server using the set_value callback function from the a_v3_extfn_aggregate_context structure:

  ```
  void (*_evaluate_superaggregate_extfn)
  (a_v3_extfn_aggregate_context *cntxt, void *args_handle);
  ```

- **NULL fields** – initialize these fields to NULL:

  ```
  void *    reserved1_must_be_null;
  void *    reserved2_must_be_null;
  void *    reserved3_must_be_null;
  void *    reserved4_must_be_null;
  void *    reserved5_must_be_null;
  ```

- **Status indicator bit field** – a bit field containing indicators that allow the engine to optimize the algorithm used to process the aggregate.

  ```
  a_sql_uint32  indicators;
  ```

- **_calculation_context_size** – the number of bytes for the server to allocate for each UDF calculation context. The server may allocate multiple calculation contexts during query processing. The currently active group context is available in a_v3_extfn_aggregate_context_user_calculation_context.

  ```
  short _calculation_context_size;
  ```

- **_calculation_context_alignment** – specifies the alignment requirement for the user's calculation context. Valid values include 1, 2, 4, or 8.

  ```
  short _calculation_context_alignment;
  ```

- **External memory requirments** – the following fields allow the optimizer to consider the cost of externally allocated memory. With these values, the optimizer can consider the degree to which multiple simultaneous calculations can be made. These counters should be estimates based on a typical row or group, and should not be maximum values. If no memory is allocated by the UDF, set these fields to zero.
  - external_bytes_per_group – The amount of memory allocated to a group at the start of each aggregate. Typically, any memory allocated during the reset() call.
  - external_bytes_per_row – The amount of memory allocated by the UDF for each row of a group. Typically, the amount of memory allocated during next_value().

    ```
    double            external_bytes_per_group;
    double            external_bytes_per_row;
    ```

- **Reserved fields for future use** – initialize these fields:

```
a_sql_uint64      reserved6_must_be_null;
a_sql_uint64      reserved7_must_be_null;
a_sql_uint64      reserved8_must_be_null;
a_sql_uint64      reserved9_must_be_null;
a_sql_uint64      reserved10_must_be_null;
```

- **Closing syntax** – Complete the descriptor with this syntax:

```
//---------- For Server Internal Use Only ----------
void  * _for_server_internal_use;
} a_extfn_aggregate;
```

Calculation context

The _user_calculation_context field allows the server to concurrently execute calculations on multiple groups of data.

A UDAF must keep intermediate counters for calculations as it is processing rows. The simple model for managing these counters is to allocate memory at the start API function, store a pointer to it in the aggregate context's _user_data field, then release the memory at the aggregate's finish API. An alternative method, based on the _user_calculation_context field, allows the server to concurrently execute calculations on multiple groups of data.

The _user_calculation_context field is a server-allocated memory pointer, created by the server for each concurrent processing group. The server ensures that the _user_calculation_context always points to the correct calculation context for the group of rows currently being processed. Between UDF API calls, depending on the data, the server may allocate new _user_calculation_context values. The server may save and restore calculation context areas to disk while processing a query.

The UDF stores all intermediate calculation values in this field. This illustrates a typical usage:

```
struct my_average_context
{
        int    sum;
        int    count;
};

reset(a_v3_aggregate_context *context)
{
    mycontext  = (my_average_context *) context-
>_user_calculation_context;
    mycontext->count = 0;
    mycontext->sum = 0;
}

    next_value(a_v3_aggregate_context *context, void *args_handle)
{
    mycontext  = (my_average_context *) context-
>_user_calculation_context;
    mycontext->count++;
```

```
    ..
}
```

In this model, the _user_data field can still be used, but no values relating to intermediate result calculations can be stored there. The _user_calculation_context is NULL at both the start and finish entry points.

To use the _user_calculation_context to enable concurrent processing, the UDF must specify the size and alignment requirements for its calculation context, and define a structure to hold its values and set a_v3_extfn_aggregate and _calculation_context_size to the sizeof() of that structure.

The UDF must also specify the data alignment requirements of _user_calculation_context through _calculation_context_alignment. If user_calculation_context memory contains only a character byte array, no particular alignment is necessary, and you can specify an alignment of 1. Likewise, double floating point values might require an 8-byte alignment. Alignment requirements vary by platform and data type. Specifying a larger alignment than necessary always works; however, using the smallest alignment uses memory more efficiently.

UDAF context structure

The aggregate UDF context structure, a_v3_extfn_aggregate_context, has exactly the same set of callback function pointers as the scalar UDF context structure.

In addition, it has a read/write _user_data pointer just like the scalar UDF context, and a set of read-only data fields that describe the current usage and location. Each unique instance of the UDF within a statement has one aggregate UDF context instance that is passed to each of the functions specified within the aggregate UDF descriptor structure when they are called. The aggregate context structure is defined as:

- **typedef struct a_v3_extfn_aggregate_context** – One created for each instance of an external function referenced within a query. If used within a parallelized subtree within a query, there is a separate context for parallel subtree.
- **Callbacks available via the context** – Common arguments to the callback routines include:
  - **arg_handle** – A handle to function instance and arguments provided by the server.
  - **arg_num** – The argument number. Return values are 0..N.
  - **data** – The pointer to argument data.

  The context must call get_value before get_piece, but needs to call get_piece only if piece_len is less than total_len.

```
short (SQL_CALLBACK *get_value)(
  void *          arg_handle,
        a_sql_uint32    arg_num,
        an_extfn_value *value
        );
short (SQL_CALLBACK *get_piece)(
        void *          arg_handle,
        a_sql_uint32    arg_num,
        an_extfn_value *value,
```

```
        a_sql_uint32    offset
        );
```

- **Determining whether an argument is a constant** – The UDF can ask whether a given argument is a constant. This can be useful, for example, to allow work to be done once at the first call to the _next_value function rather than for every call to the _next_value function.

```
short (SQL_CALLBACK *get_value_is_constant)(
  void *          arg_handle,
        a_sql_uint32    arg_num,
        a_sql_uint32 *  value_is_constant
        );
```

- **Returning a null value** – To return a null value, set "data" to NULL in an_extfn_value. The total_len field is ignored on calls to set_value, the data supplied becomes the value of the argument if append is FALSE; otherwise, the data is appended to the current value of the argument. It is expected that set_value is called with append=FALSE for an argument before being called with append=TRUE for the same argument. The append field is ignored for fixed-length data types (in other words, all numeric data types).

```
short (SQL_CALLBACK *set_value)(
  void *          arg_handle,
        an_extfn_value *value,
        short              append
        );
```

- **Determining whether the statement was interrupted** – If a UDF entry point performs work for an extended period of time (many seconds), then it should, if possible, call the get_is_cancelled callback every second or two to see if the user has interrupted the current statement. If the statement has been interrupted, a nonzero value is returned, and the UDF entry point should then immediately perform. Eventually, the _finish_extfn function is called to do any necessary cleanup, but no other UDF entry points are subsequently called.

```
 a_sql_uint32 (SQL_CALLBACK *get_is_cancelled)
(a_v3_extfn_aggregate_context * cntxt);
```

- **Sending error messages** – If a UDF entry point encounters some error that should result in an error message being sent back to the user and the current statement being shut down, the set_error callback routine should be called. set_error causes the current statement to roll back; the user sees `Error from external UDF: <error_desc_string>`, and the SQLCODE is the negated form of <error_number>. After a call to set_error, the UDF entry point immediately performs a return. Eventually, _finish_extfn is called to perform any necessary cleanup, but no other UDF entry points are subsequently called.

```
void (SQL_CALLBACK *set_error)(
  a_v3_extfn_aggregate_context * cntxt,
        a_sql_uint32    error_number,
        // use error_number values >17000 & <100000
        const char *    error_desc_string
        );
```

- **Writing messages to the message log** – Messages longer than 255 bytes may be truncated.

```
void (SQL_CALLBACK *log_message)(
  const char *msg,
```

```
        short msg_length
  );
```

- **Converting one data type to another** – for input:
  - **an_extfn_value.data** – input data pointer.
  - **an_extfn_value.total_len** – length of input data.
  - **an_extfn_value.type** – DT_ datatype of input.

