User-Defined Functions

SAP Sybase IQ 16.0 SP03

DOCUMENT ID: DC01034-01-1603-01
LAST REVISED: December 2013

Contents

Audience

The User-Defined Functions guide is intended for SQL analysts, C developers, C++ developers, and Java developers who want to extend the functionality of SAP® Sybase® IQ.

As a developer, use the tasks, concepts, and API reference material to program non-SQL external user-defined functions.

As a SQL analyst, use this guide to develop SQL queries that reference non-SQL external user-defined functions.

Understanding User-Defined Functions

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

SAP Sybase IQ allows user defined functions (UDFs), which execute within the database container. The UDF execution feature is available as an optional component for use within SAP Sybase IQ.

You must be specifically licensed to use these external C/C++ UDFs interfaces.

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

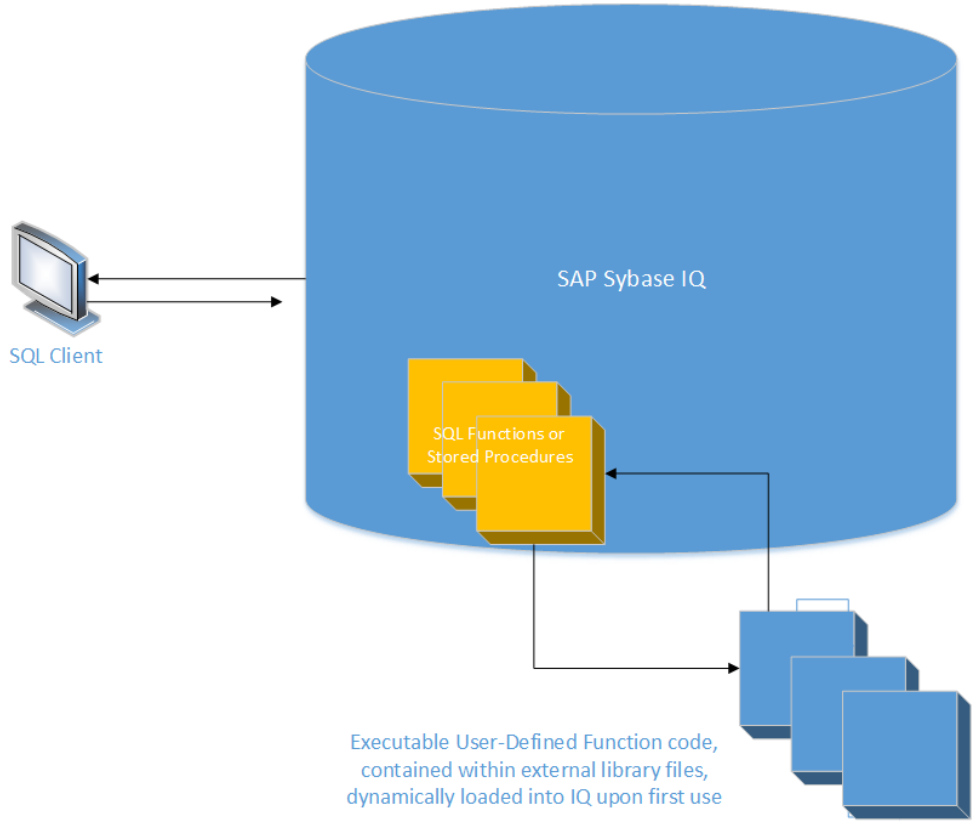
UDFs that execute within SAP Sybase IQ take advantage of the extreme performance of the server, while also providing users the flexibility of analyzing their data with the flexibility of a programmatic solution. User-Defined Functions consist of two components:

- UDF declaration, and
- UDF executable code

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

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

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



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

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

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

To facilitate the construction of UDFs, SAP Sybase IQ includes a C-based API. The API comprises a set of predefined entry points for the UDFs, a well-defined context data structure, and a series of SQL callback functions that provide a communication mechanism from the UDF back to the server. The SAP Sybase IQ UDF API allows software vendors and expert end-users to develop, package, and sell their own UDFs.

Learning Roadmap: Types of UDFs

The types of user-defined functions (UDFs) available in SAP Sybase IQ.

UDF Type	Description	Required Licence	See
UDF (SQL)	A user-defined function written in SQL.	none	*Administration: Database > Create Procedures and Batches > Introduction to User-Defined Functions*
Scalar C or C++ UDF	V3 external C or C++ procedure that operates on a single value.	IQ_UDF	*Learning Roadmap: Types of External C and C++ UDFs* on page 6
Scalar C or C++ UDF	V4 external C or C++ procedure that operates on a single value.	IQ_IDA	*Learning Roadmap: Types of External C and C++ UDFs* on page 6
Aggregate C or C++ UDF	V3 external C or C++ procedure that operates on multiple values. Aggregate UDFs are also sometimes known as UDAs or UDAFs. The context structure for coding aggregate UDFs is slightly different than the context structure used for coding scalar UDFs.	IQ_UDF	*Learning Roadmap: Types of External C and C++ UDFs* on page 6
Aggregate C or C++ UDF	V4 external C or C++ procedure that operates on multiple values. Aggregate UDFs are also sometimes known as UDAs or UDAFs. The context structure for coding aggregate UDFs is slightly different than the context structure used for coding scalar UDFs.	IQ_IDA	*Learning Roadmap: Types of External C and C++ UDFs* on page 6
Table UDF	External C or C++ procedure that produces a set of rows and can be used as a table expression in the **FROM** clause of a SQL statement	IQ_IDA	*Learning Roadmap: Types of External C and C++ UDFs* on page 6

UDF Type	Description	Required Licence	See
Table parameterized function	A table UDF that accepts table (non-scalar) parameters in addition to scalar parameters, and can be executed in parallel over partitions of row-sets. Also known as table parameterized user-defined functions.	IQ_IDA	*Learning Roadmap: Types of External C and C++ UDFs* on page 6
Java scalar UDF	An out-of-process (external environment) scalar user-defined function implemented in Java code.	none	*The Java External Environment* on page 341
Java table UDF	An out-of-process (external environment) table UDF implemented in Java code.	none	*The Java External Environment* on page 341

Learning Roadmap: Types of External C and C++ UDFs

The high-performance, in-process, external C and C++ user-defined functions available with the IQ_IDA license.

The v3 API requires either the IQ_UDF or IQ_IDA license. The v4 API requires the IQ_IDA license.

UDF Type	Input Parameters	Return	API	See:
Scalar UDF	Scalar	Single scalar value	v3, v4	*Declaring and Defining Scalar User-Defined Functions* on page 32
Aggregate UDF	Scalar	Single scalar value	v3, v4	*Declaring and Defining Aggregate UDFs* on page 46
Table UDF	Scalar	Table	v4	*Table UDFs and TPFs* on page 97
Table parameterized function (TPF)	Scalar and table	Table	v4	*Table Parameterized Functions* on page 136

These UDFs can be deterministic or nondeterministic. The result of a function can be determined by the input parameters and data (deterministic), or by some random behavior (nondeterministic). Parameters of nondeterministic UDFs typically need a random seed as one of the input parameters.

User-Defined Functions Compliance with SAP Sybase IQ Databases

Develop user-defined functions to work with SAP Sybase IQ databases.

- **Seamless Execution –** UDFs must run seamlessly within the database container. Although SAP Sybase IQ is a complex product consisting of many files, the main user interaction is through a server process (iqsrv16.0), using industry-standard Structured Query Language (SQL). Execution of UDFs should be accomplished entirely through SQL commands; the user does not need to understand the underlying implementation method to use the UDFs.

  The `EXTFN_V3_API` and `EXTFN_V4_API` provide callback functions enabling the UDF to write to the message file (`.iqmsg`).

  UDFs should manage memory and temporary results as defined by the `EXTFN_V3_API` and `EXTFN_V4_API`.

  SAP Sybase IQ is a multiuser application. Many users can simultaneously execute the same UDF. Certain OLAP queries cause a UDF to be executed multiple times within the same query, sometimes in parallel. For details on setting UDFs to run in parallel, see *Aggregate UDF calling patterns* on page 84.
- **Internationalization –** SAP Sybase IQ has been internationalized for global use. Error messages are in external files, which allows you to localize error messages to new languages without having to make extensive code changes.

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

  SAP Sybase IQ has also been localized for a few non-English languages. To support localization to the same languages that SAP Sybase IQ supports, internationalize UDFs so that an independent organization can localize them at a later date.

  For details about international language support in SAP Sybase IQ, see *International Languages and Character Sets* in *Administration: Globalization* .

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

- **Platform Differences –** Develop UDFs to run on a variety of platforms supported by SAP Sybase IQ. The SAP Sybase IQ 16.0 server runs on 64-bit architectures, and is supported under several platforms of the MS Windows (64-bit) family of operating systems. SAP Sybase IQ is also supported on versions of UNIX (64-bit), including Solaris, HP-UX, AIX, and Linux.

Practices to Avoid

Learn good practices for creating user-defined functions.

- Do not write ambiguous code, or constructs that can unexpectedly loop forever, without providing a mechanism for the user to cancel the UDF invocation (see the function **'get_is_cancelled()'**.
- Do not perform complex, or memory-intensive operations that are repeated every invocation. When a UDF call is made against a table that contains many thousands of rows, efficient execution becomes paramount. Allocate blocks of memory for a thousand to several thousand rows at a time, rather than on a row-by-row basis.
- Do not open a database connection, or perform database operations from within a UDF. All parameters and data required for UDF execution must be passed as parameters to the UDF.
- Do not use reserved words when naming UDFs.

Note: Use source control software for C++ UDFs and Java UDFs to track changes to:

- The source code (`.java` files/`.cpp` files)
- The `class/jar/dll/so` files that may be deployed to the database or mentioned in the UDF stored procedure definition.
- The Syntax for the UDF stored procedure definition itself.
- Deployment instructions, 3rd party library versions and special deployment notes such as security specifics.

See also

- *get_is_cancelled* on page 298

Naming Conventions for User-Defined Functions

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

SAP Sybase IQ identifiers have a maximum length of 128 bytes. For simplicity of use, UDF names should start with an alphabetic character. Alphabetic characters as defined by SAP Sybase IQ include the letters of the alphabet, plus underscore (_), at sign (@), number or pound sign (#) and dollar sign ($). UDF names should consist entirely of these alphabetic characters as well as digits (the numbers 0 through 9). UDF names should not conflict with

SQL reserved words. For a list of SQL reserved words in SAP Sybase IQ see *Reserved Words* in *Reference: Building Blocks, Tables, and Procedures*.

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

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

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

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

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:

SQL Data Type	C or C++ Data Type Identifier	C or C++ Typedef	Description
`UNSIGNED BIGINT`	DT_UNSBIGINT	a_sql_uint64	An unsigned 64-bit integer, requiring 8 bytes of storage.
`BIGINT`	DT_BIGINT	a_sql_int64	A signed 64-bit integer, requiring 8 bytes of storage.

SQL Data Type	C or C++ Data Type Identifier	C or C++ Typedef	Description
UNSIGNED INT	DT_UNSINT	a_sql_uint32	An unsigned 32-bit integer, requiring 4 bytes of storage.
INT	DT_INT	a_sql_int32	A signed 32-bit integer, requiring 4 bytes of storage.
SMALLINT	DT_SMALLINT	short	A signed 16-bit integer, requiring 2 bytes of storage.
TINYINT	DT_TINYINT	unsigned char	An unsigned 8-bit integer, requiring 1 byte of storage.
DOUBLE	DT_DOUBLE	double	A signed 64-bit double-precision floating point number, requiring 8 bytes of storage.
REAL	DT_FLOAT	float	A signed 32-bit floating point number, requiring 4 bytes of storage.
FLOAT	DT_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.
CHAR(<n>)	DT_FIXCHAR	char	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.
VARCHAR(<n>)	DT_VARCHAR	char	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_len* field within the an_extfn_value structure. Similarly, for a UDF result of this type, the actual length must be set in the *total_len* field.

SQL Data Type	C or C++ Data Type Identifier	C or C++ Typedef	Description
LONG VARCHAR(<n>) or CLOB	DT_VARCHAR	char	A varying-length character string, in the database default character set. Use the LONG VARCHAR data type only as an input argument, not as a return-value data type. The maximum possible length, "<n>", is 4GB (gigabytes) for v3 UDFs. The data is not null-byte terminated. LONG VARCHAR data type can have a WD or TEXT index. For UDF input arguments, the actual length, when the value is not NULL, must be retrieved from the *total_len* field within the an_extfn_value structure. You need not rebuild or recompile an existing scalar or aggregate UDF to use a LOB data type as an input parameter, if the function contains a loop that reads pieces of the value via the get_value() and get_piece() methods. The loop continues until *remain_len* > 0 or until 4GB is reached for v3 UDFs (there is no 4GB limit in v4). Table UDFs and TPFs do not use the get_piece() method to process and retrieve data. Table UDFs and TPFs must use the Blob (a_v4_extfn_blob) API instead. Use blob_length to determine length of input parameters. Large object data support requires a separately licensed SAP Sybase IQ option.
BINARY(<n>)	DT_BINARY	unsigned char	A fixed-length null-byte padded binary, value with a maximum possible binary length, "<n>", of 32767. The data is not null-byte terminated.
VARBINARY(<n>)	DT_BINARY	unsigned char	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_len* field within the an_extfn_value structure. Similarly, for a UDF result of this type, you must set the actual length in the *total_len* field. The data is not null-byte terminated.

SQL Data Type	C or C++ Data Type Identifier	C or C++ Typedef	Description
LONG BINARY(<n>) or BLOB	DT_BINARY	unsigned char	A fixed-length null-byte padded binary, value with a maximum possible binary length, "<n>", of 4GB (gigabytes) for v3 UDFs. Use the LONG BINARY data type only as an input argument, not as a return-value data type. You need not rebuild or recompile an existing scalar or aggregate UDF to use a LOB data type as an input parameter, if the function contains a loop that reads pieces of the value via the get_value() and get_piece() methods. The loop continues until *remain_len* > 0 or until 4GB is reached for v3 UDFs (there is no 4GB limit in v4). Table UDFs and TPFs do not use the get_piece() method to process and retrieve data. Table UDFs and TPFs must use the Blob (a_v4_extfn_blob) API instead. Use blob_length to determine length of input parameters. Large object data support requires a separately licensed SAP Sybase IQ option.

SQL Data Type	C or C++ Data Type Identifier	C or C++ Typedef	Description
DATE	DT_TIMESTAMP_STRUCT	unsigned integer	A calendar date value, which is passed to or from a UDF as an unsigned integer. The value given to the UDF is guaranteed to be usable in comparison and sorting operations. A larger value indicates a later date. If the actual date components are required, the UDF must invoke the `convert_value` function in order to convert to the type DT_TIMESTAMP_STRUCT. This date type represents date and time with this structure: `typedef struct sqldatetime { unsigned short year; /* e.g. 1992*/ unsigned char month; /* 0-11 */ unsigned char day_of_week; /* 0-6 0=Sunday, 1=Monday, ... */ unsigned short day_of_year; /* 0-365 */ unsigned char day; /* 1-31 */ unsigned char hour; /* 0-23 */ unsigned char minute; /* 0-59 */ unsigned char second; /* 0-59 */ a_sql_uint32 microsecond; /* 0-999999 */ } SQLDATETIME;`
	DT_TIMESTAMP_STRUCT	unsigned bigint	A value that precisely describes a moment within a given day. . The value given to the UDF is guaranteed to be usable in comparison and sorting operations. A larger value indicates a later time. If the actual time components are required, the UDF must invoke the `convert_value` function to convert to the type `DT_TIMESTAMP_STRUCT`.

SQL Data Type	C or C++ Data Type Identifier	C or C++ Typedef	Description
DATETIME, SMALLDATETIME, or TIMESTAMP	DT_TIMESTAMP_STRUCT	unsigned bigint	A calendar date and time value. The value given to the UDF is guaranteed to be usable in comparison and sorting operations. A larger value indicates a later datetime. If the actual time components are required, the UDF must invoke the convert_value function to convert to the type DT_TIMESTAMP_STRUCT.
TABLE	DT_EXTFN_TABLE	a_v4_extfn_table	Represents an input TABLE parameter result set. This datatype is only available on TPFs.

See also

- *Blob (a_v4_extfn_blob)* on page 199
- *Blob Input Stream (a_v4_extfn_blob_istream)* on page 203
- *convert_value* on page 300
- *Table (a_v4_extfn_table)* on page 310

Unsupported Data Types

Certain SQL data types cannot be used in a UDF declaration, either as data types for arguments to a UDF, or as return-value data types.

- **BIT –** Should typically be handled in the UDF declaration as a TINYINT data type, and then the implicit data type conversion from BIT automatically handles the value translation.
- **DECIMAL –** (<precision>, <scale>) or NUMERIC(<precision>, <scale>) – depending on the usage, DECIMAL is typically handled as a DOUBLE data type, but various conventions may be imposed to enable the use of INT or BIGINT data types.
- **LONG VARCHAR –** (CLOB) – supported only as an input argument, not as a return-value data type. An exception exists for pass-through TPFs, where LONG VARCHAR is supported as a return-value data type.
- **LONG BINARY –** (BLOB) – supported only as an input argument, not as a return-value data type. An exception exists for pass-through TPFs, where LONG BINARY is supported as a return-value data type.
- **TEXT –** not currently supported.

Building UDFs

Design, build, and test UDFs.

Design Basics of User-Defined Functions

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

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

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

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

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

Sample Code

Sample UDF source code is delivered with the product. The newest version of the sample code is always delivered with the most current version of SAP Sybase IQ.

On UNIX platforms, the sample UDF code is in `$SYBASE/IQ-16.0/samples/udf` (where $SYBASE is the installation root).

On Windows platforms, the sample UDF code is in `C:\Documents and Settings \All Users\SybaseIQ\samples\udf`.

The sample UDF code documented in the *User-Defined Functions* guide may not be the latest version as delivered with the SAP Sybase IQ product. Last-minute changes to the sample UDF source code are documented in the Release Bulletin for your operating system platform.

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_V4_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 UDFs, only new interface styles `EXTFN_V3_API` and `EXTFN_V4_API` are supported.

Upgrading to the v4 API

Upgrade to the v4 API included with 16.0.

Prerequisites

Install SAP Sybase IQ server version 16.0.

Task

If you have existing scalar or aggregate UDFs developed for SAP Sybase IQ server versions 15.1, 15.2, or 15.3, those UDFs use the V3 API interface style and reference the `extfnapiv3.h` header file. Modify your legacy C or C++ external library files to reference the `extfnapiv4.h` header file.

Existing v3 scalar and aggregate functions continue to work as designed. However, to take advantage of scalar and aggregate distribution in PlexQ, you must upgrade the header file and library version to v4. You need not change the name of the typedefs for your scalar or aggregate function.

1. Open the C or C++ external library file defining the scalar or aggregate user-defined function.
2. Locate all instances of `#include 'extfnapiv3.h'` and change to `#include 'extfnapiv4.h'`.
3. Set the dynamic library interface to `EXTFN_V4_API`.
4. Rebuild.

Next

Partners must ensure the library exports `extfn_get_license_info` as an entry point.

See also

- *External Function Prototypes* on page 93
- *License Information (a_v4_extfn_license_info)* on page 305
- *Defining an Aggregate UDF* on page 53
- *Defining a Scalar UDF* on page 37
- *Developing a Table UDF* on page 103
- *Developing a TPF* on page 136

Library Version (extfn_get_library_version)

Use the `extfn_get_library_version` method to extract the library version from the current multiplex node. The server considers partitioning a query across multiplex nodes only if the installed library is compatible with the other nodes.

Implementation

A v4 library can define this optional entry point:

```
size_t extfn_get_library_version( uint8 *buff, size_t len );
```

Description

Library versioning methods are at the library level, and do not have the `a_v4` prefix in their method name.

If the v4 library defines the optional entry point, the server allows query distribution to other nodes. The entry point populates the supplied buffer with the library version string (a C-style character string containing only ASCII characters, terminated with **\0**) and returns the actual size of the populated version string, which is constrained to a maximum of 256 bytes.

If an entry point is not defined, the server does not distribute the UDF to the other nodes in the multiplex.

See also

- *Library Version Compatibility (extfn_check_version_compatibility)* on page 17
- *Setting the Dynamic Library Interface* on page 15

Library Version Compatibility (extfn_check_version_compatibility)

Use the `extfn_check_version_compatibility` method to define compatibility criteria for library versions across nodes in a multiplex.

Implementation

A v4 library can define this optional entry point:

```
a_bool extfn_check_version_compatibility( uint8 *buff, size_t
len );
```

Description

Library versioning methods are at the library level, and do not have the `a_v4` prefix in their method name.

This optional entry point accepts a buffer containing the version string and the version string length. It returns whether or not the library version on the target node is compatible with the version string parameter. The library developer defines the compatibility criteria.

Interaction with `extfn_get_library_version`

The leader node calls `extfn_get_library_version` before checking version compatibility. If `extfn_get_library_version` is not implemented on the leader node, then there is no distribution. If `extfn_get_library_version` is implemented on the leader node, then the UDF or TPF is eligible for distribution. Being eligible for distribution is not a guarantee that distributed query processing will occur.

The `extfn_get_library_version` method can return a 0-length string; however, this does not mean that `extfn_get_library_version` is not implemented.

Note: A TPF or UDF is still eligible for distribution if `extfn_get_library_version` returns a 0-length string.

If `extfn_get_library_version` returns a 0-length string, whether or not the worker node accepts the distributed work depends on the `extfn_check_version_compatibility` implementation on the worker node. A worker node requires a compatible library to process distributed work.

See also

- *Library Version (extfn_get_library_version)* on page 17
- *Setting the Dynamic Library Interface* on page 15

License Information (extfn_get_license_info)

If you are a design partner, implement the `extfn_get_license_info` library-level function to enable the server to obtain licensing information from a v4 UDF.

Data Type

`an_extfn_license_info`

Implementation

```
(_entry an_extfn_get_license_info) ( an_extfn_license_info
**license_info );
```

Parameters

license_info is an output parameter that returns the license information as received from the library. You define the license information in the `a_v4_extfn_license_info` structure.

Description
Design partners must specify the SAP-supplied license key in the `a_v4_extfn_license_info` structure, and must ensure that the library exports `extfn_get_license_info` as an entry point.

Adding the extfn_get_license_info Method

If you are a design partner, populate strings in `a_v4_extfn_license_info` and define `extfn_get_license_info` as a v4 entry point.

1. In the `a_v4_extfn_license_info` structure, specify your company name. The maximum length is 255 characters.
2. In the `a_v4_extfn_license_info` structure, specify additional library information such as library version and build numbers. The maximum length is 255 characters.
3. In the `a_v4_extfn_license_info` structure, enter the license key provided by SAP.
4. Ensure the library exports `extfn_get_license_info` as an entry point.

```
a_v4_extfn_license_info my_info = {
    1,
    "Company Name",
    "Library Info String",
    (void *)"KEY_STRING"
};

void SQL_CALLBACK extfn_get_license_info( an_extfn_license_info
**license_info )
/
************************************************************************
**********************/
{
    *license_info = (an_extfn_license_info *)& my_info;
}
```

Compile and Link Source Code to Build Dynamically Linkable Libraries

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

Warning! Use fully-qualified path names for UDF libraries. In multiplex implementations, ensure the relative path is the same for all nodes.

1. A UDF dynamically linkable library must include an implementation of the function **extfn_use_new_api()**. The source code for this function is in *Setting the dynamic library interface* on page 15. This function informs the server of the API style that all functions in

the library adhere to. The sample source file `my_main.cxx` contains this function; you can use it without modification.

2. A UDF dynamically linkable library must also contain object code for at least one UDF function. A UDF dynamically linkable library may optionally contain multiple UDFs.
3. Link together the object code for each UDF as well as the **extfn_use_new_api()** to form a single library.

   For example, to build the library "libudfex:"

   - Compile each source file to produce an object file:

     ```
     my_main.cxx
     my_bit_or.cxx
     my_bit_xor.cxx
     my_interpolate.cxx
     my_plus.cxx
     my_plus_counter.cxx
     my_sum.cxx
     my_byte_length.cxx
     my_md5.cxx
     my_toupper.cxx
     tpf_agg.cxx
     tpf_blob.cxx
     tpf_dt.cxx
     tpf_filt.cxx
     tpf_oby.cxx
     tpf_pby.cxx
     tpf_rg_1.cxx
     tpf_rg_2.cxx
     udf_blob.cxx
     udf_main.cxx
     udf_rg_1.cxx
     udf_rg_2.cxx
     udf_rg_3.cxx
     udf_utils.cxx
     ```

   - Link together each object produced into a single library.

   After the dynamically linkable library has been compiled and linked:

   - Update the **CREATE FUNCTION ... EXTERNAL NAME** or **CREATE PROCEDURE ... EXTERNAL NAME** to include an explicit path name for the UDF library.
4. Run `iqdir16/samples/udf/build.bat` on Windows. Run `iqdir16/samples/udf/build.sh` on UNIX.

Compiling and Linking the Sample UDFs for Windows

Run the `build.bat` script to compile and link the sample scalar and aggregate UDFs, table UDFs, and TPFs found in the `samples\udf` directory.

1. Navigate to `%ALLUSERSPROFILE%\samples\udf`.
2. Run `build.bat`:

Parameter	Description
-clean	Deletes the object and the build directory
-v3	Builds sample scalar and aggregate UDFs with the v3 API
-v4	(Default) Builds sample table UDFs and TPFs with the v4 API

Compiling and Linking the Sample UDFs for UNIX

Run the `build.sh` script to compile and link the sample scalar and aggregate UDFs, table UDFs, and TPFs found in the `samples/udf` directory.

1. Navigate to `$IQDIR15/samples/udf`.
2. Run `build.sh`:

Parameter	Description
-clean	Deletes the object and the build directory
-v3	Builds sample scalar and aggregate UDFs with the v3 API
-v4	(Default) Builds sample table UDFs and TPFs with the v4 API

AIX Switches

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

xlC `10.0` on a PowerPC

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

Note: To compile on AIX 6.1 systems, the minimum level of the xlC compiler is `10.0`.

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 6.24 on Itanium

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

compile switches

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

link switches

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

Linux Switches

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

g++ 4.1.1 on x86

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

compile switches

```
-fPIC -fsigned-char -fno-exceptions -pthread -fno-omit-frame-
pointer
-Wno-deprecated -Wno-ctor-dtor-privacy -O2 -Wall
```

Note: When compiling C++ applications for building shared libraries on Linux, adding the **-O2** and **-Wall** switches to the list of compile UDF switches decreases computation time.

link switches

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

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

Examples

- *Example 1*

  ```
  g++ -c my_interpolate.cxx -fPIC -fsigned-char -fno-exceptions -
  pthread
        -fno-omit-frame-pointer -Wno-deprecated -Wno-ctor-dtor-
  privacy
        -I${IQDIR16}/sdk/include/
  ```

- *Example 2*

  ```
  g++ -c my_main.cxx -fPIC -fsigned-char -fno-exceptions -pthread
        -fno-omit-frame-pointer -Wno-deprecated -Wno-ctor-dtor-
  ```

```
privacy
      -I${IQDIR16}/sdk/include/
```

- *Example 3*

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

xlC `10.0` on a PowerPC

compile switches

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

link switches

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

Solaris Switches

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

Sun Studio 12 on SPARC

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

compile switches

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

link switches

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

Sun Studio 12 on x86

compile switches

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

link switches

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

Windows Switches

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

Visual Studio 2008 on x86

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

compile and link switches

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

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

Example

Environment setup

```
set VCBASE=c:\dev\vc9
      set MSSDK=C:\dev\mssdk6.0a
      set IQINSTALLDIR=C:\Sybase\IQ
      set OBJ_DIR=%IQINSTALLDIR%\IQ-16_0\samples\udf\objs
      set SRC_DIR=%IQINSTALLDIR%\IQ-16_0\samples\udf\src
      call %VCBASE%\VC\bin\vcvars32.bat
```

- *Example 1*

```
%VCBASE%\VC\bin\amd64\cl -c -nologo -DNDEBUG -DWINNT -D_USRDLL
      -D_WINDLL -D_WIN64 -DWIN64 -
D_WIN32_WINNT=_WIN32_WINNT_WINXP
     -DWINVER=_WIN32_WINNT_WINXP -D_MBCS -GS -W3 -Zi -favor:AMD64
      -DSYB_LP64 -D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=64 -
DHMSWNT
      -D_CRT_SECURE_NO_DEPRECATE -D_CRT_NONSTDC_NO_DEPRECATE
      -DPOINTERS_ARE_64BITS -DLONG_IS_64BITS -
D_RWSTD_NO_EXCEPTIONS
      -I"%VCBASE%\VC\include" -I"%MSSDK%\include "-I"%MSSDK%\Lib
\AMD64"
      -I"%VCBASE%\VC\lib\amd64" -DMSDCXX -DINT64_WORKAROUND
      -DSUPPORTS_UDAF -Od -Zi -MD -I"%IQINSTALLDIR%\IQ-16_0\sdk
\include"
     -Fo"%OBJ_DIR%\my_interpolate.o" %SRC_DIR%\my_interpolate.cxx
```

- *Example 2*

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

```
D_RWSTD_NO_EXCEPTIONS
      -I"%VCBASE%\VC\include" -I"%MSSDK%\include "-I"%MSSDK%\Lib
\AMD64"
      -I"%VCBASE%\VC\lib\amd64" -DMSDCXX -DINT64_WORKAROUND
      -DSUPPORTS_UDAF -Od -Zi -MD -I"%IQINSTALLDIR%\IQ-16_0\sdk
\include"
      -Fo"%OBJ_DIR%\my_main.o" %SRC_DIR%\my_main.cxx
```

- *Example 3*

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

- *Example 4*

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

Testing User-Defined Functions

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

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

Enabling and Disabling User-Defined Functions

Use the `inmemory_external_procedure` security feature to enable or disable the server's ability to make use of high performance in-process UDFs.

A database should maintain data integrity. Under no circumstances should data be lost, modified, augmented, or corrupted. Since UDF execution happens within the SAP Sybase IQ server, there is a risk of corrupting data; practice caution with memory management and any other use of pointers. Install and execute UDFs within a read-only multiplex node. For additional protection, use the secured feature (**-sf**) startup option with each server to enable or disable the execution of UDF.

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

Administrators can enable v3 and v4 UDFs for any server by specifying this in the server startup command or in the configuration file:

```
-sf -inmemory_external_procedure
```

Administrators can disable v3 and v4 UDFs for any server by specifying this in the server startup command or in the configuration file:

```
-sf inmemory_external_procedure
```

Initially Executing a User-Defined Function

To ensure the safest environment possible, install and invoke UDFs from a read-only server node in a multiplex installation.

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

- **Libraries using the stored procedure SA_EXTERNAL_LIBRARY_UNLOAD –** These libraries are not reloaded when the SAP Sybase IQ server is stopped and restarted.

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

Managing External Libraries

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

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

Note: Unload existing libraries from a running SAP Sybase IQ server before replacing the dynamic link library. The server may fail, if you do not unload the library. Before replacing a dynamically linkable library, either shut down the SAP Sybase IQ server or use the **sa_external_library_unload** function to unload the library.

For Windows, unload an external function library using:

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

For UNIX, unload an external function library using:

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

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

In Windows, use

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

In UNIX, use

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

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

Controlling Error Checking and Call Tracing

The **external_UDF_execution_mode** option controls the amount of error checking and call tracing that is performed when statements involving v3 and v4 external user-defined functions are evaluated.

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

Allowed Values
0, 1, 2

Default Value
0

Scope
Can be set as public, temporary, or user.

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

When set to 1, external UDFs are evaluated to validate the information passed back and forth to each UDF. This setting is intended for scalar and aggregate UDFs.

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. This setting is intended for all C or C++ external UDFs. Memory tracing is turned on for table UDFs and TPFs.

Viewing SAP Sybase IQ Log Files

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

Log files for the database are generally located with the database file and configuration file. On UNIX platforms, there are two files named after the database instance, one with a `.stderr` extension and one with a `.stdout` extension. On Windows, by default, the `stderr` file is not generated.

To capture the `stderr` messages along with the `stdout` messages under Windows, redirect the `stdout` and `stderr`:

```
iqsrv16.exe @iqdemo.cfg iqdemo.db 2>&1 > iqdemo.stdout
```

The Windows output messages are slightly different from the output messages generated on UNIX platforms.

Using Microsoft Visual Studio Debugger for User-Defined Functions

Microsoft Visual Studio 2008 developers use Microsoft Visual Studio Debugger to step through the user-defined function code.

1. Attach the debugger to a running server:
   ```
   devenv /debugexe "%IQDIR16%\Bin64\iqsrv16.exe"
   ```
2. Goto Debug | Attach to Process
3. To start the server and debugger together:
   ```
   devenv /debugexe "%IQDIR16%\bin32\iqsrv16.exe" [commandline
   options for your server]
   ```

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

Modifying the UDF at Runtime

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

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

1. Ensure all users who invoke UDFs do not have any pending queries in progress

2. Revoke the execute privilege from users, and drop the SQL functions and stored procedures which reference external UDF code modules
3. Unload the library from the SAP Sybase IQ server, using the **call sa_external_library_unload** command (shutting down the IQ server also automatically unloads the library).
4. Perform the desired maintenance on the external library files (copy, move, update, delete).
5. Edit SQL function and stored procedure definitions in the registration scripts to reflect external library locations, if the libraries were moved.
6. Grant the execute privilege to users, and run registration scripts to re-create the SQL functions and stored procedures which reference external UDF code modules.
7. Invoke a SQL function or stored procedure that references the external UDF code to ensure the SAP Sybase IQ server can dynamically load the external library.

Granting the Privilege To Run a Procedure

Grant the privilege to execute or call a procedure.

Prerequisites

At least one of these conditions:

- You created the table.
- You have been granted privileges on the table with the ADMIN OPTION.
- You have been granted the EXECUTE ANY PROCEDURE system privilege.
- You have been granted LOAD and TRUNCATE object privileges.
- You have been granted the MANAGE ANY OBJECT PRIVILEGE system privilege. If the LOAD or TRUNCATE object privilege is granted using the **WITH GRANT OPTION** clause, the grantee can then grant the object privilege to other users, but is limited to those tables specified in the original GRANT statement. Under this scenario, the grantee does not need the MANAGE ANY OBJECT PRIVILEGE system privilege.

Task

Procedures execute with the privileges of their owner. Any procedure that updates information on a table executes successfully only if the owner of the procedure has UPDATE privileges on the table.

As long as the procedure owner has the proper privileges, the procedure executes successfully when called by any user assigned privilege to execute it, whether or not he or she has privileges on the underlying table. You can use procedures to allow users to carry out well-defined activities on a table, without having any general privileges on the table.

To grant the **EXECUTE** privilege, enter:

```
GRANT EXECUTE ON procedure_name
TO usreID
```

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 procedure, or a user with the DROP ANY PROCEDURE or DROP ANY OBJECT system privilege, can drop a function or procedure from the database.

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

```
DROP FUNCTION fullname
```

To remove a table UDF or TPF named *fullname* from the database, enter:

```
DROP PROCEDURE fullname
```

Scalar and Aggregate UDFs

Scalar and aggregate user-defined functions return a single value to the calling environment.

Note: Scalar and aggregate UDFs are a licensable option, and require the IQ_UDF or IQ_IDA license. Installing the license enables user-defined functions.

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

Scalar and Aggregate UDF Restrictions

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

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

Creating a Scalar or Aggregate UDF

Learn how to create and configure external C or C++ scalar and aggregate user-defined functions.

1. Declare the UDF to the server by using the **CREATE FUNCTION** or **CREATE AGGREGATE FUNCTION** statements. Write and execute these statements as commands, or use Sybase Control Center.

   The external C/C++ form of the CREATE FUNCTION statement requires the CREATE EXTERNAL REFERENCE system privilege. Therefore, standard users do not have the authority to declare any UDFs of this type.
2. *Write the UDF library identification function* on page 15.
3. Define the UDF as a set of C or C++ functions. See *Defining a scalar UDF* on page 37 or *Defining an aggregate UDF* on page 53.
4. Implement the function entry points in C/C++.
5. *Compile the UDF functions and the library identification functions* on page 19.
6. Link the compiled file into a dynamically linkable library.

Any reference to a UDF in a SQL statement first, if necessary, links the dynamically linkable library. The *calling patterns* on page 82 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 SAP Sybase IQ server can explicitly *enable or disable this functionality* on page 25.

Declaring and Defining Scalar User-Defined Functions

SAP 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 `$IQDIR16/samples/udf`. You can also find them in the `$IQDIR16/lib64/libudfex` dynamically linkable library.

Declaring a Scalar UDF

The system privileges required to declare an in-process external UDF vary depending on the owner of the UDF. There is also a server startup option that allows an administrator to enable or disable this style of user-defined function.

To declare an in-process external UDF owned by themselves, a user requires both the CREATE PROCEDURE and CREATE EXTERNAL REFERENCE system privileges. To declare an in-process external UDF which is owned by another user requires either the CREATE ANY PROCEDURE or CREATE ANY OBJECT system privilege, as well as the CREATE EXTERNAL REFERENCE system privilege.

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

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

Note: To declare a UDF function owned by themselves, a user must have the CREATE PROCEDURE system privilege. To declare a UDF function owned by others requires either the CREATE ANY PROCEDURE or CREATE ANY OBJECT system privilege . If the UDF function contains an external reference, the CREATE EXTERNAL REFERENCE system privilege is also required, regardless of who declares the UDF function.

The syntax for creating a 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, use a fully qualified path name to a secure directory for the library name portion of the EXTERNAL NAME clause.

SQL Security

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

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

External Name

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

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

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

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

See also

- *Defining a Scalar UDF* on page 37

UDF Example: my_plus Declaration

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

my_plus declaration

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

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

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

section. That descriptor structure contains the required and optional function pointers that embody the implementation of this UDF.

