SYBASE®

User-Defined Functions Guide

Sybase IQ 15.1

DOCUMENT ID: DC01034-01-1510-02
LAST REVISED: May 2009

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

Contents

About these topics

Learn about the related documents, documentation conventions, and certifications available for Sybase IQ.

Subject

Sybase® IQ is a high-performance decision-support database server designed specifically for data warehouses and data marts. This book, *User-Defined Functions Guide*, presents concepts about and procedures for programming scalar and aggregate user-defined functions with Sybase IQ.

Audience

This guide is for application developers who access data in Sybase IQ databases. Familiarity with relational database systems and introductory user-level experience with Sybase IQ is assumed. Use this guide with other manuals in the documentation set.

Related documents

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

Sybase IQ documents

The Sybase IQ 15.1 documentation set includes:

- *Release Bulletin* provides information about last-minute changes to the product and documentation.
- *Installation and Configuration Guide* provides platform-specific instructions on installing, migrating to a new version, and configuring Sybase IQ for a particular platform.
- *Advanced Security in Sybase IQ* covers the use of user encrypted columns within the Sybase IQ data repository. You need a separate license to install this product option.
- *Error Messages* lists Sybase IQ error messages referenced by Sybase error code, SQLCode, and SQLState, and SQL preprocessor errors and warnings.
- *IMSL Numerical Library 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 only available to RAP – The Trading Edition™ Enterprise users.
- *Introduction to Sybase IQ* includes hands-on exercises for those unfamiliar with Sybase IQ or with the Sybase Central™ database management tool.

- *Large Objects Management in Sybase IQ* explains storage and retrieval of Binary Large Objects (BLOBs) and Character Large Objects (CLOBs) within the Sybase IQ data repository. You need a separate license to install this product option.
- *New Features in Sybase IQ 15.0* documents new features and behavior changes for version 15.0.
- *New Features Summary Sybase IQ 15.1* summarizes new features and behavior changes for the current version.
- *Performance and Tuning Guide* describes query optimization, design, and tuning issues for very large databases.
- *Quick Start* lists steps to build and query the demo database provided with Sybase IQ for validating the Sybase IQ software installation. Includes information on converting the demo database to multiplex.
- *Reference Manual* – Includes two reference guides to Sybase IQ:
  - *Reference: Building Blocks, Tables, and Procedures* describes SQL, stored procedures, data types, and system tables that Sybase IQ supports.
  - *Reference: Statements and Options* describes the SQL statements and options that Sybase IQ supports.
- *System Administration Guide* – Includes two volumes:
  - *System Administration Guide: Volume 1* describes startup, connections, database creation, population and indexing, versioning, collations, system backup and recovery, troubleshooting, and database repair.
  - *System Administration Guide: Volume 2* describes writing and running procedures and batches, programming with OLAP, accessing remote data, setting up IQ as an Open Server, scheduling and event handling, programming with XML, and debugging.
- *Using Sybase IQ Multiplex* tells how to use multiplex capability, designed to manage large query loads across multiple nodes.
- *Utility Guide* provides Sybase IQ utility program reference material, such as available syntax, parameters, and options.

To access the Infocenter Web site, go to *SyBooks Online Help*.

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).

You can also refer to the SQL Anywhere documentation in the SQL Anywhere Studio 11.0 collection at *Product Manuals* and in *DocCommentXchange*.

Documentation for Sybase Software Asset Management (SySAM) includes:

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

Other sources of information

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

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

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

  Refer to the *SyBooks Installation Guide* on the Getting Started CD, or the `README.txt` file on the SyBooks CD for instructions on installing and starting SyBooks.
- The Sybase Product Manuals Web site is an online version of the SyBooks CD that you can access using a standard Web browser. In addition to product manuals, you will find links to

EBFs/Maintenance, Technical Documents, Case Management, Solved Cases, newsgroups, and the Sybase Developer Network.

To access the Sybase Product Manuals Web site, go to *Product Manuals*.

Sybase certifications on the Web

Find updates to Sybase IQ product and documentation on the Sybase Web site.

Finding the latest information on product certifications

Download the latest product updates from the Sybase Web site.

1. Point your Web browser to *Technical Documents*.
2. Either select the product family and product under Search by Base Product; or select the platform and product under Search by Platform.
3. Select Search to display the availability and certification report for the selection.

Finding the latest information on component certifications

Download the latest component updates from the Sybase Web site.

1. Point your Web browser to *Availability and Certification Reports*.
2. Either select the product family and product under Search by Base Product; or select the platform and product under Search by Platform.
3. Select Search to display the availability and certification report for the selection.

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

Set up a MySybase profile to create a personalized view of Sybase Web pages.