  For output:
  - **an_extfn_value.data** – UDF-supplied output data pointer.
  - **an_extfn_value.piece_len** – maximum length of output data.
  - **an_extfn_value.total_len** – server set length of converted output.
  - **an_extfn_value.type** – DT_ datatype of desired output.

```
short (SQL_CALLBACK *convert_value)(
  an_extfn_value   *input,
        an_extfn_value   *output
        );
```

- **Fields reserved for future use** – These are reserved for future use:

```
void  *   reserved1;
void  *   reserved2;
void  *   reserved3;
void  *   reserved4;
void  *   reserved5;
```

- **Data available from the context** – This data pointer can be filled in by any usage with any context data the external routine requires. The UDF allocates and deallocates this memory. A single instance of _user_data is active for each statement. Do not use this memory for intermediate result values.

```
void  * _user_data;
```

- **Currently active calculation context** – UDFs should use this memory location to store intermediate values that calculate the aggregate. This memory is allocated by the server based on the size requested in the a_v3_extfn_aggregate. Intermediate calculations must be stored in this memory, since the engine may perform simultaneous calculations over more than one group. Before each UDF entry point, the server ensures that the correct context data is active.

```
void  * _user_calculation_context;
```

- **Other available aggregate information** – Available at all external function entry points, including start_extfn. Zero indicates an unknown or not-applicable value. Estimated average number of rows per partition or group.
  - **a_sql_uint64 _max_rows_in_frame;** – Calculates the maximum number of rows defined in the window frame. For range-based windows, this indicates unique values. Zero indicates an unknown or not-applicable value.
  - **a_sql_uint64 _estimated_rows_per_partition;** – Displays the estimated average number of rows per partition or group. 0 indicates an unknown or not-applicable value.

    - **a_sql_uint32 _is_used_as_a_superaggregate;** – Identifies whether this instance is a normal aggregate or a superaggregate. Returns a result of 0 if the instance is a normal aggregate.
- **Determining window specifications** – Window specifications if a window is present on the query:
    - **a_sql_uint32 _is_window_used;** – Determines if the statement is windowed.
    - **a_sql_uint32 _window_has_unbounded_preceding;** – A return value of 0 indicates the window does not have unbounded preceding.
    - **a_sql_uint32 _window_contains_current_row;** – A return value of 0 indicates the window does not contain the current row.
    - **a_sql_uint32 _window_is_range_based;** – If the return code is 1, the window is range-based. If the return code is 0, the window is row-based.
- **Available at reset_extfn() calls** – Returns the actual number of rows in current partition, or 0 for nonwindowed aggregate.

  ```
  a_sql_uint64 _num_rows_in_partition;
  ```
- **Available only at evaluate_extfn() calls for windowed aggregates** – Currently evaluated row number in partition (starting with 1). This is useful during the evaluation phase of unbounded windows.

  ```
  a_sql_uint64 _result_row_from_start_of_partition;
  ```
- **Closing syntax** – Complete the context with:

  ```
  //---------- For Server Internal Use Only ----------
    void  * _for_server_internal_use;
  } a_v3_extfn_aggregate_context;
  ```

UDAF example: my_sum definition

The "my_sum" example operates only on integers.

my_sum definition

Since my_sum, like SUM, can be used in any context, all the optimized optional entry points have been supplied. In this example, the normal _evaluate_extfn function can also be used as the _evaluate_superaggregate_extfn function.

```
#include "extfnapiv3.h"
#include <stdlib.h>
#include <assert.h>


//  Simple aggregate UDF that adds up a set of
//  integer arguments, and whenever asked returns
//  the resulting big integer total.  For int
//  arguments, the only difference between this
//  UDF and the SUM built-in aggregate is that this
//  UDF will return NULL if there are no input rows.
//
//  The start function creates a little structure for
//  the running total, and the finish function then
//  deallocates it.
```

```
//
//  Since there are no aggregate usage restrictions
//  for this UDAF, the corresponding SQL declaration
//  will look like:
//
//              CREATE AGGREGATE FUNCTION my_sum(IN arg1 INT)
//                      RETURNS BIGINT
//                      ON EMPTY INPUT RETURNS NULL
//                      EXTERNAL NAME 'my_integer_sum@libudfex'


typedef struct my_total {
  a_sql_int64   _total;
  a_sql_uint64  _num_nonnulls_seen;
} my_total;


extern "C"
void my_integer_sum_start(a_v3_extfn_aggregate_context *cntxt)
{
}


extern "C"
void my_integer_sum_finish(a_v3_extfn_aggregate_context *cntxt)
{
}


extern "C"
void my_integer_sum_reset(a_v3_extfn_aggregate_context *cntxt)
{
  my_total *cptr = (my_total *)cntxt->_user_calculation_context;
  cptr->_total = 0;
  cptr->_num_nonnulls_seen = 0;
}


extern "C"
void my_integer_sum_next_value(a_v3_extfn_aggregate_context *cntxt,
                               void *arg_handle)
{
  an_extfn_value  arg;
  a_sql_int32 arg1;

  my_total *cptr = (my_total *)cntxt->_user_calculation_context;

  //  Get the one argument, and if non-NULL then add it to the total
  //
  if (cntxt->get_value( arg_handle, 1, &arg) && arg.data) {
    arg1 = *((a_sql_int32 *)arg.data);
    cptr->_total += arg1;
    cptr->_num_nonnulls_seen++;
  }
}
```

```
extern "C"
void my_integer_sum_drop_value(a_v3_extfn_aggregate_context *cntxt,
                               void *arg_handle)
{
  an_extfn_value  arg;
  a_sql_int32 arg1;
  my_total *cptr = (my_total *)cntxt->_user_calculation_context;

  //  Get the one argument, and if non-NULL then subtract it from the
total
  if (cntxt->get_value( arg_handle, 1, &arg) && arg.data) {
    arg1 = *((a_sql_int32 *)arg.data);
    cptr->_total -= arg1;
    cptr->_num_nonnulls_seen--;
  }
}


extern "C"
void my_integer_sum_evaluate(a_v3_extfn_aggregate_context *cntxt,
                             void *arg_handle)
{
  an_extfn_value  outval;
  my_total *cptr = (my_total *)cntxt->_user_calculation_context;

  //  Set the output result value.  If the inputs
  //  were all NULL, then set the result as NULL.
  //
  outval.type = DT_BIGINT;
  outval.piece_len = sizeof(a_sql_int64);
  if (cptr->_num_nonnulls_seen > 0) {
    outval.data = &cptr->_total;
  } else {
    outval.data = 0;
  }
  cntxt->set_value( arg_handle, &outval, 0 );
}


extern "C"
void my_integer_sum_cum_evaluate(
                       a_v3_extfn_aggregate_context *cntxt,
                       void *arg_handle)
{
  an_extfn_value  outval;
  an_extfn_value  arg;
  int arg1;
  my_total *cptr = (my_total *)cntxt->_user_calculation_context;

  //  Get the one argument, and if non-NULL then add it into the
total.
  //
  if (cntxt->get_value( arg_handle, 1, &arg) && arg.data) {
    arg1 = *((a_sql_int32 *)arg.data);
    cptr->_total += arg1;
```

```
    cptr->_num_nonnulls_seen++;
  }

  //  Then set the output result value.  If the inputs
  //  were all NULL, then set the result as NULL.
  //
  outval.type = DT_BIGINT;
  outval.piece_len = sizeof(a_sql_int64);
  if (cptr->_num_nonnulls_seen > 0) {
    outval.data = &cptr->_total;
  } else {
    outval.data = 0;
  }
  cntxt->set_value( arg_handle, &outval, 0 );
}


extern "C"
void my_integer_sum_next_subagg_value(
                a_v3_extfn_aggregate_context *cntxt,
                void *arg_handle)
{
  an_extfn_value  arg;
  a_sql_int64 arg1;

  my_total *cptr = (my_total *)cntxt->_user_calculation_context;

  //  Get the one argument, and if non-NULL then add it to the total
  //
  if (cntxt->get_value( arg_handle, 1, &arg) && arg.data) {
    arg1 = *((a_sql_int64 *)arg.data);
    cptr->_total += arg1;
    cptr->_num_nonnulls_seen++;
  }
}


extern "C"
void my_integer_sum_drop_subagg_value(
                a_v3_extfn_aggregate_context *cntxt,
                void *arg_handle)
{
  an_extfn_value  arg;
  a_sql_int64 arg1;

  my_total *cptr = (my_total *)cntxt->_user_calculation_context;

  //  Get the one argument, and if non-NULL then subtract it from the
total
  //
  if (cntxt->get_value( arg_handle, 1, &arg) && arg.data) {
    arg1 = *((a_sql_int64 *)arg.data);
    cptr->_total -= arg1;
    cptr->_num_nonnulls_seen--;
  }
}
```

```
a_v3_extfn_aggregate my_integer_sum_descriptor =
{
        &my_integer_sum_start,
        &my_integer_sum_finish,
        &my_integer_sum_reset,
        &my_integer_sum_next_value,
        &my_integer_sum_evaluate,
        &my_integer_sum_drop_value,
        &my_integer_sum_cum_evaluate,
        &my_integer_sum_next_subagg_value,
        &my_integer_sum_drop_subagg_value,
        &my_integer_sum_evaluate,
        NULL, // reserved1_must_be_null
        NULL, // reserved2_must_be_null
        NULL, // reserved3_must_be_null
        NULL, // reserved4_must_be_null
        NULL, // reserved5_must_be_null
        0, // indicators
        ( short )sizeof( my_total ), // context size
        8, // context alignment
        0.0, //external_bytes_per_group
        0.0, // external bytes per row
        0, // reserved6_must_be_null
        0, // reserved7_must_be_null
        0, // reserved8_must_be_null
        0, // reserved9_must_be_null
        0, // reserved10_must_be_null
        NULL // _for_server_internal_use
};

extern "C"
a_v3_extfn_aggregate *my_integer_sum()
{
  return &my_integer_sum_descriptor;
}
```