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

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

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

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

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

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

```
SELECT my_plus(t.x, t.y), count(*)
FROM t
WHERE t.z = 2
AND my_plus(t.x, 5) > 10
AND my_plus(t.y, 5) > 10
GROUP BY my_plus(t.x, t.y)
```

UDF Example: my_plus_counter Declaration

The `my_plus_counter` example is a simple nondeterministic scalar UDF that takes a single integer argument, and returns the result of adding that argument value to an internal

integer usage counter. If the input argument value is NULL, the result is the current value of the usage counter.

my_plus_counter declaration

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

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

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

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

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

SAP Sybase 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
```

UDF Example: my_byte_length Declaration

my_byte_length is a simple scalar user-defined function that returns the size of a column in bytes.

my_byte_length declaration

When **my_byte_length** resides within the dynamically linkable library my_shared_lib, the declaration for this example is:

```
CREATE FUNCTION my_byte_length(IN arg1 LONG BINARY)
//   RETURNS UNSIGNED INT
//   DETERMINISTIC
//   IGNORE NULL VALUES
//   EXTERNAL NAME 'my_byte_length@libudfex'
```

This declaration says that **my_byte_length** is a simple scalar UDF residing in my_shared_lib with a descriptor routine named describe_my_byte_length. 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 descriptor structure contains the required and optional function pointers that embody the implementation of this UDF.

This declaration also says that **my_byte_length** accepts one `LONG BINARY` argument and returns an `UNSIGNED INT` result value.

Note: Large object data support requires a separately licensed SAP Sybase IQ option.

The declaration states that this function is deterministic, which 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 this 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_byte_length** function follows that convention, and therefore this UDF routine is not actually invoked when either of its input values is 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.

This example query with **my_byte_length** in the **SELECT** list returns a column with one row for each row in `exTable`, with an `INT` representing the size of the binary file:

```
SELECT my_byte_length(exLOBColumn)
FROM exTable
```

Defining a Scalar UDF

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

- **extfnapiv3.h** – inclusion of the UDF interface definition header file.
- **_evaluate_extfn** – An evaluation function. All evaluation functions take two arguments:
  - an instance of the scalar UDF context structure that is unique to each usage of a UDF that contains a set of callback function pointers, and a pointer where a UDF can store UDF-specific data.

    - a pointer to a data structure that allows access to the argument values and to the result value through the supplied callbacks.
- **a_v3_extfn_scalar** – an instance of the scalar UDF descriptor structure that contains a pointer to the evaluation function.
- **Descriptor function** – returns a pointer to the scalar UDF descriptor structure.

These parts are optional:

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

See also

- *Declaring a Scalar UDF* on page 33

Scalar UDF Descriptor Structure

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

```
typedef struct a_v3_extfn_scalar {    //
    // Metadata descriptor for a scalar UDF
    // supplied by the UDF library to the server
    // An optional pointer to an initialize function
    void (*_start_extfn)(a_v3_extfn_scalar_context * cntxt);
    //
    // An optional pointer to a shutdown function
    void (*_finish_extfn)(a_v3_extfn_scalar_context *    cntxt);
    //
    // A required pointer to a function that will be
    // called for each invocation of the UDF on a
    // new set of argument values
    void (*_evaluate_extfn)(a_v3_extfn_scalar_context * cntxt, void
*args_handle);
    // RESERVED FIELDS MUST BE INITIALIZED TO NULL
 void  *reserved1_must_be_null;
 void  *reserved2_must_be_null;
 void  *reserved3_must_be_null;
 void  *reserved4_must_be_null;
 void  *reserved5_must_be_null;
                    …

} a_v3_extfn_scalar;
```

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

structure should be the null pointer. Similarly, if the shutdown function is not supplied, the corresponding value in the descriptor structure should be the null pointer.

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

Scalar UDF Context Structure

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

```
typedef struct a_v3_extfn_scalar_context {
//-------- Callbacks available via the context --------
//
  short (SQL_CALLBACK *get_value)(
    void           *arg_handle,
              a_sql_uint32    arg_num,
              an_extfn_value *value
              );
 short (SQL_CALLBACK *get_piece)(
              void *          arg_handle,
              a_sql_uint32    arg_num,
              an_extfn_value *value,
              a_sql_uint32    offset
              );
 short (SQL_CALLBACK *get_value_is_constant)(
              void *          arg_handle,
              a_sql_uint32    arg_num,
              a_sql_uint32 *  value_is_constant
              );
 short (SQL_CALLBACK *set_value)(
              void *          arg_handle,
              an_extfn_value *value,
              short              append
              );
 a_sql_uint32 (SQL_CALLBACK *get_is_cancelled)(
              a_v3_extfn_scalar_context * cntxt
              );
 short (SQL_CALLBACK *set_error)(
              a_v3_extfn_scalar_context * cntxt,
              a_sql_uint32    error_number,
              const char *    error_desc_string
              );
 void (SQL_CALLBACK *log_message)(
              const char *msg,
              short msg_length
                 );
 short (SQL_CALLBACK *convert_value)(
                 an_extfn_value   *input,
                 an_extfn_value   *output
 //---------- Data available from the context ----------
 void * _user_data;   // read-write field
 //---------- For Server Internal Use Only -------------
```

```
 void  * _for_server_internal_use;
} a_v3_extfn_scalar_context;
```

Note: The **get_piece** callback is valid in v3 and v4 scalar and aggregate UDFs. For v4 table UDFs and TPFs, use the Blob (`a_v4_extfn_blob`) and Blob Input Stream (`a_v4_extfn_blob_istream`) structures instead.

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

The rest of the scalar UDF context structure is filled with the set of callback functions, supplied by the engine, for use within each of the user's UDF functions. Most of these callback functions return a success status through a short result value; a true return indicates success. Well-written UDF implementations should never cause a failure status, but during development (and possibly in all debug builds of a given UDF library), 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;
  ```

Table 1. Scalar External Function Context: a_v3_extfn_scalar_context

Method of a_v3_extfn_scalar_context structure	Description
void set_cannot_be_distributed(a_v3_extfn_scalar_context * cntxt)	Distribution can be disabled at the UDF level, even if distribution criteria are met at the library level. By default, the UDF is assumed to be distributable if the library is distributable. It is the responsibility of the UDF to push the decision to disable distribution to the server.

See also

- *Blob (a_v4_extfn_blob)* on page 199
- *Blob Input Stream (a_v4_extfn_blob_istream)* on page 203

Example: my_plus Definition

The definition for the my_plus scalar UDF 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
```

Example: my_plus _counter Definition

This scalar UDF 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
```

Example: my_byte_length Definition

The **my_byte_length** scalar UDF example computes the size of a column by streaming the data in piece by piece, then returns the size of the column in bytes.

my_byte_length definition

Note: Large object data support requires a separately licensed SAP Sybase IQ option.

```
#include "extfnapiv4.h"
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <assert.h>


//  A simple function that returns the size of a cell value in bytes
//
//    CREATE FUNCTION my_byte_length(IN arg1 LONG BINARY)
//            RETURNS UNSIGNED INT
//            DETERMINISTIC
//          IGNORE NULL VALUES
//            EXTERNAL NAME 'my_byte_length@libudfex'

#if defined __cplusplus
extern "C" {
#endif

static void my_byte_length_evaluate(a_v3_extfn_scalar_context
*cntxt,
                      void *arg_handle)
{

    if (cntxt == NULL || arg_handle == NULL)
    {
        return;
    }

    an_extfn_value  arg;
    an_extfn_value  outval;

    a_sql_uint64 total_len;

    //  Get first argument
    a_sql_uint32 fetchedLength = 0;
    (void) cntxt->get_value( arg_handle, 1, &arg );
    if (arg.data == NULL)
    {
        return;
    }

    fetchedLength += arg.piece_len;

    // saving total length as it loses scope inside get_piece
    total_len = arg.len.total_len;
```

```
    while (fetchedLength < total_len)
    {
       (void) cntxt->get_piece( arg_handle, 1, &arg, fetchedLength );
        fetchedLength += arg.piece_len;
    }

    //if this fails, the function did not get the full data from the
cell
    assert(fetchedLength == total_len);

    outval.type = DT_UNSINT;
    outval.piece_len = 4;
    outval.data = &fetchedLength;
    cntxt->set_value(arg_handle, &outval, 0);
}

static a_v3_extfn_scalar my_byte_length_descriptor = {
    0,
    0,
    &my_byte_length_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_byte_length()
{
  return &my_byte_length_descriptor;
}

#if defined __cplusplus
}
#endif
```

See also

- *Example: my_byte_length Definition* on page 45

Declaring and Defining Aggregate UDFs

SAP Sybase IQ supports aggregate UDFs. 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 aggregate UDFs that can be used anywhere the SUM aggregate can be used.

Note: The aggregate UDF examples referenced here are installed with the server, and can be found as .cxx files in `$IQDIR16/samples/udf`. You can also find them in the `$IQDIR16/lib64/libudfex` dynamically linkable library.

An aggregate function can produce either a single result, or a set of results. The number of data points in the output result set may not necessarily match the number of data points in the input set. Multiple-output aggregate UDFs must use a temporary output file to hold the results.

Declaring an Aggregate UDF

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

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

When implementing an aggregate UDF, 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 an aggregate UDF reflects these usage decisions.

The syntax for creating 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
        | WINDOW FRAME
        { { ALLOWED | REQUIRED }
            [ window-frame-constraints ... ]
          | NOT ALLOWED }
    | ON EMPTY INPUT RETURNS { NULL | VALUE }
  -- Call or skip function on NULL inputs

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

restrict:   { [ NOT ] ALLOWED } | REQUIRED

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

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

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

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

The DUPLICATE INSENSITIVE option allows the optimizer to consider removing the duplicates without affecting the result, giving the optimizer the choice on how to execute the query. Write the aggregate UDF 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, an aggregate UDF is assumed to be usable as both a simple aggregate and as an OLAP-style aggregate with any kind of window frame.

For an aggregate UDF 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 aggregate UDFs 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 an aggregate UDF 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 an aggregate UDF 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.

The 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.

See also

- *Defining an Aggregate UDF* on page 53
- *Context Storage of Aggregate User-Defined Functions* on page 81

Example: my_sum Declaration

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

Example: my_bit_xor Declaration

The “my_bit_xor” example is analogous to the SAP Sybase SQL Anywhere® 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.

Example: my_bit_or Declaration

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

my_bit_or declaration

The resulting declaration looks like:

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

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

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

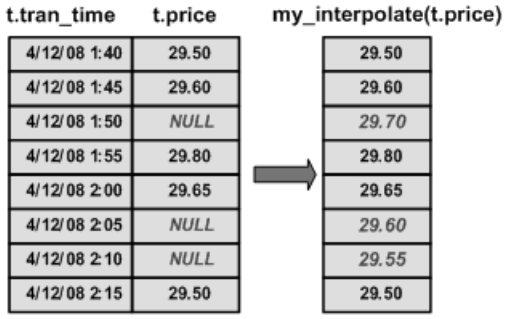
Example: my_interpolate Declaration

The “my_interpolate” example is an OLAP-style UDAF that attempts to fill in any missing values in a sequence (where missing values are denoted by NULLs) by performing linear interpolation across any set of adjacent NULL values to the nearest non-NULL value in each direction.

my_interpolate declaration

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

Figure 1: my_interpolate results



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

The resulting UDAF declaration looks like this:

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

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

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

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

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

Defining an Aggregate UDF

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

- **extfnapiv3.h** – the UDF interface definition header file. The file is `extfnapiv4.h` for the v4 API.
- **_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 an aggregate UDF. 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 aggregate UDF context structure that is unique to each usage of an aggregate UDF.
- **_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 aggregate UDF context structure that is unique to each usage of an aggregate UDF.
- **_next_value_extfn** – a function called for each new set of input arguments. _next_value_extfn takes two arguments:
  - A pointer to the aggregate UDF 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 aggregate UDF context structure, and
  - An args_handle.
- **a_v3_extfn_aggregate** – an instance of the aggregate UDF descriptor structure that contains the pointers to all of the supplied functions for this UDF.
- **Descriptor function** – a descriptor function that returns a pointer to that aggregate UDF descriptor structure.

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

- **_drop_value_extfn** – an optional function pointer that is called for each input set of argument values that has fallen out of a moving window frame. This function should not set the result of the aggregation. Use the get_value callback function to access the input argument values, and, if necessary, through repeated calls to the get_piece callback function.

  Set the function pointer to the null pointer if:

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

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

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

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

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

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

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

  Examples of such partitionable aggregates include:

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

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

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

  Access to the subaggregate input value is through the normal get_value callback function. Direct communication between subaggregates and the superaggregate is impossible; the server handles all such communication. The sub-aggregates and the super-aggregate do

not share a context structure. Instead, individual sub-aggregates are treated exactly the same as nonpartitioned aggregates. The independent super-aggregate sees a calling pattern that looks like this:

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

Or like this:

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

If neither _evaluate_superaggregate_extfn or _next_subaggregate_extfn is supplied, then the aggregate UDF 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.

See also

- *Declaring an Aggregate UDF* on page 47
- *Context Storage of Aggregate User-Defined Functions* on page 81
- *Blob (a_v4_extfn_blob)* on page 199
- *Blob Input Stream (a_v4_extfn_blob_istream)* on page 203

Aggregate UDF Descriptor Structure

The aggregate UDF descriptor structure comprises several pieces.

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

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

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

```
void (*_finish_extfn)(a_v3_extfn_aggregate_context *);
```

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

```
void (*_reset_extfn)(a_v3_extfn_aggregate_context *);
```

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

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

Note: The **get_piece** callback is valid in v3 and v4 scalar and aggregate UDFs. For v4 table UDFs and TPFs, use the Blob (`a_v4_extfn_blob`) and Blob Input Stream (`a_v4_extfn_blob_istream`) structures instead.

- **_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.

Note: The **get_piece** callback is valid in v3 and v4 scalar and aggregate UDFs. For v4 table UDFs and TPFs, use the Blob (`a_v4_extfn_blob`) and Blob Input Stream (`a_v4_extfn_blob_istream`) structures instead.

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);
  ```

Note: The **get_piece** callback is valid in v3 and v4 scalar and aggregate UDFs. For v4 table UDFs and TPFs, use the Blob (`a_v4_extfn_blob`) and Blob Input Stream (`a_v4_extfn_blob_istream`) structures instead.

- **_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 Aggregate UDF 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.

Note: The **get_piece** callback is valid in v3 and v4 scalar and aggregate UDFs. For v4 table UDFs and TPFs, use the Blob (`a_v4_extfn_blob`) and Blob Input Stream (`a_v4_extfn_blob_istream`) structures instead.

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

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

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

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

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

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

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

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

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

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

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

  ```
  a_sql_uint32  indicators;
  ```

- **_calculation_context_size** – the number of bytes for the server to allocate for each UDF calculation context. The server may allocate multiple calculation contexts during query

processing. The currently active group context is available in a_v3_extfn_aggregate_context_user_calculation_context.

```
short _calculation_context_size;
```

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

  ```
  short _calculation_context_alignment;
  ```

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

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

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

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

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

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

See also

- *Blob (a_v4_extfn_blob)* on page 199
- *Blob Input Stream (a_v4_extfn_blob_istream)* on page 203

Calculation Context

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

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

The _user_calculation_context field is a server-allocated memory pointer, created by the server for each concurrent processing group. The server ensures that the _user_calculation_context always points to the correct calculation context for the group of rows currently being processed. Between UDF API calls, depending on the data, the server

may allocate new _user_calculation_context values. The server may save and restore calculation context areas to disk while processing a query.

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

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

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

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

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

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

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

Aggregate UDF Context Structure

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

In addition, it has a read/write _user_data pointer just like the scalar UDF context, and a set of read-only data fields that describe the current usage and location. Each unique instance of the UDF within a statement has one aggregate UDF context instance that is passed to each of the

functions specified within the aggregate UDF descriptor structure when they are called. The aggregate context structure is defined as:

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

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

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

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

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

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

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

- **Determining whether the statement was interrupted** – If a UDF entry point performs work for an extended period of time (many seconds), then it should, if possible, call the get_is_cancelled callback every second or two to see if the user has interrupted the current statement. If the statement has been interrupted, a nonzero value is returned, and the UDF

entry point should then immediately perform. Eventually, the _finish_extfn function is called to do any necessary cleanup, but no other UDF entry points are subsequently called.

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

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

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

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

```
void (SQL_CALLBACK *log_message)(
  const char *msg,
        short msg_length
  );
```

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

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

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

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

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

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

```
void  * _user_data;
```

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

  ```
  void  * _user_calculation_context;
  ```

- **Other available aggregate information** – Available at all external function entry points, including start_extfn. Zero indicates an unknown or not-applicable value. Estimated average number of rows per partition or group.
  - **a_sql_uint64 _max_rows_in_frame;** – Calculates the maximum number of rows defined in the window frame. For range-based windows, this indicates unique values. Zero indicates an unknown or not-applicable value.
  - **a_sql_uint64 _estimated_rows_per_partition;** – Displays the estimated average number of rows per partition or group. 0 indicates an unknown or not-applicable value.
  - **a_sql_uint32 _is_used_as_a_superaggregate;** – Identifies whether this instance is a normal aggregate or a superaggregate. Returns a result of 0 if the instance is a normal aggregate.
- **Determining window specifications** – Window specifications if a window is present on the query:
  - **a_sql_uint32 _is_window_used;** – Determines if the statement is windowed.
  - **a_sql_uint32 _window_has_unbounded_preceding;** – A return value of 0 indicates the window does not have unbounded preceding.
  - **a_sql_uint32 _window_contains_current_row;** – A return value of 0 indicates the window does not contain the current row.
  - **a_sql_uint32 _window_is_range_based;** – If the return code is 1, the window is range-based. If the return code is 0, the window is row-based.
- **Available at reset_extfn() calls** – Returns the actual number of rows in current partition, or 0 for nonwindowed aggregate.

  ```
  a_sql_uint64 _num_rows_in_partition;
  ```

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

  ```
  a_sql_uint64 _result_row_from_start_of_partition;
  ```

- **Closing syntax** – Complete the context with:

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

Aggregate External Function Context: a_v3_extfn_aggregate_context

Method of a_v3_extfn_aggregate_context structure	Description
void set_cannot_be_distributed(a_v3_extfn_aggregate_context * cntxt)	Distribution can be disabled at the UDF level, even if distribution criteria are met at the library level. By default, the UDF is assumed to be distributable if the library is distributable. It is the responsibility of the UDF to push the decision to disable distribution to the server.

See also

- *Blob (a_v4_extfn_blob)* on page 199
- *Blob Input Stream (a_v4_extfn_blob_istream)* on page 203

Example: my_sum Definition

The aggregate UDF **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 "extfnapiv4.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 aggregate UDF, 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;
}
```

Example: my_bit_xor Definition

The aggregate UDF **my_bit_xor** example is similar to the SQL Anywhere built-in BIT_XOR, except **my_bit_xor** operates only on unsigned integers.

my_bit_xor definition

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

```
#include "extfnapiv4.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 aggregate UDF, 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
```

Example: my_bit_or Definition

The aggregate UDF **my_bit_or** example is similar to the SQL Anywhere 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 "extfnapiv4.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 aggregate UDF
//  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
```

Example: my_interpolate definition

The aggregate UDF **my_interpolate** example is an OLAP-style aggregate UDF 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 "extfnapiv4.h"
#include <stdlib.h>
#include <assert.h>

//  MY_INTERPOLATE
//
//  OLAP-style aggregate UDF that accepts a double precision
//  floating point argument.  If the current argument value is
//  not NULL, then the result value is the same as the
//  argument value.  On the other hand, if the current row's
//  argument value is NULL, then the result, where possible,
//  will be the arithmetic interpolation across the nearest
//  preceding and nearest following values that are not NULL.
//  In all cases the result is also a double precision value.
//
//  The start function creates a structure for maintaining the
//  argument values within the window including their NULLness.
```

```
//  The finish function then deallocates this structure.
//
//  Since there are some strict aggregate usage restrictions
//  for this aggregate (must be used with a row-based window
//  frame that includes the current row), the corresponding
//  SQL declaration will look like:
//
//      CREATE AGGREGATE FUNCTION my_interpolate(IN arg1 DOUBLE)
//                       RETURNS DOUBLE
//                       OVER REQUIRED
//                       WINDOW FRAME REQUIRED
//                              RANGE NOT ALLOWED
//                              PRECEDING REQUIRED
//                              UNBOUNDED PRECEDING NOT ALLOWED
//                              FOLLOWING REQUIRED
//                              UNBOUNDED FOLLOWING NOT ALLOWED
//                       EXTERNAL NAME  'my_interpolate@libudfex'


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



#if defined __cplusplus
extern "C" {
#endif


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

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


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

  // Make sure function was defined correctly
```

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

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


static void my_interpolate_finish(a_v3_extfn_aggregate_context
*cntxt)
{
  if (cntxt->_user_data) {
    my_window *cptr = (my_window *)cntxt->_user_data;
    if (cptr->_is_null) {
      free(cptr->_is_null);
      cptr->_is_null = 0;
    }
    if (cptr->_dbl_val) {
      free(cptr->_dbl_val);
      cptr->_dbl_val = 0;
    }
    free(cntxt->_user_data);
    cntxt->_user_data = 0;
```

```
  }
}


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

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

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


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

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


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

  an_extfn_value  outval;
  my_window *cptr = (my_window *)cntxt->_user_data;
  double  result;
  int     result_is_null = 1;
```

```
  double  preceding_value;
  int     preceding_value_is_null = 1;
  double  preceding_distance = 0;
  double  following_value;
  int     following_value_is_null = 1;
  double  following_distance = 0;
  int j;

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

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




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

```
     tmp_cell_num = ((curr_cell_num + cptr->_allocated_elem - j - 1)
                     % cptr->_allocated_elem);
      if (cptr->_is_null[tmp_cell_num] == 0) {
          preceding_value = cptr->_dbl_val[tmp_cell_num];
          preceding_value_is_null = 0;
          preceding_distance = j + 1;
          break;
      }
    }
    //  Finally, see what we can come up with for a result value
    //
    if (preceding_value_is_null && !following_value_is_null) {
      //
     //  No choice but to mirror the nearest following non-NULL value
      //  Example:
      //
      //    Inputs:  NULL    Result of my_interpolate:  40.0
      //             NULL                               40.0
      //             40.0                               40.0
      //
      result = following_value;
      result_is_null = 0;
      //
   } else if (!preceding_value_is_null && following_value_is_null) {
      //
     //  No choice but to mirror the nearest preceding non-NULL value
      //  Example:
      //
      //    Inputs:  10.0    Result of my_interpolate:  10.0
      //             NULL                               10.0
      //
      result = preceding_value;
      result_is_null = 0;
      //
   } else if (!preceding_value_is_null && !following_value_is_null)
{
      //
      //  Here we get to do real interpolation based on the
      //  nearest preceding non-NULL value, the nearest following
      //  non-NULL value, and the relative distances to each.
      //  Examples:
      //
      //    Inputs:  10.0    Result of my_interpolate:  10.0
      //             NULL                               20.0
      //             NULL                               30.0
      //             40.0                               40.0
      //
      //    Inputs:  10.0    Result of my_interpolate:  10.0
      //             NULL                               25.0
      //             40.0                               40.0
      //
      result = (  preceding_value
                + (  (following_value - preceding_value)
                     * (  preceding_distance
                          / (preceding_distance +
following_distance))));
```

```
      result_is_null = 0;
    }
  }

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


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

a_v3_extfn_aggregate *my_interpolate()
  { return &my_interpolate_descriptor; }

#if defined __cplusplus
  }
#endif
```

Context Storage of Aggregate User-Defined Functions

The context area is used to transfer or communicate data between multiple invocations of the UDF within the same query (particularly within OLAP-style queries).

Context variables control whether the intermediate results of aggregate functions are to be managed by the UDF itself (forcing the SAP Sybase IQ server to run the UDFs serially), or whether the memory is to be managed by the SAP Sybase IQ server.

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

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

For details on context storage, refer to the description of _calculation_context_size and _calculation_context_alignment in the section *Aggregate UDF descriptor structure* on page 55. These variables are near the end of the descriptor structure.

For a detailed discussion about the use of context storage, see *Calculation context* on page 59.

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

See also

- *Declaring an Aggregate UDF* on page 47
- *Defining an Aggregate UDF* on page 53

Calling Scalar and Aggregate UDFs

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

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

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

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

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

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

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

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

Scalar and Aggregate UDF Calling Patterns

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

Scalar and Aggregate UDF Callback Functions

The set of callback functions are supplied by the engine through the `a_v3_extfn_scalar_context` structure and used within the user's UDF functions.

- **get_value –** The function used within an evaluation method to retrieve the value of each input argument. For narrow argument data types (smaller than 256 bytes), a call to get_value is sufficient to retrieve the entire argument value. For wider argument data types, if the piece_len field within the an_extfn_value structure passed to this callback comes back with a value smaller than the value in the total_len field, use the get_piece callback to retrieve the rest of the input value.
- **get_piece –** The function used to retrieve subsequent fragments of a long argument input value.

  Note: The **get_piece** callback is valid in v3 and v4 scalar and aggregate UDFs. For v4 table UDFs and TPFs, use the Blob (`a_v4_extfn_blob`) and Blob Input Stream (`a_v4_extfn_blob_istream`) structures instead.
- **get_is_constant –** A function that determines whether the specified input argument value is a constant. This can be useful for optimizing a UDF, for example, where work can be performed once during the first call to the _evaluate_extfn function, rather than for every evaluation call.

- **set_value –** The function used within an evaluation function to tell the server the result value of the UDF for this call. If the result data type is narrow, one call to set_value is sufficient. However, if the result data value is wide, then multiple calls to set_value are required to pass the entire value, and the append argument to the callback should be true for each fragment except the last. To return a NULL result, the UDF should set the data field within the result value's an_extfn_value structure to the null pointer.
- **get_is_cancelled –** A function to determine whether the statement has been cancelled. If a UDF entry point is performing work for an extended period of time (many seconds), then it should, if possible, call the get_is_cancelled callback every second or two to see if the user has interrupted the current statement. The return value is 0 if the statement has not been interrupted.

  SAP Sybase IQ can handle extremely large data sets, and some queries can run for long periods of time. Occasionally, a query takes an unusually long time to execute. The SQL client lets the user cancel a query if it is taking too long to complete. Native functions track when a user has canceled a query. UDFs must also be written in a manner that tracks whether a query has been canceled by the user. In other words, UDFs should support the ability for users to cancel long-running queries that invoke UDFs.
- **set_error –** A function that can be used to communicate an error back to the server, and eventually to the user. Call this callback routine if a UDF entry point encounters an error that should result in an error message being sent back to the user. When called, set_error rolls back the current statement, and the user receives `Error from external UDF: error_desc_string`, and the SQLCODE is the negated form of the supplied error_number. To avoid collisions with existing errors, UDFs should use error_number values between 17000 and 99999. The maximum length of “error_desc_string” is 140 characters.
- **log_message –** The function used to send a message to the server's message log. The string must be a printable text string no longer than 255 bytes.
- **convert_value –** The function allows data conversion between data types. The primary use is the conversion between DT_DATE, DT_TIME, and DT_TIMESTAMP, and DT_TIMESTAMP_STRUCT. An input and output an_extfn_value is passed to the function.

See also

- *Scalar UDF Calling Pattern* on page 84
- *Aggregate UDF Calling Patterns* on page 84
- *Blob (a_v4_extfn_blob)* on page 199
- *Blob Input Stream (a_v4_extfn_blob_istream)* on page 203

Scalar UDF Calling Pattern

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

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

See also

- *Scalar and Aggregate UDF Callback Functions* on page 82
- *Aggregate UDF Calling Patterns* on page 84

Aggregate UDF Calling Patterns

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

The examples that follow this table definition:

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

The following abbreviation is used:

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

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

See also

- *Scalar and Aggregate UDF Callback Functions* on page 82
- *Scalar UDF Calling Pattern* on page 84

Simple Aggregate Ungrouped

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

Query

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

Calling pattern

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

Result

```
my_sum(a)
21
```

Simple Aggregate Grouped

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

Query

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

Calling pattern

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

Result

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

OLAP-Style Aggregate Calling Pattern with Unbounded Window

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

are set to 1 for _window_has_unbounded_preceding and _window_has_unbounded_following

Query

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

Calling pattern

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

Result

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

OLAP-Style Unoptimized Cumulative Window Aggregate

If _evaluate_cumulative_extfn is not supplied, this cumulative sum is evaluated through this 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 is supplied, this moving window sum is evaluated using this calling pattern, which is more efficient than using _drop_value_extfn.

Query

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

Calling pattern

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

Result

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

OLAP-Style Unoptimized Moving Window Following Aggregate

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

Query

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

Calling pattern

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

Result

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

OLAP-Style Optimized Moving Window Following Aggregate

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

Query

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

Calling pattern

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

Result

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

OLAP-Style Unoptimized Moving Window without Current

Assume the UDF my_sum works like the built-in SUM. If _drop_value_extfn function is not supplied, this moving window count is evaluated through the following calling pattern. This case is similar to the previous moving window examples, but the current row is not part of the window frame.

Query

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

Calling pattern

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

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

Result

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

OLAP-Style Optimized Moving Window without Current

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

Query

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

Calling pattern

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

```
_evaluate_extfn(cntxt, args)          returns 12
_finish_extfn(cntxt)
```

Result

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

External Function Prototypes

Define the API by a header file named `extfnapiv3.h` (`extfnapiv4.h` for the v4 API) in the subdirectory of your SAP 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.v4h`.

Each library should implement and export this function as:

```
unsigned int extfn_use_new_api(void)
    {
        return EXTFN_V4_API;
    }
```

The presence of this function, and that it returns `EXTFN_V4_API` informs the SAP Sybase IQ engine that the library contains UDFs written to the new API documented in this book.

Function prototypes

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

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

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

The `an_extfn_api` structure has this form:

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

Note: The **get_piece** callback is valid in v3 and v4 scalar and aggregate UDFs. For v4 table UDFs and TPFs, use the Blob (`a_v4_extfn_blob`) and Blob Input Stream (`a_v4_extfn_blob_istream`) structures instead.

The an_extfn_value structure has this form:

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

Notes

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

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

To return NULL, set data to NULL in `an_extfn_value`.

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

The header file itself contains additional notes.

See also

- *Blob (a_v4_extfn_blob)* on page 199
- *Blob Input Stream (a_v4_extfn_blob_istream)* on page 203

Table UDFs and TPFs

Table UDFs are external user-defined C, C++, or Java table functions. Unlike scalar and aggregate UDFs, table UDFs produce row sets as output. SQL queries consume a table UDF's output sets as table expressions.

Scalar and aggregate UDFs can use either the v3 or v4 API, but table UDFs can use only v4.

Declare a table UDF SQL function using the **CREATE PROCEDURE** statement. Scalar and aggregate UDFs use the **CREATE FUNCTION** statement.

Table parameterized functions (TPFs) are enhanced table UDFs that accept either scalar values or row sets as input.

See also

- *Table Parameterized Functions* on page 136
- *Declaring and Defining Scalar User-Defined Functions* on page 32
- *Declaring and Defining Aggregate UDFs* on page 46
- *Learning Roadmap: Types of External C and C++ UDFs* on page 6
- *Creating a Java Table UDF* on page 354

User Roles

Two types of users work with table UDFs: UDF developers, and SQL analysts.

- **UDF developer –** develops the table UDF in C or C++.
- **SQL analyst –** develops and analyzes the SQL queries that reference the table expression in the **FROM** clause. The table expression is the set of rows produced by the table UDF.

See also

- *Learning Roadmap for Table UDF Developers* on page 97
- *Learning Roadmap for SQL Analysts* on page 98

Learning Roadmap for Table UDF Developers

Use annotated examples to learn how to develop a C or C++ table UDF. After completing the development tasks, the SQL analyst can then reference your UDF in a SQL query.

This roadmap assumes:

- You have a C or C++ development environment on your machine.

- You are familiar with standard programming practices.

Task	See
Become familiar with table UDF and TPF restrictions.	*Table UDF Restrictions* on page 99
Create a table UDF.	*Developing a Table UDF* on page 103
(Optional) Define the library version validators for distributed query processing (DQP).	*Library Version (extfn_get_library_version)* on page 17 *Library Version Compatibility (extfn_check_version_compatibility)* on page 17
Compile and link source code.	*Compile and Link Source Code to Build Dynamically Linkable Libraries* on page 19
Declare the UDF to the server using the **CREATE PROCEDURE** statement. Write and execute these statements as commands, or use Sybase Control Center.	*Learning Roadmap for SQL Analysts* on page 98

Learning Roadmap for SQL Analysts

Reference a C or C++ table UDF in your SQL query.

Task	See:
Obtain the `.dll` or `.so` file (for example, `myudf.dll`) from the UDF developer. Place the `.dll` file in the `bin64` directory; place the `.so` file in the `lib64` or `LD_LIBRARY_PATH` directory.	Not applicable.
Define the **CREATE PROCEDURE** statement, referencing the `.dll` file and the callback function. For example: `CREATE PROCEDURE my_udf( IN num_row INT ) RESULT( id INT ) EXTERNAL NAME 'udf_rg_proc@myudf.dll'`	*CREATE PROCEDURE Statement (Table UDF)* on page 169
Select rows from the UDF. For example: `SELECT * FROM my_udf(5)`	*SELECT Statement* on page 188 *FROM Clause* on page 180

See also

- *SQL Reference for Table UDF and TPF Queries* on page 166

Table UDF Restrictions

Table UDFs and TPFs have some restrictions.

- The **TEMPORARY PROCEDURE** clause is not allowed for any external procedures. Attempting to create a temporary external procedure results in an error at creation time.
- The **NO RESULT SET** clause is not allowed. Table UDFs and TPFs must explicitly declare the contents of their results.
- If the optional **DYNAMIC RESULT SETS integer-expression** clause is specified, the value must be set to 1. Table UDFs and TPFs do not return multiple result sets.
- A table UDF or TPF cannot be referenced in a **CALL** SQL statement or **EXEC** embedded SQL statement. A table UDF or TPF can be referenced only in a **FROM** clause of a SQL statement.
- The **LANGUAGE** clause cannot be used for table UDFs or TPFs. If the **LANGUAGE** clause is present, syntax errors are reported at execution time.
- The **parameter** clause is limited to keyword **IN**; **INOUT** and **OUT** keywords are not supported for table UDFs or TPFs.
- The **EXTERNAL NAME** clause has the same syntax as scalar and aggregate UDFs.

Get Started

Familiarize yourself with sample files, concepts, and restrictions before developing table UDFs and TPFs.

Sample Files

Sample table UDF files are installed with the server. Use the samples as models when defining your own table UDFs.

Sample files are located in:

- %ALLUSERSPROFILE%\SybaseIQ\samples\udf (Windows)
- $SYBASE/IQ-16_0/samples/udf (UNIX)

File	Description
apache_log_reader.cxx	Implementation of a table UDF that reads an Apache log file and presents the rows from the file in table format. This UDF illustrates a real-world example of how you can use a UDF to make computer-generated data available to a SQL query writer in real time.
build.sh / build.bat	Script that compiles and links the sample scalar and aggregate UDFs, table UDFs, and TPFs found in the samples/udf directory.
my_md5.cxx	A simple deterministic scalar UDF that calculates the MD5 hash value of an input file (a LOB binary argument).
tpf_agg.cxx	Consumes rows from an input table, performs an aggregation on the input data, and returns rows back to the server.
tpf_blob.cxx	Implementation of a TPF that reads LOB data from an input table and passes the data to the result set, if an even number of the specified character or digit is present. This TPF illustrates how to read LOB data and how a user can pass LOB datatypes through to the result set.
tpf_dt.cxx	
tpf_filt.cxx	Illustrates how a TPF can be used to filter rows. The example uses the row block provided by caller and passes it to the input TABLE parameter. The input table schema must match the output result set of this function.
tpf_oby.cxx	Illustrates how a TPF can generate ordered output and pass it along.
tpf_pby.cxx	Illustrates how a TPF can generate partitioned output and pass it along.
tpf_rg_1.cxx	Similar to the table UDF sample udf_rg_2.cxx. It produces rows of data based on an input parameter.
tpf_rg_2.cxx	Builds upon the sample in tpf_rg_1.cxx, but uses fetch_into instead of fetch_block to read rows from the input table.

File	Description
udf_main.cxx	This file is linked into all of the examples and includes a common set of required entry points for the v4 API. This allows you to reuse the code rather than including it in each example.
udf_rg_1.cxx	A simple table UDF that generates rows of integer data.
udf_rg_2.cxx	A simple table UDF that generates rows of integer data that uses `describes` to ensure the schema defined in SQL matches the UDF's implementation. It also describes some optimizer attributes.
udf_rg_3.cxx	A simple table UDF that generates integer data in blocks of 100 using the `_fetch_block` fetch method.
udf_utils.cxx	A set of utility functions and macros that are useful to UDF/TPF developers. The examples rely on items in this file.
udf_utils.h	A set of utility functions and macros that are useful to UDF/TPF developers. The examples rely on items in this file.

Understanding Producers Versus Consumers

The server and UDF form a producer and consumer relationship when exchanging rows of data.

Production and consumption refer to table row data. The producer produces table rows; the consumer consumes table rows.

The server executes scalar and aggregate UDFs once for each matching row of a query. These UDFs consume input scalar parameters and produce, and return, a single scalar parameter. This data exchange occurs during the `evaluate` method using the `get_value()` and `set_value()` APIs.

However, scalar production and consumption is an inefficient method of data exchange if your UDF must produce or consume a table. Table UDFs that produce a table, and TPFs that consume a table, use the `row block` data structure of the v4 API. Row blocks allow for bulk row and column data exchange. The row block is populated by a producer, and read from by a consumer.

In this example, the table UDF `my_table_udf()` is a producer of data. SAP Sybase IQ, the server, is the consumer of the data:

```
SELECT * FROM my_table_udf()
```

In general, a table UDF is always a producer of data. The server, however, may not always be the consumer:

```
SELECT * FROM my_tpf( TABLE( SELECT * FROM my_table_udf() ) )
```

The outer TPF, `my_tpf()`, is the consumer for the table input parameter specified by **SELECT * from my_table_udf()**. SAP Sybase IQ is the consumer of the table produced by the `my_tpf()` TPF. A TPF, therefore, can be both a consumer and a producer.

The TPF does not have to consume from a table UDF. In this example, the TPF consumes the table data produced by the inner query, which is produced by the SAP Sybase IQ server:

```
SELECT * FROM my_tpf( TABLE( SELECT * FROM my_table where my_table.c1
< 10 ) )
```

In a TPF, therefore, SAP Sybase IQ can be both the consumer and producer of table data.

In the v4 API, a row block defines a memory area where data is produced to, and consumed from. In general, the layout of a row block conceptually matches the row and column format of the table; a row block consists of a number of rows, and each row consists of a number of columns. Either the producer or consumer must allocate the row block, and must also deallocate it when the time comes.

Rows and column have their own specific attributes that only apply to them. For example, rows have a status flag which indicates if the row is present or not. This flag lets a TPF change the row status without having to move the column data. Columns have a null mask that indicates if the data value is null or not. Row blocks also have some additional attributes: maximum number of rows, and current number of rows, for example. These row block attributes are useful when a UDF wants to create a row block to handle a large set of rows, but produce a smaller number of rows as required.

The process of consuming a row is handled via one of the two fetch APIs:

- `fetch_into`
- `fetch_block`

The `fetch_into` is called when the consumer allocates the row block and passes it to the producer. The producer is then requested to populate as many rows as possible, up to the maximum number of rows. The `fetch_block` is called when the consumer wants the producer to allocate the row block. `Fetch_block` is efficient if you are developing a TPF that filters rows of data. The server (consumer) allocates the row block and fetches from the TPF using the `fetch_into` API. The TPF can then pass the same row block to the input parameter using the `fetch_block` API.

See also

- *Row Block Data Exchange* on page 128

Developing a Table UDF

The general steps for developing a table UDF include determining input and output, declaring the v4 library, defining the `a_v4_extfn_proc` descriptor, defining a library entry point function, defining how the server gets row information, implementing the `a_v4_extfn_proc` structure functions, and implement the `a_v4_extfn_table_func` structure functions.

1. Determine the input and output for the table UDF.

   The input is defined by the parameters the procedure accepts, and the output is defined by how the **RESULT** clause for the procedure is declared. The declaration of the table UDF in SQL is separate from the implementation of the table UDF. This means that a particular implementation of a table UDF may be bound to a specific declaration. When developing a table UDF, ensure that the implementation and declaration match.
2. Declare the library as a v4 Library.

   For SAP Sybase IQ to recognize the library as a v4 library, the library must include the `extfnapiv4.h` header file located in the subdirectory of your SAP Sybase IQ installation directory.

   This header defines the v4 API features and functions and is a superset of the v3 API; `extfnapiv4.h` includes `extfnapiv3.h`.