1. Point your Web browser to *Technical Documents*.
2. Click MySybase and create a MySybase profile.

Finding the latest information on EBFs and software maintenance

Use your MySybase account to find the latest information on EBFs and software maintenance.

1. Point your Web browser to the *Sybase Support Page*.
2. Select EBFs/Maintenance. If prompted, enter your MySybase user name and password.
3. Select a product.

4. Specify a time frame and click Go. A list of EBF/Maintenance releases is displayed.

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

Syntax conventions

Learn about syntax conventions used in this document.

- *Keywords* SQL keywords are shown in UPPERCASE. However, SQL keywords are case-insensitive, so you can enter keywords in any case; SELECT, Select, and select are equivalent.
- *Placeholders* Items that must be replaced with appropriate identifiers or expressions are shown in *italics*.
- *Continuation* Lines beginning with an ellipsis (...) are a continuation of the statements from the previous line.
- *Repeating items* Lists of repeating items are shown with an element of the list followed by an ellipsis (...). One or more list elements are allowed. If multiple elements are specified, they must be separated by commas.
- *Optional portions* Optional portions of a statement are enclosed by square brackets. For example:

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

  The square brackets indicate that the *savepoint-name* is optional. Do not type the brackets.
- *Options* When none or only one of a list of items must be chosen, the items are separated by vertical bars and the list enclosed in square brackets. For example:

  ```
  [ ASC | DESC ]
  ```

  The square brackets indicate that you can choose ASC, DESC, or neither. Do not type the brackets.
- *Alternatives* When precisely one of the options must be chosen, the alternatives are enclosed in curly braces. For example:

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

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

Typographic conventions

Learn about the typographic conventions used in this documentation.

Table 1. Typographic conventions

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

Accessibility features

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

Sybase IQ 15.1 and the HTML documentation have been tested for compliance with U.S. government Section 508 Accessibility requirements. Documents that comply with Section 508 generally also meet non-U.S. accessibility guidelines, such as the World Wide Web Consortium (W3C) guidelines for Web sites.For information about accessibility support in the Sybase IQ plug-in for Sybase Central, see the plug-in online help, which you can navigate using a screen reader.

Note: You might need to configure your accessibility tool for optimal use. Some screen readers pronounce text based on its case; for example, they pronounce ALL UPPERCASE TEXT as initials, and MixedCase Text as words. You might find it helpful to configure your tool to announce syntax conventions. Consult the documentation for your tool and see *Introduction to Sybase IQ* for information on using screen readers.

For information about how Sybase supports accessibility, see *Sybase Accessibility*. The Sybase Accessibility site includes links to information on Section 508 and W3C standards.

For a Section 508 compliance statement for Sybase IQ, go to *Sybase Accessibility*.

The demo database

Sybase IQ includes scripts to create a demo database.

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

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

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

If you need help

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

Installing and configuring user-defined functions

Learn how to install and create user-defined functions.

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

Sybase IQ overview

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

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. These UDF definitions can be compiled and linked into a dynamically linkable library. The dynamically linkable library can be loaded into a running IQ server, and the defined UDFs can be used directly within queries or other SQL statements.

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. For instructions on creating UDFs using Interactive SQL see Chapter 1, “Using Procedures and Batches,” in System Administration Guide: Volume 2.

To build and use a UDF from a dynamically linkable library:

1. Declare the UDF to the server by using the CREATE FUNCTION or CREATE AGGREGATE FUNCTION statements. These statements can either be directly written and executed as a command, or the appropriate CREATE statement can be built using the *Sybase Central New Function wizard* on page 11.

   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. Define the UDF as a set of C or C++ functions. See *Defining a scalar UDF* on page 25 or *Defining an aggregate UDF* on page 39.
3. *Write the UDF library identification function.* on page 10
4. *Compile the UDF functions and the library identification functions.* on page 15
5. Link the compiled file into a dynamically linkable library.

Once these steps have been completed, any reference to that UDF in a SQL statement first, if necessary, links the dynamically linkable library. The *calling patterns* on page 67 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 10.

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 version 3 interface style (EXTFN_V3_API) is supported.

Enabling and disabling user-defined functions

Sybase IQ 15.1 includes a security feature, external_procedure_v3, which enables or disables the ability of a server to make use of the high performance in-process UDFs.

Administrators can enable version 3 UDFs for any server by specifying

```
-sf -external_procedure_v3
```

in the server startup command or in the configuration file.

Administrators can disable version 3 UDFs for any server by specifying

```
-sf external_procedure_v3
```

in the server startup command or in the configuration file.

Additional information on the -sf flag is available in the *SQL Anywhere Server - Database Administration* guide. The values listed in the SQL Anywhere document are not applicable to Sybase IQ 15.1, and should not be used.