UDAF example: my_bit_xor definition

The "my_bit_xor" example is similar to the SA built-in BIT_XOR, except my_bit_xor operates only on unsigned integers.

my_bit_xor definition

Because the input and the output data types are identical, use the normal _next_value_extfn and _evaluate_extfn functions to accumulate subaggregate values and produce the superaggregate result.

```
#include "extfnapiv3.h"
#include <stdlib.h>
#include <assert.h>
```

```
//  Generic aggregate UDF that exclusive-ORs a set of
//  unsigned integer arguments, and whenever asked
//  returns the resulting unsigned integer result.
//
//  The start function creates a little structure for
//  the running result, and the finish function then
//  deallocates it.
//
//  Since there are no aggregate usage restrictions
//  for this UDAF, the corresponding SQL declaration
//  will look like:
//
//            CREATE AGGREGATE FUNCTION my_bit_xor(IN arg1 UNSIGNED
INT)
//                      RETURNS UNSIGNED INT
//                      ON EMPTY INPUT RETURNS NULL
//                      EXTERNAL NAME 'my_bit_xor@libudfex'


typedef struct my_xor_result {
  a_sql_uint64 _num_nonnulls_seen;
  a_sql_uint32 _xor_result;
} my_xor_result;

#if defined __cplusplus
extern "C" {
#endif

static void my_xor_start(a_v3_extfn_aggregate_context *cntxt)
{
}


static void my_xor_finish(a_v3_extfn_aggregate_context *cntxt)
{
}


static void my_xor_reset(a_v3_extfn_aggregate_context *cntxt)
{
  my_xor_result *cptr = (my_xor_result *)cntxt-
>_user_calculation_context;
  cptr->_xor_result = 0;
  cptr->_num_nonnulls_seen = 0;
}


static void my_xor_next_value(a_v3_extfn_aggregate_context *cntxt,
                              void *arg_handle)
{
  an_extfn_value  arg;
  a_sql_uint32 arg1;

  my_xor_result *cptr = (my_xor_result *)cntxt-
>_user_calculation_context;
```

```
  //  Get the one argument, and add it to the total
  if (cntxt->get_value( arg_handle, 1, &arg) && arg.data) {
    arg1 = *((a_sql_uint32 *)arg.data);
    cptr->_xor_result ^= arg1;
    cptr->_num_nonnulls_seen++;
  }
}


static void my_xor_drop_value(a_v3_extfn_aggregate_context *cntxt,
                              void *arg_handle)
{
  an_extfn_value  arg;
  a_sql_uint32 arg1;
  my_xor_result *cptr = (my_xor_result *)cntxt-
>_user_calculation_context;

  //  Get the one argument, and remove it from the total
  if (cntxt->get_value( arg_handle, 1, &arg) && arg.data) {
    arg1 = *((a_sql_uint32 *)arg.data);
    cptr->_xor_result ^= arg1;
    cptr->_num_nonnulls_seen--;
  }
}


static void my_xor_evaluate(a_v3_extfn_aggregate_context *cntxt,
                            void *arg_handle)
{
  an_extfn_value  outval;
  my_xor_result *cptr = (my_xor_result *)cntxt-
>_user_calculation_context;

  outval.type = DT_UNSINT;
  outval.piece_len = sizeof(a_sql_uint32);
  if (cptr->_num_nonnulls_seen > 0) {
    outval.data = &cptr->_xor_result;
  } else {
    outval.data = 0;
  }
  cntxt->set_value( arg_handle, &outval, 0 );
}


static void my_xor_cum_evaluate(a_v3_extfn_aggregate_context
*cntxt,
                                void *arg_handle)
{
  an_extfn_value  outval;
  an_extfn_value  arg;
  a_sql_uint32 arg1;
  my_xor_result *cptr = (my_xor_result *)cntxt-
>_user_calculation_context;

  //  Get the one argument, and include it in the result,
  //  unless that input value is null.
```

```
  //
  if (cntxt->get_value( arg_handle, 1, &arg) && arg.data) {
    arg1 = *((a_sql_uint32 *)arg.data);
    cptr->_xor_result ^= arg1;
    cptr->_num_nonnulls_seen++;
  }

  //  Then set the output result value
  outval.type = DT_UNSINT;
  outval.piece_len = sizeof(a_sql_uint32);
  if (cptr->_num_nonnulls_seen > 0) {
    outval.data = &cptr->_xor_result;
  } else {
    outval.data = 0;
  }
  cntxt->set_value( arg_handle, &outval, 0 );
}


static a_v3_extfn_aggregate my_xor_descriptor =
    {
        &my_xor_start,
        &my_xor_finish,
        &my_xor_reset,
        &my_xor_next_value,
        &my_xor_evaluate,
        &my_xor_drop_value,
        &my_xor_cum_evaluate,
        &my_xor_next_value,
        &my_xor_drop_value,
        &my_xor_evaluate,
        NULL, // reserved1_must_be_null
        NULL, // reserved2_must_be_null
        NULL, // reserved3_must_be_null
        NULL, // reserved4_must_be_null
        NULL, // reserved5_must_be_null
        0,    // indicators
        ( short )sizeof( my_xor_result ), // context size
        8,    // context alignment
        0.0,  // external_bytes_per_group
        0.0,  // external bytes per row
        0,    // reserved6_must_be_null
        0,    // reserved7_must_be_null
        0,    // reserved8_must_be_null
        0,    // reserved9_must_be_null
        0,    // reserved10_must_be_null
        NULL  // _for_server_internal_use
    };

a_v3_extfn_aggregate *my_bit_xor()
{
  return &my_xor_descriptor;
}

#if defined __cplusplus
```

```
}
#endif
```

UDAF example: my_bit_or definition

The "my_bit_or" example is similar to the SA built-in BIT_OR, except my_bit_or operates only on unsigned integers, and can be used only as a simple aggregate.