   To create table UDF or TPFs, the library must provide the `extfn_use_new_api()` entry point. For v4 libraries, `extfn_use_new_api()` must return `EXTFN_V4_API`.
3. Define the `a_v4_extfn_proc` descriptor.

   When developing a v4 table UDF or TPF, the library must declare what functions are available for the server to call.

   Create a variable of type `a_v4_extfn_proc` and set each member of this structure to the address of the function within the table UDF that implements the function. The information in this variable is made available to the server via a library entry pointer. Not all members of `a_v4_extfn_proc` are required and there are two reserved fields which you must set to NULL.

   Use this descriptor function as a model when developing your own function:

```
static a_v4_extfn_proc udf_proc_descriptor =
{
    udf_proc_start,      // optional
    udf_proc_finish,     // optional
    udf_proc_evaluate,   // required
    udf_proc_describe,   // required
    udf_proc_enter_state,// optional
    udf_proc_leave_state,// optional
```

```
    NULL,                   // Reserved: must be NULL
    NULL                    // Reserved: must be NULL
};
```

4. Define a library entry-point function.

   The table UDF library must provide a function entry point that returns an `a_v4_extfn_proc` descriptor pointer. This is the same descriptor as described in step 3.

   This callback function is the main required entry point for the library.

   Use this function as a model when developing your own library entry point:

```
extern "C"
a_v4_extfn_proc * SQL_CALLBACK udf_rg_proc()
/*****************************************/
{
    return &udf_proc_descriptor;
}
```

5. Define how the server gets row information from the table UDF.

   When developing a v4 table UDF or TPF, the library must declare how row information is transferred to the server.

   Create a variable of type `a_v4_extfn_table_func` and set each member of this structure to the address of the function within the table UDF that implements the function. The information in this variable is made available to the server at runtime.

   Not all members of `a_v4_extfn_table_func` are required and there are two reserved fields which must be set to NULL.

   Use this descriptor as a model when developing your own table UDF:

```
{
    udf_table_func_open,        // required
    udf_table_func_fetch_into,  // one of fetch_into or
fetch_block required
    udf_table_func_fetch_block, // one of fetch_into or
fetch_block required
    udf_table_func_rewind,      // optional
    udf_table_func_close,       // required
    NULL,                       // Reserved: must be NULL
    NULL                        // Reserved: must be NULL
};
```

   At the start of execution, the server calls the `a_v4_extfn_proc` function `_evaluate_extfn` to give the table UDF an opportunity to tell the server what table functions it is implementing. To do this, the table UDF must create an instance of `a_v4_extfn_table` that is given to the server. This structure contains a pointer to the `a_v4_extfn_table_func` descriptor and the number of columns in the result set.

   Use this descriptor as a model when developing your own table UDF:

```
static a_v4_extfn_table_udf_rg_table = {
    &udf_table_funcs, // Table function descriptor
    1                 // number_of_columns
};
```

6. Implement the `a_v4_extfn_proc` structure functions.

   The table UDF must provide an implementation for each of the `a_v4_extfn_proc` functions that it declares in the `a_v4_extfn_proc` descriptor in step 3.

7. Implement the `a_v4_extfn_table_func` structure functions.

   The table UDF must provide an implementation for each of the `a_v4_extfn_table_func` functions that it declares in the `a_v4_extfn_table_func` descriptor in step 5.

See also

- *Scalar and Aggregate UDF Calling Patterns* on page 82
- *udf_rg_2* on page 111
- *udf_rg_3* on page 115
- *Implementing Sample Table UDF udf_rg_1* on page 106
- *Table UDF Implementation Examples* on page 105
- *External Function (a_v4_extfn_proc)* on page 288
- *Table Functions (a_v4_extfn_table_func)* on page 319
- *_evaluate_extfn* on page 290

Table UDF Implementation Examples

Implementation examples start with a simple table UDF and increase in complexity.

The table UDF implementation examples are included in the samples directory. These examples start with a simple table UDF and build upon its complexity and functionality as the examples progress.

The examples are available in a precompiled dynamic library called `libv4apiex`. (The extension of this library name is platform dependent.) This library has linked in the functions defined in `udf_main.cxx`, which contains the library level functions, such as **extfn_use_new_api**. Put `libv4apiex` in a directory the server can read.

See also

- *Running the Sample Table UDF in udf_rg_1.cxx* on page 111
- *Running the Sample Table UDF in udf_rg_2.cxx* on page 114
- *Running the Sample Table UDF in udf_rg_3.cxx* on page 118

Implementing Sample Table UDF udf_rg_1

The sample table UDF called **udf_rg_1** illustrates how a v4 Table UDF can generate n rows of data. The implementation of the table UDF is in the samples directory in `udf_rg_1.cxx`.

1. Determine the input and output for the table UDF.

   This example produces *n* rows of data based on the value of an input parameter. The input is a single integer parameter and the output is rows that consist of a single column of type `integer`.

   The **CREATE PROCEDURE** statement required to define this procedure is:

   ```
   CREATE OR REPLACE PROCEDURE udf_rg_1( IN num INT )
   RESULT( c1 INT )
   EXTERNAL NAME 'udf_rg_1@libv4apiex'
   ```

2. Declare the library as a v4 library.

   In this example, `udf_rg_1.cxx` includes the `extfnapiv4.h` header file:

   ```
   #include "extfnapiv4.h"
   ```

   To inform the server that this library contains v4 table UDFs, this function export is defined in `udf_main.cxx`:

   ```
   a_sql_uint32 SQL_CALLBACK extfn_use_new_api( void )
   /*********************************************/
   {
       return EXTFN_V4_API;
   }
   ```

3. Define the `a_v4_extfn_proc` descriptor.

   This declares the necessary descriptor in `udf_rg_1.cxx`:

   ```
   static a_v4_extfn_proc udf_rg_descriptor =
   {
       NULL,             // _start_extfn
       NULL,             // _finish_extfn
       udf_rg_evaluate,  // _evaluate_extfn
       udf_rg_describe,  // _describe_extfn
       NULL,             // _leave_state_extfn
       NULL,             // _enter_state_extfn
       NULL,             // Reserved: must be NULL
       NULL              // Reserved: must be NULL
   };
   ```

4. Define a library entry point function.

   This callback function declares the main entry point function. It simply returns a pointer to the `a_v4_proc_descriptor` variable *udf_rg_descriptor*.

   ```
   extern "C"
   a_v4_extfn_proc * SQL_CALLBACK udf_rg_1_proc()
   /*********************************************/
   {
   ```

```
    return &udf_rg_descriptor;
}
```

5. Define how the server gets row information from the table UDF.

   This declares the `a_v4_extfn_table_func` descriptor that is used to tell the server how to retrieve row data from the table UDF:

```
static a_v4_extfn_table_func udf_rg_table_funcs =
{
    udf_rg_open,       // _open_extfn
    udf_rg_fetch_into, // _fetch_into_extfn
    NULL,              // _fetch_block_extfn
    NULL,              // _rewind_extfn
    udf_rg_close,      // _close_extfn
    NULL,              // Reserved: must be NULL
    NULL               // Reserved: must be NULL
};
```

   In this example, the `_fetch_into_extfn` function transfers row data to the server. This is the easiest data transfer method to understand and implement. This document refers to data transfer methods as *rowblock data exchange*. There are two rowblock data exchange functions: `_fetch_into_extfn` and `_fetch_block_extfn`.

   At runtime, when the `_evaluate_extfn` function is called, the UDF publishes the table functions descriptor by setting the result set parameter. To do this, the UDF must create an instance of `a_v4_extfn_table`:

```
static a_v4_extfn_table udf_rg_table = {
    &udf_rg_table_funcs,    // Table function descriptor
    1                     // number_of_columns
};
```

   This structure contains a pointer to the `udf_rg_table_funcs` structure and the number of columns in the result set. This table UDF produces a single column in its result set.

6. Implement the a_v4_extfn_proc structure functions.

   In this example, the required function `_describe_extfn` function does not do anything. Other examples demonstrate how a table UDF can use the `describe` function:

```
static void UDF_CALLBACK udf_rg_describe(
       a_v4_extfn_proc_context *ctx )
/
*************************************************************
/
{
   // This required function is not needed in this simple example.
}
```

   The `_evaluate_extfn` method sends the server information about getting the result set from the UDF. This is done by calling the `a_v4_extfn_proc_context` method `set_value` on argument 0. Argument 0 represents the return value, which for a table

UDF is a DT_EXTFN_TABLE. This method constructs `an_extfn_value` structure, setting the data type to DT_EXTFN_TABLE and setting the value pointer of this to point to the `a_v4_extfn_table` object created in step 5. For table UDFs, the type must always be DT_EXTFN_TABLE.

```
static void UDF_CALLBACK udf_rg_evaluate(
          a_v4_extfn_proc_context *ctx,
        void *args_handle )
/*********************************/
{
    an_extfn_value    result_table = { &udf_rg_table,
                      sizeof( udf_rg_table ),
                      sizeof( udf_rg_table ),
                      DT_EXTFN_TABLE };

    // Tell the server what functions table functions are being
    // implemented and how many columns are in our result set.
    ctx->set_value( args_handle, 0, &result_table );
}
```

7. Implement the `a_v4_extfn_table_func` structure functions.

   In this example, the table UDF needs to read in the parameter passed in that contains the number of rows to generate, and cache this information to be used later. Because the `_open_extfn` method is called for each new value that the parameter has, this is an appropriate place to get this information.

   In addition to the total number of rows to generate, the table UDF must also remember the next row to generate. When the server begins fetching rows from the table UDF, it may need to repeatedly call the `_fetch_into_extfn` method. This means that the table UDF must remember the last row that was generated.

   This structure is created in `udf_rg_1.cxx` to contain the state information between calls:

   ```
   struct udf_rg_state {
       a_sql_int32    next_row;  // The next row to produce
       a_sql_int32    max_row;   // The number of rows to generate.
   };
   ```

   The open method first reads in the value of argument 1 using the `a_v4_proc_context` method `get_value`. An instance of `udf_rg_state` is allocated using the `a_v4_proc_context` function `alloc`. table UDFs should use the memory management functions (`alloc` and `free`) on the `a_v4_proc_context` structure whenever possible to manage their memory. The state object is then saved in the *user_data* field of `a_v4_proc_context`. Memory stored in this field is available to the table UDF until execution finishes.

```
static short UDF_CALLBACK udf_rg_open(
          a_v4_extfn_table_context * tctx )
/*************************************/
{
    an_extfn_value     value;
```

```
    udf_rg_state *    state = NULL;

    // Read in the value of the input parameter and store it away in a
    // state object.  Save the state object in the context.
    if( !tctx->proc_context->get_value( tctx->args_handle,
                    1,
                    &value ) ) {

    // Send an error to the client if we could not get the value.
    tctx->proc_context->set_error(
        tctx->proc_context,
        17001,
        "Error: Could not get the value of parameter 1" );

    return 0;
    }

    // Allocate memory for the state using the a_v4_extfn_proc_context
    // function alloc.
    state = (udf_rg_state *)
    tctx->proc_context->alloc( tctx->proc_context,
                   sizeof( udf_rg_state ) );

    // Start generating at row zero.
    state->next_row = 0;

    // Save the value of parameter 1
    state->max_row = *(a_sql_int32 *)value.data;

    // Save the state on the context
    tctx->user_data = state;

    return 1;
}
```

The `_fetch_info_extfn` method returns row data to the server. This method is called repeatedly until it returns false. For this example, the table UDF retrieves the state information from the *user_data* field of the `a_v4_extfn_proc_context` object to determine the next row to generate and the total number of rows to generate. This method is free to generate up to the maximum number of rows indicated in the rowblock structure passed in.

For this example, the table UDF generates a single column of type `INT`. It copies the data for the *next_row* saved in the state into the data pointer of the first column. Each time through the loop, the table UDF copies a new value into the data pointer and stops when either the maximum number of rows to produce is reached or the row block is full.

```
static short UDF_CALLBACK udf_rg_fetch_into(
          a_v4_extfn_table_context *tctx,
          a_v4_extfn_row_block *rb)
/*****************************************/
{
    udf_rg_state    *state = (udf_rg_state *)tctx->user_data;

    // Because we are implementing fetch_into, the server has provided
    // us with a row block. We need to inform the server how many rows
    // this call to _fetch_into has produced.
    rb->num_rows = 0;
```

```
    // The server provided row block structure contains a max_rows
    // field.  This field is the maximum number of rows that this row
    // block can handle.  We can not exceed this number.  We will also
    // stop producing rows when we have produced the number of rows
    // required as per the max_row in the state.
    while( rb->num_rows < rb->max_rows && state->next_row < state->max_row ) {

            // Get the current row from the row block data.
            a_v4_extfn_row      &row = rb->row_data[ rb->num_rows ];

            // Get the column data for the current row.
            a_v4_extfn_column_data    &col0 = row.column_data[ 0 ];

            // Copy the integer value for the next row to generate
            // into the column data for the current row.
            memcpy( col0.data, &state->next_row, col0.max_piece_len );

            state->next_row++;
            rb->num_rows++;
    }

    // If we produced any rows, return true.
    return( rb->num_rows > 0 );
}
```

The table UDF calls the `_close_extfn` method once per new value for the parameters, after all the rows have been fetched. In other words, for each `_open_extfn` call, there is a subsequent `_close_extfn` call. In this example, the table UDF must free the memory allocated during the `_open_extfn` call which it does by retrieving the state from the *user_data* field of `a_v4_extfn_proc_context` object and calling the `free` method.

```
static short UDF_CALLBACK udf_rg_close(
          a_v4_extfn_table_context *tctx)
/***********************************/
{
    udf_rg_state *    state = NULL;

    // Retrieve the state that was saved in user_data
    state = (udf_rg_state *)tctx->user_data;

    // Free the memory for the state using the
a_v4_extfn_proc_context
    // function free.
    tctx->proc_context->free( tctx->proc_context, state );
    tctx->user_data = NULL;

    return 1;
}
```

See also

- *udf_rg_2* on page 111
- *udf_rg_3* on page 115
- *Row Block Data Exchange* on page 128
- *Describe API* on page 208
- *_evaluate_extfn* on page 290

- *fetch_into* on page 313
- *Table (a_v4_extfn_table)* on page 310
- *External Procedure Context (a_v4_extfn_proc_context)* on page 292
- *_open_extfn* on page 321
- *_close_extfn* on page 324

Running the Sample Table UDF in udf_rg_1.cxx

The sample `udf_rg_1` is included in a precompiled dynamic library called `libv4apiex` (extension is platform-dependent). Its implementation is in the `samples` directory in `udf_rg_1.cxx`.

1. Put the library `libv4apiex` in a directory that can be read by the server.
2. To declare the table UDF to the server, issue:

```
CREATE PROCEDURE udf_rg_1( IN num INT )
RESULT( c1 INT )
EXTERNAL NAME 'udf_rg_1@libv4apiex'
```

3. Select rows from the table UDF:

```
SELECT * FROM udf_rg_1( 5 );
```

udf_rg_2

The sample table UDF `udf_rg_2` builds on the sample in `udf_rg_1.cxx` and has the same behavior. The procedure is called **udf_rg_2** and its implementation is in the samples directory in `udf_rg_2.cxx`.

The table UDF **udf_rg_2** provides an alternate implementation of the `_describe_extfn` method in the `a_v4_extfn_proc` descriptor.

```
static void UDF_CALLBACK udf_rg_describe(
        a_v4_extfn_proc_context *ctx )
/*************************************************************/
{
    a_sql_int32        desc_rc;

    // The following describes will ensure that the schema defined
    // by the user matches the schema supported by this table udf.
    // This is achieved by telling the server what our schema is
    // using describe_xxxx_set methods.
    if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {

    a_sql_data_type        type     = DT_NOTYPE;
    a_sql_uint32        num_cols    = 0;
    a_sql_uint32        num_parms    = 0;


    // Inform the server that we support a single input
    // parameter.
    num_parms = 1;
    desc_rc = ctx->describe_udf_set
        ( ctx,
```

```
        EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS,
        &num_parms,
        sizeof( num_parms ) );

    // Checks the return code and sets an error if the
    // describe was unsuccessful for any reason.
    UDF_CHECK_DESCRIBE( ctx, desc_rc );

    // Inform the server that the type of parameter 1 is int.
    type = DT_INT;
    desc_rc = ctx->describe_parameter_set
        ( ctx,
          1,
          EXTFNAPIV4_DESCRIBE_PARM_TYPE,
          &type,
          sizeof( type ) );

    UDF_CHECK_DESCRIBE( ctx, desc_rc );

    // Inform the server that the number of columns in our
    // result set is 1.
    num_cols = 1;
    desc_rc = ctx->describe_parameter_set
        ( ctx,
          0,
          EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS,
          &num_cols,
          sizeof( num_cols ) );

    UDF_CHECK_DESCRIBE( ctx, desc_rc );

    // Inform the server that the type of column 1 in our
    // result set is int.
    type = DT_INT;
    desc_rc = ctx->describe_column_set
        ( ctx,
          0,
          1,
          EXTFNAPIV4_DESCRIBE_COL_TYPE,
          &type,
          sizeof( type ) );

    UDF_CHECK_DESCRIBE( ctx, desc_rc );

    }

    // The following describes will inform the server of various
    // optimizer related characteristics.
    if( ctx->current_state == EXTFNAPIV4_STATE_OPTIMIZATION ) {

    an_extfn_value        p1_value;
    a_v4_extfn_estimate    num_rows;

    // If the value of parameter 1 was constant, then we can
    // inform the server how many distinct values will be.
    desc_rc = ctx->describe_parameter_get
```

```
        ( ctx,
          1,
          EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE,
          &p1_value,
          sizeof( p1_value ) );

    UDF_CHECK_DESCRIBE( ctx, desc_rc );

    if( desc_rc != EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE ) {

        // Inform the server that this UDF will produce n rows.
        num_rows.value = *(a_sql_int32 *)p1_value.data;
        num_rows.confidence = 1;
        desc_rc = ctx->describe_parameter_set
        ( ctx,
          0,
          EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS,
          &num_rows,
          sizeof( num_rows ) );

        UDF_CHECK_DESCRIBE( ctx, desc_rc );

        // Inform the server that this UDF will produce n distinct
        // values for column 1 of its result set.
        desc_rc = ctx->describe_column_set
        ( ctx,
          0,
          1,
          EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES,
          &num_rows,
          sizeof( num_rows ) );

        UDF_CHECK_DESCRIBE( ctx, desc_rc );

    }

    }
}
```

This `describe` method has two primary functions:

- Inform the server what schema it supports.
- Inform the server of some known optimization attributes.

The `describe` function is called during several states. However, not all `describe` attributes are usable in every state. The `describe` method determines the state in which it is executing by checking the *current_state* variable on the `a_v4_extfn_proc` structure.

During the Annotation state, the **udf_rg_2** table udf informs the server that it has one parameter of type `INTEGER` and its result set contains a single column of type `INTEGER`. This is accomplished by setting these attributes:

- EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS

- EXTFNAPIV4_DESCRIBE_PARM_TYPE
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS
- EXTFNAPIV4_DESCRIBE_COL_TYPE

If the information set in these `describe` methods does not match the procedure definition from the **CREATE PROCEDURE** statement, the `describe_parameter_set` and `describe_column_set` methods return `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE`. The describe method then sets an error to indicate to the client there is a mismatch.

This example uses the macro `UDF_CHECK_DESCRIBE` defined in `udf_utils.h` to check the return value from a describe and set an error, if it is not successful.

During optimization, the **udf_rg_2** table udf informs the server that it returns the same number of rows indicated in parameter one. Since the generated rows increment, the values are also unique. During optimization, only parameters that have a constant value are available. Use the `describe` attribute `EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE` to obtain the value of a constant parameter. Once the table udf determines that the attribute value is available, **udf_rg_2** sets `EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS` and `EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES` to the value obtained.

See also

- *udf_rg_3* on page 115
- *Implementing Sample Table UDF udf_rg_1* on page 106

Running the Sample Table UDF in udf_rg_2.cxx

The sample `udf_rg_2` is included in a pre-compiled dynamic library called `libv4apiex` (extension is platform-dependent). Its implementation is in the `samples` directory in `udf_rg_2.cxx`.

1. To declare the table UDF to the server, issue:

```
CREATE OR REPLACE PROCEDURE udf_rg_2( IN num INT )
RESULT( c1 INT )
EXTERNAL NAME 'udf_rg_2@libv4apiex'
```

2. Select rows from the table UDF:

```
SELECT * FROM udf_rg_2( 5 );
```

3. To see how the `describe` affects behavior, issue a **CREATE PROCEDURE** statement that has a different schema than the one published by the table UDF. For example:

```
CREATE OR REPLACE PROCEDURE udf_rg_2( IN num INT, IN extra INT )
RESULT( c1 INT )
EXTERNAL NAME 'udf_rg_2@libv4apiex'
```

4. Select rows from the table UDF:

```
SELECT * FROM udf_rg_2( 5 );
```

IQ returns an error.

udf_rg_3

The sample table UDF **udf_rg_3** builds upon **udf_rg_2** and has similar behavior. The procedure is called **udf_rg_3** and its implementation is in the `samples` directory in `udf_rg_3.cxx`.

The difference between the behavior of table UDFs **udf_rg_3** and **udf_rg_2** is that **udf_rg_3** generates only 100 unique values from 0 to 99, then repeats the sequence as necessary. This table UDF provides `_start_extfn` and `_finish_extfn` methods and has a modified version of `_describe_extfn` to account for the different semantics of the function.

Using `fetch_block` instead of `fetch_into` allows the table UDF to own the row block structure and use its own data layout. To illustrate this, the numbers generated are pre-allocated in an array. When a fetch is performed, rather than copying data into the server provided row block, the table UDF points the row block data pointers directly to the memory containing the data, thus preventing additional copies.

The following ancillary structure stores the numbers array. This structure also keeps a pointer to the allocated row block, which deallocates the row block.

```
#define MAX_ROWS 100
struct RowData {

    a_sql_int32            numbers[MAX_ROWS];
    a_sql_uint32           piece_len;
    a_v4_extfn_row_block * rows;

    void Init()
    {
           rows = NULL;
           piece_len = sizeof( a_sql_int32 );
           for( int i = 0; i < MAX_ROWS; i++ ) {
               numbers[i] = i;
           }
    }
};
```

This structure is allocated when execution of the table UDF starts, and deallocated when execution finishes, by providing `_start_extfn` and `_finish_extfn` methods in the `a_v4_extfn_proc_context`.

```
static void UDF_CALLBACK udf_rg_start(
          a_v4_extfn_proc_context *ctx )
/***********************************/
{
    // The start_extfn method is a good place to allocate our row
    // data.  This method is called only once at the beginning of
    // execution.
    RowData *row_data = (RowData *)
          ctx->alloc( ctx, sizeof( RowData ) );
    row_data->Init();
```

```
    ctx->_user_data = row_data;
}
```

The finish method performs two functions:

- Deallocates the RowData structure.
- Destroys the row block, if the table UDF encounters an error during fetch and cannot destroy the row block.

```
static void UDF_CALLBACK udf_rg_finish(
          a_v4_extfn_proc_context *ctx )
/*************************************/
{
    if( ctx->_user_data != NULL ) {

           RowData *row_data = (RowData *)ctx->_user_data;


           // If rows is non-null here, it means an error occurred and
           // fetch_block did not complete.
           if( row_data->rows != NULL ) {
               DestroyRowBlock( ctx, row_data->rows, 0, false );
           }

           ctx->free( ctx, ctx->_user_data );
           ctx->_user_data = NULL;
    }
}
```

The `fetch_block` method is:

```
static short UDF_CALLBACK udf_rg_fetch_block(
          a_v4_extfn_table_context *tctx,
          a_v4_extfn_row_block **rows )
/*****************************************/
{
    udf_rg_state *    state         = (udf_rg_state*)tctx->user_data;
    RowData *        row_data    = (RowData *)tctx->proc_context->_user_data;

    // First call, we need to build the row block
    if( *rows == NULL ) {

          // This function will build a row block structure that holds
          // MAX_ROWS rows of data.  See udf_utils.cxx for details.
          *rows = BuildRowBlock( tctx->proc_context, 0, MAX_ROWS, false );

          // This pointer gets saved here because in some circumstances
          // when an error occurs, its possible we may have allocated
          // the rowblock structure but then never called back into
          // fetch_block to deallocate it.  In this case, when the finish
          // method is called, we will end up deallocating it there.
          row_data->rows = *rows;
    }

    (*rows)->num_rows = 0;

    // The row block we allocated contains a max_rows member that was
    // set to the macro MAX_ROWS (100 in this case). This field is the
    // maximum number of rows that this row block can handle.  We can
    // not exceed this number.  We will also stop producing rows when
```

```
    // we have produced the number of rows required as per the max_row
    // in the state.
    while( (*rows)->num_rows < (*rows)->max_rows &&
             state->next_row < state->max_row ) {

          a_v4_extfn_row      &row = (*rows)->row_data[ (*rows)->num_rows ];
          a_v4_extfn_column_data    &col0 = row.column_data[ 0 ];

          // Row generation here is a matter of pointing the data
          // pointer in the rowblock to our pre-allocated array of
          // integers that was stored in the proc_context.
          col0.data = &row_data->numbers[(*rows)->num_rows % MAX_ROWS];
          col0.max_piece_len = sizeof( a_sql_int32 );
          col0.piece_len = &row_data->piece_len;
          state->next_row++;
          (*rows)->num_rows++;
    }
    if( (*rows)->num_rows > 0 ) {
          return 1;
    } else {
         // When we are finished generating data, we can destroy the
         // row block structure.
         DestroyRowBlock( tctx->proc_context, *rows, 0, false );
         row_data->rows = NULL;
         return 0;
    }
}
```

The first time this method is called, a row block is allocated using the helper function **BuildRowBlock**, which is in `udf_utils.cxx`. A pointer to this row block is saved in the `RowData` structure for later use.

Row generation is achieved by setting the data pointer for the column data to the address of the next number in sequence in the previously allocated numbers array. The `piece_len` pointer for the column data must also be initialized, by setting it to the address of the `piece_len` member of `RowData`. Since the rows are a fixed data length, this number is the same for all rows.

When fetch is called the last time and there is no more data to produce, the row block structure is destroyed using the **DestroyRowBlock** helper function in `udf_utils.cxx`.

To accommodate this table UDF generating only 100 unique values, `EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES` is set to a value of 100. This code excerpt from the describe method demonstrates this:

```
static void UDF_CALLBACK udf_rg_describe(
          a_v4_extfn_proc_context *ctx )
/*************************************************************/
{
...
...
...

    a_v4_extfn_estimate distinct = {
          MAX_ROWS, 1.0
    };
```

```
    // Inform the server that this UDF will produce MAX_ROWS
    // distinct values for column 1 of its result set.
    desc_rc = ctx->describe_column_set
          ( ctx,
            0,
            1,
            EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES,
            &distinct,
            sizeof( distinct ) );

    UDF_CHECK_DESCRIBE( ctx, desc_rc );
...
...
...

}
```

See also

- *udf_rg_2* on page 111
- *Implementing Sample Table UDF udf_rg_1* on page 106

Running the Sample Table UDF in udf_rg_3.cxx

The sample `udf_rg_3` is included in a precompiled dynamic library called `libv4apiex` (extension is platform-dependent). Its implementation is in the `samples` directory in `udf_rg_3.cxx`.

1. To declare the Table UDF to the server, issue:

   ```
   CREATE OR REPLACE PROCEDURE udf_rg_3( IN num INT )
   RESULT( c1 INT )
   EXTERNAL NAME 'udf_rg_3@libv4apiex'
   ```

2. Select rows from the table UDF:

   ```
   SELECT * FROM udf_rg_3( 200 );
   ```

   This query produces values for `c1` from 0…99 followed by 0…99.

apache_log_reader

The sample table UDF `apache_log_reader` reads the contents of an Apache access log or an Apache error log into table data. It is implemented in the file `apache_log_reader.cxx` in the `samples` directory.

A sample access log (`apache_access.log`) and sample error log (`apache_error.log`) are included in the `samples` directory.

The `apache_log_reader` sample opens the log file in the `_open_extfn` method. It reads in the data and parses it into the schema supported by the procedure in the `_fetch_into_extfn` method. It then closes the log file using the `_close_extfn` method.

See also

- *_open_extfn* on page 321
- *_fetch_into_extfn* on page 322
- *_close_extfn* on page 324

Running the Sample Table UDF in apache_log_reader.cxx

The sample `apache_log_reader` is included in a precompiled dynamic library called `libv4apiex` (extension is platform-dependent). Its implementation is in the `samples` directory in `apache_log_reader.cxx`.

1. To declare the table UDF to the server, issue:

```
create procedure apache_log_reader
(
    in file_name  varchar(4000),
    in log_format varchar(32),
    in ip_padding varchar(1)
)
result
(
    ip_address  varchar(15),
    log_name    varchar(4000),
    user_name   varchar(4000),
    access_time datetime,
    time_zone   int,
    request     varchar(4000),
    response    int,
    bytes_sent  int,
    referer     varchar(4000),
    browser     varchar(4000),
    error_type  varchar(4000),
    error_msg   varchar(4000)
)
external name 'apache_log_reader@libv4apiex'
```

2. Select rows from the table UDF. Use the full path to the access log when executing the SQL query.

```
SELECT * FROM apache_log_reader( 'apache_access.log', 'access',
null );
```

udf_blob

The sample table UDF `udf_blob` illustrates how a table UDF or TPF can read LOB input parameters using the `blob` API.

`udf_blob` counts the number of occurrences of a letter in the first input parameter. The data type of parameter **1** can be `LONG VARCHAR` or `VARCHAR(64)`. If the type is `LONG VARCHAR`, the table UDF uses the `blob` API to read in the value. If the type is `VARCHAR(64)`, the entire value is available using `get_value`.

This code snippett from the `_open_extfn` method illustrates how parameter **1** is read using the `blob` API:

```
static short UDF_CALLBACK udf_blob_open(
    a_v4_extfn_table_context * tctx )
/*************************************/
{
…
…
    a_v4_extfn_blob     *blob         = NULL;

    ret = tctx->proc_context->get_value( tctx->args_handle, 2,
&value );
    UDF_SQLERROR_RT( tctx->proc_context,
              "get_value for argument 2 failed",
              ret == 1,
              0 );

    letter_to_find = *(char *)value.data;

    ret = tctx->proc_context->get_value( tctx->args_handle, 1,
&value );
    UDF_SQLERROR_RT( tctx->proc_context,
              "get_value for argument 1 failed",
              ret == 1,
              0 );

    if( EXTFN_IS_NULL(value) || EXTFN_IS_EMPTY(value) ) {
    state->return_value = 0;
    return 1;
    }

    if( EXTFN_IS_INCOMPLETE(value) ) {
    // If the value is incomplete, then that means we
// are dealing with a blob.
    tctx->proc_context->get_blob( tctx->args_handle, 1, &blob );
    return_value = ProcessBlob( tctx->proc_context,
 blob,
 letter_to_find );
    blob->release( blob );
    } else {
    // The entire value was put into the value pointer.
    return_value = CountNum( (char *)value.data,
value.piece_len,
letter_to_find );
    }
…
…
}
```

Parameter **1** is retrieved using `get_value`. If the value is empty or NULL, then no further processing is required. If the value is determined to be a `blob` using the macro `EXTFN_IS_INCOMPLETE`, then the Table UDF gets an instance of `a_v4_extfn_blob` using the `get_blob` method of `a_v4_extfn_proc_context`. The `ProcessBlob`

method reads from the `blob` to determine how many occurrences of the specified letter are present.

See also

- *Blob (a_v4_extfn_blob)* on page 199
- *_open_extfn* on page 321
- *get_blob* on page 304
- *External Procedure Context (a_v4_extfn_proc_context)* on page 292

Running the Sample Table UDF udf_blob.cxx

The sample `udf_blob` is included in a precompiled dynamic library called `libv4apiex` (extension is platform-dependent). Its implementation is in the `samples` directory in `udf_blob.cxx`.

1. ITo declare the table UDF to the server, issue:

```
CREATE PROCEDURE udf_blob( IN data long varchar, letter char(1) )
RESULT ( c1 BIGINT )
EXTERNAL NAME 'udf_blob@libv4apiex'
```

2. Select rows from the table UDF:

```
set temporary option Enable_LOB_Variables = 'On';
create variable testblob long varchar;
set testblob = 'aaaaaaaaaabbbbbbbbbbb';
select * from udf_blob(testblob, 'a');
```

   The supplied string contains the letter "a" 10 times.

Query Processing States

The SQL statement that references a UDF goes through query processing states in the SAP Sybase IQ server. In each of these states, the server uses the v4 API to communicate and negotiate with the UDF.

See also

- *Generic describe_column Errors* on page 325
- *EXTFNAPIV4_DESCRIBE_COL_TYPE (Set)* on page 228
- *EXTFNAPIV4_DESCRIBE_COL_TYPE (Get)* on page 211

Initial State

Initial state on the server. The only UDF method called during the Initial state is `_start_extfn`.

The server calls the start method for each instance of the UDF created. If a query is executed by a single server thread, then the start method is called once. If a query is handled by several

threads, or distributed across several nodes, the server creates different UDF instances and, as a result, the start method is called several times.

UDFs can set function instance level data within the `_user_data` field of the `a_v4_extfn_proc_context` structure, which is the argument to the start method.

Annotation State

During the annotation state the server updates the parse tree with the metadata necessary for efficient and correct query optimization.

The `[_enter_state]`, `_describe_extfn`, and `[_leave_state]` methods are called. The `_enter_state` and `_leave_state` methods are optional and called if provided by the UDF.

The annotation state is represented in the v4 API by `EXTFNAPIV4_STATE_ANNOTATION` from the `a_v4_extfn_state` enumeration:

```
typedef enum a_v4_extfn_state {
    … EXTFNAPIV4_STATE_ANNOTATION, …
} a_v4_extfn_state;
```

As a UDF developer, you can perform some initial schema negotiation in this phase. Schema negotiation can occur either through the UDF describing to the server what it supports, or the UDF asking the server how it was declared.

When the UDF describes itself to the server, the server detects mismatches and returns SQL errors back to the client. For example, if a UDF describes that it requires four parameters and the SQL writer only declared the UDF with two, the server detects this and returns a SQL error back to the client.

When the UDF itself performs the validation by asking the server how it was declared, it adjusts its runtime execution accordingly: it either matches the declaratio, or it returns an error via the `set_error` v4 API. For example, assume you build a UDF that returns the maximum value of up to five input scalar integers. At runtime, the UDF determines how many input parameters were provided and adjusts its internal logic accordingly. SQL analysts could then create the procedure as:

```
CREATE PROCEDURE my_sum_2( IN a INT, IN b INT ) EXTERNAL
"my_sum@my_lib"
CREATE PROCEDURE my_sum_3( IN a INT, IN b INT, IN c INT ) EXTERNAL
"my_sum@my_lib"
```

Both functions use the same underling implementation of `my_sum`. The UDF recognizes that there are only two parameters for `my_sum_2`, and attempts to sum parameters 1 and 2. For `my_sum_3`, the UDF sums parameters 1, 2 and 3.

As a UDF developer, you can obtain values for constant literal parameters only in the Annotation state. No other values are available until the Execution state. To get parameter

values during the annotation state use the `describe_parameter_get` method with the `PARM_CONSTANT_VALUE` and `PARM_IS_CONSTANT` attributes.

In the Annotation state, UDFs have access to schema `describe` attributes:

- `EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS`
- `EXTFNAPIV4_DESCRIBE_PARM_NAME`
- `EXTFNAPIV4_DESCRIBE_PARM_TYPE`
- `EXTFNAPIV4_DESCRIBE_PARM_WIDTH`
- `EXTFNAPIV4_DESCRIBE_PARM_SCALE`
- `EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT`
- `EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE`
- `EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS`
- `EXTFNAPIV4_DESCRIBE_COL_NAME`
- `EXTFNAPIV4_DESCRIBE_COL_TYPE`
- `EXTFNAPIV4_DESCRIBE_COL_WIDTH`
- `EXTFNAPIV4_DESCRIBE_COL_SCALE`
- `EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT`
- `EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE`

During the Annotation phase the UDF can set the above values to define its schema to the server. If the server detects a mismatch between what the UDF describes and the SQL procedure declaration, it returns an error. This technique is referred to as *self-describing*.

An alternative technique, schema validation, can be employed by the UDF. This involves the UDF getting the values for the schema describe types, and then setting an error if a mismatch is detected. With this approach, validation is left to the UDF, but the UDF can choose to support multiple schemas with a single implementation (for example, the ability to support multiple datatypes for a given parameter or being able to support varying number of parameters).

See also

- *EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS Attribute (Get)* on page 278
- *EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS Attribute (Set)* on page 280
- *EXTFNAPIV4_DESCRIBE_PARM_NAME Attribute (Get)* on page 243
- *EXTFNAPIV4_DESCRIBE_PARM_NAME Attribute (Set)* on page 262
- *EXTFNAPIV4_DESCRIBE_PARM_TYPE Attribute (Get)* on page 244
- *EXTFNAPIV4_DESCRIBE_PARM_TYPE Attribute (Set)* on page 263
- *EXTFNAPIV4_DESCRIBE_PARM_WIDTH Attribute (Get)* on page 245
- *EXTFNAPIV4_DESCRIBE_PARM_WIDTH Attribute (Set)* on page 264
- *EXTFNAPIV4_DESCRIBE_PARM_SCALE Attribute (Get)* on page 246
- *EXTFNAPIV4_DESCRIBE_PARM_SCALE Attribute (Set)* on page 265
- *EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT Attribute (Get)* on page 251
- *EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT Attribute (Set)* on page 267

- *EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE Attribute (Get)* on page 252
- *EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE Attribute (Set)* on page 268
- *EXTFNAPIV4_DESCRIBE_COL_NAME (Get)* on page 210
- *EXTFNAPIV4_DESCRIBE_COL_NAME (Set)* on page 227
- *EXTFNAPIV4_DESCRIBE_COL_TYPE (Get)* on page 211
- *EXTFNAPIV4_DESCRIBE_COL_TYPE (Set)* on page 228
- *EXTFNAPIV4_DESCRIBE_COL_WIDTH (Get)* on page 212
- *EXTFNAPIV4_DESCRIBE_COL_WIDTH (Set)* on page 229
- *EXTFNAPIV4_DESCRIBE_COL_SCALE (Get)* on page 212
- *EXTFNAPIV4_DESCRIBE_COL_SCALE (Set)* on page 230
- *EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT (Get)* on page 217
- *EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT (Set)* on page 234
- *EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE (Get)* on page 218
- *EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE (Set)* on page 234

Query Optimization State

During the Optimization state, the server is in the initial process of constructing a query plan. The server collects schema information and some preliminary statistical information.

The `[_enter_state]`, `_describe_extfn`, and `[_leave_state]` methods are called. The `_enter_state` and `_leave_state` methods are optional, and called if provided by the UDF.

The query optimization state is represented in the v4 API by `EXTFNAPIV4_STATE_OPTIMIZATION` from the `a_v4_extfn_state` enumeration:

```
typedef enum a_v4_extfn_state {
    … EXTFNAPIV4_STATE_OPTIMIZATION, …
} a_v4_extfn_state;
```

Negotiations during the query optimization state include:

- The server and UDF determine the partitioning/ordering/clustering already specified for input tables.
- The server and UDF determine the partitioning/ordering required for input tables.
- The UDF declares physical properties (such as an ordering property) for the result table.
- The UDF describes any properties and statistics (for example, cost estimates) which can be used during the query optimization process.
  - Table scope estimates include:
    - **Number of rows –** the total number of rows present in the UDF during the execution state. This value is available for both the input TABLE parameter and the returned table.