Creating and executing user-defined functions

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

There are four steps required to create and use a UDF:

1. Declare the user-defined function using the CREATE AGGREGATE FUNCTION statement or the CREATE FUNCTION statement.
2. Implement the function entry point(s) in C/C++.
3. *Compile the implementation and link it as a shared library* on page 15.
4. Use the function within a SQL statement anywhere that you would use a built-in SQL function.

Creating user-defined functions

Use the **CREATE FUNCTION** or **CREATE AGGREGATE FUNCTION** statements to create user-defined functions. You must have Resource authority to execute this statement.

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. From Sybase Central's View menu, select Folders.
3. In the left pane, right-click Procedures & Functions and select New > Function.
4. In the Welcome dialog, type 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 which user will 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, untis, 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 if the function is allowed to be used in an **OVER** clause, is required to be used in an **OVER** clause, or is 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, proceed with 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 must have the following restrictions.

- All UDFs should be written in a manner that allows the functions 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 is responsible for implementing 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.
- All aggregate UDFs and all deterministic scalar UDFs should be written such that the receipt of the same input values will always produce 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.

Calling user-defined functions

You can use a user-defined function, subject to permissions, in 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.

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 that user can execute it without permission. The owner of a user-defined function can grant permissions to other users with the **GRANT EXECUTE** command.

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

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

Or to revoke permissions by issuing:

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

For more information on managing user permissions on functions, see “Granting permissions on procedures” in Chapter 8, “Managing User IDs and Permissions,” *System Administration Guide: Volume 1*.

Compile and link switches for building dynamically linkable libraries

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

The following steps are required to build a UDF dynamically linkable library:

1. A UDF dynamically linkable library must include an implementation of the function "extfn_use_new_api()". The source code for this function is listed under *Setting the dynamic library interface* on page 10. 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 and can be used 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 instance, steps for building the example dynamically linkable library "libudfex" would consist of:

- Compiling 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
```

- Linking together each object produced into a single library.

The next section lists platform specific recommendations for compiling source files and linking objects to form a UDF dynamically linkable library. Other versions of compilers may work, these specific examples are provided as a guide

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

- Update the CREATE FUNCTION ... EXTERNAL NAME to include an explicit path name for the UDF library. (Recommended)
- Place the UDF library file into the directory where all the IQ libraries are stored.
- Start up the IQ server 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.

AIX switches

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

xlC 8.0 on a PowerPC

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

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

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

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

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
```

Windows switches

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

Visual Studio 2008 on x86

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:iqudf.dll
```

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 whose maximum possible binary length, “<n>”, is 32767. The data is not null-byte terminated. The data type identifier is DT_FIXBINARY, 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_VARBINARY, 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/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 datetype represent date and time with the following 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/sorting operations. A larger value indicates a later time. If the actual time components are required, the UDF must invoke the convert_value api in order 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/sorting operations. A larger value indicates a later datetime. If the actual time components are required, the UDF must invoke the convert_value api in order 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, this 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.

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 new server startup option that allows an administrator to enable or disable this style of user-defined functions.

Note: You can also *create the user-defined function declaration in Sybase Central* on page 11.

By default, all user-defined functions are accessed using the access permissions of the owner of the UDF.

The supported IQ 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 recommendeds 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.

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.

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 'describe_my_plus@my_shared_lib'
```

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 resutls 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.

There are two optional pieces:

- **_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 are defined to 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 are defined to 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, then 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.

When you are allocating memory, make sure UDF libraries do not overload the C++ new operator. On some platforms, overloading the new operator has a global effect that may adversely affect the proper operation of the server.

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 following is 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 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 UDAF

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

Note: You can also *create the user-defined function declaration in Sybase Central* on page 11.

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 all 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. The UDAF must be written 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 only as 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

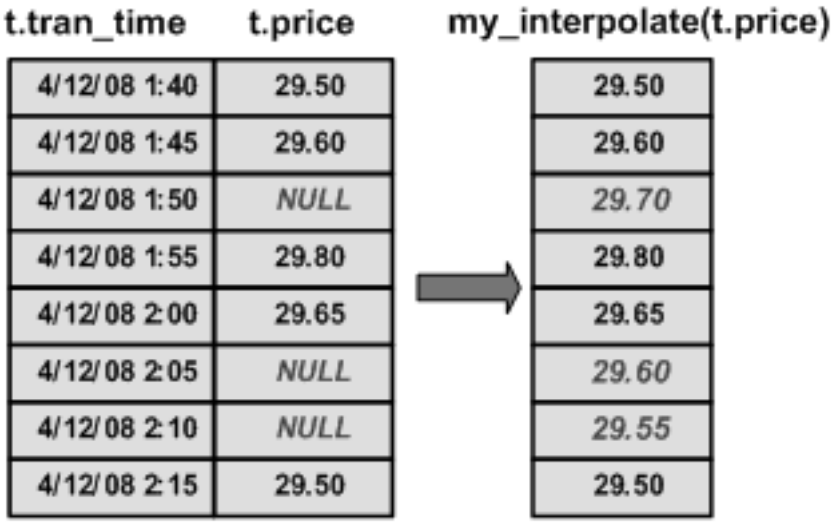
interpolation across any set of adjacent NULL values to the nearest non-NULL value in each direction.