my_bit_or definition

The "my_bit_or" definition is somewhat simpler than the "my_bit_xor" example.

```
#include "extfnapiv3.h"
#include <stdlib.h>
#include <assert.h>


//  A simple (non-OLAP) aggregate UDF that ORs a set
//  of unsigned integer arguments, and whenever asked
//  returns the resulting unsigned integer result.
//
//  The start function creates a little structure for
//  the running result, and the finish function then
//  deallocates it.
//
//  The aggregate usage restrictions for this UDAF
//  only allow its use as a simple aggregate, so the
//  corresponding SQL declaration will look like:
//
//      CREATE AGGREGATE FUNCTION my_bit_or(IN arg1 UNSIGNED INT)
//            RETURNS UNSIGNED INT
//                      ON EMPTY INPUT RETURNS NULL
//            OVER NOT ALLOWED
//            EXTERNAL NAME 'my_bit_or@libudfex'


typedef struct my_or_result {
  a_sql_uint32 _or_result;
  a_sql_uint32 _non_null_seen;
} my_or_result;



#if defined __cplusplus
extern "C" {
#endif

static void my_or_start(a_v3_extfn_aggregate_context *cntxt)
{
}

static void my_or_finish(a_v3_extfn_aggregate_context *cntxt)
{
}

static void my_or_reset(a_v3_extfn_aggregate_context *cntxt)
```

```
{
  my_or_result *cptr = (my_or_result *)cntxt-
>_user_calculation_context;
  cptr->_or_result = 0;
  cptr->_non_null_seen = 0;
}


static void my_or_next_value(a_v3_extfn_aggregate_context *cntxt,
                             void *arg_handle)
{
  an_extfn_value  arg;
  a_sql_uint32 arg1;

  my_or_result *cptr = (my_or_result *)cntxt-
>_user_calculation_context;

  //  Get the one argument, and add it to the total
  if (cntxt->get_value( arg_handle, 1, &arg ) && arg.data)
  {
      arg1 = *((a_sql_uint32 *)arg.data);
      cptr->_or_result |= arg1;
      cptr->_non_null_seen = 1;
  }
}


static void my_or_evaluate(a_v3_extfn_aggregate_context *cntxt,
                           void *arg_handle)
{
  an_extfn_value  outval;
  my_or_result *cptr = (my_or_result *)cntxt-
>_user_calculation_context;

  outval.type = DT_UNSINT;
  outval.piece_len = sizeof(a_sql_uint32);
  if (cptr->_non_null_seen)
  {
      outval.data = &cptr->_or_result;
  }
  else
  {
      // Return null if no values seen
      outval.data = 0;
  }
  cntxt->set_value( arg_handle, &outval, 0 );
}



static a_v3_extfn_aggregate my_or_descriptor =
{
    &my_or_start,
    &my_or_finish,
    &my_or_reset,
    &my_or_next_value,
```

```
    &my_or_evaluate,
    NULL, // drop_val_extfn
    NULL, // cume_eval,
    NULL, // next_subaggregate_extfn
    NULL, // drop_subaggregate_extfn
    NULL, // evaluate_superaggregate_extfn
    NULL, // reserved1_must_be_null
    NULL, // reserved2_must_be_null
    NULL, // reserved3_must_be_null
    NULL, // reserved4_must_be_null
    NULL, // reserved5_must_be_null
    0, // indicators
    ( short )sizeof( my_or_result ), // context size
    8, // context alignment
    0.0, //external_bytes_per_group
    0.0, // external bytes per row
    0, // reserved6_must_be_null
    0, // reserved7_must_be_null
    0, // reserved8_must_be_null
    0, // reserved9_must_be_null
    0, // reserved10_must_be_null
    NULL // _for_server_internal_use
};


extern "C"
a_v3_extfn_aggregate *my_bit_or()
{
  return &my_or_descriptor;
}


#if defined __cplusplus
}
#endif
```

UDAF example: my_interpolate definition

The "my_interpolate" example is an OLAP-style UDAF that attempts to fill in NULL values within a sequence by performing linear interpolation across any set of adjacent NULL values to the nearest non-NULL value in each direction.

my_interpolate definition

To operate at a sensible cost, my_interpolate must run using a fixed-width, row-based window, but the user can set the width of the window based on the maximum number of adjacent NULL values expected. If the input at a given row is not NULL, the result for that row is the same as the input value. This function takes a set of double-precision floating-point values and produces a resulting set of doubles.

```
#include "extfnapiv3.h"
#include <stdlib.h>
#include <assert.h>
```

```
//  MY_INTERPOLATE
//
//  OLAP-style aggregate UDF that accepts a double precision
//  floating point argument.  If the current argument value is
//  not NULL, then the result value is the same as the
//  argument value.  On the other hand, if the current row's
//  argument value is NULL, then the result, where possible,
//  will be the arithmetic interpolation across the nearest
//  preceding and nearest following values that are not NULL.
//  In all cases the result is also a double precision value.
//
//  The start function creates a structure for maintaining the
//  argument values within the window including their NULLness.
//  The finish function then deallocates this structure.
//
//  Since there are some strict aggregate usage restrictions
//  for this aggregate (must be used with a row-based window
//  frame that includes the current row), the corresponding
//  SQL declaration will look like:
//
//      CREATE AGGREGATE FUNCTION my_interpolate(IN arg1 DOUBLE)
//                      RETURNS DOUBLE
//                      OVER REQUIRED
//                      WINDOW FRAME REQUIRED
//                              RANGE NOT ALLOWED
//                              PRECEDING REQUIRED
//                              UNBOUNDED PRECEDING NOT ALLOWED
//                              FOLLOWING REQUIRED
//                              UNBOUNDED FOLLOWING NOT ALLOWED
//                      EXTERNAL NAME  'my_interpolate@libudfex'


typedef struct my_window {
  int      _allocated_elem;
  int      _first_used;
  int      _next_insert_loc;
  int     *_is_null;
  double  *_dbl_val;
  int      _num_rows_in_frame;
} my_window;



#if defined __cplusplus
extern "C" {
#endif


static void my_interpolate_reset(a_v3_extfn_aggregate_context
*cntxt)
{
  assert(cntxt->_user_data);
  my_window *cptr = (my_window *)cntxt->_user_data;

  cptr->_first_used = 0;
  cptr->_next_insert_loc = 0;
```

```
  cptr->_num_rows_in_frame = 0;
  for (int i=0; i<cptr->_allocated_elem; i++) {
    cptr->_is_null[i] = 1;
  }
}


static void my_interpolate_start(a_v3_extfn_aggregate_context
*cntxt)
{
  my_window *cptr = (my_window *)cntxt->_user_data;

  // Make sure function was defined correctly
  if (!cntxt->_is_window_used)
  {
      cntxt->set_error(cntxt, 20001, "Function requires window");
      return;
  }
  if (cntxt->_window_has_unbounded_preceding ||
      cntxt->_window_has_unbounded_following)
  {
      cntxt->set_error(cntxt, 20002, "Window cannot be unbounded");
      return;
  }
  if (cntxt->_window_is_range_based)
  {
      cntxt->set_error(cntxt, 20003, "Window must be row based");
      return;
  }

  if (!cptr) {
    //
    cptr = (my_window *)malloc(sizeof(my_window));
    if (cptr) {
      cptr->_is_null = 0;
      cptr->_dbl_val = 0;
      cptr->_num_rows_in_frame = 0;
      cptr->_allocated_elem = ( int )cntxt->_max_rows_in_frame;
      cptr->_is_null = (int *)malloc(cptr->_allocated_elem
                                     * sizeof(int));
      cptr->_dbl_val = (double *)malloc(cptr->_allocated_elem
                                        * sizeof(double));
      cntxt->_user_data = cptr;
    }
  }
  if (!cptr || !cptr->_is_null || !cptr->_dbl_val) {
      // Terminate this query
      cntxt->set_error(cntxt, 20000, "Unable to allocate memory");
      return;
  }
  my_interpolate_reset(cntxt);
}


static void my_interpolate_finish(a_v3_extfn_aggregate_context
*cntxt)
```

```
{
  if (cntxt->_user_data) {
    my_window *cptr = (my_window *)cntxt->_user_data;
    if (cptr->_is_null) {
      free(cptr->_is_null);
      cptr->_is_null = 0;
    }
    if (cptr->_dbl_val) {
      free(cptr->_dbl_val);
      cptr->_dbl_val = 0;
    }
    free(cntxt->_user_data);
    cntxt->_user_data = 0;
  }
}


static void my_interpolate_next_value(a_v3_extfn_aggregate_context
*cntxt,
                                      void *arg_handle)
{
  an_extfn_value  arg;
  double arg1;
  my_window *cptr = (my_window *)cntxt->_user_data;

  //  Get the one argument, and stash its value
  //  within the rotating window arrays
  //
  int curr_cell_num = cptr->_next_insert_loc % cptr-
>_allocated_elem;
  if (cntxt->get_value( arg_handle, 1, &arg ) && arg.data != NULL ) {
    arg1 = *((double *)arg.data);
    cptr->_dbl_val[curr_cell_num] = arg1;
    cptr->_is_null[curr_cell_num] = 0;
  } else {
    cptr->_is_null[curr_cell_num] = 1;
  }

  //  Then increment the insertion location and number of rows in
frame
  cptr->_next_insert_loc = ((cptr->_next_insert_loc + 1)
                            % cptr->_allocated_elem);
  cptr->_num_rows_in_frame++;
}


static void my_interpolate_drop_value(a_v3_extfn_aggregate_context
*cntxt,
                                      void * /*arg_handle*/)
{
  my_window *cptr = (my_window *)cntxt->_user_data;

  //  Drop one value from the window by incrementing past it and
  // decrement the number of rows in the frame
  cptr->_first_used = ((cptr->_first_used + 1) % cptr-
>_allocated_elem);
```

```
  cptr->_num_rows_in_frame--;
}


static void my_interpolate_evaluate(a_v3_extfn_aggregate_context
*cntxt,
                                    void *arg_handle)
{

  an_extfn_value  outval;
  my_window *cptr = (my_window *)cntxt->_user_data;
  double  result;
  int     result_is_null = 1;
  double  preceding_value;
  int     preceding_value_is_null = 1;
  double  preceding_distance = 0;
  double  following_value;
  int     following_value_is_null = 1;
  double  following_distance = 0;
  int j;

  //  Determine which cell is the current cell
  int curr_cell_num =
      ((int)(cntxt->_result_row_from_start_of_partition-1))%cptr-
>_allocated_elem;
  int tmp_cell_num;

  int result_row_offset_from_start_of_frame = cptr->_first_used <=
curr_cell_num ?
          ( curr_cell_num - cptr->_first_used ) :
          ( curr_cell_num + cptr->_allocated_elem - cptr-
>_first_used );




  //  Compute the result value
  if (cptr->_is_null[curr_cell_num] == 0) {
    //
    //  If the current rows input value is not NULL, then there is
    //  no need to interpolate, just use that input value.
    //
    result = cptr->_dbl_val[curr_cell_num];
    result_is_null = 0;
    //
  } else {
    //
    //  If the current rows input value is NULL, then we do
    //  need to interpolate to find the correct result value.
    //  First, find the nearest following non-NULL argument
    //  value after the current row.
    //
    int rows_following = cptr->_num_rows_in_frame -
                 result_row_offset_from_start_of_frame - 1;
    for (j=0; j<rows_following; j++) {
      tmp_cell_num = ((curr_cell_num + j + 1) % cptr-
```

```
>_allocated_elem);
      if (cptr->_is_null[tmp_cell_num] == 0) {
          following_value = cptr->_dbl_val[tmp_cell_num];
          following_value_is_null = 0;
          following_distance = j + 1;
          break;
      }
    }
    //  Second, find the nearest preceding non-NULL
    //  argument value before the current row.
    //
    int rows_before = result_row_offset_from_start_of_frame;
    for (j=0; j<rows_before; j++) {
     tmp_cell_num = ((curr_cell_num + cptr->_allocated_elem - j - 1)
                     % cptr->_allocated_elem);
      if (cptr->_is_null[tmp_cell_num] == 0) {
          preceding_value = cptr->_dbl_val[tmp_cell_num];
          preceding_value_is_null = 0;
          preceding_distance = j + 1;
          break;
      }
    }
    //  Finally, see what we can come up with for a result value
    //
    if (preceding_value_is_null && !following_value_is_null) {
      //
     //  No choice but to mirror the nearest following non-NULL value
      //  Example:
      //
      //    Inputs:  NULL    Result of my_interpolate:  40.0
      //             NULL                               40.0
      //             40.0                               40.0
      //
      result = following_value;
      result_is_null = 0;
      //
   } else if (!preceding_value_is_null && following_value_is_null) {
      //
     //  No choice but to mirror the nearest preceding non-NULL value
      //  Example:
      //
      //    Inputs:  10.0    Result of my_interpolate:  10.0
      //             NULL                               10.0
      //
      result = preceding_value;
      result_is_null = 0;
      //
   } else if (!preceding_value_is_null && !following_value_is_null)
{
      //
      //  Here we get to do real interpolation based on the
      //  nearest preceding non-NULL value, the nearest following
      //  non-NULL value, and the relative distances to each.
      //  Examples:
      //
      //    Inputs:  10.0    Result of my_interpolate:  10.0
```

```
      //             NULL                                  20.0
      //             NULL                                  30.0
      //             40.0                                  40.0
      //
      //    Inputs:  10.0    Result of my_interpolate:  10.0
      //             NULL                                  25.0
      //             40.0                                  40.0
      //
      result = (  preceding_value
                  + (  (following_value - preceding_value)
                       * (  preceding_distance
                            / (preceding_distance +
following_distance))));
      result_is_null = 0;
    }
  }

  //  And last, pass the result value out
  outval.type = DT_DOUBLE;
  outval.piece_len = sizeof(double);
  if (result_is_null) {
    outval.data = 0;
  } else {
    outval.data = &result;
  }
  cntxt->set_value( arg_handle, &outval, 0 );
}


static a_v3_extfn_aggregate my_interpolate_descriptor =
    {
        &my_interpolate_start,
        &my_interpolate_finish,
        &my_interpolate_reset,
        &my_interpolate_next_value, //( timeseries_expression )
        &my_interpolate_evaluate,
        &my_interpolate_drop_value,
       NULL, // cume_eval,
       NULL, // next_subaggregate_extfn
       NULL, // drop_subaggregate_extfn
       NULL, // evaluate_superaggregate_extfn
       NULL, // reserved1_must_be_null
       NULL, // reserved2_must_be_null
       NULL, // reserved3_must_be_null
       NULL, // reserved4_must_be_null
       NULL, // reserved5_must_be_null
       0, // indicators
       0, // context size
       0, // context alignment
       0.0, //external_bytes_per_group
       ( double )sizeof( double ), // external bytes per row
       0, // reserved6_must_be_null
       0, // reserved7_must_be_null
       0, // reserved8_must_be_null
       0, // reserved9_must_be_null
       0, // reserved10_must_be_null
```

```
       NULL // _for_server_internal_use
    };

a_v3_extfn_aggregate *my_interpolate()
  { return &my_interpolate_descriptor; }

#if defined __cplusplus
  }
#endif
```