    - **Row size –** an estimate of the average number of bytes in each row.
  - Column scope estimates include:
    - **Distinct count –** the number of distinct values in a column over the total number of rows in a table. This value is available for both the input TABLE parameter and the returned table.

In the Optimization state, UDFs have access to `describe` attributes:

- `EXTFNAPIV4_DESCRIBE_PARM_NAME`
- `EXTFNAPIV4_DESCRIBE_PARM_TYPE`
- `EXTFNAPIV4_DESCRIBE_PARM_WIDTH`
- `EXTFNAPIV4_DESCRIBE_PARM_SCALE`
- `EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT`
- `EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE`
- `EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS`
- `EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS`
- `EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY`
- `EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY`
- `EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND`
- `EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND`
- `EXTFNAPIV4_DESCRIBE_COL_NAME`
- `EXTFNAPIV4_DESCRIBE_COL_TYPE`
- `EXTFNAPIV4_DESCRIBE_COL_WIDTH`
- `EXTFNAPIV4_DESCRIBE_COL_SCALE`
- `EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL`
- `EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT`
- `EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE`
- `EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER`
- `EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT`

See also

- *DEFAULT_TABLE_UDF_ROW_COUNT Option* on page 179
- *EXTFNAPIV4_DESCRIBE_PARM_NAME Attribute (Get)* on page 243
- *EXTFNAPIV4_DESCRIBE_PARM_NAME Attribute (Set)* on page 262
- *EXTFNAPIV4_DESCRIBE_PARM_TYPE Attribute (Get)* on page 244
- *EXTFNAPIV4_DESCRIBE_PARM_TYPE Attribute (Set)* on page 263
- *EXTFNAPIV4_DESCRIBE_PARM_WIDTH Attribute (Get)* on page 245
- *EXTFNAPIV4_DESCRIBE_PARM_WIDTH Attribute (Set)* on page 264
- *EXTFNAPIV4_DESCRIBE_PARM_SCALE Attribute (Get)* on page 246
- *EXTFNAPIV4_DESCRIBE_PARM_SCALE Attribute (Set)* on page 265
- *EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT Attribute (Get)* on page 251

- *EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER (Set)* on page 235
- *EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT (Get)* on page 225
- *EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT (Set)* on page 240

Plan Building State

During the plan building state, the server builds the query execution plan based on the best plan found during the query optimization state.

The `[_enter_state]`, `_describe_extfn`, and `[_leave_state]` methods are called. The `_enter_state` and `_leave_state` methods are optional and called if provided by the UDF.

The plan building state is represented in the v4 API by `EXTFNAPIV4_STATE_PLAN_BUILDING` from the `a_v4_extfn_state` enumeration:

```
typedef enum a_v4_extfn_state {
    … EXTFNAPIV4_STATE_PLAN_BUILDING, …
} a_v4_extfn_state;
```

At this point in query processing, the server determines what columns are needed from the UDF, and requests information about the columns needed from the TABLE parameters.

If the UDF supports parallel processing, and if the server agrees that the query is eligible for parallelism, the server creates multiple instances of the UDF for distributed query processing.

In the Plan Building state, UDFs have access to all describe attributes.

As an example, the following code fragment queries the server to determine which columns are used:

```
    a_sql_int32     rc;
    rg_udf         *rgUdf = (rg_udf *)ctx->_user_data;
    rg_table       *rgTable = rgUdf->rgTable;
    a_sql_uint32    buffer_size = 0;

    buffer_size = sizeof(a_v4_extfn_column_list)   ( rgTable-
>number_of_columns - 1 ) * sizeof(a_sql_uint32);
    a_v4_extfn_column_list *ulist = (a_v4_extfn_column_list *)ctx-
>alloc(
                                    ctx,
                                    buffer_size );
    memset(ulist, 0, buffer_size);

    rc = ctx->describe_parameter_get( ctx,
                0,
                EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS,
                ulist,
                buffer_size );

    if( rc != buffer_size ) {
```

```
        ctx->free( ctx, ulist );
       UDF_SQLERROR( PC(ctx), "Describe parameter type get failure.",
rc == buffer_size );
    } else {
        rgTable->unused_col_list = ulist;
    }
```

Assuming the above code fragment is from a Table UDF that produces 4 result set columns, and assuming the SQL statement was

```
SELECT c1, c2 FROM my_table_proc();
```

then the `describe` API returns only c1 and c2. This lets the UDF optimize the production of the result set values.

See also

- *Describe API* on page 208

Execution State

During the execution state, the server makes an execution call into the UDF.

The execution plan, created in the plan building state, is used in the execution state to compute the result set of the SQL query.

These methods can be called: `[_enter_state]`, `_describe_extfn`, `evaluate_extfn`, `_open_extfn`, `_fetch_into_extfn`, `_fetch_block_extfn`, `_close_extfn`, `[_leave_state]`, and `_finish_extfn`.

The execution state is represented in the `a_v4_extfn_state` API by this enumeration:

```
typedef enum a_v4_extfn_state {
    … EXTFNAPIV4_STATE_EXECUTING, …
} a_v4_extfn_state;
```

In the execution state:

- Input TABLE parameter rows and nonconstant scalar input parameter values are available.
- The UDF can open a result set on input TABLE parameters, and fetch rows.

Executing Partition State

If an input TABLE parameter exists, and if a **PARTITION BY** clause exists in the SQL query, then the server invokes the UDF once per available partition.

Row Block Data Exchange

A row block is the data transfer area between the producer and the consumer.

A table UDF can only produce rows. It can use an existing row block, or it can build its own row block.

A TPF can both produce and consume rows. A TPF produces rows in the same way a table UDF produces rows and can use an existing row block or build its own row block. A TPF can consume rows from an input table and can provide the producer with a row block, or request the producer to create its own row block.

See also

- *Row Block (a_v4_extfn_row_block)* on page 309
- *Table (a_v4_extfn_table)* on page 310
- *Table Functions (a_v4_extfn_table_func)* on page 319
- *_open_extfn* on page 321
- *_fetch_into_extfn* on page 322
- *_fetch_block_extfn* on page 322
- *_rewind_extfn* on page 323
- *_close_extfn* on page 324

Fetch Methods for Row Blocks

The fetch methods for row blocks are `_fetch_into_extfn` and `_fetch_block_extfn`. These methods are part of the `a_v4_extfn_table_func` structure.

When producing data, if the table UDF or TPF builds its own row block, the UDF must provide the `fetch_block` API method. If the UDF does not build its own row block, the UDF must provide the `fetch_into` API method.

When consuming data, if the TPF builds its own row block, the UDF calls the `fetch_into` method on the producer. If the TPF does not build its own row block, the TPF must call the `fetch_block` method on the producer.

The UDF can select which fetch method to use for data production and consumption. In general, these guidelines apply:

- **fetch_into –** Use this API when the consumer owns the memory for the data transfer area and requests that the producer use this area. In this scenario, the consumer cares about how the data transfer area is set up, and the producer performs the necessary data copies into this area.
- **fetch_block –** Use this API when the consumer does not care about the format of the data transfer area. `fetch_block` requests the producer to create a data transfer area and provides a pointer to that area. The consumer owns the memory and the consumer is responsible for copying data from this area.

See also

- *Table Parameterized Functions* on page 136
- *fetch_into* on page 313
- *fetch_block* on page 316

The fetch_block Method

Use the `fetch_block` method for underlying data storage.

The `fetch_block` method is used as an entry point when the consumer does not need the data in a particular format. `fetch_block` requests that the producer create a data transfer area and provide a pointer to that area. The consumer owns the memory and takes responsibility for copying data from this area.

The `fetch_block` method is more efficient than `fetch_into`, if the consumer does not need a specific layout. The `fetch_block` call provides a row block that can be populated, and this block is passed on the next `fetch_block` call. This method is part of the `a_v4_extfn_table_context` structure.

If the underlying data storage does not map easily to the row block structure, the UDF can simply point the row block to addresses in its memory. This prevents unnecessary data copies to satisfy another memory layout scheme.

The API uses a data transfer area that is defined by the structure `a_v4_extfn_row_block`, which is defined as a set of rows, where each row is defined as a set of columns. The row block creator can allocate enough storage to hold a single row or a set of rows. The producer can fill the rows, but cannot exceed the maximum number of rows allocated for the row block. If the producer has additional rows, the producer informs the consumer by returning the numeric value 1 from the `fetch` method.

Fetch is executed against a table object, which is either the object produced as the result set of a table UDF or the object consumed as a result set of an input TABLE parameter.

See also

- *Using a Row Block to Produce Data* on page 131
- *fetch_block* on page 316

The fetch_into Method

Use the `fetch_into` API when the consumer owns the memory for the data transfer area and requests that the producer use this area.

The `fetch_into` method is useful when the producer does not know how data should be arranged in memory. This method is used as an entry point when the consumer has a transfer area with a specific format. The `fetch_into()` function writes the fetched rows into the provided row block. This method is part of the `a_v4_extfn_table_context` structure.

The API uses a data transfer area that is defined by the structure `a_v4_extfn_row_block`, which is defined as a set of rows, where each row is defined as a set of columns. The creator of the row block can allocate enough storage to hold a single row or a set of rows. The producer can fill the rows, but cannot exceed the maximum number of rows allocated for the row block. If the producer has additional rows, the producer informs the consumer by returning the numeric value 1 from the fetch method.

This API enables consumers to optionally construct the row block, such that the data pointers refer to its own data structures. This allows the producer to directly populate memory within the consumer. A consumer may not want to do this, if data cleansing or validation checks are required first.

Fetch is executed against a table object, which is either the object produced as the result set of a table UDF or the object consumed as a result set of an input TABLE parameter.

See also

- *Using a Row Block to Produce Data* on page 131
- *fetch_into* on page 313

Using a Row Block to Produce Data

A table UDF or TPF can use row block structures to produce data.

The `a_v4_extfn_row_block` row block has three fields:

- **max_rows –** How many table rows the row block can store in a piece of memory.
- **num_rows –** The number of rows actually produced or available for consumption. Cannot be larger than `max_rows`.
- **row_data –** The array of rows produced or available for consumption. Each row is an `a_v4_extfn_row` structure.

See also

- *Table UDF Implementation Examples* on page 105
- *fetch_into* on page 313
- *fetch_block* on page 316
- *Row Block (a_v4_extfn_row_block)* on page 309
- *Row (a_v4_extfn_row)* on page 309
- *Column Data (a_v4_extfn_column_data)* on page 204

Producing Data Using fetch_into

Use the `fetch_into` API method to produce data.

1. Set `num_rows` to a value based on the number of rows produced in the fetch call.
2. For each row produced, set the `row_status` flag of `a_v4_extfn_row` to 1 (available) or 0 (not available). The default value is 1.
3. For each column (`a_v4_extfn_column_data`) in the row set:

Options	Description
is_null	Set to true, if the value returned is NULL. The default is false.

Options	Description
data	The data returned must be copied into this pointer
piece_len	The actual length of data returned. For fixed-length data types, this cannot exceed *max_piece_len*. Defaults to *max_piece_len* for fixed data types.

4. For each column, return 1 to indicate rows produced, and return 0 to indicate otherwise.

Producing Data Using fetch_block

Use the `fetch_block` API method to produce data.

1. Set `max_rows` to the number of rows the producer-allocated row block structure can hold.
2. On the first fetch call, allocate a row block structure that can hold `max_rows`.
3. Set `num_rows` to a value based on the number of rows produced in the fetch call.
4. For each row produced, set the `row_status` flag of `a_v4_extfn_row` to 1 (available) or 0 (not available). The default value is 1.
5. For each column (`a_v4_extfn_column_data`) in the row set:

null_value	Indicates the value is used to indicate NULL
null_mask	Identifies the bits that represent the NULL value.
is_null	If the value is NULL, set is_null to a value such that `(*(cd->is_null) & cd->null_mask) == cd->null_value ).`
data	Set this pointer to the area in the producer's memory containing the data to be returned.
piece_len	The actual length of data being returned. For fixed-length data types, this cannot exceed *max_piece_len*. This value defaults to *max_piece_len* for fixed data types.

6. Return 1 from `fetch_into` to indicate rows were produced, and return 0 to indicate otherwise. On the last fetch call, deallocate any memory that is allocated for the row block structure.

Row Block Allocation

Row block allocation is required when a producer produces data using the `fetch_block` method or when the consumer uses the `fetch_into` method to retrieve data.

`udf_utils.cxx` contains sample code that illustrates how to allocate and deallocate a row block.

These relevant data structures in the `extfnapiv4.h` header file are used when allocating a row block:

```
typedef struct a_v4_extfn_column_data {
    a_sql_byte          *is_null;
    a_sql_byte          null_mask;
    a_sql_byte          null_value;

    void                *data;
    a_sql_uint32        *piece_len;
    size_t              max_piece_len;

    void                *blob_handle;
} a_v4_extfn_column_data;

typedef struct a_v4_extfn_row {
    a_sql_uint32                *row_status;
    a_v4_extfn_column_data      *column_data;
} a_v4_extfn_row;

typedef struct a_v4_extfn_row_block {
    a_sql_uint32                max_rows;
    a_sql_uint32                num_rows;
    a_v4_extfn_row              *row_data;
} a_v4_extfn_row_block;
```

When allocating a row block, the developer must decide how many rows the row block is capable of holding, how many columns each row has, and the number of bytes required for each of those columns.

For a row block of size *m*, where each row has *n* columns, the developer must allocate an array of `m a_v4_extfn_row` structures. For each row in this array, the developer must allocate *n* `a_v4_extfn_column_data` structures.

These tables outline allocation requirements for each member of the row block structures:

Table 2. a_v4_extfn_row_block Structure

Field	Requirement
max_rows	Set to the number of rows this row block can hold
num_rows	Initialize to zero. Is set to the number of actual rows a row block contains during usage
*row_data	Allocate an array containing max_rows of `a_v4_extfn_row` structures

Table 3. a_v4_extfn_row Structure

Field	Requirement
*row_status	Allocate enough memory to hold an **a_sql_uint32**

Field	Requirement
*column_data	Allocate an array containing the number of columns in the result set of `a_v4_extfn_column_data` structures

Table 4. a_v4_extfn_column_data Structure

Field	Requirement
*is_null	Allocate enough memory to hold an **a_sql_byte**
null_mask	Set to a value such that the formula (*is_null & null_mask) == null_value indicates a column value is NULL
null_value	Set to a value such that the formula (*is_null & null_mask) == null_value indicates a column value is NULL
*data	Allocate an array of bytes large enough to hold the data for the data type of the column
*piece_len	Allocate enough memory to hold an **a_sql_uint32**
max_piece_len	Set to the maximum width for the column
*blob_handle	Always owned by the server. Initialize to NULL.

See also

- *SQL Data Types* on page 9
- *External Procedure Context (a_v4_extfn_proc_context)* on page 292

Table UDF Query Plan Objects

The table UDF values and TPF values visible in the query plan.

- **Blocks Fetched –** shows the number of chunks used to transfer all the data produced by the UDF. This value equates to the number of times the server called the fetch method of the UDF.
- **Maximum rows per `_fetch_into_extfn` –** (visible ony if the UDF is using `_fetch_into_extfn`.) Displays the maximum number of rows a UDF can produce on each call to `_fetch_into_extfn` as determined by the server.
- **Minimum/Maximum values for an output column –** displays minimum/maximum values per column if the UDF has set them via `extfnapiv4_describe_col_maximum_value`. Minimum/maximum appear only for arithmetic data type columns.

- **ORDER BY node (TPF only) –** for a TPF, the query plan shows an ORDER BY node as a child of the TPF SubQuery node. The ORDER BY node indicates that the data is ordered as it flows into the TABLE parameter.
- **Output Row Width –** (visible only if the UDF is using `_fetch_into_extfn`.) Shows the width of an output column in bytes. This value is used in calculating the maximum number of rows.
- **TableUDF node –** represents an instance of a table UDF in the query. The TableUDF node is a leaf node.
- **TPF node (TPF only) –** same as the TableUDF node except that TPF node permits use of an input TABLE parameter. Unlike a TableUDF node which is a leaf node, the TPF is an interior node with at most one child.
- **TPF SubQuery node (TPF only) –** child of the TPF node. Represents the subquery for the input table argument.
- **UDF Library –** UDF library file name. Shows the full path on disk from which the dynamic library implementing the UDF was loaded.
- **Uniqueness of an output column –** reflects the value set by `extfnapiv4_describe_col_is_unique`.
- **TABLE_UDF_ROW_BLOCK_SIZE_KB –** option value displays in query plan statistics if you specify a value other than 128KB.

Enabling Memory Tracking

Enable memory tracking to help you locate memory leaks in your UDFs, and to free the leaked memory. Memory tracking imposes a performance penalty.

Enabling memory tracking tracks all invocations of `a_v4_extfn_proc_context alloc` and `a_v4_extfn_proc_context free`. An allocations without a matching free is logged to the `iqmsg` file.

1. Ensure the **external_UDF_execution_mode** is set to 1 or 2 (validation mode or tracing mode).
2. Use the `alloc` and `free` methods of `a_v4_extfn_proc_context`.

See also

- *alloc* on page 301
- *free* on page 302

Table Parameterized Functions

A Table parameterized function (TPF) is an extension of a table UDF that accepts table input parameters in addition to scalar input parameters.

You can configure user-specified partitioning for your TPF. The UDF developer can declare a partitioning scheme that breaks down the dataset into smaller pieces of query processing that you distribute across multiplex nodes. This enables you to execute the TPF in parallel in a distributed server environment over partitions of row sets. The query engine supports massive parallelization of TPF processing.

Note: Multiplex requires a separate license. See *Administration: Multiplex*.

Learning Roadmap for TPF Developers

Develop a C or C++ TPF.

This roadmap assumes:

- You have a C or C++ development environment on your machine.
- For the optional data partitioning capability, you have a multiplex environment. See *Administration: Multiplex*.

Task	See
Familiarize yourself with table UDF development.	*Learning Roadmap for Table UDF Developers* on page 97
Follow the recommended procedure for creating a TPF.	*Developing a TPF* on page 136 *TPF Implementation Examples* on page 156
Establish a table context for the input table and consume table rows from it.	*Consume TABLE Parameters* on page 137
(Optional) Order incoming data.	*Order Input Table Data* on page 140
(Optional) Partition the incoming data to enable parallel TPF processing in your multiplex.	*Partitioning Input Data* on page 140

Developing a TPF

Review the major steps required to develop a TPF.

1. Perform the same steps required to develop a table UDF.
2. Consume input parameters.
3. (Optional) Order input table data.
4. (Optional) Partition input table data.

5. (Optional) Enable parallel TPF processing.

See also

- *Consume TABLE Parameters* on page 137
- *Order Input Table Data* on page 140
- *Partitioning Input Data* on page 140
- *_open_extfn* on page 321
- *_fetch_into_extfn* on page 322
- *_fetch_block_extfn* on page 322
- *_rewind_extfn* on page 323
- *_evaluate_extfn* on page 290
- *Developing a Table UDF* on page 103

Consume TABLE Parameters

A TABLE parameter is a non-constant parameter. This means that the TPF must be in the execution state to retrieve TABLE parameters.

The TPF can retrieve the TABLE parameter from these methods:

- `_open_extfn`
- `_fetch_into_extfn`
- `_fetch_block_extfn`
- `_rewind_extfn`
- `_evaluate_extfn`

To consume a TABLE parameter, the TPF must:

Obtain a Table Object

The TPF obtains a table object for the TABLE parameter using the `get_value` method of `a_v4_extfn_proc_context`.

A table object (`a_v4_extfn_table`) can initiate retrieving rows from an input table. The following code snippet illustrates how `get_value` obtains a table object for parameter **1**. For simplicity, this code assumes that parameter **1** is a table.

```
a_v4_extfn_value        value;
a_v4_extfn_table *      table;

ctx->get_value( args_handle,
                        1,
                        &value );

table = (a_v4_extfn_table *)value.data;
```

See also

- *get_value* on page 294

Open the Result Set

Once a table object has been obtained using `get_value`, the TPF must open a result set on the table object using the `open_result_set` method of `a_v4_extfn_proc_context` before it can fetch any rows.

Calling `open_result_set` returns an instance of `a_v4_extfn_table_context` that the TPF can use to process table data. It also saves the table object in the table member of the `a_v4_extfn_table_context` object.

The following code snippet illustrates how `open_result_set` gets an instance of `a_v4_extfn_table_context` for fetching rows:

```
a_v4_extfn_table_context *  rs  = NULL;

ctx->open_result_set( ctx,
                     (a_v4_extfn_table *)value.data,
                      &rs ) );
```

See also

- *open_result_set* on page 303
- *Table Context (a_v4_extfn_table_context)* on page 311

Fetch from the Result Set

The TPF fetches table data from an input table using an open result set.

Fetching is accomplished by calling either `fetch_block` or `fetch_into` on the `a_v4_extfn_table_context` object returned from `open_result_set`. The TPF can choose which fetch method to use. If `fetch_block` is used, the server is responsible for rowblock allocation. If `fetch_into` is used, the TPF is responsible for row block allocation.

Each call to the fetch method returns either nothing, which is indicated by a return value of false, or returns a populated row block structure. The row block structure can then be used to consume the table data.

See also

- *fetch_into* on page 313
- *fetch_block* on page 316
- *Row Block Data Exchange* on page 128

Consume Table Data Using a Row Block

The TPF consumes table data using the `fetch_into` or `fetch_block` row block structures.

Each successful call to either `fetch_into` or `fetch_block` populates a `a_v4_extfn_row_block` structure.

The `a_v4_extfn_row_block` members are:

- **max_rows –** the number of table rows the row block can store in a piece of memory.
- **num_rows –** the number of rows actually produced or available for consumption. Cannot be larger than `max_rows`.
- **row_data –** the array of rows produced or available for consumption. Each row is an `a_v4_extfn_row` structure.

Each row of table data in `row_data` has these members:

- **row_stats –** indicates whether values for this row are present. A value of 1 means the values are present; 0 means the values are not.
- **column_data –** the column data associated with this row.

The `column_data` members are:

Member	Description
`null_value`	The value representing NULL
`null_mask`	One or more bits used to represent the NULL value
`data`	Pointer to the data for the column. Depending on the type of fetch mechanism, either points to an address in the consumer, or an address where the data is stored in the UDF.
`piece_len`	The actual length of data for variable-length data types
`blob`	A non-NULL value means that the data for this column must be read using the `blob` API

See also

- *Column Data (a_v4_extfn_column_data)* on page 204
- *Row Block (a_v4_extfn_row_block)* on page 309
- *Row (a_v4_extfn_row)* on page 309
- *get_blob* on page 318

Close the Result Set

Once the TPF is finished processing table data, it closes the open result set using the `close_result_set` method of `a_v4_extfn_proc_context`.

This code snippet illustrates `close_result_set` closing a result set.

```
ctx->close_result_set( ctx,
                       rs ) );
```

Order Input Table Data

Either a SQL Analyst or the UDF developer can order incoming data.

A SQL Analyst controls ordering by including the **ORDER BY** clause in a **SELECT** statement.

The UDF developer controls ordering by using the **DESCRIBE_PARM_TABLE_ORDERBY** attribute.

Both methods result in the server ordering the incoming data, the results of which can be see in the query plan in the Order node.

See also

- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY Attribute (Get)* on page 255
- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY Attribute (Set)* on page 270

Partitioning Input Data

Use the **PARTITION BY** clause to express and declare invocation partitioning in your parallel TPF.

As a SQL analyst, you can efficiently utilize system resources by leveraging the server query parallelism and distribution features available with the **PARTITION BY** clause in your SQL queries. Depending on the clause specified, the server may partition data into distinct value-based sets of rows or by row-range sets of rows.

- **Value-based partitions –** determined by key values on an expression. These partitions provide value when a computation depends on seeing all rows of the same value for aggregation.
- **Row-based partitions –** simple and efficient means to divide a computation into multiple streams of work. Used when a query must be executed in parallel.

You can express a design for partition via the **PARTITION BY <expr>** clause on the **TABLE** parameter to a TPF. UDF developers can utilize the TABLE parameter metadata attribute **EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY** to programmatically declare that the UDF requires partitioning before invocation can proceed. The UDF can inquire to the partition to enforce it, or to dynamically adapt the partitioning.

See also

- *Parallel TPF PARTITION BY Examples Using EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY* on page 143
- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY (Get)* on page 256
- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY (Set)* on page 272
- *V4 API describe_parameter and EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY* on page 141

V4 API describe_parameter and EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY

You can use `describe_parameter_set` and `describe_parameter_get` for partitioning an input TABLE parameter for required columns.

Declaration

The `describe_parameter` API has two declarations.

describe_parameter_set Declaration

```
a_sql_int32 (SQL_CALLBACK *describe_parameter_set)(
            a_v4_extfn_proc_context        *cntxt,
            a_sql_uint32                   arg_num,
            a_v4_extfn_describe_parm_type  describe_type,
            void                           *describe_buffer,
            size_t                         describe_buffer_
            )
```

describe_parameter_get Declaration

```
a_sql_int32 (SQL_CALLBACK *describe_parameter_get)(
            a_v4_extfn_proc_context           *cntxt,
            a_sql_uint32                   arg_num,
            a_v4_extfn_describe_parm_type  describe_type,
            const void                     *describe_buffer,
            size_t                            describe_buffer_
            )
```

Usage

In order to use these APIs, the **arg_num** must refer to a TABLE parameter, and the **describe_buffer** must refer to the type of memory block `a_v4_extfn_column_list` structure.

```
typedef struct a_v4_extfn_column_list {
    a_sql_int32         number_of_columns;
    a_sql_uint32        column_indexes[1];
} a_v4_extfn_column_list;
```

The structure field **number_of_columns** must have one of these values:

- Positive integer N, where N indicates the number of columns present in the partition by list.

- 0, which indicates **PARTITION BY ANY**.
- -1, which indicates **NO PARTITION BY**.

This enumerated type is defined in the `extfnapiv4.h` header file:

```
typedef struct a_v4_extfn_column_list {
    a_sql_int32         number_of_columns;
    a_sql_uint32        column_indexes[1];
} a_v4_extfn_column_list;
```

You can use the `v4_extfn_partitionby_col_num` enumerated type to build the column list structure and execute the `describe_parameter_set` and `describe_parameter_get` API to inform the server of its requirements and to determine which input columns have been partitioned. The execution of `describe_parameter_set` and `describe_parameter_get` APIs can have following scenarios:

describe_parameter_set Scenarios

column Index Scenarios	Description
{ 1, 1 }	Input table column #1 is partitioned as per UDF request.
{ 2, 3, 1 }	Input table columns #3 and #1 are partitioned as per UDF request.
{ 0 }	UDF can support any form of input table partitioning as per UDF request.

describe_parameter_get Scenarios

column Index Scenarios	Description
{ 1, 2 }	Input table column #2 is being partitioned on.
{ 2, 1, 2}	Input table columns #1 and #2 are being partitioned on.
{ 0 }	Input table being partitioned by a noncolumn based scheme.
NULL	No runtime partitioning is provided.

Note: A **PARTITION BY** expression other than **PARTITION BY ANY** or **PARTITION BY NONE** must appear in the select list for the input query.

See also

- *Describe API* on page 208
- *Partition By Column Number (a_v4_extfn_partitionby_col_num)* on page 307
- *EXTFNAPIV4_DESCRIBE_PARM_TYPE Attribute (Get)* on page 244

- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS Attribute (Get)* on page 253
- *EXTFNAPIV4_DESCRIBE_PARM_TYPE Attribute (Set)* on page 263
- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS Attribute (Set)* on page 268
- *EXTFNAPIV4_DESCRIBE_COL_TYPE (Get)* on page 211
- *EXTFNAPIV4_DESCRIBE_COL_TYPE (Set)* on page 228

Parallel TPF PARTITION BY Examples Using EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY

Develop partitioning using the **PARTITION BY <expr>** clause on the **TABLE** parameter to a TPF function. As a UDF developer, use the TABLE parameter metadata attribute `EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY` to programmatically declare that the UDF requires partitioning before invoking it.

The examples illustrate:

- Various SQL writer scenarios where the UDF describes partitioning requirements to the server
- Valid queries and invalid queries (SQL exceptions) for each scenario
- How the server detects mismatches
- The various possible combinations that arise from usage of the **PARTITION BY** SQL clause and the `EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY` UDF attribute

See also

- *Partitioning Input Data* on page 140

Example Procedure Definition

An example procedure definition that supports TPF **PARTITION BY** clause examples.

All TPF **PARTITION BY** clause examples in this section assume that you first execute this procedure definition:

```
CREATE PROCEDURE my_tpf( arg1 TABLE( c1 INT, c2 INT ) )
  RESULTS ( r1 INT, r2 INT, r3 INT )
  EXTERNAL 'my_tpf_proc@mylibrary';

CREATE TABLE T( x INT, y INT, z INT );
```

See also

- *describe_parameter_set Example # 1: One-Column Partitioning on Column 1* on page 144
- *describe_parameter_set Example # 2: Two-Column Partitioning* on page 146
- *describe_parameter_set Example # 3: Any-Column Partitioning* on page 148
- *describe_parameter_set Example # 4: No Support for PARTITION BY ANY Clause* on page 150

- *describe_parameter_set Example # 5: No Partitioning Support* on page 151
- *describe_parameter_set Example # 6: One-Column Partitioning on Column 2* on page 153

describe_parameter_set Example # 1: One-Column Partitioning on Column 1

An example UDF that informs the server to perform partitioning on column 1 (c1).

```
void UDF_CALLBACK my_tpf_proc_describe( a_v4_extfn_proc_context
*ctx )
{
    if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {
        a_sql_int32            rc = 0;
        a_v4_extfn_column_list pbcol =
            { 1,        // 1 column in the partition by list
              1 };    // column index 1 requires partitioning

        // Describe partitioning for argument 1 (the table)
        rc = ctx->describe_parameter_set(
ctx,
                  1,
                  EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
                  &pbcol,
                  sizeof(pbcol) );

    if( rc == 0 ) {
          ctx->set_error( ctx, 17000,
            "Runtime error, unable set partitioning requirements for
column." );
      }
   }
}
```

See also

- *Example Procedure Definition* on page 143
- *describe_parameter_set Example # 2: Two-Column Partitioning* on page 146
- *describe_parameter_set Example # 3: Any-Column Partitioning* on page 148
- *describe_parameter_set Example # 4: No Support for PARTITION BY ANY Clause* on page 150
- *describe_parameter_set Example # 5: No Partitioning Support* on page 151
- *describe_parameter_set Example # 6: One-Column Partitioning on Column 2* on page 153

SQL Writer Semantics for One-Column Partitioning on Column 1

Example queries valid for one-column partitioning on column 1 (c1).

Example 1

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.x ) )
```

In this example, the UDF describes to the server that the data is partitioned by the first column (T.x) and the SQL writer also explicitly requests partitioning on the same column. When the two columns match, the above query proceeds without any errors using this negotiated query:

```
my_tpf( TABLE( SELECT T.x, T.y FROM T )
     OVER ( PARTITION BY T.x ) )
V4 describe_parameter_get API returns: { 1, 1 }
```

Example 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY ANY ) )
```

In this example, the UDF describes to the server that the data is partitioned by the first column (T.x) and the SQL writer only wants the query engine to execute the UDF on partitions. The server uses the UDF's preference for partitioning and as a result the same effective query in Example 1 is executed.

Example 3

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T ) )

SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER ( PARTITION BY DEFAULT ) )
```

This example shows that the SQL writer does not include the **PARTITION BY** clause or the **PARTITION BY DEFAULT** clause as part of the input table query specification. In this case, the specification requested by the UDF applies, which is to perform partitioning on column T.x.

SQL Exceptions for One Column Partitioning on Column 1

Example queries not valid for one column partitioning on column 1 (c1). Each example raises a SQL exception.

Example 1

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.y ))
```

In this example the UDF describes to the server that the data is partitioned by the first column (T.x) and that the SQL writer is also explicitly requesting partitioning on a different column (T.y) which conflicts with what the UDF is requesting and as a result the server returns a SQL error.

Example 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( NO PARTITION BY ))
```

This example conflicts with the request made by the UDF because the SQL writer does not want the input table partitioned and as a result the server returns a SQL error.

Example 3

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.x, T.y ))
```

In this example the UDF describes to the server that the data is partitioned by the first column (T.x) and the SQL writer requests partitioning on columns (T.x and T.y) which conflicts with what the UDF is requesting and as a result the server returns a SQL error.

describe_parameter_set Example # 2: Two-Column Partitioning

An example UDF that informs the server to perform partitioning on column 1 (c1) and column 2 (c2).

```
void UDF_CALLBACK my_tpf_proc_describe( a_v4_extfn_proc_context
*ctx )
{
    if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {
        a_sql_int32            rc = 0;
        a_v4_extfn_column_list pbcol =
{ EXTFNAPIV4_PARTITION_BY_COLUMN_ANY };


        // Describe partitioning for argument 1 (the table)
        rc = ctx->describe_parameter_set(
ctx,
                  1,
                  EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
                  &pbcol,
                  sizeof(pbcol) );

    if( rc == 0 ) {
          ctx->set_error( ctx, 17000,
            "Runtime error, unable set partitioning requirements for
column." );
      }
   }
}
```

See also

- *Example Procedure Definition* on page 143
- *describe_parameter_set Example # 1: One-Column Partitioning on Column 1* on page 144
- *describe_parameter_set Example # 3: Any-Column Partitioning* on page 148
- *describe_parameter_set Example # 4: No Support for PARTITION BY ANY Clause* on page 150
- *describe_parameter_set Example # 5: No Partitioning Support* on page 151
- *describe_parameter_set Example # 6: One-Column Partitioning on Column 2* on page 153

SQL Writer Semantics for Two-Column Partitioning

Example queries valid for two-column partitioning on column 1 (c1) and column 2 (c2).

Example 1

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.y, T.x ))
```

In this example, the UDF describes to the server that the data is partitioned by columns T.y and T.x. The SQL writer also requests partitioning on the same column. When the two columns match, the above query proceeds without any errors using this negotiated query:

```
my_tpf( TABLE( SELECT T.x, T.y FROM T )
     OVER ( PARTITION BY T.y, T.x ) )
V4 describe_parameter_get API returns: { 2, 2, 1 }
```

Example 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY ANY ))
```

In this example, the SQL writer does not specify a specific column for partitioning. Instead the SQL writer partitions the input table. The UDF requests partitioning on columns T.y and T.x, and as a result, the server partitions the input data on the columns T.y and T.x.

Example 3

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )

SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER ( PARTITION BY DEFAULT ))
```

This example shows that the SQL writer does not include **PARTITION BY** clause or the **PARTITION BY DEFAULT** clause. The server uses the partition requested by the UDF, and since

the UDF describes that it requires partitioning on columns T.y and T.x, the server executes the query by performing partitioning on columns T.y and T.x.

Example 4

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.x,T.y))
```

This example is semantically identical to Example 1. The ordering of the two columns are different, but within a given partition, the values for columns T.x and T.y stay the same. Both columns (T.x, T.y) and columns (T.y, T.x) result in the same logical partitioning of data.

SQL Exceptions for Two-Column Partitioning

Invalid example queries for two-column partitioning on column 1 (c1) and column 2 (c2). Each example raises a SQL exception.

Example 1

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( NO PARTITION BY ) )
```

This example conflicts with the request made by the UDF because the SQL writer does not want the input table partitioned. As a result, the server returns a SQL error.

Example 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.x ) )

SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.y ) )
```

In this example the UDF describes to the server that the data is partitioned by columns T.y and T.x, while the SQL writer requests the partitioning on either column T.y or T.x. which conflicts with what the UDF is requesting. As a result, the server returns a SQL error.

describe_parameter_set Example # 3: Any-Column Partitioning

An example UDF that informs the server it can perform partitioning on any column.

```
void UDF_CALLBACK my_tpf_proc_describe( a_v4_extfn_proc_context
*ctx )
{
    if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {
        a_sql_int32            rc = 0;
        a_v4_extfn_column_list pbcol =
{ EXTFNAPIV4_PARTITION_BY_COLUMN_ANY };
```

```
        // Describe partitioning for argument 1 (the table)
        rc = ctx->describe_parameter_set(
ctx,
                  1,
                  EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
                  &pbcol,
                  sizeof(pbcol) );

    if( rc == 0 ) {
          ctx->set_error( ctx, 17000,
            "Runtime error, unable set partitioning requirements for
column." );
       }
   }
}
```

See also

- *Example Procedure Definition* on page 143
- *describe_parameter_set Example # 1: One-Column Partitioning on Column 1* on page 144
- *describe_parameter_set Example # 2: Two-Column Partitioning* on page 146
- *describe_parameter_set Example # 4: No Support for PARTITION BY ANY Clause* on page 150
- *describe_parameter_set Example # 5: No Partitioning Support* on page 151
- *describe_parameter_set Example # 6: One-Column Partitioning on Column 2* on page 153

SQL Writer Semantics for Any-Column Partitioning

Example queries valid for any-column partitioning.

Example 1

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.x ) )
```

In this example, the UDF describes to the server that the data is partitioned by the first column (T.x) and the SQL writer also explicitly requests partitioning on the same column. When the two columns match, the above query proceeds without any errors using this negotiated query:

```
my_tpf( TABLE( SELECT T.x, T.y FROM T )
     OVER ( PARTITION BY T.y, T.x ) )
V4 describe_parameter_get API returns: { 2, 2, 1 }
```

Example 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY ANY ) )
```

In this example, nether the SQL writer nor the UDF specify a specific column for partitioning. Iinstead the SQL writer partitions the input table and, as a result, the server arranges partitioning in a nonvalue-based scheme and the data is partitioned over ranges of rows.

describe_parameter_set Example # 4: No Support for PARTITION BY ANY Clause
An example UDF that informs the server that it cannot perform partitioning on any columns, because the UDF does not support the **PARTITION BY ANY** clause.

```
void UDF_CALLBACK my_tpf_proc_describe( a_v4_extfn_proc_context
*ctx )
{
    // No describe calls
}
```

See also

- *Example Procedure Definition* on page 143
- *describe_parameter_set Example # 1: One-Column Partitioning on Column 1* on page 144
- *describe_parameter_set Example # 2: Two-Column Partitioning* on page 146
- *describe_parameter_set Example # 3: Any-Column Partitioning* on page 148
- *describe_parameter_set Example # 5: No Partitioning Support* on page 151
- *describe_parameter_set Example # 6: One-Column Partitioning on Column 2* on page 153

SQL Writer Semantics for No Support for PARTITION BY ANY Clause
Example queries valid when the UDF does not support the **PARTITION BY ANY** clause.

Example 1

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T ))
```

This example shows that the SQL writer does not include **PARTITION BY** clause. The server uses the partition requested by the UDF and since the UDF does not supports any partitioning requirements, the server executes the query without performing any partitioning.

Example 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( NO PARTITION BY ))
```

In this example, the SQL writer requests the **NO PARTITION BY** clause as part of the input table query specification. As a result, the server executes the query with no runtime partitioning.

Example 3

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
```

```
  OVER( PARTITION BY T.x))
```

In this example the UDF does not describe any partitioning requirements. However, the SQL writer requests partitioning by column T.x and as a result the server executes the query by performing partitioning on column T.x.

Example 4

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.y))
```

In this example, the UDF does not describe any partitioning requirements. However, the SQL writer requests partitioning by column T.y. As a result, the server executes the query by performing partitioning on column T.y.