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.

This table illustrates the effect of my_interpolate on a small set of input rows:

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 only 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; 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.

The eight mandatory pieces are:

- **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 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 this function 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 this function 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.

  This function 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 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 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 sub-aggregates 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 has the following 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, this is 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, this is used to deallocate some structure whose address was 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, this is used to reset some values in the structure whose 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. Then 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, this would be 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 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._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 is more memory-efficient.

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. It also has 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 where this allows 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 do a return. 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. Zero 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 zero 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 zero indicates the window does not have unbounded preceding.
  - **a_sql_uint32 _window_contains_current_row;** – A return value of zero indicates the window does not contain the current row.
  - **a_sql_uint32 _window_is_range_based;** – If the return code is one, the window is range-based. If the return code is zero, the window is row-based.

- **Available at reset_extfn() calls** – Returns the actual number of rows in current partition, or zero 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 sub-aggregate values and produce the super-aggregate 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  preceeding_value;
  int     preceeding_value_is_null = 1;
  double  preceeding_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 preceeding 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) {
          preceeding_value = cptr->_dbl_val[tmp_cell_num];
          preceeding_value_is_null = 0;
          preceeding_distance = j + 1;
          break;
      }
    }
    //  Finally, see what we can come up with for a result value
    //
    if (preceeding_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 (!preceeding_value_is_null && following_value_is_null)
{
      //
      //  No choice but to mirror the nearest preceeding non-NULL
value
      //  Example:
      //
      //    Inputs:  10.0    Result of my_interpolate:  10.0
      //             NULL                               10.0
      //
      result = preceeding_value;
      result_is_null = 0;
      //
   } else if (!preceeding_value_is_null && !following_value_is_null)
{
      //
      //  Here we get to do real interpolation based on the
      //  nearest preceeding 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 = (  preceeding_value
                + (  (following_value - preceeding_value)
                     * (  preceeding_distance
                          / (preceeding_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
```

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 (< 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 less than the value in the total_len field, then the get_piece callback, below, must be used 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 can be used to determine 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, then 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 that can be used 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.
- **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 causes the current statement to be rolled back and the user receives an error where the message text is “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 limited to 140 characters according to existing IQ internals.

- **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

In general, the expected calling pattern for these supplied function pointers for a scalar UDF look like this:

```
_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. This section shows example calling sequences that IQ UDAFs encounter for different SQL statements. These examples use 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.

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. The

_window_has_unbounded_preceding and _window_has_unbounded_following context indicators are set to 1.

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)
_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
_drop_value_extfn(cntxt)                          -- input a=1
_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
_next_value_extfn(cntxt, args)      -- input a=5
_evaluate_extfn(cntxt, args)                -- returns 9
_drop_value_extfn(cntxt)                       -- input a=4
_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 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
```

UDF specific functions and statements

User-defined functions use specific statements and functions. Learn to create function prototypes, and the syntax and usage of the statements.

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 follows:

```
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 version 3 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 some additional notes.

Finance specific functions

If you have RAP – The Trading Edition™: Enterprise, you can use these user-defined aggregate functions for financial analysis:

- **ts_arma_ar**
- **ts_arma_const**
- **ts_arma_ma**
- **ts_autocorrelation**
- **ts_auto_uni_ar**
- **ts_box_cox_xform**
- **ts_difference**
- **ts_estimate_missing**
- **ts_lack_of_fit**
- **ts_lack_of_fit_p**
- **ts_max_arma_ar**
- **ts_max_arma_const**
- **ts_max_arma_ma**
- **ts_max_arma_likelihood**
- **ts_outlier_identification**
- **ts_partial_autocorrelation**
- **ts_vwap**

For detailed information on these functions, see *Reference: Building Blocks, Tables, and Procedures*.

See *User-defined aggregate functions* on page 33 for information on syntax, parameters, and usage of the user-defined aggregate functions.

Managing external libraries

External libraries are loaded by the server the first time a UDF that requires it is invoked. A loaded library remains loaded by the server for the life of the server once it has been loaded after first being needed. 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 will only be successful 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. The syntax for unloading an external function library.

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

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

If a registered function uses a complete path, for example '/abc/def/library.dll', unregister the function or the library will not be unloaded.

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

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 will optimize 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.

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X