Context storage of aggregate user-defined functions

Context variables control whether the intermediate results of aggregate functions are to be managed by the UDF itself (forcing the IQ server to run the UDFs serially), or whether the memory is to be managed by the IQ server. The context area is used to transfer or communicate data between multiple invocations of the UDF within the same query (particularly within OLAP-style queries).

If the _calculation_context_size is set to 0, then the UDF is required to manage all interim results in memory, (forcing the IQ server to invoke the UDF sequentially over the data (instead of being able to invoke many instances of the UDF in parallel during an OLAP query).

If the _calculation_context_size is set to a nonzero value, the IQ server manages a separate context area for each invocation of the UDF, allowing multiple instances of the UDF to be invoked in parallel. To make the most efficient use of memory, consider setting the _calculation_context_alignment a value smaller than the default (depending on the size of the context storage needed).

For details on context storage, refer to the description of _calculation_context_size and _calculation_context_alignment in the section *Aggregate UDF descriptor structure* on page 49. These variables are near the end of the descriptor structure.

For a detailed discussion about the use of context storage, see *Calculation context* on page 53.

Important: To store intermediate results in memory within an aggregate UDF, initialize the memory with the _start_extfn function, and clean up and de-allocate any memory with the _finish_extfn function.

UDF callback functions and calling patterns

Calling patterns are steps the functions perform as results are gathered.

UDF and UDAF callback functions

The set of callback functions supplied by the engine through the a_v3_extfn_scalar_context structure and used within the user's UDF functions include:

- **get_value** – the function used within an evaluation method to retrieve the value of each input argument. For narrow argument data types (smaller than 256 bytes), a call to get_value is sufficient to retrieve the entire argument value. For wider argument data types, if the piece_len field within the an_extfn_value structure passed to this callback comes back with a value smaller than the value in the total_len field, use the get_piece callback to retrieve the rest of the input value.
- **get_piece** – the function used to retrieve subsequent fragments of a long argument input value.
- **get_is_constant** – a function that determines whether the specified input argument value is a constant. This can be useful for optimizing a UDF, for example, where work can be performed once during the first call to the _evaluate_extfn function, rather than for every evaluation call.
- **set_value** – the function used within an evaluation function to tell the server the result value of the UDF for this call. If the result data type is narrow, one call to set_value is sufficient. However, if the result data value is wide, then multiple calls to set_value are required to pass the entire value, and the append argument to the callback should be true for each fragment except the last. To return a NULL result, the UDF should set the data field within the result value's an_extfn_value structure to the null pointer.
- **get_is_cancelled** – a function to determine whether the statement has been cancelled. If a UDF entry point is performing work for an extended period of time (many seconds), then it should, if possible, call the get_is_cancelled callback every second or two to see if the user has interrupted the current statement. The return value is 0 if the statement has not been interrupted.