Example 5

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.y, T.x))
```

In this example, the UDF does not describe any partitioning requirements. However, the SQL writer requests partitioning by columns T.y and T.x. As a result, the server executes the query by performing partitioning on columns T.y and T.x.

Example 6

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY ANY ))
```

In this example, the SQL writer requests **PARTITION BY ANY** partitioning. However, the UDF does not support any partitioning requirements. As a result, the server executes the query by performing row range partitioning.

describe_parameter_set Example # 5: No Partitioning Support

An example UDF that informs the server that it does not support any partitioning.

```
void UDF_CALLBACK my_tpf_proc_describe( a_v4_extfn_proc_context
*ctx )
{
    if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {
        a_sql_int32            rc = 0;
        a_v4_extfn_column_list pbcol =
{ EXTFNAPIV4_PARTITION_BY_COLUMN_NONE };


        // Describe partitioning for argument 1 (the table)
        rc = ctx->describe_parameter_set(
```

```
ctx,
                    1,
                    EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
                    &pbcol,
                    sizeof(pbcol) );

    if( rc == 0 ) {
          ctx->set_error( ctx, 17000,
            "Runtime error, unable set partitioning requirements for
column." );
       }
   }
}
```

See also

- *Example Procedure Definition* on page 143
- *describe_parameter_set Example # 1: One-Column Partitioning on Column 1* on page 144
- *describe_parameter_set Example # 2: Two-Column Partitioning* on page 146
- *describe_parameter_set Example # 3: Any-Column Partitioning* on page 148
- *describe_parameter_set Example # 4: No Support for PARTITION BY ANY Clause* on page 150
- *describe_parameter_set Example # 6: One-Column Partitioning on Column 2* on page 153

SQL Writer Semantics for No Partitioning Support

Valid example queries when the UDF does not support any partitioning.

Example 1

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY ANY )
```

In this example, the SQL writer requests **PARTITION BY ANY** partitioning. However, the UDF does not support any partitioning, and as a result, the server executes the query without runtime partitioning.

Example 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )

SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER ( PARTITION BY DEFAULT )
```

This example shows that the SQL writer does not include the **PARTITION BY** clause or the **PARTITION BY DEFAULT** clause. The server uses the partition requested by the UDF and since the UDF does not supports any partitioning, the server executes the query without performing any partitioning.

Example 3

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( NO PARTITION BY )
```

In this example, the SQL writer requests no partitioning, and as a result, the server executes the query without runtime partitioning.

SQL Exceptions for No Partitioning Support

Invalid example queries, because the UDF does not support any partitioning. Each example raises a SQL exception.

Example 1

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.x ))
```

This example results in a SQL error because the SQL writer requested partitioning on column T.x, and the UDF does not support any partitioning on any columns.

Example 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.y ))
```

This example results in a SQL error because the SQL writer requested partitioning on column T.y, and the UDF does not support any partitioning on any columns.

Example 3

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.y, T.x ))
```

This example results in a SQL error because the SQL writer requested partitioning on columns T.y and T.x, and the UDF does not support any partitioning on any columns.

describe_parameter_set Example # 6: One-Column Partitioning on Column 2

An example UDF that informs the server to perform partitioning on column 2 (c2).

```
void UDF_CALLBACK my_tpf_proc_describe( a_v4_extfn_proc_context
*ctx )
{
    if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {
        a_sql_int32            rc = 0;
        a_v4_extfn_column_list pbcol =
            { 1,        // 1 column in the partition by list
              2 };    // column index 2 requires partitioning
```

```
        // Describe partitioning for argument 1 (the table)
        rc = ctx->describe_parameter_set(
ctx,
                  1,
                  EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
                  &pbcol,
                  sizeof(pbcol) );

    if( rc == 0 ) {
          ctx->set_error( ctx, 17000,
            "Runtime error, unable set partitioning requirements for
column." );
       }
   }
}
```

See also

- *Example Procedure Definition* on page 143
- *describe_parameter_set Example # 1: One-Column Partitioning on Column 1* on page 144
- *describe_parameter_set Example # 2: Two-Column Partitioning* on page 146
- *describe_parameter_set Example # 3: Any-Column Partitioning* on page 148
- *describe_parameter_set Example # 4: No Support for PARTITION BY ANY Clause* on page 150
- *describe_parameter_set Example # 5: No Partitioning Support* on page 151

SQL Writer Semantics for One-Column Partitioning on Column 2

Valid example queries for one-column partitioning on column 2 (c2).

Example 1

```
SELECT * FROM my_tpf(
TABLE( SELECT T.x, T.y FROM T )
OVER( PARTITION BY T.y )
```

In this example, the UDF describes to the server that the data is partitioned by the first column (T.y), and the SQL writer also explicitly requests partitioning on the same column. When the two columns match, the above query proceeds without any errors using this negotiated query:

```
my_tpf( TABLE( SELECT T.x, T.y FROM T )
     OVER ( PARTITION BY T.y ) )
V4 describe_parameter_get API returns: { 1, 2 }
```

Example 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY ANY )
```

In this example the SQL writer does not specify a specific column for partitioning. Instead the SQL writer partitions the input table. The UDF requests partitioning on column T.y, and as a result, the server partitions the input data on the column T.y.

Example 3

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )

SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER ( PARTITION BY DEFAULT )
```

This example shows that the SQL writer does not include the **PARTITION BY** clause or the **PARTITION BY DEFAULT** clause as part of the input table query specification. In this case, the specification requested by the UDF applies, which is to perform partitioning on column T.y.

SQL Exceptions for One-Column Partitioning on Column 2

Invalid example queries for one-column partitioning on column 2 (c2). Each example raises a SQL exception.

Example 1

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.x )
```

In this example, the UDF describes to the server that the data is partitioned by the first column (T.y) ,and that the SQL writer is also explicitly requesting partitioning on a different column (T.x), which conflicts with what the UDF is requesting. As a result the server returns a SQL error.

Example 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( NO PARTITION BY )
```

This example conflicts with the request made by the UDF because the SQL writer does not want the input table partitioned. As a result the server returns a SQL error.

Example 3

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.x, T.y )
```

In this example, the UDF describes to the server that the data is partitioned by the first column (T.y), and the SQL writer requests partitioning on columns (T.x and T.y), which conflicts with what the UDF is requesting. As a result the server returns a SQL error.

TPF Implementation Examples

Implementation examples start with a simple TPF and increase in complexity and functionality as the examples progress.

The TPF implementation examples are in the `samples` directory.

The examples are available in a precompiled dynamic library called `libv4apiex`. The extension of this library name is platform-dependent. This library includes the functions defined in `udf_main.cxx`, which contains the library-level functions, such as `extfn_use_new_api`. Put `libv4apiex` in a directory the server can read.

tpf_rg_1

TPF sample `tpf_rg_1.cxx` is similar to the table UDF sample `udf_rg_2.cxx`. It produces rows of data based on an input parameter.

The number of rows generated is the sum of the values of the rows in a single input table. The output is the same as `udf_rg_2.cxx`.

The majority of the code for this sample is the same as `udf_rg_2.cxx`. The main differences are:

- The names of the implementing functions have the prefix `tpf_rg` instead of `udf_rg`. See the file `tpf_rg_1.cxx` for details.
- The implementation of `_describe_extfn` validates the schema of this example but does not estimate the number of rows generated.
- The implementation of `_open_extfn` reads rows from an input table to determine the number of rows to generate.

The `_describe_extfn` method accommodates the schema differences between `udf_rg_2.cxx` and this example. In particular, parameter 1 is a table with one integer column. This code snippet illustrates `_describe_extfn`:

```
static void UDF_CALLBACK tpf_rg_describe(
        a_v4_extfn_proc_context *ctx )
/*************************************/
{
    a_sql_int32        desc_rc;

    // The following describes will ensure that the schema defined
    // by the user matches the schema supported by this TPF
    // This is achieved by telling the server what our schema is
    // using describe_xxxx_set methods.
    if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {

        …
…

        // Inform the server that the type of parameter 1 is a TABLE
        type = DT_EXTFN_TABLE;
        desc_rc = ctx->describe_parameter_set
```

```
            ( ctx,
                  1,
              EXTFNAPIV4_DESCRIBE_PARM_TYPE,
              &type,
                  sizeof( type ) );

        UDF_CHECK_DESCRIBE( ctx, desc_rc );

        // Inform the server that the input table should have a single
        // column.
        num_cols = 1;
        desc_rc = ctx->describe_parameter_set
            ( ctx,
                  1,
              EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS,
              &num_cols,
                  sizeof( num_cols ) );

        UDF_CHECK_DESCRIBE( ctx, desc_rc );

        // Inform the server that the input table column is an integer
        type = DT_INT;
        desc_rc = ctx->describe_column_set
            ( ctx,
              1,
                  1,
              EXTFNAPIV4_DESCRIBE_COL_TYPE,
              &type,
                  sizeof( type ) );

        UDF_CHECK_DESCRIBE( ctx, desc_rc );

…
…
    }
}
```

In `udf_rg_2.cxx`, the number of rows generated by the UDF may be available during the describe phase if the value is a constant. A table argument can never be constant, so its value is unavailable until the Execution state. For this reason, no optimizer estimate for the number of rows being generated is provided during the describe phase.

Calls to the describe only during the Annotation state have an effect in this example. Such calls will do nothing in the other states.

The `_open_extfn` method reads in rows from the input table and sums their values. As in the `udf_rg_2.cxx` example, the value of the first input parameter is retrieved using `get_value`. The difference here is that the type of the parameter is `a_v4_extfn_table` pointer. This code snippet illustrates `_open_extfn`:

```
static short UDF_CALLBACK tpf_rg_open(
    a_v4_extfn_table_context * tctx )
/************************************/
{
```

```
    an_extfn_value             value;
    tpf_rg_state *            state             = NULL;
    a_v4_extfn_table_context *    rs             = NULL;
    a_sql_uint32                num_to_generate    = 0;

    // Read in the value of the input parameter and store it away in a
    // state object.  Save the state object in the context.
    if( !tctx->proc_context->get_value( tctx->args_handle,
                    1,
                    &value ) ) {

        // Send an error to the client if we could not get the value.
        tctx->proc_context->set_error(
            tctx->proc_context,
            17001,
            "Error: Could not get the value of parameter 1" );

        return 0;
    }

    // Open a result set for the input table.
    if( !tctx->proc_context->open_result_set( tctx->proc_context,
                          ( a_v4_extfn_table * )value.data,
                          &rs ) ) {
        // Send an error to the client if we could not open the result
        // set.
        tctx->proc_context->set_error(
            tctx->proc_context,
            17001,
            "Error: Could not open result set on input table." );

        return 0;
    }

    a_v4_extfn_row_block *          rbfb            = NULL;
    a_v4_extfn_row *                rfb            = NULL;
    a_v4_extfn_column_data *        cdfb            = NULL;
    // When using fetch block to read rows from an input table, the
    // server will manage the row block allocation.
    while( rs->fetch_block( rs, &rbfb ) ) {

        // Each successful call to fetch will fill rows in the server
        // allocated row block.  The number of rows retrieved is
        // indicated by the num_rows member.
        for( unsigned int i = 0; i < rbfb->num_rows; i++ ) {
            rfb = &(rbfb->row_data[i]);
            cdfb = &(rfb->column_data[0]);

                // Only consider non-null values.  To determine null we
                // have to use the following logic.
                if( (*cdfb->is_null & cdfb->null_mask) != cdfb-
>null_value ) {
            num_to_generate += *(a_sql_int32 *)cdfb->data;
                }
}
    }
```

```
   if( !tctx->proc_context->close_result_set( tctx->proc_context, rs ) )
{
        // Send an error to the client if we could not close the
        // result set.
        tctx->proc_context->set_error(
            tctx->proc_context,
            17001,
            "Error: Could not close result set on input table." );

        return 0;
    }

    // Allocate memory for the state using the a_v4_extfn_proc_context
    // function alloc.
    state = (tpf_rg_state *)
    tctx->proc_context->alloc( tctx->proc_context,
                  sizeof( tpf_rg_state ) );
    // Start generating at row zero.
    state->next_row = 0;

    // Save the value of parameter 1
    state->max_row = num_to_generate;

    // Save the state on the context
    tctx->user_data = state;

    return 1;
}
```

Once you retrieve the table object using `get_value`, call `open_result_set` to read in rows of data from the table.

To read rows from the input table, the UDF can use `fetch_into` or `fetch_block`. When a UDF is fetching rows from an input table, it becomes a consumer of data. If the consumer (the UDF in this case) wants to be responsible for managing the row block structure, then the consumer must allocate their own row block structure and use `fetch_into` to retrieve the data. Alternatively, if the consumer wants the producer (the server in this case) to manage the row block structure, then use `fetch_block`. `tpf_rg_1` demonstrates the latter.

Using an open result set, `tpf_rg_1` retrieves rows of data from the server by calling `fetch_block` repeatedly. Each successful call to `fetch_block` populates the server allocated row block structure with up to `num_rows` rows. In `tpf_rg_1`, the value of column 1 for each row is added to a total. As in the `udf_rg_2.cxx` example, this total is saved in the `a_v4_extfn_proc_context` state to be used later.

See also

- *Describe API* on page 208
- *_open_extfn* on page 321
- *Table (a_v4_extfn_table)* on page 310
- *_fetch_block_extfn* on page 322

Running the Sample TPF in tpf_rg_1

The sample `tpf_rg_1` is included in a precompiled dynamic library called `libv4apiex` (extension is platform-dependent). Its implementation is in the `samples` directory in `tpf_rg_1.cxx`.

1. Declare the TPF to the server.

```
CREATE OR REPLACE PROCEDURE tpf_rg_1( IN tab TABLE( num INT ) )
RESULT( c1 INT )
EXTERNAL NAME 'tpf_rg_1@libv4apiex';
```

2. Declare a table to use as input to the TPF.

```
CREATE TABLE test_table( val int );
```

3. Insert rows into the table:

```
INSERT INTO test_table values(1);
INSERT INTO test_table values(2);
INSERT INTO test_table values(3);
COMMIT;
```

4. Select rows from the TPF.
   The table test_table has three rows with values 1,2,3. The sum of these values is 6. The example generates 6 rows.

```
SELECT * from tpf_rg_1( TABLE( select val from test_table ) );
```

   a) To see how the `describe` affects the behavior, issue a **CREATE PROCEDURE** statement that has a different schema than the schema the TPF publishes in the `describe`:

```
CREATE OR REPLACE PROCEDURE tpf_rg_1( IN tab TABLE( num INT,
num2 INT ) )
RESULT( c1 INT )
EXTERNAL NAME 'tpf_rg_1@libv4apiex';
```

   b) Select rows from the TPF:

```
// This will return an error that the number of columns in
select list
does not match input table param schema
SELECT * from tpf_rg_1( TABLE( select val from test_table ) );
```

tpf_rg_2

TPF sample `tpf_rg_2.cxx` builds on the sample in `tpf_rg_1.cxx` and has similar behavior. It produces rows of data based on an input parameter.

This sample provides an alternate implementation of the `_open_extfn` method in the `a_v4_extfn_func` descriptor. The behavior is the same as `tpf_rg_1` but the TPF uses `fetch_into` instead of `fetch_block` to read rows from the input table.

This code snippet from the `_open_extfn` method shows `fetch_into` retrieving rows from the input table:

```
static short UDF_CALLBACK tpf_rg_open(
    a_v4_extfn_table_context * tctx )
/*************************************/
{
…
…
…
   // This block of code will create a statically allocated row block
    // that can contain at most 1 row of data.

    a_sql_uint32            c1_data;
    a_sql_byte            c1_null     = 0x0;
    a_sql_uint32            c1_len     = 0;
    a_sql_byte            null_mask    = 0x1;
    a_sql_byte            null_value     = 0x1;
    a_v4_extfn_column_data    cd[1] =
    {
    { &c1_null,        // is_null
      null_mask,        // null_mask
      null_value,        // null_value
      &c1_data,        // data
      &c1_len,        // piece_len
      sizeof(c1_data),    // max_piece_len
      NULL            // blob
    }
    };

    a_sql_uint32        r_status;

    a_v4_extfn_row        row =
    {
    &r_status, &cd[0]
    };

    a_v4_extfn_row_block    rb =
    {
    1, 0, &row
    };

    // We are providing a row block structure that was statically
    // allocated to have a single row.  This means that each call to
    // fetch_into will return at most 1 row.
    while( rs->fetch_into( rs, &rb ) ) {

// Only consider non-null rows.  They way the column data has
// been defined allows us to treat c1_null as a boolean.
if( !c1_null ) {
    num_to_generate += c1_data;
}

    }

    …
    …
}
```

When using `fetch_into` to retrieve rows from an input table, the TPF manages the row block structure. In this example, a static row block structure is created that retrieves one row at a time. Alternatively, you can allocate a dynamic structure that simultaneously supports an arbitrary number of rows.

In the code snippet, the row block structure defined to store the value of the column from the input table in the variable *c1_data*. If a NULL row is encountered, the variable *c1_null* is set to 1 to indicate this.

See also

- *_open_extfn* on page 321
- *_fetch_into_extfn* on page 322

Running the Sample TPF in tpf_rg_2

The sample `tpf_rg_2` is included in a pre-compiled dynamic library called `libv4apiex` (extension is platform-dependent). Its implementation is in the `samples` directory in `tpf_rg_2.cxx`.

1. Issue a **CREATE PROCEDURE** statement to declare the TPF to the server.

   ```
   CREATE OR REPLACE PROCEDURE tpf_rg_2( IN tab TABLE( num INT ) )
   RESULT( c1 INT )
   EXTERNAL NAME 'tpf_rg_2@libv4apiex';
   ```

2. Issue a **CREATE TABLE** statement to declare a table to use as input to the TPF.

   ```
   CREATE TABLE test_table( val INT );
   ```

3. Insert rows into the table.

   ```
   INSERT INTO test_table VALUES(1);
   INSERT INTO test_table VALUES(2);
   INSERT INTO test_table VALUES(3);
   COMMIT;
   ```

4. Select rows from the TPF.

   ```
   SELECT * FROM tpf_rg_2( TABLE( SELECT val FROM test_table ) );
   ```

   The table test_table has three rows with values 1,2,3. The sum of these values is 6. The example generates 6 rows.

Pass-Through TPF in tpf_blob

The TPF sample `tpf_blob.cxx` demonstrates advanced UDF LOB and CLOB handling. The example is available in the `samples` directory. This examination of `tpf_blob` illustrates concepts not already covered by the simpler examples in `tpf_rg_1` and `tpf_rg_2`; only the relevant portions are discussed.

A table UDF or TPF can not produce LOB or CLOB data. However, using a concept known as *pass-through*, LOB or CLOB data can be passed from an input table to an output table. In fact, any data type can be passed through from an input table to the result set. This allows a TPF to *filter* rows, meaning that the output is a subset of the input table rows.

The **CREATE PROCEDURE** statement supported by `tpf_blob` is:

```
CREATE PROCEDURE tpf_blob( IN tab TABLE( num INT, s [LONG] <VARCHAR |
BINARY >,
                           IN pattern char(1) )
RESULT SET ( num INT, s [LONG] <VARCHAR | BINARY > )
EXTERNAL NAME 'tpf_blob@libv4apiex'
```

The procedure supports multiple schemas. The data types for column **s** in the result set and input table can be one of `VARCHAR`, `BINARY`, `LONG VARCHAR`, or `LONG BINARY`.

Dynamic Schema Support

The schema for the `tpf_blob` procedure is dynamic.

The data types for column **s** in the result set and input table can be one of `VARCHAR`, `BINARY`, `LONG VARCHAR`, or `LONG BINARY`. You accomplish this using the `describe_column_get` method of `a_v4_extfn_proc_context` to get the data type of the input table column. The implementation of the TPF is adjusted according to what the actual defined schema is. The interpretation of the *pattern* argument of the procedure differs depending on the data type of column **s**. For character data types, the argument is interpreted as a letter; for binary data types it is interpreted as a digit.

See also

- *External Procedure Context (a_v4_extfn_proc_context)* on page 292
- **describe_column_get* on page 209

Processing LOB and CLOB Columns in Input Tables

A `tpf_blob` example counting the number of occurrences of a pattern in each data row in the input table.

When the procedure is defined as having `LONG VARCHAR` or `LONG BINARY` for column **s**, the data must be processed using the `blob` API. This code snippet from the `fetch_into_extfn` method illustrates how a TPF can use the `blob` API to process LOB and CLOB data from an input table:

```
if( EXTFN_COL_IS_BLOB(cd, 1) ) {
    ret = state->rs->get_blob( state->rs, &cd[1], &blob );

    UDF_SQLERROR_RT( tctx->proc_context,
             "Failed to get blob",
             (ret == 1 && blob != NULL),
             0 );

    if( state->data_type == DT_VARCHAR ||
        state->data_type == DT_LONGVARCHAR ) {

       num = ProcessBlob( tctx->proc_context, blob, state->pattern );
    } else {
        char i = (char)atoi( &(state->pattern) );
        num = ProcessBlob( tctx->proc_context, blob, i );
```

```
    }
    ret = blob->release( blob );
    UDF_SQLERROR_RT( tctx->proc_context,
             "Failed to release blob",
             (ret == 1),
             0 );
    } else {
    if( state->data_type == DT_VARCHAR ||
        state->data_type == DT_LONGVARCHAR ) {
        num = CountNum( (char *)cd[1].data,
*(cd[1].piece_len),
state>pattern );
    } else {
        char i = (char)atoi( &(state->pattern) );
        num = CountNum( (char *)cd[1].data, *(cd[1].piece_len), i );
    }
    }
```

For each of the rows in the input table, the TPF checks if it is a blob using the macro `EXTFN_COL_IS_BLOB`. If it is a blob, then the TPF uses the `get_blob` method of `a_v4_extfn_table_context` to create a blob object for the specified column. On success, the `get_blob` method provides the TPF with an instance of `a_v4_extfn_blob`, which allows the TPF to read in the blob data. Once the TPF is finished with the blob, it should call `release` on it.

The `ProcessBlob` method illustrates how a blob object processes the data:

```
static a_sql_uint64 ProcessBlob(
    a_v4_extfn_proc_context *ctx,
    a_v4_extfn_blob *blob,
    char pattern)
/*****************************/
{
    char buffer[BLOB_ISTREAM_BUFFER_LEN];
    size_t len = 0;
    short ret = 0;

    a_sql_uint64 num = 0;

    a_v4_extfn_blob_istream *is = NULL;

    ret = blob->open_istream( blob, &is );
    UDF_SQLERROR_RT( ctx,
             "Failed to open blob istream",
             (ret == 1 && is != NULL),
             0 );

    for(;;) {
    len = is->get( is, buffer, BLOB_ISTREAM_BUFFER_LEN );
    if( len == 0 ) {
        break;
    }
    num += CountNum( buffer, len, pattern );
    }
```

```
    ret = blob->close_istream( blob, is );
    UDF_SQLERROR_RT( ctx,
             "Failed to close blob istream",
             (ret == 1),
             0 );
    return num;
}
```

The `open_istream` method on the blob object creates an instance of `a_v4_extfn_blob_istream`, which can then be used to read a specified amount of the blob into a buffer using the `get` method.

See also

- *Blob Input Stream (a_v4_extfn_blob_istream)* on page 203
- *Blob (a_v4_extfn_blob)* on page 199
- *get_blob* on page 318
- *fetch_into* on page 313

Passing Input Table Columns to the Result Set

A `tpf_blob` illustrating how a TPF can pass the rows from an input table to the result table, and how to use the `row_status` flag to indicate if a row is present.

This allows the TPF to filter out unwanted rows.

1. During the describe phase, ensure the TPF uses the `describe_column_set` method of `EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT` to inform the server that specific result-set rows are a subset of rows from an input table.
   This code snippet from the `describe_extfn` method illustrates filtering:

```
else if( ctx->current_state == EXTFNAPIV4_STATE_OPTIMIZATION ) {

    // The output columns of this TPF are the same as the first
    // argument's input table columns.  The following describe
    // informs the consumer of this fact.
    a_v4_extfn_col_subset_of_input colMap;

    for( short i = 1; i <= 2; i++ ) {
        colMap.source_table_parameter_arg_num = 1;
        colMap.source_column_number = i;

        desc_rc = ctx->describe_column_set( ctx,
            0, i,
            EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT,
            &colMap, sizeof(a_v4_extfn_col_subset_of_input) );

        UDF_CHECK_DESCRIBE( ctx, desc_rc );
    }
}
```

2. Pass the call to `fetch_into` for the input table the same rowblock structure that was passed into the `fetch_into_extfn` method. This ensures that the rowblock structure for the result set is the same as for the input tables.

See also

- *EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT (Set)* on page 240
- *fetch_into* on page 313
- *_fetch_into_extfn* on page 322

Running the Sample TPF in tpf_blob.cxx

The sample `tpf_blob` is included in a precompiled dynamic library called `libv4apiex` (extension is platform-dependent). Its implementation is in the `samples` directory in `tpf_blob.cxx`.

1. Declare the TPF to the server:

```
CREATE OR REPLACE PROCEDURE tpf_blob( IN tab TABLE( num INT,
                                                    s long
varchar ),
                                                IN pattern char(1) )
RESULT( num INT, s long varchar )
EXTERNAL NAME 'tpf_blob@libv4apiex';
```

2. Declare a table to use as input to the TPF:

```
CREATE TABLE test_table( val INT, str LONG VARCHAR );
```

3. Insert rows into the table:

```
INSERT INTO test_table VALUES(1, 'aaaaaaaaaabbbbbbbbbb');
INSERT INTO test_table VALUES(2, 'aaaaaaaaaaabbbbbbbbbb');
INSERT INTO test_table VALUES(3, 'aaaaaaaaaaaabbbbbbbbbb');
INSERT INTO test_table VALUES(4, 'aaaaaaaaaaaaabbbbbbbbbb');
INSERT INTO test_table VALUES(5, 'aaaaaaaaaaaaaabbbbbbbbbb');
COMMIT;
```

4. Select rows from the TPF:

```
SELECT * FROM tpf_blob( TABLE( SELECT val,str FROM test_table ),
'a' );
```

The table test_table has three rows with an even number of **a**s. Row 1 has 10, row 3 has 12, and row 5 has 14.

SQL Reference for Table UDF and TPF Queries

SQL statement reference for queries referencing table UDFs and TPFs.

ALTER PROCEDURE Statement

Replaces an existing procedure with a modified version. Include the entire modified procedure in the **ALTER PROCEDURE** statement, and reassign user permissions on the procedure.

Quick Links:

Go to Parameters on page 167

Go to Usage on page 168

Go to Standards on page 168

Go to Permissions on page 169

Syntax

Syntax 1

```
ALTER PROCEDURE [ owner.]procedure-name procedure-definition
```

Syntax 2

```
ALTER PROCEDURE [ owner.]procedure-name
   REPLICATE { ON | OFF }
```

Syntax 3

```
ALTER PROCEDURE [ owner.]procedure-name
   SET HIDDEN
```

Syntax 4

```
ALTER PROCEDURE [ owner.]procedure-name
   RECOMPILE
```

Syntax 5

```
ALTER PROCEDURE
   [ owner.]procedure-name ( [ parameter, …] )
   [ RESULT (result-column, ...)]
   EXTERNAL NAME 'external-call' [ LANGUAGE JAVA [ environment-name ] }

external-call – (back to Syntax 5)
   [column-name:]function-name@library; ...

environment-name – (back to Syntax 5)
   DISALLOW | ALLOW SERVER SIDE REQUESTS
```

Parameters

(back to top) on page 167

- **procedure-definition – CREATE PROCEDURE** syntax following the name.

- **REPLICATE –** if a procedure needs to be relocated to other sites using SAP Sybase Replication Server, use the REPLICATE ON clause.
- **SET HIDDEN –** to obfuscate the definition of the associated procedure and cause it to become unreadable. The procedure can be unloaded and reloaded into other databases.

  Note: This setting is irreversible. It is recommended that you retain the original procedure definition outside of the database.
- **RECOMPILE –** recompiles a stored procedure. When you recompile a procedure, the definition stored in the catalog is re-parsed and the syntax is verified.

  The procedure definition is not changed by recompiling. You can recompile procedures with definitions hidden with the SET HIDDEN clause, but their definitions remain hidden.
- **RESULT –** for procedures that generate a result set but do not include a RESULT clause, the database server attempts to determine the result set characteristics for the procedure and stores the information in the catalog. This can be useful if a table referenced by the procedure has been altered to add, remove, or rename columns since the procedure was created.
- **environment-name –** DISALLOW is the default. ALLOW indicates that server-side connections are allowed.

  Note:

  - Do not specify ALLOW unless necessary. Use of teh ALLOW clause slows down certain types of SAP Sybase IQ table joins.
  - Do not use UDFs with both ALLOW SERVER SIDE REQUESTS and DISALLOW SERVER SIDE REQUESTS clauses in the same query.

Usage

(back to top) on page 167

The **ALTER PROCEDURE** statement must include the entire new procedure. You can use **PROC** as a synonym for **PROCEDURE**. Both Watcom and Transact-SQL® dialect procedures can be altered through the use of **ALTER PROCEDURE**. Existing permissions on the procedure are not changed. If you execute **DROP PROCEDURE** followed by **CREATE PROCEDURE**, execute permissions are reassigned.

You cannot combine Syntax 2 with Syntax 1.

When using the **ALTER PROCEDURE** statement for table UDFs, the same set of restrictions apply as for the ***CREATE PROCEDURE*** *Statement (External Procedures).*

Standards

(back to top) on page 167

- SQL—Vendor extension to ISO/ANSI SQL grammar.
- SAP Sybase Database product—Not supported by SAP Adaptive Server® Enterprise.

Permissions

(back to top) on page 167

Alter a Watcom-SQL or Transcat-SQL procedure – Requires one of:

- ALTER ANY PROCEDURE system privilege.
- ALTER ANY OBJECT system privilege.
- You own the procedure.

Alter an external C/C++ or external environment procedure – Requires CREATE EXTERNAL REFERENCE system privilege. Also requires one of:

- ALTER ANY PROCEDURE system privilege.
- ALTER ANY OBJECT system privilege.
- You own the procedure.

See also

- *Table UDF Restrictions* on page 99
- *CREATE PROCEDURE Statement (Table UDF)* on page 169

CREATE PROCEDURE Statement (Table UDF)

Creates an interface to an external table user-defined function (table UDF). Users must be specifically licensed to use table UDFs.

For **CREATE PROCEDURE** reference information for external procedures, see *CREATE PROCEDURE Statement (External Procedures)*. For **CREATE PROCEDURE** reference information for Java UDFs, see *CREATE PROCEDURE Statement (Java UDF)*

Quick Links:

Go to Parameters on page 170

Go to Usage on page 171

Go to Standards on page 172

Go to Permissions on page 172

Syntax

```
CREATE [ OR REPLACE ] PROCEDURE
[ owner.]procedure-name ( [ parameter[, …]] )
| RESULT  result-column [, …] )
[ SQL SECURITY { INVOKER | DEFINER } ]
EXTERNAL NAME 'external-call'
```

```
parameter – (back to Syntax)
    [ IN ] parameter-name data-type [ DEFAULT expression ]
    | [ IN  ] parameter-name table-type

table-type – (back to parameter)
    TABLE ( column-name data-type [, ...] )

external-call – (back to Syntax)
    [column-name:]function-name@library; ...
```

Parameters

(back to top) on page 169

- **IN –** the parameter is an object that provides a value for a scalar parameter or a set of values for a TABLE parameter to the UDF.

  Note: TABLE parameters cannot be declared as INOUT or OUT. You can only have one TABLE parameter (the position of which is not important).

- **OR REPLACE –** specifying **OR REPLACE** (**CREATE OR REPLACE PROCEDURE**) creates a new procedure, or replaces an existing procedure with the same name. This clause changes the definition of the procedure, but preserves existing permissions. An error is returned if you attempt to replace a procedure that is already in use.

- **RESULT –** declares the column names and their data types for the result set of the external UDF. The data types of the columns must be a valid SQL data type (e.g., a column in the result set cannot have TABLE as data type). The set of datums in the result implies the TABLE. External UDFs can only have one result set of type TABLE.

  Note: TABLE is not an output value. A table UDF cannot have LONG VARBINARY or LONG VARCHAR data types in its result set, but a table parameterized function (TPF) can have large object (LOB) data in its result set.

  A TPF cannot produce LOB data, but can have columns in the result set as LOB data types. However, the only way to get LOB data in the output is to pass a column from an input table to the output table. The describe attribute `EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT` allows this, as illustrated in the sample file `tpf_blob.cxx`.

- **SQL SECURITY –** defines whether the procedure is executed as the INVOKER (the user who is calling the UDF), or as the DEFINER (the user who owns the UDF). The default is DEFINER.

  When SQL SECURITY INVOKER is specified, more memory is used because annotation must be done for each user that calls the procedure. Also, when SQL SECURITY INVOKER is specified, name resolution is done as the invoker as well. Therefore, care should be taken to qualify all object names (tables, procedures, and so on) with their appropriate owner. For example, suppose user1 creates this procedure:

```
CREATE PROCEDURE user1.myProcedure()
    RESULT( columnA INT )
    SQL SECURITY INVOKER
    BEGIN
        SELECT columnA FROM table1;
    END;
```

If user2 attempts to run this procedure and a table user2.table1 does not exist, a table lookup error results. Additionally, if a user2.table1 does exist, that table is used instead of the intended user1.table1. To prevent this situation, qualify the table reference in the statement (user1.table1, instead of just table1).

- **EXTERNAL NAME –** An external UDF must have EXTERNAL NAME clause which defines an interface to a function written in a programming language such as C. The function is loaded by the database server into its address space.

  The library name can 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. This is a formal example.

  ```
  CREATE PROCEDURE mystring( IN instr CHAR(255),
      IN input_table TABLE(A INT) )
      RESULT (CHAR(255))
  EXTERNAL NAME
  'mystring@mylib.dll;Unix:mystring@mylib.so'
  ```

  A simpler way to write the preceding EXTERNAL NAME clause, using platform-specific defaults, is as follows:

  ```
  CREATE PROCEDURE mystring( IN instr CHAR(255),
      IN input_table TABLE(A INT) )
      RESULT (CHAR(255))
  EXTERNAL NAME 'mystring@mylib'
  ```

Usage

(back to top) on page 169

You define table UDFs using the `a_v4_extfn` API. **CREATE PROCEDURE** statement reference information for external procedures that do not use the `a_v3_extfn` or `a_v4_extfn` APIs is located in a separate topic. **CREATE PROCEDURE** statement reference information for Java UDFs is located in a separate topic.

The **CREATE PROCEDURE** statement creates a procedure in the database. To create a procedure for themselves, a user must have the CREATE PROCEDURE system privilege. To create a procedure for others, a user must specify the owner of the procedure and must have either the CREATE ANY PROCEDURE or CREATE ANY OBJECT system privilege. If the procedure contains an external reference, the user must have the CREATE EXTERNAL REFERENCE system privilege in addition to previously mentioned system privileges, regardless of who owns the procedure.

If a stored procedure returns a result set, it cannot also set output parameters or return a return value.

When referencing a temporary table from multiple procedures, a potential issue can arise if the temporary table definitions are inconsistent and statements referencing the table are cached. Use caution when referencing temporary tables within procedures.

You can use the **CREATE PROCEDURE** statement to create external table UDFs implemented in a different programming language than SQL. However, be aware of the table UDF restrictions before creating external UDFs.

The data type for a scalar parameter, a result column, and a column of a TABLE parameter must be a valid SQL data type.

Parameter names must conform to the rules for other database identifiers such as column names. They must be a valid SQL data type.

TPFs support a mix scalar parameters and single TABLE parameter. A TABLE parameter must define a schema for an input set of rows to be processed by the UDF. The definition of a TABLE parameter includes column names and column data types.