  Sybase IQ can handle extremely large data sets, and some queries can run for long periods of time. Occasionally, a query takes an unusually long time to execute. The SQL client lets the user cancel a query if it is taking too long to complete. Native functions track when a user has canceled a query. UDFs must also be written in a manner that tracks whether a query has been canceled by the user. In other words, UDFs should support the ability for users to cancel long-running queries that invoke UDFs.
- **set_error** – A function that can be used to communicate an error back to the server, and eventually to the user. Call this callback routine if a UDF entry point encounters an error that should result in an error message being sent back to the user. When called, set_error

rolls back the current statement, and the user receives `Error from external UDF: error_desc_string`, and the SQLCODE is the negated form of the supplied error_number. To avoid collisions with existing errors, UDFs should use error_number values between 17000 and 99999. The maximum length of “error_desc_string” is 140 characters.

- **log_message** – The function used to send a message to the server's message log. The string must be a printable text string no longer than 255 bytes.
- **convert_value** – The function allows data conversion between data types. The primary use is the conversion between DT_DATE, DT_TIME, and DT_TIMESTAMP, and DT_TIMESTAMP_STRUCT. An input and output an_extfn_value is passed to the function.

Scalar UDF calling pattern

eExpected calling pattern for supplied function pointers for a scalar UDF calling pattern.

```
_start_extfn(if supplied)
_evaluate_extfn (repeated 0 to numerous times)
_finish_extfn(if supplied)
```

Aggregate UDF calling patterns

The calling patterns for the user-supplied aggregate UDF functions are more complex and varied than the scalar calling patterns.

The examples that follow this table definition:

```
create table t (a int, b int, c int)
insert into t values (1, 1, 1)
insert into t values (2, 1, 1)
insert into t values (3, 1, 1)
insert into t values (4, 2, 1)
insert into t values (5, 2, 1)
insert into t values (6, 2, 1)
```

The following abbreviation is used:

RR = a_v3_extfn_aggregate_context. _result_row_offset_from_start_of_partition – This value indicates the current row number inside the current partition for which a value is calculated. The value is set during windowed aggregates and is intended to be used during the evaluation step of unbounded windows; it is available at all evaluate calls.

Sybase IQ is a multi user application. Many users can simultaneously execute the same UDF. Certain OLAP queries excute UDFs multiple times within the same query, sometimes in parallel.

Simple aggregate ungrouped

The simple aggregate ungrouped calling pattern totals the input values of all rows and produces a result.

Query

```
select my_sum(a) from t
```

Calling pattern

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)  -- input a=1
_next_value_extfn(cntxt, args)  -- input a=2
_next_value_extfn(cntxt, args)  -- input a=3
_next_value_extfn(cntxt, args)  -- input a=4
_next_value_extfn(cntxt, args)  -- input a=5
_next_value_extfn(cntxt, args)  -- input a=6
_evaluate_extfn(cntxt, args)    -- returns 21
_finish_extfn(cntxt)
```

Result

```
my_sum(a)
21
```

Simple aggregate grouped

The simple aggregate grouped calling pattern totals the input values of all rows in the group and produces a result. **_reset_extfn** identifies the beginning of a group.

Query

```
select b, my_sum(a) from t group by b order by b
```

Calling pattern

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)  -- input a=1
_next_value_extfn(cntxt, args)  -- input a=2
_next_value_extfn(cntxt, args)  -- input a=3
_evaluate_extfn(cntxt, args)    -- returns  6
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)  -- input a=4
_next_value_extfn(cntxt, args)  -- input a=5
_next_value_extfn(cntxt, args)  -- input a=6
_evaluate_extfn(cntxt, args)    -- returns 15
_finish_extfn(cntxt)
```

Result

```
b,   my_sum(a)
1,   6
2,   15
```

OLAP-style aggregate calling pattern with unbounded window

Partitioning on “b” creates the same partitions as grouping on “b”. An unbounded window causes the “a” value to be evaluated for each row of the partition. Because this is an unbounded query, all values are fed to the UDF first, followed by an evaluation cycle. Context indicators are set to 1 for _window_has_unbounded_preceding and _window_has_unbounded_following

Query

```
select b, my_sum(a) over (partition by b rows between
unbounded preceding and
unbounded following)
from t
```

Calling pattern

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)    input a=1
_next_value_extfn(cntxt, args)    input a=2
_next_value_extfn(cntxt, args)    input a=3
_evaluate_extfn(cntxt, args)        rr=1  returns 6
_evaluate_extfn(cntxt, args)        rr=2  returns 6
_evaluate_extfn(cntxt, args)      rr=3  returns 6
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)    input a=4
_next_value_extfn(cntxt, args)    input a=5
_next_value_extfn(cntxt, args)    input a=6
_evaluate_extfn(cntxt, args)      rr=1  returns 15
_evaluate_extfn(cntxt, args)      rr=2  returns 15
_evaluate_extfn(cntxt, args)      rr=3  returns 15
_finish_extfn(cntxt)
```

Result

```
b,  my_sum(a)
1,  6
1,  6
1,  6
2,  15
2,  15
2,  15
```

OLAP-style unoptimized cumulative window aggregate

If _evaluate_cumulative_extfn is not supplied, this cumulative sum is evaluated through the following calling pattern, which is less efficient than _evaluate_cumulative_extfn.

Query

```
select b, my_sum(a) over (partition by b
rows between unbounded preceding and current row)
from t
order by b
```

Calling pattern

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)     -- input a=1
_evaluate_extfn(cntxt, args)    -- returns 1
_next_value_extfn(cntxt, args)  -- input a=2
_evaluate_extfn(cntxt, args)    -- returns 3
_next_value_extfn(cntxt, args)  -- input a=3
_evaluate_extfn(cntxt, args)    -- returns 6
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)  -- input a=4
_evaluate_extfn(cntxt, args)    -- returns 4
_next_value_extfn(cntxt, args)  -- input a=5
_evaluate_extfn(cntxt, args)    -- returns 9
_next_value_extfn(cntxt, args)  -- input a=6
_evaluate_extfn(cntxt, args)    -- returns 15
_finish_extfn(cntxt)
```

Result

```
b,  my_sum(a)
1,  1
1,  3
1,  6
2,  4
2,  9
2,  15
```

OLAP-style optimized cumulative window aggregate

If _evaluate_cumulative_extfn is supplied, this cumulative sum is evaluated where the next_value/evaluate sequence is combined into a single _evaluate_cumulative_extfn call for each row within each partition.

Query

```
select b, my_sum(a) over (partition by b rows between unbounded
preceding and current row)
```

```
from t
order by b
```

Calling pattern

```
_start_extnfn(cntxt)
_reset_extfn(cntxt)
_evaluate_cumulative_extfn(cntxt, args) -- input a=1 returns 1
_evaluate_cumulative_extfn(cntxt, args) -- input a=2 returns 3
_evaluate_cumulative_extfn(cntxt, args) -- input a=3 returns 6
_reset_extfn(cntxt)
_evaluate_cumulative_extfn(cntxt, args) -- input a=4 returns 4
_evaluate_cumulative_extfn(cntxt, args) -- input a=5 returns 9
_evaluate_cumulative_extfn(cntxt, args) -- input a=6 returns 15
_finish_extfn(cntxt)
```

Result

```
b,  my_sum(a)
1,  1
1,  3
1,  6
2,  4
2,  9
2,  15
```

OLAP-style unoptimized moving window aggregate

If _drop_value_extfn function is not supplied, this moving window sum is evaluated through this significantly less efficient than using _drop_value_extfn:

Query

```
select b, my_sum(a) over (partition by b rows between 1 preceding and
current row)
from t
```

Calling pattern

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)         input a=1
_evaluate_extfn(cntxt, args)           returns 1
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)         input a=1
_next_value_extfn(cntxt, args)         input a=2
_evaluate_extfn(cntxt, args)           returns 3
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args )        input a=2
_next_value_extfn(cntxt, args )        input a=3
_evaluate_extfn(cntxt, args)           returns 5
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)         input a=4
```

```
_evaluate_extfn(cntxt, args)              returns 4
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)         input a=4
_next_value_extfn(cntxt, args)         input a=5
_evaluate_extfn(cntxt, args)              returns 9
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)         input a=5
_next_value_extfn(cntxt, args)         input a=6
_evaluate_extfn(cntxt, args)              returns 11
_finish_extfn(cntxt)
```

Result

```
b,  my_sum(a)
1,  1
1,  3
1,  5
2,  4
2,  9
2,  11
```

OLAP-style optimized moving window aggregate

If the _drop_value_extfn function was supplied, this moving window sum is evaluated using this calling pattern, which is more efficient than using _drop_value_extfn

Query

```
select b, my_sum(a) over (partition by b rows between 1 preceding and
current row)
from t
```

Calling pattern

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
_evaluate_aggregate_extfn(cntxt, args)              -- returns  1
_evaluate_aggregate_extfn(cntxt, args)                 -- returns  3
_drop_value_extfn(cntxt)                            -- input a=1
_next_value_extfn(cntxt, args)      -- input a=3
_evaluate_aggregate_extfn(cntxt, args)                 -- returns  5
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)      -- input a=4
_evaluate_aggregate_extfn(cntxt, args)                 -- returns  4
_next_value_extfn(cntxt, args)      -- input a=5
_evaluate_aggregate_extfn(cntxt, args)                 -- returns 9
_drop_value_extfn(cntxt)                         -- input a=4
_next_value_extfn(cntxt, args)      -- input a=6
_evaluate_aggregate_extfn(cntxt, args)              -- returns  11
_finish_extfn(cntxt)
```

Result

```
b,  my_sum(a)
1,  1
```

```
1,  3
1,  5
2,  4
2,  9
2,  11
```

OLAP-style unoptimized moving window following aggregate

If _drop_value_extfn function is not supplied, this moving window sum is evaluated through the following calling pattern. This case is similar to the previous moving window example, but the row being evaluated is not the last row given by next value function.

Query

```
select b, my_sum(a) over (partition by b rows between 1 preceding and
1 following)
from t
```

Calling pattern

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)        input a=1
_next_value_extfn(cntxt, args)      input a=2
_evaluate_extfn(cntxt, args)        returns 3
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)      input a=1
_next_value_extfn(cntxt, args)      input a=2
_next_value_extfn(cntxt, args)      input a=3
_evaluate_extfn(cntxt, args)        returns 6
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)      input a=2
_next_value_extfn(cntxt, args)      input a=3
_evaluate_extfn(cntxt, args)        returns 5
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)      input a=4
_next_value_extfn(cntxt, args)      input a=5
_evaluate_extfn(cntxt, args)        returns 9
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)      input a=4
_next_value_extfn(cntxt, args)      input a=5
_next_value_extfn(cntxt, args)      input a=6
_evaluate_extfn(cntxt, args)        returns 15
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)      input a=5
_next_value_extfn(cntxt, args)      input a=6
_evaluate_extfn(cntxt, args)        returns 11
_finish_extfn(cntxt)
```

Result

```
b,  my_sum(a)
1,  3
1,  6
```

```
1,  5
2,  9
2,  15
2,  11
```

OLAP-style optimized moving window following aggregate

If _drop_value_extfn function is supplied, this moving window sum is evaluated through the following calling pattern. Again, this case is similar to the previous moving window example, but the row being evaluated is not the last row given by next value function.

Query

```
select b, my_sum(a) over (partition by b rows between 1 preceding and
1 following)
from t
```

Calling pattern

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)         input a=1
_next_value_extfn(cntxt, args)      input a=2
_evaluate_extfn(cntxt, args)        returns 3
_next_value_extfn(cntxt, args)      input a=3
_evaluate_extfn(cntxt, args)        returns 6
_dropvalue_extfn(cntxt)                          input a=1
_evaluate_extfn(cntxt, args)        returns 5
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)      input a=4
_next_value_extfn(cntxt, args)      input a=5
_evaluate_extfn(cntxt, args)        returns 9
_next_value_extfn(cntxt, args)      input a=6
_evaluate_extfn(cntxt, args)        returns 15
_dropvalue_extfn(cntxt)                             input a=4
_evaluate_extfn(cntxt, args)        returns 11
_finish_extfn(cntxt)
```

Result

```
b,  my_sum(a)
1,  3
1,  6
1,  5
2,  9
2,  15
2,  11
```

OLAP-style unoptimized moving window without current

Assume the UDF my_sum works like the built-in SUM. If _drop_value_extfn function is not supplied, this moving window count is evaluated through the following calling pattern. This

case is similar to the previous moving window examples, but the current row is not part of the window frame.

Query

```
select b, my_sum(a) over (rows between 3 preceding and 1 preceding)
from t
```

Calling pattern

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
_evaluate_extfn(cntxt, args)          returns NULL
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)      input a=1
_evaluate_extfn(cntxt, args)          returns 1
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)      input a=1
_next_value_extfn(cntxt, args)      input a=2
_evaluate_extfn(cntxt, args)          returns 3
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)      input a=1
_next_value_extfn(cntxt, args)      input a=2
_next_value_extfn(cntxt, args)      input a=3
_evaluate_extfn(cntxt, args)          returns 6
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)      input a=2
_next_value_extfn(cntxt, args)      input a=3
_next_value_extfn(cntxt, args)      input a=4
_evaluate_extfn(cntxt, args)          returns 9
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)      input a=3
_next_value_extfn(cntxt, args)      input a=4
_next_value_extfn(cntxt, args)      input a=5
_evaluate_extfn(cntxt, args)          returns 12
_finish_extfn(cntxt)
```

Result

```
b           my_sum(a)
----------- -------------
          1          NULL
          1             1
          1             3
          2             6
          2             9
          2            12
```

OLAP-style optimized moving window without current

If _drop_value_extfn function is supplied, this moving window count is evaluated through the following calling pattern. This case is similar to the previous moving window examples, but the current row is not part of the window frame.

Query

```
select b, my_sum(a) over (rows between 3 preceding and 1 preceding)
from t
```

Calling pattern

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
_evaluate_extfn(cntxt, args)          returns NULL
_next_value_extfn(cntxt, args)      input a=1
_evaluate_extfn(cntxt, args)          returns 1
_next_value_extfn(cntxt, args)      input a=2
_evaluate_extfn(cntxt, args)          returns 3
_next_value_extfn(cntxt, args)      input a=3
_evaluate_extfn(cntxt, args)          returns 6
_dropvalue_extfn(cntxt)                 input a=1
_next_value_extfn(cntxt, args)      input a=4
_evaluate_extfn(cntxt, args)          returns 9
_dropvalue_extfn(cntxt)                 input a=2
_next_value_extfn(cntxt, args)      input a=5
_evaluate_extfn(cntxt, args)          returns 12
_finish_extfn(cntxt)
```

Result

```
b           my_sum(a)
----------- -------------
          1          NULL
          1             1
          1             3
          2             6
          2             9
          2            12
```

External function prototypes

Define the API by a header file named `extfnapiv3.h`, in the subdirectory of your Sybase IQ installation directory. This header file handles the platform-dependent features of external function prototypes.

To notify the database server that the library is not written using the old API, provide a function as follows:

```
uint32 extfn_use_new_api( )
```

This function returns an unsigned 32-bit integer. If the return value is nonzero, the database server assumes that you are using the new API.

If the DLL does not export this function, the database server assumes that the old API is in use. When using the new API, the returned value must be the API version number defined in `extfnapi.v3h`.

Each library should implement and export this function as:

```
unsigned int extfn_use_new_api(void)
    {
        return EXTFN_V3_API;
    }
```

The presence of this function, and that it returns EXTFN_V3_API informs the IQ engine that the library contains UDFs written to the new API documented in this book.

Function prototypes

The name of the function must match that referenced in the **CREATE PROCEDURE** or **CREATE FUNCTION** statement. Declare the function as:

```
void function-name ( an_extfn_api *api, void *argument-handle )
```

The function must return void, and must take as arguments a structure used to pass the arguments, and a handle to the arguments provided by the SQL procedure.

The an_extfn_api structure has this form:

```
typedef struct an_extfn_api {
short (SQL_CALLBACK *get_value)(
                    void *          arg_handle,
                    a_sql_uint32    arg_num,
                    an_extfn_value *value
                    );
 short (SQL_CALLBACK *get_piece)(
                    void *          arg_handle,
                    a_sql_uint32    arg_num,
                    an_extfn_value *value,
                    a_sql_uint32    offset
                    );
 short (SQL_CALLBACK *set_value)(
                    void *          arg_handle,
                    a_sql_uint32    arg_num,
                    an_extfn_value *value
                    short          append
                    );
void (SQL_CALLBACK *set_cancel)(
                    void *    arg_handle,
                    void *    cancel_handle
                    );
} an_extfn_api;
```

The an_extfn_value structure has this form:

```
typedef struct an_extfn_value {
void *                data;
  a_sql_uint32         piece_len;
  union {
  a_sql_uint32    total_len;
  a_sql_uint32    remain_len;
} len;
  a_sql_data_type        type;
} an_extfn_value;
```

Notes

Calling get_value on an OUT parameter returns the data type of the argument, and returns data as NULL.