```
TABLE(c1 INT, c2 CHAR(20))
```

The above example defines a schema with the two columns c1 and c2 of types INT and CHAR(20). Each row processed by the UDF must be a tuple with two (2) values. TABLE parameters, unlike scalar parameters cannot be assigned a default value.

Standards

(back to top) on page 169

- SQL—ISO/ANSI SQL compliant.
- SAP Sybase Database product—The Transact-SQL **CREATE PROCEDURE** statement is different.
- SQLJ—The syntax extensions for Java result sets are as specified in the proposed SQLJ1 standard.

Permissions

(back to top) on page 169

Unless creating a temporary procedure, a user must have the CREATE PROCEDURE system privilege to create a UDF for themselves. To create a UDF for others, they must specify the owner of the procedure and must have either the CREATE ANY PROCEDURE or CREATE ANY OBJECT system privilege. If the procedure contains an external reference, a user must also have the CREATE EXTERNAL REFERENCE system privilege, in addition to the previously mentioned system privileges.

See also

- *Sample Files* on page 99

CREATE FUNCTION Statement

Creates a user-defined function in the database. A function can be created for another user by specifying an owner name. Subject to permissions, a user-defined function can be used in exactly the same way as other non-aggregate functions.

Quick Links:

Go to Parameters on page 174

Go to Examples on page 177

Go to Usage on page 178

Go to Standards on page 178

Go to Permissions on page 179

Syntax

Syntax 1

```
CREATE [ OR REPLACE ] [ TEMPORARY ] FUNCTION [ owner.]function-name
( [ parameter, … ] )
    [ SQL SECURITY { INVOKER | DEFINER } ]
    RETURNS data-type ON EXCEPTION RESUME
    | [ NOT ] DETERMINISTIC
    { compound-statement | AS tsql-compound-statement
    | EXTERNAL NAME library-call
    | EXTERNAL NAME java-call LANGUAGE JAVA }
```

Syntax 2

```
CREATE FUNCTION [ owner.]function-name ( [ parameter, … ] )
    RETURNS data-type
    URL url-string
    [ HEADER header-string ]
    [ SOAPHEADER soap-header-string ]
    [ TYPE { 'HTTP[:{ GET | POST } ] ' | 'SOAP[:{ RPC | DOC } ]' } ]
    [ NAMESPACE namespace-string ]
    [ CERTIFICATE certificate-string ]
    [ CLIENTPORT clientport-string ]
    [ PROXY proxy-string ]

parameter – (back to Syntax 1) or (back to Syntax 2)
    IN parameter-name data-type [ DEFAULT expression ]

tsql-compound-statement – (back to Syntax 1)
    sql-statement
    sql-statement …

library-call – (back to Syntax 1)
    '[ operating-system:]function-name@library; …'

operating-system – (back to library-call)
    UNIX
```

```
java-call – (back to Syntax 1)
    '[ package-name.]class-name.method-name method-signature'

method-signature – (back to java-call)
    ( [ field-descriptor, ….] ) return-descriptor

field-descriptor and return-descriptor – (back to method-signature)
    Z | B | S | I | J | F | D | C | V | [descriptor | L class-name;

url-string – (back to Syntax 2)
    ' { HTTP | HTTPS | HTTPS_FIPS }://[user:password@]hostname[:port][/
path] '
```

Parameters

(back to top) on page 173

- **CREATE [OR REPLACE] –** parameter names must conform to the rules for database identifiers. They must have a valid SQL data type and be prefixed by the keyword IN, signifying that the argument is an expression that provides a value to the function.

  The CREATE clause creates a new function, while the OR REPLACE clause replaces an existing function with the same name. When a function is replaced, the definition of the function is changed but the existing permissions are preserved. You cannot use the OR REPLACE clause with temporary functions.

- **TEMPORARY –** the function is visible only by the connection that created it, and that it is automatically dropped when the connection is dropped. Temporary functions can also be explicitly dropped. You cannot perform **ALTER**, **GRANT**, or **REVOKE** operations on them, and unlike other functions, temporary functions are not recorded in the catalog or transaction log.

  Temporary functions execute with the permissions of their creator (current user), and can only be owned by their creator. Therefore, do not specify owner when creating a temporary function. They can be created and dropped when connected to a read-only database.

- **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 INVOKER is specified, more memory is used because annotation must be done for each user that calls the procedure. Also, name resolution is done as the invoker as well. Therefore, take care to qualify all object names (tables, procedures, and so on) with their appropriate owner.

- **data-type –** LONG BINARY and LONG VARCHAR are not permitted as return-value data types.

- **compound-statement –** a set of SQL statements bracketed by **BEGIN** and **END**, and separated by semicolons. See *BEGIN … END Statement*.
- **tsql-compound-statement –** a batch of Transact-SQL statements.
- **external-name –** a wrapper around a call to a function in an external library and 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.
- **LANGUAGE JAVA –** a wrapper around a Java method. For information on calling Java procedures, see *CREATE PROCEDURE Statement*.
- **ON EXCEPTION RESUME –** uses Transact-SQL-like error handling. See *CREATE PROCEDURE Statement*.
- **[NOT] DETERMINISTIC –** function is re-evaluated each time it is called in a query. The results of functions not specified in this manner may be cached for better performance, and re-used each time the function is called with the same parameters during query evaluation.

  Functions that have side effects, such as modifying the underlying data, should be declared as NOT DETERMINISTIC. For example, a function that generates primary key values and is used in an **INSERT … SELECT** statement should be declared NOT DETERMINISTIC:

```
CREATE FUNCTION keygen( increment INTEGER )
RETURNS INTEGER
NOT DETERMINISTIC
BEGIN
  DECLARE keyval INTEGER;
  UPDATE counter SET x = x + increment;
  SELECT counter.x INTO keyval FROM counter;
  RETURN keyval
END
INSERT INTO new_table
SELECT keygen(1), ...
FROM old_table
```

  Functions may be declared as DETERMINISTIC if they always return the same value for given input parameters. All user-defined functions are treated as deterministic unless they are declared NOT DETERMINISTIC. Deterministic functions return a consistent result for the same parameters and are free of side effects. That is, the database server assumes that two successive calls to the same function with the same parameters will return the same result without unwanted side-effects on the semantics of the query.
- **URL –** for use only when defining an HTTP or SOAP web services client function. Specifies the URL of the web service. The optional user name and password parameters provide a means of supplying the credentials needed for HTTP basic authentication. HTTP basic authentication base-64 encodes the user and password information and passes it in the “Authentication” header of the HTTP request.

For web service client functions, the return type of SOAP and HTTP functions must one of the character data types, such as VARCHAR. The value returned is the body of the HTTP response. No HTTP header information is included. If more information is required, such as status information, use a procedure instead of a function.

Parameter values are passed as part of the request. The syntax used depends on the type of request. For HTTP:GET, the parameters are passed as part of the URL; for HTTP:POST requests, the values are placed in the body of the request. Parameters to SOAP requests are always bundled in the request body.

- **HEADER –** when creating HTTP web service client functions, use this clause to add or modify HTTP request header entries. Only printable ASCII characters can be specified for HTTP headers, and they are case-insensitive. For more information about how to use this clause, see the HEADER clause of the *CREATE PROCEDURE Statement*.
- **SOAPHEADER –** when declaring a SOAP Web service as a function, use this clause to specify one or more SOAP request header entries. A SOAP header can be declared as a static constant, or can be dynamically set using the parameter substitution mechanism (declaring IN, OUT, or INOUT parameters for hd1, hd2, and so on). A web service function can define one or more IN mode substitution parameters, but cannot define an INOUT or OUT substitution parameter.
- **TYPE –** specifies the format used when making the web service request. If SOAP is specified or no type clause is included, the default type SOAP:RPC is used. HTTP implies HTTP:POST. Since SOAP requests are always sent as XML documents, HTTP:POST is always used to send SOAP requests.
- **NAMESPACE –** applies to SOAP client functions only and identifies the method namespace usually required for both SOAP:RPC and SOAP:DOC requests. The SOAP server handling the request uses this namespace to interpret the names of the entities in the SOAP request message body. The namespace can be obtained from the WSDL description of the SOAP service available from the web service server. The default value is the procedure's URL, up to but not including the optional path component.
- **CERTIFICATE –** to make a secure (HTTPS) request, a client must have access to the certificate used by the HTTPS server. The necessary information is specified in a string of semicolon-separated key/value pairs. The certificate can be placed in a file and the name of the file provided using the file key, or the whole certificate can be placed in a string, but not both. These keys are available:

Key	Abbreviation	Description
file		File name of certificate
certificate	cert	The certificate
company	co	Company specified in the certificate

Key	Abbreviation	Description
unit		Company unit specified in the certificate
name		Common name specified in the certificate

Certificates are required only for requests that are either directed to an HTTPS server or can be redirected from an insecure to a secure server.

- **CLIENTPORT –** identifies the port number on which the HTTP client procedure communicates using TCP/IP. It is provided for and recommended only for connections across firewalls, as firewalls filter according to the TCP/UDP port. You can specify a single port number, ranges of port numbers, or a combination of both; for example, CLIENTPORT '85,90-97'.
- **PROXY –** specifies the URI of a proxy server. For use when the client must access the network through a proxy. Indicates that the procedure is to connect to the proxy server and send the request to the web service through it.

Examples

(back to top) on page 173

- **Example 1 –** concatenates a `firstname` string and a `lastname` string:

```
CREATE FUNCTION fullname (
  firstname CHAR(30),
  lastname CHAR(30) )
RETURNS CHAR(61)
BEGIN
  DECLARE name CHAR(61);
  SET name = firstname || ' ' || lastname;
  RETURN (name);
END
```

This example illustrates the use of the **fullname** function.

- Return a full name from two supplied strings:

```
SELECT fullname ('joe','smith')
```

fullname('joe', 'smith')
joe smith

- List the names of all employees:

```
SELECT fullname (givenname, surname)
FROM Employees
```

fullname (givenname, surname)
Fran Whitney
Matthew Cobb
Philip Chin
Julie Jordan
Robert Breault
...

- **Example 2 –** uses Transact-SQL syntax:

```
CREATE FUNCTION DoubleIt ( @Input INT )
RETURNS INT
AS
DECLARE @Result INT
SELECT @Result = @Input * 2
RETURN @Result
```

  The statement `SELECT DoubleIt( 5 )` returns a value of `10`.

- **Example 3 –** creates an external function written in Java:

```
CREATE FUNCTION dba.encrypt( IN name char(254) )
RETURNS VARCHAR
EXTERNAL NAME
'Scramble.encrypt (Ljava/lang/String;)Ljava/lang/String;'
LANGUAGE JAVA
```

Usage

(back to top) on page 173

To modify a user-defined function, or to hide the contents of a function by scrambling its definition, use the **ALTER FUNCTION** statement.

When functions are executed, not all parameters need to be specified. If a default value is provided in the **CREATE FUNCTION** statement, missing parameters are assigned the default values. If an argument is not provided by the caller and no default is set, an error is given.

Side Effects

- Automatic commit

Standards

(back to top) on page 173

- SQL—ISO/ANSI SQL compliant.
- SAP Sybase Database product—Not supported by Adaptive Server.

Permissions

(back to top) on page 173

For function to be owned by self – Requires the CREATE PROCEDURE system privilege.

For function to be owned by any user – Requires one of:

- CREATE ANY PROCEDURE system privilege.
- CREATE ANY OBJECT system privilege.

To create a function containing an external reference, regardless of whether or not they are the owner of the function, also requires the CREATE EXTERNAL REFERENCE system privilege.

DEFAULT_TABLE_UDF_ROW_COUNT Option

Enables you to override the default estimate of the number of rows to return from a table UDF (either a C, C++, or Java table UDF).

Allowed Values
0 to 4294967295

Default
200000

Scope
Option can be set at the database (PUBLIC) or user level. When set at the database level, the value becomes the default for any new user, but has no impact on existing users. When set at the user level, overrides the PUBLIC value for that user only. No system privilege is required to set option for self. System privilege is required to set at database level or at user level for any user other than self.

Requires the SET ANY PUBLIC OPTION system privilege to set this option. Can be set temporary for an individual connection or for the PUBLIC role. Takes effect immediately.

Remarks
A table UDF can use the **DEFAULT_TABLE_UDF_ROW_COUNT** option to give the query processor an estimate for the number of rows that a table UDF will return. This is the only way a Java table UDF can convey this information. However, for a C or C++ table UDF, the UDF developer should consider publishing this information in the `describe` phase using the `EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS` describe parameter to publish the number of rows it expects to return. The value of `EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS` always overrides the value of the **DEFAULT_PROXY_TABLE_UDF_ROW_COUNT** option.

See also

- *Query Processing States* on page 121

TABLE_UDF_ROW_BLOCK_CHUNK_SIZE_KB Option

Controls the size, in kilobytes, for server-allocated row blocks. Row blocks are used by Table UDFs and TPFs.

Allowed Values
0 to 4294967295

Default
128

Scope
Option can be set at the database (PUBLIC) or user level. When set at the database level, the value becomes the default for any new user, but has no impact on existing users. When set at the user level, overrides the PUBLIC value for that user only. No system privilege is required to set option for self. System privilege is required to set at database level or at user level for any user other than self.

Requires the SET ANY PUBLIC OPTION system privilege to set this option. Can be set temporary for an individual connection or for the PUBLIC role. Takes effect immediately.

Description
Specifies the row block size, in kilobytes, to fetch from the server.

The server allocates row blocks when you use `fetch_into` to fetch rows from a table UDF, and when you use `fetch_block` to fetch rows from a TPF input table.

The row block contains as many rows as will fit into the specified size. If you specify a row block size smaller than the size required for a single row, the server allocates the size of one row.

FROM Clause

Specifies the database tables or views involved in a **SELECT** statement.

Quick Links:

Go to Parameters on page 183

Go to Examples on page 186

Go to Usage on page 186

Go to Standards on page 187

Go to Permissions on page 188

Syntax

```
...FROM table-expression [,...]
```

```
table-expression – (back to Syntax)
    table-name
    | view-name
    | procedure-name
    | common-table-expression
    | (subquery) [[ AS ] derived-table-name [ column_name, ...) ]]
    | derived-table
    | join-expression
    | ( table-expression , ... )
    | openstring-expression
    | apply-expression
    | contains-expression
    | dml-derived-table

table-name – (back to table-expression)
    [ userid.] table-name ]
    [ [ AS ] correlation-name ]
    [ FORCE INDEX ( index-name ) ]

view-name – (back to table-expression)
    [ userid.]view-name [ [ AS ] correlation-name ]

procedure-name – (back to table-expression)
    [  owner, ] procedure-name ([  parameter, ...])
    [  WITH(column-name datatype, )]
    [ [ AS ] correlation-name ]

parameter – (back to procedure-name)
    scalar-expression | table-parameter

table-parameter – (back to parameter)
    TABLE (select-statement) [ OVER ( table-parameter-over )]

table-parameter-over – (back to table-parameter)
    [ PARTITION BY {ANY
    | NONE| table-expression } ]
    [ ORDER BY { expression | integer }
    [ ASC | DESC ] [, ...] ]

derived-table – (back to table-expression)
    ( select-statement )
        [ AS ] correlation-name [ ( column-name, ... ) ]

join-expression – (back to table-expression)
    table-expression join-operator table-expression
        [ ON join-condition ]

join-operator – (back to join-expression)
    [ KEY | NATURAL ] [ join-type ] JOIN | CROSS JOIN

join-type – (back to join-operator)
    INNER
      | LEFT [ OUTER ]
      | RIGHT [ OUTER ]
      | FULL [ OUTER ]
```

```
openstring-expression – (back to table-expression)
    OPENSTRING ( { FILE | VALUE } string-expression )
      WITH ( rowset-schema )
        [ OPTION ( scan-option ... ) ]
        [ AS ] correlation-name

apply-expression – (back to table-expression)
    table-expression { CROSS | OUTER } APPLY table-expression

contains-expression – (back to table-expression)
    { table-name | view-name } CONTAINS
    ( column-name [,...], contains-query )
    [ [ AS ] score-correlation-name ]

rowset-schema – (back to openstring-expression)
    column-schema-list
        | TABLE [owner.]table-name [ ( column-list ) ]

column-schema-list – (back to rowset-schema)
    { column-name user-or-base-type | filler( ) } [ , ... ]

column-list – (back to rowset-schema)
    { column-name | filler( ) } [ , ... ]

scan-option – (back to openstring-expression)
    BYTE ORDER MARK { ON | OFF }
    | COMMENTS INTRODUCED BY comment-prefix
    | DELIMITED BY string
    | ENCODING encoding
    | ESCAPE CHARACTER character
    | ESCAPES { ON | OFF }
    | FORMAT { TEXT | BCP }
    | HEXADECIMAL { ON | OFF }
    | QUOTE string
    | QUOTES { ON | OFF }
    | ROW DELIMITED BY string
    | SKIP integer
    | STRIP { ON | OFF | LTRIM | RTRIM | BOTH }

contains-query – (back to contains-expression)
    string

dml-derived-table – (back to table-expression)
    ( dml-statement ) REFERENCING ( [ table-version-names | NONE ] )

dml-statement – (back to dml-derived-table)
    insert-statement
    update-statement
    delete-statement

table-version-names – (back to dml-derived-table)
    OLD [ AS ] correlation-name [ FINAL [ AS ] correlation-name ]
      | FINAL [ AS ] correlation-name
```

Parameters

(back to top) on page 180

- **table-name –** a base table or temporary table. Tables owned by a different user can be qualified by specifying the user ID. Tables owned by groups to which the current user belongs are found by default without specifying the user ID.
- **view-name –** specifies a view to include in the query. As with tables, views owned by a different user can be qualified by specifying the user ID. Views owned by groups to which the current user belongs are found by default without specifying the user ID. Although the syntax permits table hints on views, these hints have no effect.
- **procedure-name –** a stored procedure that returns a result set. This clause applies to the FROM clause of SELECT statements only. The parentheses following the procedure name are required even if the procedure does not take parameters. DEFAULT can be specified in place of an optional parameter.
- **parameter –** specifies a scalar-parameter or table-parameter clause. A scalar-parameter are any objects of a valid SQL datatype. A table-parameter can be specified using a table, view or common table-expression name which are treated as new instance of this object if the object is also used outside the table-parameter.

  This query illustrates a valid **FROM** clause where the two references to the same table T are treated as two different instances of the same table T.

```
SELECT * FROM T, my_proc(TABLE(SELECT T.Z, T.X FROM T)
OVER(PARTITION BY T.Z));
```

  Table Parameterized Function (TPF) Example—This query illustrates a valid **FROM** clause.

```
SELECT * FROM R, SELECT * FROM my_udf(1);
SELECT * FROM my_tpf(1, TABLE(SELECT c1, c2 FROM t))
   (my_proc(R.X, TABLE T OVER PARTITION BY T.X)) AS XX;
```

  If a subquery is used to define the TABLE parameter, then the following restrictions must hold:

  - The table-parameter clause must be of type IN.
  - PARTITION BY or ORDER BY clauses must refer to the columns of the derived table and outer references. An expression in the expression-list can be an integer K which refers to the Kth column of the TABLE input parameter.

Note: A Table UDF can only be referenced in a **FROM** clause of a SQL statement.

- **PARTITION BY –** logically specifies how the invocation of the function will be performed by the execution engine. The execution engine must invoke the function for each partition and the function must process a whole partition in each invocation.

  PARTITION BY clause also specifies how the input data must be partitioned such that each invocation of the function will process exactly one partition of data. The function

must be invoked the number of times equal to the number of partitions. For TPF, the parallelism characteristics are established through dynamic negotiation between the server and the UDF at the runtime. If the TPF can be executed in parallel, for N input partitions, the function can be instantiated M times, with M <=N. Each instantiation of the function can be invoked more than once, each invocation consuming exactly one partition.

You can specify only one TABLE input parameter for PARTITION BY *expression-list* or PARTITION BY ANY clause. For all other TABLE input parameters you must specify, explicit or implicit PARTITION BY NONE clause.

Note: The execution engine can invoke the function in any order of the partitions and the function is assumed to return the same result sets regardless of the partitions order. Partitions cannot be split among two invocations of the function.

- **ORDER BY –** specifies that the input data in each partition is expected to be sorted by *expression-list* by the execution engine. The UDF expects each partition to have this physical property. If only one partition exists, the whole input data is ordered based on the ORDER BY specification. ORDER BY clause can be specified for any of the TABLE input parameters with PARTITION BY NONE or without PARTITION BY clause.
- **derived-table –** you can supply a SELECT statement instead of table or view name in the FROM clause. A SELECT statement used in this way is called a derived table, and it must be given an alias. For example, the following statement contains a derived table, MyDerivedTable, which ranks products in the Products table by UnitPrice.

```
SELECT TOP 3 *
  FROM ( SELECT Description,
         Quantity,
         UnitPrice,
         RANK() OVER ( ORDER BY UnitPrice ASC )
         AS Rank
         FROM Products ) AS MyDerivedTable
         ORDER BY Rank;
```

- **join-expression, join-operator, join-type –** the join-type keywords are:

Keyword	Description
CROSS JOIN	Returns the Cartesian product (cross product) of the two source tables
NATURAL JOIN	Compares for equality all corresponding columns with the same names in two tables (a special case equijoin; columns are of same length and data type)
KEY JOIN	Restricts foreign-key values in the first table to be equal to the primary-key values in the second table
INNER JOIN	Discards all rows from the result table that do not have corresponding rows in both tables

Keyword	Description
LEFT OUTER JOIN	Preserves unmatched rows from the left table, but discards unmatched rows from the right table
RIGHT OUTER JOIN	Preserves unmatched rows from the right table, but discards unmatched rows from the left table
FULL OUTER JOIN	Retains unmatched rows from both the left and the right tables

Do not mix comma-style joins and keyword-style joins in the **FROM** clause. The same query can be written two ways, each using one of the join styles. The ANSI syntax keyword style join is preferable.

This query uses a comma-style join:

```
SELECT *
  FROM Products pr, SalesOrders so, SalesOrderItems si
  WHERE pr.ProductID = so.ProductID
    AND pr.ProductID = si.ProductID;
```

The same query can use the preferable keyword-style join:

```
SELECT *
  FROM Products pr INNER JOIN SalesOrders so
    ON (pr.ProductID = so.ProductID)
  INNER JOIN SalesOrderItems si
    ON (pr.ProductID = si.ProductID);
```

The ON clause filters the data of inner, left, right, and full joins. Cross joins do not have an ON clause. In an inner join, the ON clause is equivalent to a WHERE clause. In outer joins, however, the ON and WHERE clauses are different. The ON clause in an outer join filters the rows of a cross product and then includes in the result the unmatched rows extended with nulls. The WHERE clause then eliminates rows from both the matched and unmatched rows produced by the outer join. You must take care to ensure that unmatched rows you want are not eliminated by the predicates in the WHERE clause.

You cannot use subqueries inside an outer join ON clause.

- **openstring-expression –** Specify an OPENSTRING clause to query within a file or a BLOB, treating the content of these sources as a set of rows. When doing so, you also specify information about the schema of the file or BLOB for the result set to be generated, since you are not querying a defined structure such as a table or view. This clause applies to the FROM clause of a SELECT statement. It is not supported for UPDATE or DELETE statements.
- **apply-expression –** Use this clause to specify a join condition where the right table-expression is evaluated for every row in the left table-expression. For example, you can use an apply expression to evaluate a function, procedure, or derived table for each row in a table expression.

- **contains-expression –** Use the CONTAINS clause after a table name to filter the table, and return only those rows matching the full text query specified with contains-query. Every matching row of the table is returned, along with a score column that can be referred to using score-correlation-name, if it is specified. If score-correlation-name is not specified, then the score column can be referred to by the default correlation name, contains.
- **dml-derived-table –** Supports the use of a DML statement (INSERT, UPDATE, or DELETE) as a table expression in a query's FROM clause.

Examples

(back to top) on page 180

- **Example 1 –** these are valid **FROM** clauses:

```
...
FROM Employees
...
...
FROM Employees NATURAL JOIN Departments
...
...
FROM Customers
KEY JOIN SalesOrders
KEY JOIN SalesOrderItems
KEY JOIN Products
...
```

- **Example 2 –** this query illustrates how to use derived tables in a query:

```
SELECT Surname, GivenName, number_of_orders
FROM Customers JOIN
     ( SELECT CustomerID, count(*)
       FROM SalesOrders
        GROUP BY CustomerID )
     AS sales_order_counts ( CustomerID,
                             number_of_orders )
ON ( Customers.ID = sales_order_counts.cust_id )
WHERE number_of_orders > 3
```

Usage

(back to top) on page 180

The **SELECT** statement requires a table list to specify which tables are used by the statement.

Note: Although this description refers to tables, it also applies to views unless otherwise noted.

The **FROM** table list creates a result set consisting of all the columns from all the tables specified. Initially, all combinations of rows in the component tables are in the result set, and the number of combinations is usually reduced by join conditions and/or **WHERE** conditions.

Tables owned by a different user can be qualified by specifying the *userid*. Tables owned by roles to which the current user belongs are found by default without specifying the user ID.

The correlation name is used to give a temporary name to the table for this SQL statement only. This is useful when referencing columns that must be qualified by a table name but the table name is long and cumbersome to type. The correlation name is also necessary to distinguish between table instances when referencing the same table more than once in the same query. If no correlation name is specified, then the table name is used as the correlation name for the current statement.

If the same correlation name is used twice for the same table in a table expression, that table is treated as if it were only listed once. For example, in:

```
SELECT *
FROM SalesOrders
KEY JOIN SalesOrderItems,
SalesOrders
KEY JOIN Employees
```

The two instances of the `SalesOrders` table are treated as one instance that is equivalent to:

```
SELECT *
FROM SalesOrderItems
KEY JOIN SalesOrders
KEY JOIN Employees
```

By contrast, the following is treated as two instances of the `Person` table, with different correlation names HUSBAND and WIFE.

```
SELECT *
FROM Person HUSBAND, Person WIFE
```

Join columns require like data types for optimal performance.

- **Performance Considerations –** Depending on the query, SAP Sybase IQ allows between 16 and 64 tables in the **FROM** clause with the optimizer turned on; however, performance might suffer if you have more than 16 to 18 tables in the **FROM** clause in very complex queries.

  Note: If you omit the **FROM** clause, or if all tables in the query are in the `SYSTEM` dbspace, the query is processed by SQL Anywhere instead of SAP Sybase IQ and might behave differently, especially with respect to syntactic and semantic restrictions and the effects of option settings.

  If you have a query that does not require a **FROM** clause, you can force the query to be processed by SAP Sybase IQ by adding the clause **FROM iq_dummy**, where `iq_dummy` is a one-row, one-column table that you create in your database.

Standards

(back to top) on page 180

- SQL—ISO/ANSI SQL compliant.
- SAP Sybase Database product—The **JOIN** clause is not supported in some versions of Adaptive Server. Instead, you must use the **WHERE** clause to build joins.

Permissions

(back to top) on page 180

Must be connected to the database.

SELECT Statement

Retrieves information from the database.

Quick Links:

Go to Parameters on page 189

Go to Examples on page 195

Go to Usage on page 196

Go to Standards on page 197

Go to Permissions on page 197

Syntax

```
SELECT [ ALL | DISTINCT ] [ row-limitation-option1 ] select-list
    … [ INTO { host-variable-list | variable-list | table-name } ]
    … [ INTO LOCAL TEMPORARY TABLE { table-name } ]
    … [ FROM table-list ]
    … [ WHERE search-condition ]
    … [ GROUP BY [ expression [, ...]
          | ROLLUP ( expression [, ...] )
          | CUBE ( expression [, ...] ) ] ]
    … [ HAVING search-condition ]
    … [ ORDER BY { expression | integer } [ ASC | DESC ] [, ...] ]
    | [ FOR JSON json-mode ]
    … [ row-limitation-option ]

select-list – (back to Syntax)
    { column-name
    | expression [ [ AS ] alias-name ]
    | * }

row-limitation-option1 – (back to Syntax)
    FIRST
    | TOP {ALL | limit-expression} [START AT startat-expression ]

limit-expression – (back to row-limitation-option1) or (back to row-
limitation-option2)
     simple-expression

startat-expression – (back to row-limitation-option1)
```

```
    simple-expression

row-limitation-option2 – (back to Syntax)
    LIMIT { [ offset-expression , ] limit-expression
    | limit-expression OFFSET offset-expression }

offset-expression – (back to row-limitation-option2)
    simple-expression

simple-expression – (back to startat-expression) or (back to offset-
expression) or (back to limit-expression)
   integer
   | variable
   | ( simple-expression )
   | ( simple-expression { + | - | * } simple-expression )
```

Parameters

(back to top) on page 188

- **ALL or DISTINCT –** filters query results. If neither is specified, all rows that satisfy the clauses of the **SELECT** statement are retrieved. If DISTINCT is specified, duplicate output rows are eliminated. This is called the projection of the result of the statement. In many cases, statements take significantly longer to execute when DISTINCT is specified, so reserve the use of DISTINCT for cases where it is necessary.

  If DISTINCT is used, the statement cannot contain an aggregate function with a DISTINCT parameter.
- **row-limitation-option1 –** specifies the number of rows returned from a query. FIRST returns the first row selected from the query. TOP returns the specified number of rows from the query where *number-of-rows* is in the range 1 – 2147483647 and can be an integer constant or integer variable.

  Note: You cannot use TOP and LIMIT in the same query.

  FIRST and TOP are used primarily with the ORDER BY clause. If you use these keywords without an ORDER BY clause, the result might vary from run to run of the same query, as the optimizer might choose a different query plan.

  FIRST and TOP are permitted only in the top-level SELECT of a query, so they cannot be used in derived tables or view definitions. Using FIRST or TOP in a view definition might result in the keyword being ignored when a query is run on the view.

  Using FIRST is the same as setting the **ROW_COUNT** database option to 1. Using TOP is the same as setting the **ROW_COUNT** option to the same number of rows. If both TOP and **ROW_COUNT** are set, then the value of TOP takes precedence.

  The **ROW_COUNT** option could produce inconsistent results when used in a query involving global variables, system functions or proxy tables. See *ROW_COUNT Option* for details.

- ***select-list* –** is a comma delimited list of expressions that specify what is retrieved from the database. If an asterisk (*) is specified, all columns of all tables in the FROM clause (`table-name` all columns of the named table) are selected. Aggregate functions and analytical functions are allowed in the *select-list*.

  Note: In SAP Sybase IQ, scalar subqueries (nested selects) are allowed in the select list of the top level SELECT, as in SQL Anywhere and Adaptive Server. Subqueries cannot be used inside a conditional value expression (for example, in a **CASE** statement).

  Subqueries can also be used in a WHERE or HAVING clause predicate (one of the supported predicate types). However, inside the WHERE or HAVING clause, subqueries cannot be used inside a value expression or inside a CONTAINS or LIKE predicate. Subqueries are not allowed in the ON clause of outer joins or in the GROUP BY clause.

- ***alias-names* –** can be used throughout the query to represent the aliased expression. Alias names are also displayed by Interactive SQL at the top of each column of output from the **SELECT** statement. If the optional *alias-name* is not specified after an expression, Interactive SQL displays the expression. If you use the same name or expression for a column alias as the column name, the name is processed as an aliased column, not a table column name.
- **INTO *host-variable-list* –** specifies where the results of the **SELECT** statement goes. There must be one *host-variable* item for each item in the *select-list*. Select list items are put into the host variables in order. An indicator host variable is also allowed with each *host-variable* so the program can tell if the select list item was NULL. Used in Embedded SQL only.
- **INTO *variable-list* –** specifies where the results of the **SELECT** statement go. There must be one variable for each item in the select list. Select list items are put into the variables in order. Used in procedures only
- **INTO *table-name* –** creates a table and fills the table with data.

  If the table name starts with #, the table is created as a temporary table. Otherwise, the table is created as a permanent base table. For permanent tables to be created, the query must satisfy these conditions:

  - The *select-list* contains more than one item, and the INTO target is a single *table-name* identifier, or
  - The select-list contains a * and the INTO target is specified as *owner.table*.

  To create a permanent table with one column, the table name must be specified as *owner.table*. Omit the owner specification for a temporary table.

  This statement causes a **COMMIT** before execution as a side effect of creating the table. Requires the CREATE TABLE system privilege to execute this statement. No permissions are granted on the new table: the statement is a short form for **CREATE TABLE** followed by **INSERT... SELECT**.

A **SELECT INTO** from a stored procedure or function is not permitted, as **SELECT INTO** is an atomic statement and you cannot do **COMMIT**, **ROLLBACK**, or some **ROLLBACK TO SAVEPOINT** statements in an atomic statement.

Tables created using this statement do not have a primary key defined. You can add a primary key using **ALTER TABLE**. A primary key should be added before applying any updates or deletes to the table; otherwise, these operations result in all column values being logged in the transaction log for the affected rows.

Use of this clause is restricted to valid SQL Anywhere queries. SAP Sybase IQ extensions are not supported.

- **INTO LOCAL TEMPORARY TABLE –** creates a local, temporary table and populates it with the results of the query. When you use this clause, you do not need to start the temporary table name with `#`.
- **FROM *table-list* –** retrieves rows and views specified in the *table-list*. Joins can be specified using join operators. For more information, see *FROM Clause*. A **SELECT** statement with no FROM clause can be used to display the values of expressions not derived from tables. For example:

```
SELECT @@version
```

displays the value of the global variable @@version. This is equivalent to:

```
SELECT @@version
FROM DUMMY
```

Note: If you omit the FROM clause, or if all tables in the query are in the `SYSTEM` dbspace, the query is processed by SQL Anywhere instead of SAP Sybase IQ and might behave differently, especially with respect to syntactic and semantic restrictions and the effects of option settings.

If you have a query that does not require a FROM clause, you can force the query to be processed by SAP Sybase IQ by adding the clause "`FROM iq_dummy`," where `iq_dummy` is a one-row, one-column table that you create in your database.

- **WHERE *search-condition* –** specifies which rows are selected from the tables named in the FROM clause. It is also used to do joins between multiple tables. This is accomplished by putting a condition in the WHERE clause that relates a column or group of columns from one table with a column or group of columns from another table. Both tables must be listed in the FROM clause.

  The use of the same **CASE** statement is not allowed in both the SELECT and the WHERE clause of a grouped query.

  SAP Sybase IQ also supports the disjunction of subquery predicates. Each subquery can appear within the WHERE or HAVING clause with other predicates and can be combined using the AND or OR operators.
- **GROUP BY –** groups columns, alias names, or functions. GROUP BY expressions must also appear in the select list. The result of the query contains one row for each distinct set of

values in the named columns, aliases, or functions. The resulting rows are often referred to as groups since there is one row in the result for each group of rows from the table list. In the case of GROUP BY, all NULL values are treated as identical. Aggregate functions can then be applied to these groups to get meaningful results.

GROUP BY must contain more than a single constant. You do not need to add constants to the GROUP BY clause to select the constants in grouped queries. If the GROUP BY expression contains only a single constant, an error is returned and the query is rejected.

When GROUP BY is used, the select list, HAVING clause, and ORDER BY clause cannot reference any identifiers except those named in the GROUP BY clause. This exception applies: The *select-list* and HAVING clause may contain aggregate functions.

- **ROLLUP operator –** subtotals GROUP BY expressions that roll up from a detailed level to a grand total.

  The ROLLUP operator requires an ordered list of grouping expressions to be supplied as arguments. ROLLUP first calculates the standard aggregate values specified in the GROUP BY. Then ROLLUP moves from right to left through the list of grouping columns and creates progressively higher-level subtotals. A grand total is created at the end. If *n* is the number of grouping columns, ROLLUP creates *n+1* levels of subtotals.

  Restrictions on the ROLLUP operator:

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

  GROUPING is used with the ROLLUP operator to distinguish between stored NULL values and NULL values in query results created by ROLLUP.

  ROLLUP syntax:

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

  GROUPING takes a column name as a parameter and returns a Boolean value:

Table 5. Values Returned by GROUPING with the ROLLUP Operator

If the Value of the Result Is	GROUPING Returns
NULL created by a ROLLUP operation	1 (TRUE)
NULL indicating the row is a subtotal	1 (TRUE)

If the Value of the Result Is	GROUPING Returns
not created by a ROLLUP operation	0 (FALSE)
a stored NULL	0 (FALSE)

- **CUBE operator –** analyzes data by forming the data into groups in more than one dimension. CUBE requires an ordered list of grouping expressions (dimensions) as arguments and enables the **SELECT** statement to calculate subtotals for all possible combinations of the group of dimensions. The CUBE operator is part of the GROUP BY clause.

  Restrictions on the CUBE operator:

  - CUBE supports all of the aggregate functions available to the GROUP BY clause, but CUBE does not currently support COUNT DISTINCT or SUM DISTINCT.
  - CUBE does not currently support the inverse distribution analytical functions **PERCENTILE_CONT** and **PERCENTILE_DISC**.
  - CUBE can be used only in the **SELECT** statement; you cannot use CUBE in a SELECT subquery.
  - A multiple GROUPING specification that combines ROLLUP, CUBE, and GROUP BY columns in the same GROUP BY clause is not currently supported.
  - Constant expressions as GROUP BY keys are not supported.

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

  CUBE syntax:

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

  GROUPING takes a column name as a parameter and returns a Boolean value:

Table 6. Values Returned by GROUPING with the CUBE Operator

If the Value of the Result Is	GROUPING Returns
NULL created by a CUBE operation	1 (TRUE)
NULL indicating the row is a subtotal	1 (TRUE)
not created by a CUBE operation	0 (FALSE)
a stored NULL	0 (FALSE)

  When generating a query plan, the SAP Sybase IQ optimizer estimates the total number of groups generated by the GROUP BY CUBE hash operation. The **MAX_CUBE_RESULTS** database option sets an upper boundary for the number of estimated rows the optimizer considers for a hash algorithm that can be run. If the actual number of rows exceeds the **MAX_CUBE_RESULT** option value, the optimizer stops processing the query and returns

the error message "`Estimate number: nnn exceed the DEFAULT_MAX_CUBE_RESULT of GROUP BY CUBE or ROLLUP`", where *nnn* is the number estimated by the optimizer. See *MAX_CUBE_RESULT Option* for information on setting the **MAX_CUBE_RESULT** option.

- **HAVING *search-condition* –** based on the group values and not on the individual row values. The HAVING clause can be used only if either the statement has a GROUP BY clause or if the select list consists solely of aggregate functions. Any column names referenced in the HAVING clause must either be in the GROUP BY clause or be used as a parameter to an aggregate function in the HAVING clause.
- **ORDER BY –** orders the results of a query. Each item in the ORDER BY list can be labeled as ASC for ascending order or DESC for descending order. Ascending is assumed if neither is specified. If the expression is an integer **n**, then the query results are sorted by the **n**th item in the select list.

  In Embedded SQL, the **SELECT** statement is used for retrieving results from the database and placing the values into host variables with the INTO clause. The **SELECT** statement must return only one row. For multiple row queries, you must use cursors.

  You cannot include a Java class in the SELECT list, but you can, for example, create a function or variable that acts as a wrapper for the Java class and then select it.

  FOR JSON clause specifies that the result set is to be returned in JSON format. The JSON format depends on the mode you specify. This clause cannot be used with the FOR UPDATE or FOR READ ONLY clause. Cursors declared with FOR JSON are implicitly READ ONLY.

  When you specify RAW mode, each row in the result set is returned as a flattened JSON representation.

  AUTO mode returns the query results as nested JSON objects based on query joins.

  EXPLICIT mode allows you to control the form of the generated JSON objects. Using EXPLICIT mode offers more flexibility in specifying columns and nested hierarchical objects to produce uniform or heterogeneous arrays.
- ***row-limitation-option2* –** returns a subset of rows that satisfy the WHERE clause. Only one row-limitation clause can be specified at a time. When specifying this clause, an ORDER BY clause is required to order the rows in a meaningful manner. The row limitation clause is valid only in the top query block of a statement.

  The LIMIT argument must be an integer or integer variable The OFFSET argument must evaluate to a value greater than or equal to 0. If *offset-expression* is not specified, the default is 0.

  The row limitation clause LIMIT *offset-expression*, *limit-expression* is equivalent to LIMIT *limit-expression* OFFSET *offset-expression*.

  The LIMIT keyword is disabled by default. Use the **RESERVED_KEYWORDS** option to enable the LIMIT keyword.

Note: You cannot specify TOP and LIMIT in the same query.

Examples

(back to top) on page 188

- **Example 1 –** list all tables and views in the system catalog:

```
SELECT tname
FROM SYS.SYSCATALOG
WHERE tname LIKE 'SYS%' ;
```

- **Example 2 –** list all customers and the total value of their orders:

```
SELECT CompanyName,
  CAST( sum(SalesOrderItems.Quantity *
  Products.UnitPrice) AS INTEGER) VALUE
FROM Customers
  LEFT OUTER JOIN SalesOrders
  LEFT OUTER JOIN SalesOrderItems
  LEFT OUTER JOIN Products
GROUP BY CompanyName
ORDER BY VALUE DESC
```

- **Example 3 –** list the number of employees:

```
SELECT count(*)
FROM Employees;
```

- **Example 4 –** an Embedded SQL SELECT statement:

```
SELECT count(*) INTO :size FROM Employees;
```

- **Example 5 –** list the total sales by year, model, and color:

```
SELECT year, model, color, sum(sales)
FROM sales_tab
GROUP BY ROLLUP (year, model, color);
```

- **Example 6 –** select all items with a certain discount into a temporary table:

```
SELECT * INTO #TableTemp FROM lineitem
WHERE l_discount < 0.5
```

- **Example 7 –** return information about the employee that appears first when employees are sorted by last name:

```
SELECT FIRST *
FROM Employees
ORDER BY Surname;
```

- **Example 8 –** return the first five employees when their names are sorted by last name:

```
SELECT TOP 5 *
FROM Employees
ORDER BY Surname;
```

```
SELECT *
FROM Employees
```

```
ORDER BY Surname
LIMIT 5;
```

- **Example 9 –** list the fifth and sixth employees sorted in descending order by last name:

```
SELECT *
FROM Employees
ORDER BY Surname DESC
LIMIT 4,2;
```

Usage

(back to top) on page 188

You can use a **SELECT** statement in Interactive SQL to browse data in the database or to export data from the database to an external file.

You can also use a **SELECT** statement in procedures or in Embedded SQL. The **SELECT** statement with an INTO clause is used for retrieving results from the database when the **SELECT** statement returns only one row. (Tables created with SELECT INTO do not inherit IDENTITY/AUTOINCREMENT tables.) For multiple-row queries, you must use cursors. When you select more than one column and do not use *#table*, SELECT INTO creates a permanent base table. SELECT INTO *#table* always creates a temporary table regardless of the number of columns. SELECT INTO table with a single column selects into a host variable.

Note: When writing scripts and stored procedures that SELECT INTO a temporary table, wrap any select list item that is not a base column in a CAST expression. This guarantees that the column data type of the temporary table is the required data type.

Tables with the same name but different owners require aliases. A query without aliases returns incorrect results:

```
SELECT * FROM user1.t1
WHERE NOT EXISTS
(SELECT *
FROM user2.t1
WHERE user2.t1.col1 = user1.t.col1);
```

For correct results, use an alias for each table:

```
SELECT * FROM user1.t1 U1
WHERE NOT EXISTS
(SELECT *
FROM user2.t1 U2
WHERE U2.col1 = U1.col1);
```

The INTO clause with a *variable-list* is used only in procedures.

In **SELECT** statements, a stored procedure call can appear anywhere a base table or view is allowed. Note that CIS functional compensation performance considerations apply. For example, a **SELECT** statement can also return a result set from a procedure.

Standards

(back to top) on page 188

- SQL—ISO/ANSI SQL compliant.
- SAP Sybase Database product—Supported by SAP Sybase IQ, with some differences in syntax.

Permissions

(back to top) on page 188

Requires SELECT privilege on the named tables and views.

API Reference for a_v4_extfn

Reference information for `a_v4_extfn` functions, methods, and attributes.

Blob (a_v4_extfn_blob)

Use the `a_v4_extfn_blob` structure to represent a free-standing blob object.

Implementation

```
typedef struct a_v4_extfn_blob {
    a_sql_uint64 (SQL_CALLBACK *blob_length)(a_v4_extfn_blob *blob);
    void (SQL_CALLBACK *open_istream)(a_v4_extfn_blob *blob,
a_v4_extfn_blob_istream **is);
    void (SQL_CALLBACK *close_istream)(a_v4_extfn_blob *blob,
a_v4_extfn_blob_istream *is);
    void (SQL_CALLBACK *release)(a_v4_extfn_blob *blob);
} a_v4_extfn_blob;
```

Method Summary

Method Name	Data Type	Description
blob_length	a_sql_uint64	Returns the length, in bytes, of the specified blob.
open_istream	void	Opens an input stream that can be used to begin reading from the specified blob.
close_istream	void	Closes the input stream for the specified blob.

Method Name	Data Type	Description
release	void	Indicates that the caller is done with this blob and that the blob owner is free to release resources. After **release()**, referencing the blob results in an error. The owner usually deletes the memory when **release()** is called.

Description
The object `a_v4_extfn_blob` is used when:

- a table UDF needs to read LOB or CLOB data from a scalar input value
- a TPF needs to read LOB or CLOB data from a column in an input table

Restrictions and Limitations
None.

blob_length

Use the `blob_length` v4 API method to return the length, in bytes, of the specified blob.

Declaration

```
a_sql_uint64 blob_length(
    a_v4_extfn_blob *
)
```

Usage
Returns the length, in bytes, of the specified blob.

Parameters

Parameter	Description
blob	The blob to return the length of.

Returns
The length of the specified blob.

See also

- *open_istream* on page 201
- *close_istream* on page 201

- *release* on page 202

open_istream

Use the `open_istream` v4 API method to open an input stream to read from a blob.

Declaration

```
void open_istream(
    a_v4_extfn_blob *blob,
    a_v4_extfn_blob_istream **is
    )
```

Usage

Opens an input stream that can be used to begin reading from the specified blob.

Parameters

Parameter	Description
blob	The blob to open the input stream on.
is	An output parameter identifying the returned open input stream.

Returns

Nothing.

See also

- *blob_length* on page 200
- *close_istream* on page 201
- *release* on page 202

close_istream

Use the `close_istream` v4 API method to close the input stream for the specified blob.

Declaration

```
void close_istream(
    a_v4_extfn_blob *blob,
    a_v4_extfn_blob_istream *is
    )
```

Usage

Closes the input stream previously opened with the `open_istream` API.

Parameters

Parameter	Description
blob	The blob to close the input stream on.
is	A parameter identifying the input stream to close.

Returns
Nothing.

See also

- *blob_length* on page 200
- *open_istream* on page 201
- *release* on page 202

release

Use the `release` v4 API method to indicate that the caller is done with the currently selected blob. Releasing enables the owner to free memory.

Declaration

```
void release(
a_v4_extfn_blob *blob
)
```

Usage
Indicates that the caller is done with this blob and that the blob owner is free to release resources. After **release()**, referencing the blob results in an error. The owner usually deletes the memory when **release()** is called.

Parameters

Parameter	Description
blob	The blob to release.

Returns
Nothing.

See also

- *blob_length* on page 200
- *open_istream* on page 201
- *close_istream* on page 201

Blob Input Stream (a_v4_extfn_blob_istream)

Use the `a_v4_extfn_blob_istream` structure to read blob data for a LOB or CLOB scalar input column, or LOB or CLOB column in an input table.

Implementation

```
typedef struct a_v4_extfn_blob_istream {
    size_t (SQL_CALLBACK *get)( a_v4_extfn_blob_istream *is, void
*buf, size_t len );
    a_v4_extfn_blob     *blob;
    a_sql_byte          *beg;
    a_sql_byte          *ptr;
    a_sql_byte          *lim;
} a_v4_extfn_blob_istream;
```

Method Summary

Method Name	Data Type	Description
get	size_t	Gets a specified amount of data from a blob input stream.

Data Members and Data Types Summary

Data Member	Data Type	Description
Blob	a_v4_extfn_blob	The underlying blob structure for which this input stream was created.
Beg	a_sql_byte	A pointer to the beginning of the current chunk of data.
Ptr	a_sql_byte	A pointer to the current byte in the chunk of data.
Lim	a_sql_byte	A pointer to the end of the current chunk of data.

get

Use the `get` v4 API method to get a specified amount of data from a blob input stream.

Declaration

```
size_t get(
    a_v4_extfn_blob_istream *is,
```

```
    void *buf,
    size_t len
)
```

Usage
Gets a specified amount of data from a blob input stream.

Parameters

Parameter	Description
is	The input stream to retrieve data from.
buf	The buffer to store the data in.
len	The amount of data to retrieve.

Returns
The amount of data received.

Column Data (a_v4_extfn_column_data)

The structure `a_v4_extfn_column_data` represents a single column's worth of data. This is used by the producer when generating result set data, or by the consumer when reading input table column data.

Implementation

```
typedef struct a_v4_extfn_column_data {
    a_sql_byte          *is_null;
    a_sql_byte          null_mask;
    a_sql_byte          null_value;

    void                *data;
    a_sql_uint32        *piece_len;
    size_t              max_piece_len;

    void                *blob_handle;
} a_v4_extfn_column_data;
```

Data Members and Data Types Summary

Data Member	Data Type	Description
is_null	a_sql_byte *	Points to a byte where the NULL information for the value is stored.
null_mask	a_sql_byte	One or more bits used to represent the NULL value

Data Member	Data Type	Description
null_value	a_sql_byte	The value representing NULL
data	void *	Pointer to the data for the column. Depending on the type of fetch mechanism, either points to an address in the consumer, or an address where the data is stored in the UDF.
piece_len	a_sql_uint32 *	The actual length of data for variable-length data types
max_piece_len	size_t	The maximum data length allowed for this column.
blob_handle	void *	A non-NULL value means that the data for this column must be read using the `blob` API

Description

The `a_v4_extfn_column_data` structure represents the data values and related attributes for a specific data column. This structure is used by the producer when generating result set data. Data producers are also expected to create storage for *data*, *piece_len*, and the *is_null* flag.

The *is_null*, *null_mask*, and *null_value* data members indicate null in a column, and handle situations in which the null-bits are encoded into one byte for eight columns, or other cases in which a full byte is used for each column.

This example shows how to interpret the three fields used to represent NULL: *is_null*, *null_mask*, and *null_value*.

```
is_value_null()
   return( (*is_null & null_mask) == null_value )

set_value_null()
   *is_null = ( *is_null & ~null_mask) | null_value

set_value_not_null()
   *is_null = *is_null & ~null_mask | (~null_value & null_mask)
```

See also

- *get_blob* on page 318

Column List (a_v4_extfn_column_list)

Use the `a_v4_extfn_column_list` structure to provide a list of columns when describing **PARTITION BY** or to provide a list of columns when describing **TABLE_UNUSED_COLUMNS**.

Implementation

```
typedef struct a_v4_extfn_column_list {
    a_sql_int32         number_of_columns;
    a_sql_uint32        column_indexes[1];      // there are
number_of_columns entries
} a_v4_extfn_column_list;
```

Data Members and Data Types Summary

Data Member	Data Type	Description
number_of_columns	a_sql_uint32	The number of columns in the list.
column_indexes	a_sql_uint32 *	A contiguous array of size number_of_columns with the column indexes (1-based).

Description

The meaning of the contents of the column list changes, depending on whether the list is used with **TABLE_PARTITIONBY** or **TABLE_UNUSED_COLUMNS**.

See also

- *V4 API describe_parameter and EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY* on page 141
- *Parallel TPF PARTITION BY Examples Using EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY* on page 143
- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS Attribute (Get)* on page 260
- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS Attribute (Set)* on page 276

Column Order (a_v4_extfn_order_el)

Use the `a_v4_extfn_order_el` structure to describe the element order in a column.

Implementation

```
typedef struct a_v4_extfn_order_el {
    a_sql_uint32        column_index;    // Index of the column in the
```

```
table (1-based)
    a_sql_byte       ascending;              // Nonzero if the column
is ordered "ascending".
} a_v4_extfn_order_el;
```

Data Members and Data Types Summary

Data Member	Data Type	Description
column_index	a_sql_uint32	Index of the column in the table (1-based).
ascending	a_sql_byte	Nonzero, if the column order is "ascending."

Description

The `a_v4_extfn_order_el` structure describes a column and tells whether it should be in ascending or descending order. The `a_v4_extfn_orderby_list` structure holds an array of these structures. There is one `a_v4_extfn_order_el` structure for each column in the **ORDERBY** clause.

See also

- *Order By List (a_v4_extfn_orderby_list)* on page 307

Column Subset (a_v4_extfn_col_subset_of_input)

Use the `a_v4_extfn_col_subset_of_input` structure to declare that an output column has a value that is always taken from a particular input column to the UDF.

Implementation

```
typedef struct a_v4_extfn_col_subset_of_input {
    a_sql_uint32    source_table_parameter_arg_num;    // arg_num of
the source table parameter
    a_sql_uint32    source_column_number;        // source column of
the source table
} a_v4_extfn_col_subset_of_input;
```

Data Members and Data Types Summary

Data Member	Data Type	Description
source_table_parameter_arg_num	a_sql_uint32 *	*arg_num* of the source TABLE parameter
source_column_number	a_sql_uint32 *	Source column of the source table

Description

The query optimizer uses the subset of input to infer logical properties of the values in the output column. For example, the number of distinct values in the input column is an upper bound on the distinct values in the output column, and any local predicates on the input column also hold on the output column.

See also

- *Describe Column Type (a_v4_extfn_describe_col_type)* on page 281

Describe API

The **_describe_extfn** function is a member of `a_v4_extfn_proc`. A UDF gets and sets logical properties using the `describe_column`, `describe_parameter`, and `describe_udf` properties in the `a_v4_extfn_proc_context` object.

_describe_extfn Declaration

```
void (UDF_CALLBACK *_describe_extfn)(a_v4_extfn_proc_context
*cntxt );
)
```

Usage

The **_describe_extfn** function describes the procedure evaluation to the server.

Each of the `describe_column`, `describe_parameter`, and `describe_udf` properties has an associated get and set method, a set of attribute types, and an associated data type for each attribute. The get methods retrieve information from the server; the set methods describe the logical properties of the UDF (such as the number of output columns or the number of distinct values for a output column) to the server.

See also

- **describe_column_get* on page 209
- **describe_column_set* on page 225
- **describe_parameter_get* on page 242
- **describe_parameter_set* on page 261
- **describe_udf_get* on page 277
- **describe_udf_set* on page 279
- *External Function (a_v4_extfn_proc)* on page 288

*describe_column_get

The `describe_column_get` v4 API method is used by the table UDF to retrieve properties about an individual column of a TABLE parameter.

Declaration

```
a_sql_int32 (SQL_CALLBACK *describe_column_get)(
    a_v4_extfn_proc_context       *cntxt,
    a_sql_uint32                     arg_num,
    a_sql_uint32                        column_num,
    a_v4_extfn_describe_parm_type describe_type,
    void                                *describe_buffer,
    size_t                              describe_buffer_len );
```

Parameters

Parameter	Description
cntxt	The procedure context object for this UDF.
arg_num	The ordinal of the TABLE parameter (0 is the result table, 1 for first input argument).
column_num	The ordinal of the column starting at 1.
describe_type	A selector indicating what property to retrieve.
describe_buffer	A structure that holds the describe information for the specified property to get from the server. The specific structure or data type is indicated by the **describe_type** parameter.
describe_buffer_length	The length, in bytes, of the **describe_buffer**.

Returns

On success, returns the number of bytes written to the **describe_buffer**. If an error occurs, or no property is retrieved, this function returns one of the generic `describe_column` errors.

See also

- **describe_column_set* on page 225

Attributes for *describe_column_get

Code showing the attributes for `describe_column_get` v4 API method.

```
typedef enum a_v4_extfn_describe_col_type {
    EXTFNAPIV4_DESCRIBE_COL_NAME,
    EXTFNAPIV4_DESCRIBE_COL_TYPE,
    EXTFNAPIV4_DESCRIBE_COL_WIDTH,
    EXTFNAPIV4_DESCRIBE_COL_SCALE,
```

```
    EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL,
    EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES,
    EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE,
    EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT,
    EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE,
    EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER,
    EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE,
    EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE,
    EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT,
    } a_v4_extfn_describe_col_type;
```

EXTFNAPIV4_DESCRIBE_COL_NAME (Get)

The **EXTFNAPIV4_DESCRIBE_COL_NAME** attribute indicates the column name. Used in a `describe_column_get` scenario.

Data Type

`char[]`

Description

The column name. This property is valid only for table arguments.

Usage

If a UDF gets this property, then the name of the specified column is returned.

Returns

On success, returns the length of the column name.

On failure, returns one of the generic `describe_column` errors, or:

- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – get error returned if the query processing phase is not greater than Initial.
- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – get error returned if the buffer length has insufficient characters or is 0 length.
- `EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER` – get error returned if the parameter is not a TABLE parameter.

Query Processing Phases

Valid in:

- Annotation phase
- Query Optimization phase
- Plan Building phase
- Execution phase

See also

- *EXTFNAPIV4_DESCRIBE_COL_NAME (Set)* on page 227

- *EXTFNAPIV4_DESCRIBE_COL_TYPE (Set)* on page 228
- *EXTFNAPIV4_DESCRIBE_COL_TYPE (Get)* on page 211
- *Generic describe_column Errors* on page 325
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_COL_TYPE (Get)

The `EXTFNAPIV4_DESCRIBE_COL_TYPE` attribute indicates the data type of the column. Used in a `describe_column_get` scenario.

Data Type

`a_sql_data_type`

Description

The data type of the column. This property is valid only for table arguments.

Usage

If a UDF gets this property, then returns the data type of the specified column.

Returns

On success, the `sizeof(a_sql_data_type)` is returned.

On failure, returns one of the generic `describe_column` errors, or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – get error returned if the describe buffer is not the size of `a_sql_data_type`.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – get error returned if the query processing phase is not greater than Initial.

Query Processing Phases

- Annotation phase
- Query Optimization phase
- Plan Building phase
- Execution phase

See also

- *Generic describe_column Errors* on page 325
- *EXTFNAPIV4_DESCRIBE_COL_TYPE (Set)* on page 228
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_COL_WIDTH (Get)

The **EXTFNAPIV4_DESCRIBE_COL_WIDTH** attribute indicates the width of the column. Used in a `describe_column_get` scenario.

Data Type

`a_sql_uint32`

Description

The width of a column. Column width is the amount of storage, in bytes, required to store a value of the associated data type. This property is valid only for table arguments.

Usage

If a UDF gets this property, then returns the width of the column as defined in the **CREATE PROCEDURE** statement.

Returns

On success, returns the `sizeof(a_sql_uint32)`.

On failure, returns one of the generic `describe_column` errors, or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – get error returned if the describe buffer is not the size of `a_sql_uint32`.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – get error returned if the query processing phase is not greater than Initial.

Query Processing Phases

- Annotation phase
- Query Optimization phase
- Plan Building phase
- Execution phase

See also

- *EXTFNAPIV4_DESCRIBE_COL_ WIDTH (Set)* on page 229
- *Generic describe_column Errors* on page 325
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_COL_SCALE (Get)

The **EXTFNAPIV4_DESCRIBE_COL_SCALE** attribute indicates the scale of the column. Used in a `describe_column_get` scenario.

Data Type

`a_sql_uint32`

Description
The scale of a column. For arithmetic data types, parameter scale is the number of digits to the right of the decimal point in a number. This property is valid only for table arguments.

Usage
If the UDF gets this property, returns the scale of the column as defined in the **CREATE PROCEDURE** statement. This property is valid only for arithmetic data types.

Returns
On success, returns the `sizeof(a_sql_uint32)` if the value was returned, or:

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – get error returned if the scale is unavailable for the data type of the specified column.

On failure, returns one of the generic `describe_column` errors, or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – get error returned if the describe buffer is not the size of `a_sql_uint32`.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – get error returned if the query processing phase is not greater than Initial.

Query Processing Phases

- Annotation phase
- Query Optimization phase
- Plan Building phase
- Execution phase

See also
- *EXTFNAPIV4_DESCRIBE_COL_SCALE (Set)* on page 230
- *Generic describe_column Errors* on page 325
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL (Get)
The **EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL** attribute indicates if the column can be NULL. Used in a `describe_column_get` scenario.

Data Type
`a_sql_byte`

Description
True, if the column can be NULL. This property is valid only for table arguments. This property is valid only for argument 0.

Usage
If a UDF gets this property, returns 1 if the column can be NULL, and returns 0 if otherwise.

Returns
On success, returns the `sizeof(a_sql_byte)` if the attribute is available, or:

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – returned if the attribute was not available to get. This can happen if the column was not involved in the query.

On failure, returns one of the generic `describe_column` errors, or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – get error returned if the describe buffer is not the size of `a_sql_byte`.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – get error returned if the specified argument is an input table and the query processing phase is not greater than Plan Building phase.

Query Processing Phases
Valid in:

- Execution phase

See also
- *EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL (Set)* on page 231
- *Generic describe_column Errors* on page 325
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES (Get)
The **EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES** attribute describes the distinct values for a column. Used in a `describe_column_get` scenario.

Data Type
`a_v4_extfn_estimate`

Description
The estimated number of distinct values for a column. This property is valid only for table arguments.

Usage
If a UDF gets this property, it returns the estimated number of distinct values for a column.

Returns
On success, returns the `sizeof(a_v4_extfn_estimate)`, if it returns a value, or:

- EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE – returned if the attribute was not available to get. This can happen if the column was not involved in the query.

On failure, returns one of the generic describe_column errors, or:

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH – get error returned if the describe buffer is not the size of a_v4_extfn_estimate.
- EXTFNAPIV4_DESCRIBE_INVALID_STATE – get error returned if the specified argument is an input table and the query processing phase is greater than Optimization.

Query Processing Phases

Valid in:

- Plan Building phase
- Execution phase

Example

Consider this procedure definition and code fragment in the _describe_extfn API function:

```
CREATE PROCEDURE my_tpf( col_char char(10), col_table TABLE( c1 INT,
c2 INT ) )
RESULTS ( r1 INT, r2 INT, r3 INT )
EXTERNAL 'my_tpf_proc@mylibrary';
```

```
CREATE TABLE T( x INT, y INT, z INT );
```

```
select * from my_tpf( 'test', TABLE( select x,y from T ) )
```

This example shows how a TPF gets the number of distinct values for column one of the input table. A TPF may want to get this value, if it is beneficial for choosing an appropriate processing algorithm.

```
my_tpf_describe(a_v4_extfn_proc_context *cntxt)
{
    if( cntxt->current_state == EXTFNAPIV4_STATE_PLAN_BUILDING ) {
           a_v4_extfn_estimate num_distinct;

           a_sql_int32 ret = 0;

         // Get the number of distinct values expected from the first
column
        // of the table input parameter 'col_table'
           ret = cntxt->describe_column_get( cntxt, 2, 1
                      EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES,
                      &num_distinct,
                      sizeof(a_v4_extfn_estimate) );

           // default algorithm is 1
           _algorithm = 1;

           if( ret > 0 ) {
               // choose the best algorithm for sample size.
```

```
                if ( num_distinct.value < 100 ) {
                    // use faster algorithm for small distinct values.
                        _algorithm = 2;
                }
            }
            else {
                if ( ret < 0 ) {
                        // Handle the error
                        //   or continue with default algorithm
                } else {
                        // Attribute was unavailable
                        // We will use the default algorithm.
                }
            }
    }
}
```

See also

- *EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES (Set)* on page 232
- *Generic describe_column Errors* on page 325
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE (Get)

The **EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE** attribute indicates if a column is unique in the table. Used in a `describe_column_get` scenario.

Data Type

`a_sql_byte`

Description

True, if the column is unique within the table. This property is valid only for table arguments.

Usage

If the UDF gets this property, then returns 1 if the column is unique, and 0 otherwise.

Returns

On success, returns the `sizeof(a_sql_byte)` or:

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – if the attribute was unavailable to get. This can happen if the column was not involved in the query.

On failure, returns one of the generic `describe_column` errors, or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – get error returned if the describe buffer is not the size of `a_sql_byte`.

- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – get error returned if the query processing phase is not greater than Initial.

Query Processing Phases
Valid in:

- Annotation phase
- Query Optimization phase
- Plan Building phase
- Execution phase

See also

- *Generic describe_column Errors* on page 325
- *EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE (Set)* on page 233
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT (Get)
The **EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT** attribute indicates if a column is constant. Used in a `describe_column_get` scenario.

Data Type
`a_sql_byte`

Description
True, if the column is constant for the lifetime of the statement. This property is valid only for input table arguments.

Usage
If a UDF gets this property, the return value is 1 if the column is constant for the lifetime of the statement and 0 otherwise. Input table columns are constant, if the column in the select list for the input table is a constant expression or NULL.

Returns
On success, returns the `sizeof(a_sql_byte)`, if the value was returned, or:

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – the attribute is not available to get. Returned, if the column is not involved in the query.

On failure, returns one of the generic `describe_column` errors, or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – get error returned, if the describe buffer is not the size of `a_sql_byte`.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – get error returned, if the query processing phase is not greater than Initial.

- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – get error returned, if the specified argument is not an input table.

Query Processing Phases
Valid in:

- Annotation phase
- Query Optimization phase
- Plan Building phase
- Execution phase

See also

- *EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT (Set)* on page 234
- *Generic describe_column Errors* on page 325
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE (Get)
The **EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE** attribute indicates the constant value of a column. Used in a `describe_column_get` scenario.

Data Type
`an_extfn_value`

Description
The value of the column, if it is constant for the statement lifetime. If `EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT` for this column returns true, this value is available. This property is valid only for table arguments.

Usage
For columns of input tables that have a constant value, the value is returned. If the value is unavailable, then NULL is returned.

Returns
On success, returns the `sizeof(a_sql_byte)`, if the value was returned, or:

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – the attribute is not available to get. Returned, if the column is not involved in the query, or if the value is not considered constant.

On failure, returns one of the generic `describe_column` errors, or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – get error returned, if the describe buffer is not the size of `a_sql_byte`.

- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – get error returned, if the query processing phase is not greater than Initial.
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – get error returned, if the specified argument is not an input table.

Query Processing Phases
Valid in:

- Annotation phase
- Query Optimization phase
- Plan Building phase
- Execution phase

See also

- *Generic describe_column Errors* on page 325
- *EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE (Set)* on page 234
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER (Get)
The **EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER** attribute indicates if a column in the result table is used by the consumer. Used in a `describe_column_get` scenario.

Data Type
`a_sql_byte`

Description
Used either to determine whether a column in the result table is used by the consumer, or to indicate that a column in an input is not needed. Valid for table arguments. Allows the user to set or retrieve information about a single column, whereas the similar attribute `EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS` sets or retrieves information about all columns in a single call.

Usage
The UDF queries this property to determine if a result table column is required by the consumer. This can help the UDF avoid unnecessary work for unused columns.

Returns
On success, returns the `sizeof(a_sql_byte)` or:

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – if the attribute was unavailable to get. This can happen if the column was not involved in the query.

On failure, returns one of the generic `describe_column` errors, or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – get error returned if the describe buffer is not the size of `a_v4_extfn_estimate`.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – get error returned if the query processing phase is not greater than Initial.
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – get error returned if the argument specified is not argument 0.

Query Processing Phases
Valid during:

- Annotation phase
- Query Optimization phase
- Plan Building phase
- Execution phase

The PROCEDURE definition and code fragment in the `_describe_extfn` API function:

```
CREATE PROCEDURE my_tpf( col_char char(10), col_table TABLE( c1 INT,
c2
INT ) )
      RESULTS ( r1 INT, r2 INT, r3 INT )
      EXTERNAL 'my_tpf_proc@mylibrary';

CREATE TABLE T( x INT, y INT, z INT );

select r2,r3 from my_tpf( 'test', TABLE( select x,y from T ) )
```

When this TPF runs, it is beneficial to know if the user has selected column `r1` of the result set. If the user does not need `r1`, calculations for `r1` may be unnecessary and we do not need to produce it for the server.

```
my_tpf_describe(a_v4_extfn_proc_context *cntxt)
{
    if( cntxt->current_state > EXTFNAPIV4_STATE_INITIAL ) {
      a_sql_byte  col_is_used = 0;
      a_sql_int32 ret = 0;

      ret = cntxt->describe_column_get( cntxt, 0, 1,
            EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER,
            &col_is_used,
            sizeof(a_sql_byte) );

      if( ret < 0 ) {
          // Handle the error.
      }

    }
```

See also

- *EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER (Set)* on page 235

- *Generic describe_column Errors* on page 325
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE (Get)

The **EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE** attribute indicates the minimum value for a column. Used in a `describe_column_get` scenario.

Data Type

`an_extfn_value`

Description

The minimum value for a column, if available. Valid only for argument 0 and table arguments.

Usage

If a UDF gets the **EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE** property, the minimum value of the column data is returned in the **describe_buffer**. If the input table is a base table, the minimum value is based on all of the column data in the table and is accessible only if there is an index on the table column. If the input table is the result of another UDF, the minimum value is the `EXTFNAPIV4_DESCRIBE_COL_TYPE` set by that UDF.

The data type for this property is different for different columns. The UDF can use `EXTFNAPIV4_DESCRIBE_COL_TYPE` to determine the data type of the column. The UDF can also use `EXTFNAPIV4_DESCRIBE_COL_WIDTH` to determine the storage requirements of the column,to provide an equivalently sized buffer to hold the value.

describe_buffer_length allows the server to determine if the buffer is valid.

If the **EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE** property is unavailable, `describe_buffer` is NULL.

Returns

On success, returns the `describe_buffer_length`, or:

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – if the attribute was unavailable to get. Returned if the column was not involved in the query or the minimum value was unavailable for the requested column.

On failure, returns one of the generic `describe_column` errors, or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – Get error returned, if the describe buffer is not large enough to hold the minimum value.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – Get error returned if the state is not greater than Initial.

Query Processing States

Valid in any state except Initial state:

- Annotation state
- Query Optimization state
- Plan Building state
- Execution state

Example

The procedure definition and code fragment in the `_describe_extfn` API function:

```
CREATE PROCEDURE my_tpf( col_char char(10), col_table TABLE( c1 INT,
c2 INT ) )
    RESULTS ( r1 INT, r2 INT, r3 INT )
    EXTERNAL 'my_tpf_proc@mylibrary';

CREATE TABLE T( x INT, y INT, z INT );

select * from my_tpf( 'test', TABLE( select x,y from T ) )
```

This example illustrates how a TPF would get the minimum value for column two of the input table, for internal optimization purposes.

```
my_tpf_describe(a_v4_extfn_proc_context *cntxt)
{
      if( cntxt->current_state > EXTFNAPIV4_STATE_INITIAL ) {
           a_sql_int32 min_value = 0;
           a_sql_int32 ret = 0;

        // Get the minimum value of the second column of the
        // table input parameter 'col_table'

        ret = cntxt->describe_column_get( cntxt, 2, 2
                 EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE,
                 &min_value,
                 sizeof(a_sql_int32) );

     if( ret < 0 ) {
               // Handle the error.
        }

      }
}
```

See also

- *Query Processing States* on page 121
- *EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE (Set)* on page 237
- *EXTFNAPIV4_DESCRIBE_COL_TYPE (Get)* on page 211
- *EXTFNAPIV4_DESCRIBE_COL_TYPE (Set)* on page 228
- *EXTFNAPIV4_DESCRIBE_COL_WIDTH (Get)* on page 212
- *EXTFNAPIV4_DESCRIBE_COL_WIDTH (Set)* on page 229
- *Generic describe_column Errors* on page 325

EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE (Get)

The **EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE** attribute indicates the maximum value for the column. Used in a `describe_column_get` scenario.

Data Type

`an_extfn_value`

Description

The maximum value for a column. This property is valid only for argument 0 and table arguments.

Usage

If a UDF gets the `EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE` property, then the maximum value of the column data is returned in the **describe_buffer**. If the input table is a base table, the maximum value is based on all of the column data in the table and is accessible only if there is an index on the table column. If the input table is the result of another UDF, the maximum value is the `COL_MAXIMUM_VALUE` set by that UDF.

The data type for this property is different for different columns. The UDF can use `EXTFNAPIV4_DESCRIBE_COL_TYPE` to determine the data type of the column. The UDF can also use `EXTFNAPIV4_DESCRIBE_COL_WIDTH` to determine the storage requirements of the column, to provide an equivalently sized buffer to hold the value.

describe_buffer_length allows the server to determine if the buffer is valid.

If `EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE` is unavailable, `describe_buffer` is NULL.

Returns

On success, returns the `describe_buffer_length` or:

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – If the attribute was unavailable to get. This can happen if the column was uninvolved in the query, or if the maximum value was unavailable for the requested column.

On failure, returns one of the generic `describe_column` errors, or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – Get error returned if the describe buffer is not large enough to hold the maximum value.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – Get error returned if the query processing phase is not greater than Initial.

Query Processing Phases

Valid in any phase except Initial phase:

- Annotation phase
- Query Optimization phase
- Plan building phase
- Execution phase

Example

The **PROCEDURE** definition and code fragment in the `_describe_extfn` API function:

```
CREATE PROCEDURE my_tpf( col_char char(10), col_table TABLE( c1 INT,
c2 INT ) )
         RESULTS ( r1 INT, r2 INT, r3 INT )
         EXTERNAL 'my_tpf_proc@mylibrary';

CREATE TABLE T( x INT, y INT, z INT );

select * from my_tpf( 'test', TABLE( select x,y from T ) )
```

This example illustrates how a TPF would get the maximum value for column two of the input table, for internal optimization purposes.

```
my_tpf_describe(a_v4_extfn_proc_context *cntxt)
{
    if( cntxt->current_state > EXTFNAPIV4_STATE_INITIAL ) {
         a_sql_int32 max_value = 0;
         a_sql_int32 ret = 0;

         // Get the maximum value of the second column of the
         // table input parameter  'col_table'
         ret = cntxt->describe_column_get( cntxt, 2, 2
                  EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE,
                  &max_value,
                  sizeof(a_sql_int32) );

         if( ret < 0 ) {
             // Handle the error.
            }
       }
}
```

See also

- *Query Processing States* on page 121
- *EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE (Set)* on page 239
- *EXTFNAPIV4_DESCRIBE_COL_TYPE (Get)* on page 211
- *EXTFNAPIV4_DESCRIBE_COL_TYPE (Set)* on page 228
- *EXTFNAPIV4_DESCRIBE_COL_WIDTH (Get)* on page 212
- *EXTFNAPIV4_DESCRIBE_COL_WIDTH (Set)* on page 229
- *Generic describe_column Errors* on page 325

EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT (Get)

The **EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT** attribute sets a subset of the values specified in an input column. Using this attribute in a `describe_column_get` scenario returns an error.

Data Type

`a_v4_extfn_col_subset_of_input`

Description

Column values are a subset of the values specified in an input column.

Usage

This attribute can be set only.

Returns

Returns the error `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE`.

Query Processing States

Error `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE` is returned in any state.

See also

- *EXTFNAPIV4_DESCRIBE_COL_ VALUES_SUBSET_OF_INPUT (Set)* on page 240
- *Generic describe_column Errors* on page 325
- *Query Processing States* on page 121

*describe_column_set

The `describe_column_set` v4 API method sets UDF column-level properties on the server.

Description

Column-level properties describe various characteristics about columns in the result set or input tables in a TPF. For example, a UDF can tell the server that a column in its result set will have only ten distinct values.

Declaration

```
a_sql_int32 (SQL_CALLBACK *describe_column_set)(
    a_v4_extfn_proc_context        *cntxt,
    a_sql_uint32        arg_num,
    a_sql_uint32        column_num,
    a_v4_extfn_describe_udf_type        describe_type,
    const void        *describe_buffer,
    size_t        describe_buffer_len );
```

Parameters

Parameter	Description
cntxt	The procedure context object for this UDF.
arg_num	The ordinal of the TABLE parameter (0 is the result table, 1 for first input argument).
column_num	The ordinal of the column starting at 1.
describe_type	A selector indicating what property to set.
describe_buffer	A structure that holds the describe information for the specified property to set on the server. The specific structure or data type is indicated by the **describe_type** parameter.
describe_buffer_length	The length, in bytes, of the **describe_buffer**.

Returns

On success, returns the number of bytes written to the **describe_buffer**. If an error occurs, or no property is retrieved, this function returns one of the generic `describe_column` errors.

See also

- **describe_column_get* on page 209

Attributes for *describe_column_set

Code showing the attributes for `describe_column_set`.