The get_piece function for any given argument can be called only immediately after the get_value function for the same argument,

To return NULL, set data to NULL in an_extfn_value.

The append field of set_value determines whether the supplied data replaces (false) or appends to (true) the existing data. You must call set_value with append=FALSE before calling it with append=TRUE for the same argument. The append field is ignored for fixed-length data types.

The header file itself contains additional notes.

Index

A

aCC
    HP-UX 18
    Itanium 18
aggregate
    calculation context 53
    context structure 54
    creating user-defined function 13
    descriptor structure 49
aggregate functions
    declaring 41
    defining 47
    my_bit_or example 45, 65
    my_bit_xor example 45, 61
    my_interpolate example 45, 67
    my_sum example 44, 57
AIX
    PowerPC 17
    xIC 17
API
    declaring version 85
    external functions 85

B

BIGINT data type 21
BINARY (<n>) data type 21
BIT data type 23
building
    shared libraries 16–20

C

C/C++
    new operator 14
    restrictions 14
calculation
    aggregate context 53
call tracing
    configuring 27
calling pattern
    aggregate 76
    aggregate with unbounded window 78
    optimized cumulative moving window aggregate 81
    optimized cumulative window aggregate 79
    optimized moving window following aggregate 83
    optimized moving window without current 85
    scalar syntax 76
    simple aggregate grouped 77
    simple aggregate ungrouped 77
    unoptimized cumulative moving window aggregate 80
    unoptimized cumulative window aggregate 79
    unoptimized moving window following aggregate 82
    unoptimized moving window without current 83
CHAR(<n>) data type 21
compile
    switches 16–20
context
    aggregate structure 54
    scalar structure 34
CREATE AGGREGATE FUNCTION statement
    syntax 41
CREATE FUNCTION statement
    syntax 14, 29
creating
    aggregate user-defined function 13
    scalar user-defined function 12
    user-defined functions 11, 12
cumulative window aggregate
    OLAP-style optimized calling pattern 79
    OLAP-style unoptimized calling pattern 79

D

data types
    supported 21
    unsupported 23
DECIMAL(<precision>, <scale>) data type 23
declaration
    aggregate 41
    aggregate my_bit_or example 45
    aggregate my_bit_xor example 45
    aggregate my_interpolate example 45

aggregate my_sum example 44
scalar 29
scalar my_plus example 30
scalar my_plus_counter example 32
declaring
API version 85
definition
aggregate functions 47
aggregate my_bit_or example 65
aggregate my_bit_xor example 61
aggregate my_interpolate example 67
aggregate my_sum example 57
scalar functions 33
scalar my_plus example 36
scalar my_plus_counter example 37
description
aggregate structure 49
scalar structure 33
disable
user-defined functions 25
documentation
SQL Anywhere 4
Sybase IQ 3
DOUBLE data type 21
dropping
user-defined functions 15
dynamic library interface
configuring 15

E

enabling
user-defined functions 5, 25
error checking
configuring 27
evaluating statements 27
execute permissions
granting 15
revoking 15
external function
prototypes 85
external library
unloading 26
EXTERNAL NAME clause 29
external_udf_execution_mode option 27

F

FLOAT data type 21
functions
callback 75
external, prototypes 85
get_piece 87
get_value 87
GETUID 32
NUMBER 32
prototypes 86
user-defined 5

G

g++
Linux 18
x86 18
GETUID function 32
granting
execute permissions 15
GROUP BY clause 32

H

HAVING clause 32
HP-UX
aCC 6.17 18
Itanium 18

I

IGNORE NULL VALUES 30, 32
INT data type 21
interface
dynamic library 15
IQ_UDF license 5
Itanium
aCC 6.17 18
HP-UX 18

L

library
dynamic interface 15
external 26
interface style 15
license
IQ_UDF 5
link
switches 16–20

Linux
    g++ 4.1.1 18
    PowerPC 18
    X86 18
    xIC 8.0 18
LONG BINARY data type 23
LONG VARCHAR data type 23

M

moving window aggregate
    OLAP-style optimized calling pattern 81
    OLAP-style unoptimized calling pattern 80
moving window following aggregate
    OLAP-style optimized calling pattern 83
    OLAP-style unoptimized calling pattern 82
moving window without current
    OLAP-style optimized calling pattern 85
    OLAP-style unoptimized calling pattern 83
my_bit_or example
    declaration 45
    definition 65
my_bit_xor example
    declaration 45
    definition 61
my_interpolate example
    declaration 45
    definition 67
my_plus example
    declaration 30
    definition 36
my_plus_counter example
    declaration 32
    definition 37
my_sum example
    declaration 44
    definition 57

N

new operator
    C/C++ 14
NULL 30, 32, 37, 87
NUMBER function 32
NUMERIC(<precision>, <scale>) data type 23

O

OLAP-style calling pattern
    aggregate with unbounded window 78
    optimized cumulative moving window aggregate 81
    optimized cumulative window aggregate 79
    optimized moving window following aggregate 83
    optimized moving window without current 85
    unoptimized cumulative moving window aggregate 80
    unoptimized cumulative window aggregate 79
    unoptimized moving window following aggregate 82
    unoptimized moving window without current 83
ON clause 32
optimized calling pattern
    OLAP-style cumulative window aggregate 79
    OLAP-style moving window aggregate 81
    OLAP-style moving window following aggregate 83
    OLAP-style moving window without current 85
ORDER BY clause 13, 41
OVER clause 13, 41

P

pattern
    calling, aggregate 76
    calling, scalar 76
permissions
    granting 15
    revoking 15
    user-defined functions 15
PowerPC
    AIX 17
    Linux 18
    xIC 18
    xIC 8.0 17
prototypes
    external function 85

R

REAL data type 21
RESPECT NULL VALUES 30, 32

restrictions
C/C++ 14
revoking
execute permissions 15

S

scalar functions
callback functions 75
context structure 34
creating user-defined function 12
declaring 29
defining 33
descriptor structure 33
my_plus example 30, 36
my_plus_counter example 32, 37
security
user-defined functions 25
server
disabling UDFs 25
enabling UDFs 25
SET clause 32
shared libraries
building 16–20
simple aggregate grouped
calling pattern 77
simple aggregate ungrouped
calling pattern 77
Solaris
SPARC 19
Sun Studio 12 19
X86 19
SPARC
Solaris 19
Sun Studio 12 19
structure
aggregate context 54
aggregate descriptor 49
scalar context 34
scalar descriptor 33
Studio 12
See Sun Studio 12
Sun Studio 12
Solaris 19
SPARC 19
x86 19
switches
compile 16–20
link 16–20
Sybase IQ
description 1
syntax
aggregate context 54
aggregate declaration 41
aggregate definition 47
aggregate description 49
API version 85
calculation context 53
calling user-defined functions 14
CREATE FUNCTION statement 14
disabling user-defined functions 25
dropping user-defined functions 15
dynamic library interface 15
enabling user-defined functions 25
function prototypes 86
scalar context 34
scalar declaration 29
scalar definition 33
scalar description 33

T

TIME data type 21
TINYINT data type 21

U

UDF
See user-defined functions
unbounded window
OLAP-style aggregate calling pattern 78
unloading
external library 26
unoptimized calling pattern
OLAP-style cumulative window aggregate 79
OLAP-style moving window aggregate 80
OLAP-style moving window following aggregate 82
OLAP-style moving window without current 83
UNSIGNED data type 21
UNSIGNED INT data type 21
UPDATE statement 32
user-define functions 21
user-defined functions
aggregate, creating 13
callback functions 75
calling 14
calling pattern, aggregate 76

calling pattern, scalar 76
creating 11, 12
disabling 25
dropping 15
enabling 5, 25
execution permissions 15
my_bit_or example 45, 65
my_bit_xor example 45, 61
my_interpolate example 45, 67
my_plus example 30, 36
my_plus_counter example 32, 37
my_sum example 44, 57
scalar, creating 12
security 25
using 5

V

VARBINARY(<n>) data type 21
VARCHAR(<n>) data type 21
version
declaring for API 85
Visual Studio 2009
Windows 20
x86 20

W

WHERE clause 32
WINDOW FRAME clause 13
Windows
Visual Studio 2009 20
X86 20

X

x86
g++ 18
Linux 18
Solaris 19
Sun Studio 12 19
Visual Studio 2009 20
Windows 20
xIC
Linux 18
PowerPC 18
xIC 8.0
AIX 17
PowerPC 17