```
typedef enum a_v4_extfn_describe_col_type {
    EXTFNAPIV4_DESCRIBE_COL_NAME,
    EXTFNAPIV4_DESCRIBE_COL_TYPE,
    EXTFNAPIV4_DESCRIBE_COL_WIDTH,
    EXTFNAPIV4_DESCRIBE_COL_SCALE,
    EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL,
    EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES,
    EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE,
    EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT,
    EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE,
    EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER,
    EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE,
    EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE,
    EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT,
    } a_v4_extfn_describe_col_type;
```

EXTFNAPIV4_DESCRIBE_COL_NAME (Set)

The **EXTFNAPIV4_DESCRIBE_COL_NAME** attribute indicates a column name. Used in a `describe_column_set` scenario.

Data Type

`char[]`

Description

The column name. This property is valid only for table arguments.

Usage

For argument 0, if the UDF sets this property, the server compares the value with the name of the column supplied in the **CREATE PROCEDURE** statement. The comparison ensures that the **CREATE PROCEDURE** statement has the same column name as expected by the UDF.

Returns

On success, returns the length of the column name.

On failure, returns one of the generic `describe_column` errors, or:

- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – set error returned if the state is not Annotation.
- `EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER` – set error returned if the parameter is not a TABLE parameter.
- `EXTFNAPIV4_DESCRIBE_ INVALID_ATTRIBUTE_VALUE` – set error returned if the length of input column name exceeds 128 characters or if the input column name and column name stored in the catalog do not match.

Query Processing States

- Annotation state

Example

```
short desc_rc = 0;
     char name[7] = 'column1';
    // Verify that the procedure was created with the second column
of the result table as an int
     if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {
          desc_rc = ctx->describe_column_set( ctx, 0, 2,
EXTFNAPIV4_DESCRIBE_COL_NAME,
                                                                  name,
                                                        sizeof(name) );
          if( desc_rc < 0 ) {
          // handle the error.
          }
     }
```

See also

- *EXTFNAPIV4_DESCRIBE_COL_NAME (Get)* on page 210
- *EXTFNAPIV4_DESCRIBE_COL_TYPE (Set)* on page 228
- *EXTFNAPIV4_DESCRIBE_COL_TYPE (Get)* on page 211
- *Generic describe_column Errors* on page 325
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_COL_TYPE (Set)

The `EXTFNAPIV4_DESCRIBE_COL_TYPE` attribute indicates the data type of the column. Used in a `describe_column_set` scenario.

Data Type

`a_sql_data_type`

Description

The data type of the column. This property is valid only for table arguments.

Usage

For argument zero, if the UDF sets this property, then the server compares the value with the data type of the column supplied in the **CREATE PROCEDURE** statement. This allows the UDF to ensure the **CREATE PROCEDURE** statement has the same data type as expected by the UDF.

Returns

On success, returns the `a_sql_data_type`.

On failure, returns one of the generic `describe_column` errors, or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – Set error returned if the describe buffer is not the size of `a_sql_data_type`.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – Set error returned if the state is not Annotation.
- `EXTFNAPIV4_DESCRIBE_ INVALID_ATTRIBUTE_VALUE` – Set error returned if the input data type and the data type stored in the catalog do not match,.

Query processing states

- Annotation state

Example

```
short desc_rc = 0;
a_sql_data_type type = DT_INT;

   // Verify that the procedure was created with the second column of
the result table as an int
    if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {
```

```
        desc_rc = ctx->describe_column_set( ctx, 0, 2,
EXTFNAPIV4_DESCRIBE_COL_TYPE,
        &type,
                sizeof(a_sql_data_type) );
        if( desc_rc < 0 ) {
            // handle the error.
        }
    }
```

See also

- *Generic describe_column Errors* on page 325
- *EXTFNAPIV4_DESCRIBE_COL_TYPE (Get)* on page 211
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_COL_WIDTH (Set)

The **EXTFNAPIV4_DESCRIBE_COL_WIDTH** attribute indicates the width of the column. Used in a `describe_column_set` scenario.

Data Type

`a_sql_uint32`

Description

The width of a column. Column width is the amount of storage, in bytes, required to store a value of the associated data type. This property is valid only for table arguments.

Usage

If the UDF sets this property, the server compares the value with the width of the column supplied in the **CREATE PROCEDURE** statement. This allows the UDF to ensure the **CREATE PROCEDURE** statement has the same column width as expected by the UDF.

Returns

On success, returns the `sizeof(a_sql_uint32)`.

On failure, returns one of the generic `describe_column` errors, or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – set error returned if the describe buffer is not the size of `a_sql_uint32`.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – set error returned if the query processing state is not Annotation.
- `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` – set error returned if the input width and width stored in the catalog do not match.

Query Processing States

Valid in:

- Annotation state

See also

- *EXTFNAPIV4_DESCRIBE_COL_ WIDTH (Get)* on page 212
- *Generic describe_column Errors* on page 325
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_COL_SCALE (Set)

The **EXTFNAPIV4_DESCRIBE_COL_SCALE** attribute indicates the scale of the column. Used in a `describe_column_set` scenario.

Data Type

`a_sql_uint32`

Description

The scale of a column. For arithmetic data types, parameter scale is the number of digits to the right of the decimal point in a number. This property is valid only for table arguments.

Usage

If the UDF sets this property, the server compares the value with the scale of the column supplied in the **CREATE PROCEDURE** statement. This allows the UDF to ensure the **CREATE PROCEDURE** statement has the same column width as expected by the UDF. This property is valid only for arithmetic data types.

Returns

On success, returns the `sizeof(a_sql_uint32)`, or:

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – set error returned if the scale is not available for the data type of the specified column.

On failure, returns one of the generic `describe_column` errors, or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – set error returned if the describe buffer is not the size of `a_sql_uint32`.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – set error returned if the query processing state is not Annotation.
- `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` – set error returned if the input scale and scale stored in the catalog do not match.

Query Processing States

Valid in:

- Annotation state

Example

```
short desc_rc = 0;
a_sql_uint32 scale = 0;
```

```
            // Verify that the procedure has a scale of zero for the
second result table column.
        if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {
        desc_rc = ctx->describe_column_set( ctx, 0, 2,
EXTFNAPIV4_DESCRIBE_COL_SCALE,
            &scale,
            sizeof(a_sql_data_type) );
        if( desc_rc < 0 ) {
            // handle the error.
        }
    }
```

See also

- *EXTFNAPIV4_DESCRIBE_COL_SCALE (Get)* on page 212
- *Generic describe_column Errors* on page 325
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL (Set)

The **EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL** attribute indicates if the column can be null. Used in a `describe_column_set` scenario.

Data Type

`a_sql_byte`

Description

True, if the column can be NULL. This property is valid only for table arguments. This property is valid only for argument 0.

Usage

The UDF can set this property for a result table column if that column can be NULL. If the UDF does not explicitly set this property, it is assumed that the column can be NULL. The server can use this information during the Optimization state.

Returns

On success, returns the `sizeof(a_sql_byte)` if the attribute was set or:

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – returned if the attribute was unavailable to set, which may happen if the column was uninvolved in the query.

On failure, returns one of the generic `describe_column` errors, or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – set error returned if the describe buffer is not the size of `a_sql_byte`.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – set error returned if the state is not equal to OPTIMIZATION.

- `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` – set error returned if the UDF attempts to set this attribute to a value other than 0 or 1.

Query Processing States
Valid in:

- Optimization state

See also

- *EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL (Get)* on page 213
- *Generic describe_column Errors* on page 325
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES (Set)
The **EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES** attribute describes the distinct values for a column. Used in a `describe_column_set` scenario.

Data Type
`a_v4_extfn_estimate`

Description
The estimated number of distinct values for a column. This property is valid only for table arguments.

Usage
The UDF can set this property if it knows how many distinct values a column can have in its result table. The server uses this information during the Optimization state.

Returns
On success, returns the `sizeof(a_v4_extfn_estimate)`, if it sets the value, or:

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – returned if the attribute was unavailable to set. This can happen if the column was not involved in the query.

On failure, returns:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – set error returned if the describe buffer is not the size of `a_v4_extfn_estimate`.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – set error returned if the state is not equal to Optimization.

Query Processing States
Valid in:

- Optimization state

See also

- *EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES (Get)* on page 214
- *Generic describe_column Errors* on page 325
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE (Set)

The **EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE** attribute indicates if the column is unique in the table. Used in a `describe_column_set` scenario.

Data Type

`a_sql_byte`

Description

True, if the column is unique within the table. This property is valid only for table arguments.

Usage

The UDF can set this property if it knows the result table column value is unique. The server uses this information during the Optimization state. The UDF can set this property only for argument 0.

Returns

On success, returns the `sizeof(a_sql_byte)` or:

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – if the attribute was not available to set. This can happen if the column was not involved in the query.

On failure, returns one of the generic `describe_column` errors, or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – set error returned if the describe buffer is not the size of `a_sql_byte`.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – set error returned if the query processing state is not Optimization.
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – set error returned if the **arg_num** is not zero.
- `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` – set error returned if the UDF attempts to set this attribute to a value other than 0 or 1.

Query Processing States

Valid in:

- Optimization state

See also

- *Generic describe_column Errors* on page 325
- *EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE (Get)* on page 216

- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT (Set)
The **EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT** attribute indicates if the column is constant. Used in a `describe_column_set` scenario.

Data Type
`a_sql_byte`

Description
True, if the column is constant for the lifetime of the statement. This property is valid only for input table arguments.

Usage
This is a read only property. All attempts to set it return `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE`.

Returns

- `EXTFNAPIV4_DESCRIBE_INVALID ATTRIBUTE` – this is a read-only property; all attempts to set return this error.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – set error returned, if the state is not Optimization.
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – set error returned, if the **arg_num** is not zero.
- `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` – set error returned, if the UDF attempts to set this attribute to a value other than 0 or 1.

Query Processing States
Not applicable.

See also

- *EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT (Get)* on page 217
- *Generic describe_column Errors* on page 325
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE (Set)
The **EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE** attribute indicates the constant value of the column. Used in a `describe_column_set` scenario.

Data Type
`an_extfn_value`

Description
The value of the column, if it is constant for the statement lifetime. If `EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT` for this column returns true, this value is available. This property is valid only for table arguments.

Usage
This property is read-only.

Returns

- `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE` – this is a read-only property; all attempts to set return this error.

Query Processing States
Not applicable.

See also

- *Generic describe_column Errors* on page 325
- *EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE (Get)* on page 218
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER (Set)
The **EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER** attribute indicates if the column in the result table is used by the consumer. Used in a `describe_column_set` scenario.

Data Type
`a_sql_byte`

Description
Used either to determine whether a column in the result table is used by the consumer, or to indicate that a column in an input is not needed. Valid for table arguments. Allows the user to set or retrieve information about a single column, whereas the similar attribute `EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS` sets or retrieves information about all columns in a single call.

Usage
The UDF sets **EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER** on columns in an input table to inform the producer that it does not need values for the column.

Returns
On success, returns the `sizeof(a_sql_byte)` or:

- EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE – if the attribute was not available to set. This can happen if the column was not involved in the query.

On failure, returns one of the generic describe_column errors, or:

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH – set error returned if the describe buffer is not the size of a_v4_extfn_estimate.
- EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – set error returned if the argument specified is argument 0.
- EXTFNAPIV4_DESCRIBE_INVALID_STATE – set error returned if the state is not equal to Optimization.
- EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE – set error returned if the value the UDF is setting is not 0 or 1.

Query Processing States
Valid during:

- Optimization state

The PROCEDURE definition and code fragment in the _describe_extfn API function:

```
CREATE PROCEDURE my_tpf( col_char char(10), col_table TABLE( c1 INT,
c2
INT ) )
      RESULTS ( r1 INT, r2 INT, r3 INT )
      EXTERNAL 'my_tpf_proc@mylibrary';

CREATE TABLE T( x INT, y INT, z INT );

select r2,r3 from my_tpf( 'test', TABLE( select x,y from T ) )
```

When this TPF runs, it is beneficial for the server to know if column y is used by this TPF. If the TPF does not need y, the server can use this knowledge for optimization and does not send this column information to the TPF.

```
my_tpf_describe(a_v4_extfn_proc_context *cntxt)
{
    if( cntxt->current_state == EXTFNAPIV4_STATE_OPTIMIZATION ) {
      a_sql_byte  col_is_used = 0;
      a_sql_int32 ret = 0;

      ret = cntxt->describe_column_get( cntxt, 2, 2,
            EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER,
            &col_is_used,
            sizeof(a_sql_byte) );

      if( ret < 0 ) {
          // Handle the error.
      }

    }
}
```

See also

- *EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER (Get)* on page 219
- *Generic describe_column Errors* on page 325
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE (Set)

The **EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE** attribute indicates the minimum value for the column. Used in a `describe_column_set` scenario.

Data Type

`an_extfn_value`

Description

The minimum value a column can have, if available. Only valid for argument 0.

Usage

The UDF can set `EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE`, if it knows what the minimum data value of the column is. The server can use this information during optimization.

The UDF can use `EXTFNAPIV4_DESCRIBE_COL_TYPE` to determine the data type of the column, and `EXTFNAPIV4_DESCRIBE_COL_WIDTH` to determine the storage requirements of the column, to provide an equivalently sized buffer to hold the value to set.

Returns

On success, returns the `describe_buffer_length`, or:

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – if the attribute cannot be set. Returned if the column was not involved in the query or the minimum value was not available for the requested column.

On failure, returns one of the generic `describe_column` errors, or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – set error returned, if the describe buffer is not large enough to hold the minimum value.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – set error returned, if the state is not equal to Optimization.
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – set error returned, if the **arg_num** is not 0.

Query Processing States

Valid in:

- Optimization state

Example

The **PROCEDURE** definition and UDF code fragment that implements the `_describe_extfn` callback API function:

```
CREATE PROCEDURE my_tpf( col_char char(10), col_table TABLE( c1 INT,
c2 INT ) )
    RESULTS ( r1 INT, r2 INT, r3 INT )
    EXTERNAL 'my_tpf_proc@mylibrary';

CREATE TABLE T( x INT, y INT, z INT );

select * from my_tpf( 'test', TABLE( select x,y from T ) )
```

This example shows a TPF where it is useful to the server (or to another TPF that takes the result of this TPF as input) to know the minimum value of result set column one. In this instance, the minimum output value of column one is 27.

```
my_tpf_describe(a_v4_extfn_proc_context *cntxt)
{
    if( cntxt->current_state == EXTFNAPIV4_STATE_OPTIMIZATION ) {
    a_sql_int32 min_value = 27;
    a_sql_int32 ret = 0;

// Tell the server what the minimum value of the first column
// of our result set will be.

    ret = cntxt->describe_column_set( cntxt, 0, 1
        EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE,
        &min_value,
        sizeof(a_sql_int32) );

    if( ret < 0 ) {
        // Handle the error.
    }
}
}
```

See also

- *Query Processing States* on page 121
- *EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE (Get)* on page 221
- *EXTFNAPIV4_DESCRIBE_COL_TYPE (Set)* on page 228
- *EXTFNAPIV4_DESCRIBE_COL_TYPE (Get)* on page 211
- *EXTFNAPIV4_DESCRIBE_COL_WIDTH (Set)* on page 229
- *EXTFNAPIV4_DESCRIBE_COL_WIDTH (Get)* on page 212
- *Generic describe_column Errors* on page 325

EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE (Set)

The **EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE** attribute indicates the maximum value for the column. Used in a `describe_column_set` scenario.

Data Type

`an_extfn_value`

Description

The maximum value for a column. This property is valid only for argument 0 and table arguments.

Usage

The UDF can set `EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE`, if it knows what the maximum data value of the column is. The server can use this information during optimization.

The UDF can use `EXTFNAPIV4_DESCRIBE_COL_TYPE` to determine the data type of the column, and `EXTFNAPIV4_DESCRIBE_COL_WIDTH` to determine the storage requirements of the column, to provide an equivalently sized buffer to hold the value to set.

`describe_buffer_length` is the `sizeof()` this buffer.

Returns

On success, returns the `describe_buffer_length`, if the value was set, or:

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – if the attribute could not be set. Returned if the column was not involved in the query or the maximum value was not available for the requested column.

On failure, returns one of the generic `describe_column` errors, or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – set error returned, if the describe buffer is not large enough to hold the maximum value.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – Set error returned, if the query processing state is not equal to Optimization.
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – set error returned, if the **arg_num** is not 0.

Query Processing States

Valid in:

- Optimization state

Example

The PROCEDURE definition and and UDF code fragment that implements the `_describe_extfn` callback API function:

```
CREATE PROCEDURE my_tpf( col_char char(10), col_table TABLE( c1 INT,
c2 INT ) )
         RESULTS ( r1 INT, r2 INT, r3 INT )
         EXTERNAL 'my_tpf_proc@mylibrary';

CREATE TABLE T( x INT, y INT, z INT );

select * from my_tpf( 'test', TABLE( select x,y from T ) )
```

This example shows a TPF where it is useful to the server (or to another TPF that takes the result of this TPF as input) to know the maximum value of result set column one. In this instance, the maximum output value of column one is 500000.

```
my_tpf_describe(a_v4_extfn_proc_context *cntxt)
{
    if( cntxt->current_state == EXTFNAPIV4_STATE_OPTIMIZATION ) {
         a_sql_int32 max_value = 500000;
         a_sql_int32 ret = 0;

        // Tell the server what the maximum value of the first column
         // of our result set will be.

      ret = cntxt->describe_column_set( cntxt, 0, 1
                  EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE,
                  &max_value,
                  sizeof(a_sql_int32) );

         if( ret < 0 ) {
             // Handle the error.
         }

   }
}
```

See also

- *Query Processing States* on page 121
- *EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE (Get)* on page 223
- *EXTFNAPIV4_DESCRIBE_COL_TYPE (Get)* on page 211
- *EXTFNAPIV4_DESCRIBE_COL_TYPE (Set)* on page 228
- *EXTFNAPIV4_DESCRIBE_COL_WIDTH (Get)* on page 212
- *EXTFNAPIV4_DESCRIBE_COL_WIDTH (Set)* on page 229
- *Generic describe_column Errors* on page 325

EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT (Set)

The **EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT** attribute sets a subset of the values specified in an input column. Used in a `describe_column_set` scenario.

Data Type

`a_v4_extfn_col_subset_of_input`

Description

Column values are a subset of the values specified in an input column.

Usage

Setting this describe attribute informs the query optimizer that the indicated column values are a subset of those values specified in an input column. For example, consider a filter TPF that consumes a table and filters out rows based on a function. In such a case, the return table is a subset of the input table. Setting **EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT** for the filter TPF optimizes the query.

Returns

On success, returns the `sizeof(a_v4_extfn_col_subset_of_input).`

On failure, returns one of the generic `describe_column` errors, or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – set error returned if the buffer length is less than `sizeof (a_v4_extfn_col_subset_of_input).`
- `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` – set error returned if the column index of the source table is out of range.
- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – set error returned if the column `subset_of_input` is set on is not aplicable (for example, if the column is not in the select list).
- `EXTFNAPIV4_DESCRIBE_INVAILD_STATE` – set error returned if the query processing state is not Optimization.
- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – set error returned if the buffer length is zero.
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – set error returned if called on a parameter other than the return table.

Query Processing States

Valid in:

- Optimization state

Example

```
a_v4_extfn_col_subset_of_input colMap;

    colMap.source_table_parameter_arg_num = 4;
    colMap.source_column_number = i;

    desc_rc = ctx->describe_column_set( ctx,
        0, i,
        EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT,
        &colMap, sizeof(a_v4_extfn_col_subset_of_input) );
```

See also

- *EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT (Get)* on page 225
- *Generic describe_column Errors* on page 325
- *Query Processing States* on page 121

*describe_parameter_get

The `describe_parameter_get` v4 API method gets UDF parameter properties from the server.

Declaration

```
a_sql_int32 (SQL_CALLBACK *describe_parameter_get)(
            a_v4_extfn_proc_context             *cntxt,
            a_sql_uint32                        arg_num,
            a_v4_extfn_describe_udf_type        describe_type,
            const void                          *describe_buffer,
           size_t                             describe_buffer_len );
```

Parameters

Parameter	Description
cntxt	The procedure context object.
arg_num	The ordinal of the TABLE parameter (0 is for the result table and 1 is for first input argument)
describe_type	A selector indicating what property to set.
describe_buffer	A structure that holds the describe information for the specified property to set on the server. The specific structure or data type is indicated by the **describe_type** parameter.
describe_buffer_length	The length, in bytes, of the **describe_buffer**.

Returns

On success, returns 0 or the number of bytes written to the **describe_buffer**. A value of 0 indicates that the server was unable to get the attribute, but no error condition occurred. If an error occurred, or no property was retrieved, this function returns one of the generic **describe_parameter** errors.

Attributes for *describe_parameter_get

Code showing the attributes for `describe_parameter_get`.

```
typedef enum a_v4_extfn_describe_parm_type {
    EXTFNAPIV4_DESCRIBE_PARM_NAME,
    EXTFNAPIV4_DESCRIBE_PARM_TYPE,
```

```
    EXTFNAPIV4_DESCRIBE_PARM_WIDTH,
    EXTFNAPIV4_DESCRIBE_PARM_SCALE,
    EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL,
    EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES,
    EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT,
    EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE,

    EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS,

} a_v4_extfn_describe_parm_type;
```

EXTFNAPIV4_DESCRIBE_PARM_NAME Attribute (Get)

The `EXTFNAPIV4_DESCRIBE_PARM_NAME` attribute indicates the parameter name. Used in a `describe_parameter_get` scenario.

Data Type

`char[]`

Description

The name of a parameter to a UDF.

Usage

Gets the parameter name as defined in the **CREATE PROCEDURE** statement. Invalid for parameter 0.

Returns

On success, returns the length of the parameter name.

On failure, returns one of the generic `describe_parameter` errors or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – get error returned if the **describe_buffer** is not large enough to hold the name.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – get error returned if the query processing phase is not greater than Initial.
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – get error returned if the parameter is the result table.

Query Processing Phases

Valid in:

- Annotation phase
- Query optimization phase

- Plan building phase
- Execution phase

See also

- *EXTFNAPIV4_DESCRIBE_PARM_NAME Attribute (Set)* on page 262
- *Generic describe_parameter Errors* on page 326
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_PARM_TYPE Attribute (Get)

The **EXTFNAPIV4_DESCRIBE_PARM_TYPE** attribute returns the data type in a `describe_parameter_get` scenario.

Data Type

`a_sql_data_type`

Description

The data type of a parameter to a UDF.

Usage

Gets the data type of the parameter as defined in the **CREATE PROCEDURE** statement.

Returns

On success, returns `sizeof(a_sql_data_type)`.

On failure, returns one of the generic `describe_parameter` errors or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – get error returned if the **describe_buffer** is not the `sizeof(a_sql_data_type)`.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – get error returned if the query processing phase is not greater than Initial.

Query Processing Phases

Valid in:

- Annotation phase
- Query Optimization phase
- Plan Building phase
- Execution phase

See also

- *EXTFNAPIV4_DESCRIBE_PARM_TYPE Attribute (Set)* on page 263
- *Generic describe_parameter Errors* on page 326
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_PARM_WIDTH Attribute (Get)

The **EXTFNAPIV4_DESCRIBE_PARM_WIDTH** attribute indicates the width of a parameter. Used in a `describe_parameter_get` scenario.

Data Type

`a_sql_uint32`

Description

The width of a parameter to a UDF. `EXTFNAPIV4_DESCRIBE_PARM_WIDTH` applies only to scalar parameters. Parameter width is the amount of storage, in bytes, required to store a parameter of the associated data type.

- **Fixed length data types –** the bytes required to store the data.
- **Variable length data types –** the maximum length.
- **LOB data types –** the amount of storage required to store a handle to the data.
- **TIME data types –** the amount of storage required to store the encoded time.

Usage

Gets the width of the parameter as defined in the **CREATE PROCEDURE** statement.

Returns

On success, returns the `sizeof(a_sql_uint32)`.

On failure, returns one of the generic `describe_parameter` errors or:

- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – get error returned if the query processing phase is not greater than Initial.
- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – get error returned if the **describe_buffer** is not the size of `a_sql_uint32`.
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – get error returned if the specified parameter is a TABLE parameter. This includes parameter 0, or parameter *n* where *n* is an input table.

Query Processing Phases

Valid in:

- Annotation phase
- Query Optimization phase
- Plan Building phase
- Execution phase

Example

Sample procedure definition:

```
CREATE PROCEDURE my_udf(IN p1 INT, IN p2 char(100))
RESULT (x INT)
EXTERNAL NAME 'my_udf@myudflib';
```

Sample `_describe_extfn` API function code fragment:

```
my_udf_describe(a_v4_extfn_proc_context *cntxt)
{
    if( cntxt->current_state == EXTFNAPIV4_STATE_OPTIMIZATION ) {
    a_sql_uint32 width = 0;
    a_sql_int32 ret = 0;

    // Get the width of parameter 1
    ret = cntxt->describe_parameter_get( cntxt, 1,
    EXTFNAPIV4_DESCRIBE_PARM_WIDTH,
    &width,
    sizeof(a_sql_uint32) );

    if( ret < 0 ) {
        // Handle the error.
    }

    //Allocate some storage based on parameter width
    a_sql_byte *p = (a_sql_byte *)cntxt->alloc( cntxt, width )

    // Get the width of parameter 2
    ret = cntxt->describe_parameter_get( cntxt, 2,
        EXTFNAPIV4_DESCRIBE_PARM_WIDTH,
        &width,
        sizeof(a_sql_uint32) );
    if( ret <= 0 ) {
        // Handle the error.
    }

    // Allocate some storage based on parameter width
    char *c = (char *)cntxt->alloc( cntxt, width )

    …

}
}
```

See also

- *EXTFNAPIV4_DESCRIBE_PARM_WIDTH Attribute (Set)* on page 264
- *Generic describe_parameter Errors* on page 326
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_PARM_SCALE Attribute (Get)

The **EXTFNAPIV4_DESCRIBE_PARM_SCALE** attribute indicates the scale of a parameter. Used in a `describe_parameter_get` scenario.

Data Type

`a_sql_uint32`

Description

The scale of a parameter to a UDF. For arithmetic data types, parameter scale is the number of digits to the right of the decimal point in a number.

This attribute is not valid for:

- non-arithmetic data types
- TABLE parameters

Usage

Gets the scale of the parameter as defined in the **CREATE PROCEDURE** statement.

Returns

On success, returns the size of `(a_sql_uint32)`.

On failure, returns one of the generic `describe_parameter` errors or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – get error returned if the **describe_buffer** is not the size of `a_sql_uint32`.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – get error returned if the query processing phase is not greater than Initial.
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – get error returned if the specified parameter is a TABLE parameter. This includes parameter 0, or parameter *n* where *n* is an input table.

Query Processing Phases

Valid in:

- Annotation phase
- Query Optimization phase
- Plan Building phase
- Execution phase

Example

Sample `_describe_extfn` API function code fragment that gets the scale of parameter 1:

```
if( cntxt->current_state > EXTFNAPIV4_STATE_ANNOTATION ) {
    a_sql_uint32 scale = 0;
    a_sql_int32 ret = 0;

    ret = ctx->describe_parameter_get( ctx, 1,
EXTFNAPIV4_DESCRIBE_PARM_SCALE,
    &scale, sizeof(a_sql_uint32) );

    if( ret <= 0 ) {
    // Handle the error.
```

```
    }
}
```

See also

- *EXTFNAPIV4_DESCRIBE_PARM_SCALE Attribute (Set)* on page 265
- *Generic describe_parameter Errors* on page 326
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL Attribute (Get)

The **EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL** attribute indicates whether or not the parameter is null. Used in a `describe_parameter_get` scenario.

Data Type

`a_sql_byte`

Description

True, if the value of a parameter can be NULL at the time of execution. For a TABLE parameter or parameter 0, the value is false.

Usage

Gets whether or not the specified parameter can be null during query execution.

Returns

On success, returns the `sizeof(a_sql_byte)`.

On failure, returns one of the generic `describe_parameter` errors or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – Get error returned if the **describe_buffer** is not the size of `a_sql_byte`.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – Get error returned if the query processing phase is not greater than Plan Building.

Query Processing Phases

Valid in:

- Execution phase

Examples: EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL (Get)

Example procedure definitions, `_describe_extfn` API function code fragment, and SQL queries for getting **EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL** values.

Procedure Definition

Sample procedure definition used by the example queries in this topic:

```
CREATE PROCEDURE my_udf(IN p INT)
RESULT (x INT)
EXTERNAL NAME 'my_udf@myudflib';
```

API Function Code Fragment

Sample `_describe_extfn` API function code fragment used by the example queries in this topic:

```
my_udf_describe(a_v4_extfn_proc_context *cntxt)
{

    if( cntxt->current_state > EXTFNAPIV4_STATE_OPTIMIZATION ) {
    a_sql_byte can_be_null = 0;
    a_sql_int32 ret = 0;

    ret = cntxt->describe_parameter_get( cntxt, 1,
    EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL,
    &can_be_null,
    sizeof(a_sql_byte) );

    if( ret <= 0 ) {
        // Handle the error.
    }
}
}
```

Example 1: Without NOT NULL

This example creates a table with a single integer column without the **NOT NULL** modifier specified. The correlated subquery passes in column *c1* from the table `has_nulls`. When the procedure **my_udf_describe** is called during the Execution state, the call to `describe_parameter_get` populates **can_be_null** with a value of 1.

```
CREATE TABLE has_nulls ( c1 INT );
INSERT INTO has_nulls VALUES(1);
INSERT INTO has_nulls VALUES(NULL);
SELECT * from has_nulls WHERE (SELECT sum(my_udf.x) FROM
my_udf(has_nulls.c1)) > 0;
```

Example 2: With NOT NULL

This example creates a table with a single integer column with the **NOT NULL** modifier specified. The correlated subquery passes in column *c1* from the table `no_nulls`. When the procedure **my_udf_describe** is called during the Execution state, the call to `describe_parameter_get` populates **can_be_null** with a value of 0.

```
CREATE TABLE no_nulls ( c1 INT NOT NULL);
INSERT INTO no_nulls VALUES(1);
INSERT INTO no_nulls VALUES(2);
SELECT * from no_nulls WHERE (SELECT sum(my_udf.x) FROM
my_udf(no_nulls.c1)) > 0;
```

Example 3: With a Constant

This example calls the procedure **my_udf** with a constant. When the procedure **my_udf_describe** is called, during the Execution state, the call to `describe_parameter_get` populates **can_be_null** with a value of 0.

```
SELECT * from my_udf(5);
```

Example 4: With a NULL
This example calls the procedure **my_udf** with a NULL. When the procedure **my_udf_describe** is called, during the Execution state, the call to `describe_parameter_get` populates **can_be_null** with a value of 1.

```
SELECT * from my_udf(NULL);
```

EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES Attribute (Get)

The **EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES** attribute returns the number of distinct values. Used in a `describe_parameter_get` scenario.

Data Type
`a_v4_extfn_estimate`

Description
Returns the estimated number of distinct values across all invocations. valid only for scalar parameters.

Usage
If this information is available, the UDF returns the estimated number of distinct values with 100% confidence. If the information is not available, the UDF returns an estimate of 0 with 0% confidence.

Returns
On success, returns the `sizeof(a_v4_extfn_estimate)`.

On failure, returns one of the generic `describe_parameter` errors or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – get error returned if the **describe_buffer** is not the size of `a_v4_extfn_estimate`.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – get error returned if the query processing phase is not greater than Initial.
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – get error returned if the parameter is a TABLE parameter.

Query Processing Phases
Valid in:

- Annotation phase
- Query Optimization phase
- Plan Building phase
- Execution phase

Example
Sample `_describe_extfn` API function code fragment:

```
if( ctx->current_state >= EXTFNAPIV4_STATE_ANNOTATION ) {
    desc_est.value = 0.0;
    desc_est.confidence = 0.0;

    desc_rc = ctx->describe_parameter_get( ctx,
        1,
        EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES,
        &desc_est, sizeof(a_v4_extfn_estimate) );
}
```

See also

- *EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_ VALUES Attribute (Set)* on page 267
- *EXTFNAPIV4_DESCRIBE_PARM_TYPE Attribute (Get)* on page 244
- *Generic describe_parameter Errors* on page 326
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT Attribute (Get)

The `EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES` attribute returns whether or not the parameter is constant. Used in a `describe_parameter_get` scenario.

Data Type
`a_sql_byte`

Description
True, if the parameter is a constant for the statement. Valid only for scalar parameters.

Usage
Returns 0 if the value of the specified parameter is not a constant; returns 1 if the value of the specified parameter is a constant.

Returns
On success, returns the `sizeof(a_sql_byte)`.

On failure, returns one of the generic `describe_parameter` errors or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – get error returned if the **describe_buffer** is not the size of `a_sql_byte`.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – get error returned if the query processing phase is not greater than Initial.
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – get error returned if the parameter is a TABLE parameter.

Query Processing Phases
Valid in:

- Annotation phase
- Query Optimization phase
- Plan Building phase
- Execution phase

Example
Sample `_describe_extfn` API function code fragment:

```
if( ctx->current_state >= EXTFNAPIV4_STATE_ANNOTATION ) {
    desc_rc = ctx->describe_parameter_get( ctx,
        1,
        EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT,
        &desc_byte, sizeof( a_sql_byte ) );
}
```

See also

- *EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT Attribute (Set)* on page 267
- *EXTFNAPIV4_DESCRIBE_PARM_TYPE Attribute (Set)* on page 263
- *Generic describe_parameter Errors* on page 326
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE Attribute (Get)
The **EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE** attribute indicates the value of the parameter. Used in a `describe_parameter_get` scenario.

Data Type
`an_extfn_value`

Description
The value of the parameter if it is known at describe time. Valid only for scalar parameters.

Usage
Returns the value of the parameters.

Returns
On success, returns the `sizeof(an_extfn_value)` if the value is available, or:

- `EXTFNAPIV4_DESCRIBE_NOT_AVILABLE` – Value returned if the value is not constant.

On failure, returns one of the generic `describe_parameter` errors or:

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH – get error returned if the **describe_buffer** is not the size of an_extfn_value.
- EXTFNAPIV4_DESCRIBE_INVALID_STATE – get error returned if the phase is not greater than Initial.
- EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – get error returned if the parameter is a TABLE parameter.

Query Processing Phases
Valid in:

- Annotation phase
- Query Optimization phase
- Plan Building phase
- Execution phase

Example
Sample _describe_extfn API function code fragment:

```
if( ctx->current_state >= EXTFNAPIV4_STATE_ANNOTATION ) {
    a_sql_int32 desc_rc;
    desc_rc = ctx->describe_parameter_get( ctx,
        1,
        EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE,
        &arg,
        sizeof( an_extfn_value ) );
}
```

See also
- *EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT Attribute (Set)* on page 267
- *EXTFNAPIV4_DESCRIBE_PARM_TYPE Attribute (Get)* on page 244
- *Generic describe_parameter Errors* on page 326
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS Attribute (Get)
The EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS attribute indicates the number of columns in the table. Used in a describe_parameter_get scenario.

Data Type
a_sql_uint32

Description
The number of columns in the table. Only valid for argument 0 and table arguments.

Usage
Returns the number of columns in the specified table argument. Argument 0 returns the number of columns in the result table.

Returns
On success, returns the `sizeof(a_sql_uint32)`.

On failure, returns one of the generic `describe_parameter` errors or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – get error returned if the **describe_buffer** is not the size of `size of a_sql_uint32`.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – get error returned if the query processing phase is not greater than Initial.
- `EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER` – get error returned if the parameter is not a TABLE parameter.

Query Processing Phases
Valid in:

- Annotation phase
- Query Optimization phase
- Plan Building phase
- Execution phase

See also
- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS Attribute (Set)* on page 268
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS Attribute (Get)
The `EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS` attribute indicates the number of rows in the table. Used in a `describe_parameter_get` scenario.

Data Type
`a_v4_extfn_estimate`

Description
The estimated number of rows in the table. Only valid for argument 0 and table arguments.

Usage
Returns the estimated number of rows in the specified table argument or result set with a confidence of 100%.

Returns
On success, returns the size of `a_v4_extfn_estimate`.

On failure, returns one of the generic `describe_parameter` errors or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – get error returned if the **describe_buffer** is not the size of `a_v4_extfn_estimate`.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – get error returned if the query processing phase is not greater than Initial.
- `EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER` – get error returned if the parameter is not a TABLE parameter.

Query Processing Phases
Valid in:

- Annotation phase
- Query Optimization phase
- Plan Building phase
- Execution phase

See also

- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS Attribute (Set)* on page 269
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY Attribute (Get)
The `EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY` attribute indicates the order of rows in the table. Used in a `describe_parameter_get` scenario.

Data Type
`a_v4_extfn_orderby_list`

Description
The order of rows in the table. This property is only valid for argument 0 and table arguments.

Usage
This attribute allows the UDF code to:

- Determine if the input **TABLE** parameter has been ordered
- Declare that the result set is ordered

If the parameter number is 0, then the attribute refers to the outbound result set. If the parameter is > 0 and the parameter type is a table then the attribute refers to the input **TABLE** parameter.

The order is specified by the `a_v4_extfn_orderby_list`, which is a structure supporting a list of column ordinals and their associated ascending or descending property. If the UDF sets the order by property for the outbound result set, the server is then able to perform order by optimizations. For example, if the UDF produced ascending order on the

first result set column, the server will eliminate a redundant order by request on the same column.

If the UDF does not set the orderby property on the outbound result set, the server assumes the data is not ordered.

If the UDF sets the orderby property on the input **TABLE** parameter, the server guarantees data ordering for the input data. In this scenario, the UDF describes to the server that the input data must be ordered. If the server detects a runtime conflict it raises a SQL exception. For example, when the UDF describes that the first column of the input **TABLE** parameter must have ascending order and the SQL statement contains a descending clause, the server raises a SQL exception.

In the event that the SQL did not contain an ordering clause, the server automatically adds the ordering to ensure that input **TABLE** parameter is ordered as required.

Returns
If successful, returns the number of bytes copied from `a_v4_extfn_orderby_list`.

Query Processing States
Valid in:

- Annotation state
- Query optimization state

See also

- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY Attribute (Set)* on page 270
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY (Get)
The `EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY` attribute indicates that the UDF requires partitioning. Used in a `describe_parameter_get` scenario.

Data Type
`a_v4_extfn_column_list`

Description
UDF developers use **EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY** to programmatically declare that the UDF requires partitioning before invocation can proceed.

Usage
The UDF can inquire to the partition to enforce it, or to dynamically adapt the partitioning. It is the UDF's responsibility to allocate the **a_v4_extfn_column_list,** taking into consideration the total number of columns in the input table, and sending that data to the server.

Returns

On success, returns the size of `a_v4_extfn_column_list`. This value is equal to:

```
sizeof(a_v4_extfn_column_list) + sizeof(a_sql_uint32) *
number_of_partition_columns
```

On failure, returns one of the generic `describe_parameter` errors or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – get error returned if the buffer length is less than the expected size.

Query Processing Phases

Valid in:

- Query Optimization phase
- Plan Building phase
- Execution phase

Example

```
void UDF_CALLBACK my_tpf_proc_describe( a_v4_extfn_proc_context
*ctx )
{
    if( ctx->current_state == EXTFNAPIV4_STATE_OPTIMIZATION ) {
        a_sql_uint32    col_count   = 0;
        a_sql_uint32    buffer_size = 0;
        a_v4_extfn_column_list    *clist = NULL;

        col_count = 3;    // Set to the max number of possible pby
columns

       buffer_size = sizeof( a_v4_extfn_column_list ) + (col_count -
1) * sizeof( a_sql_uint32 );

        clist = (a_v4_extfn_column_list *)ctx->alloc( ctx,
buffer_size );

        clist->number_of_columns = 0;
        clist->column_indexes[0] = 0;
        clist->column_indexes[1] = 0;
        clist->column_indexes[2] = 0;

    args->r_api_rc = ctx->describe_parameter_get( ctx,
        args->p3_arg_num,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
    clist,
    buffer_size );
}
}
```

See also

- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY (Set)* on page 272

- *Generic describe_parameter Errors* on page 326
- *V4 API describe_parameter and EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY* on page 141
- *Parallel TPF PARTITION BY Examples Using EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY* on page 143
- *Query Processing States* on page 121
- *Partitioning Input Data* on page 140

EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND Attribute (Get)

The `EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND` attribute indicates that the consumer requests rewind of an input table. Used in a `describe_parameter_get` scenario.

Data Type

`a_sql_byte`

Description

Indicates that the consumer wants to rewind an input table. Valid only for table input arguments. By default, this property is false.

Usage

The UDF queries this property to retrieve the true/false value.

Returns

On success, returns `sizeof(a_sql_byte)`.

On failure, returns one of the generic `describe_parameter` errors, or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – get error returned if the **describe_buffer** is not the size of `a_sql_byte`.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – get error returned if the phase is not Optimization or Plan Building.
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – get error returned if the UDF attempts to get this attribute on parameter 0.
- `EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER` – get error returned if the UDF attempts to get this attribute on a parameter that is not a table.

Query Processing Phases

Valid in:

- Optimization phase
- Plan Building phase

See also

- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND Attribute (Set)* on page 273
- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND Attribute (Set)* on page 275
- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND Attribute (Get)* on page 259
- *_rewind_extfn* on page 323
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND Attribute (Get)

The `EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND` attribute indicates if the parameter supports rewind. Used in a `describe_parameter_get` scenario.

Data Type

`a_sql_byte`

Description

Indicates whether a producer can support rewind. Valid only for table arguments.

You must also provide an implementation of the rewind table callback (**_rewind_extfn()**) if you plan on setting `DESCRIBE_PARM_TABLE_HAS_REWIND` to `true`. The server will fail to execute the UDF if the callback method is not provided.

Usage

The UDF asks if a table input argument supports rewind. As a prerequisite, the UDF must request rewind using **DESCRIBE_PARM_TABLE_REQUEST_REWIND** before you can use this property.

Returns

On success, returns `sizeof(a_sql_byte)`.

On failure, returns one of the generic `describe_parameter` errors, or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – get error returned if the **describe_buffer** is not the size of `a_sql_byte`.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – get error error returned if the query processing phase is not greater than Annotation.
- `EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER` – get error returned if the UDF attempts to get this attribute on a parameter that is not a table.
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – get error returned if the UDF attempts to get this attribute on the result table.

Query Processing Phases

Valid in:

- Optimization phase
- Plan Building phase
- Execution phase

See also

- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND Attribute (Get)* on page 258
- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND Attribute (Set)* on page 273
- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND Attribute (Set)* on page 275
- *_rewind_extfn* on page 323
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS Attribute (Get)

The `EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS` attribute lists unconsumed columns. Used in a `describe_parameter_get` scenario.

Data Type

`a_v4_extfn_column_list`

Description

The list of output table columns that are not going to be consumed by the server or the UDF.

For the output TABLE parameter, the UDF normally produces the data for all the columns, and the server consumes all the columns. The same holds true for the input TABLE parameter where the server normally produces the data for all the columns, and the UDF consumes all the columns.

However, in some cases the server, or the UDF, may not consume all the columns. The best practice in such a case is for the UDF to perform a **GET** for the output table on the describe attribute **EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS**. This action queries the server for the list of output table columns that are not going to be consumed by the server. The list can then be used by the UDF when populating the column data for the output table; that is, the UDF does not attemp to populate data for unused columns.

In summary, for the output table the UDF polls the list of unused columns. For the input table, the UDF pushes the list of unused columns.

Usage

The UDF asks the server if all the columns of the output table are going to be used. The UDF must allocate a `a_v4_extfn_column_list` that includes all the columns of the output table, and then must pass it to the server. The server then marks all the unprojected column ordinals as 1. The list returned by the server can be used while producing the data.

Returns

On success, returns the size of the column list: `sizeof(a_v4_extfn_column_list) + sizeof(a_sql_uint32) * number result columns`.

On failure, returns one of the generic `describe_parameter` errors or:

- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – get error returned if the query processing phase is not greater than Plan Building.
- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – get error returned if the **describe_buffer** is not large enough to hold the returned list.
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – get error returned if the UDF attempts to get this attribute on an input table.
- `EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER` – get error returned if the UDF attempts to get this attribute on a parameter that is not a table.

Query Processing Phases

Valid in:

- Execution phase

See also

- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS Attribute (Set)* on page 276

*describe_parameter_set

The `describe_parameter_set` v4 API method sets properties about a single parameter to the UDF.

Declaration

```
a_sql_int32 (SQL_CALLBACK *describe_parameter_set)(
            a_v4_extfn_proc_context             *cntxt,
            a_sql_uint32                        arg_num,
            a_v4_extfn_describe_udf_type        describe_type,
            const void                          *describe_buffer,
           size_t                             describe_buffer_len );
```

Parameters

Parameter	Description
cntxt	The procedure context object.
arg_num	The ordinal of the TABLE parameter (0 is for the result table and 1 is for first input argument)

Parameter	Description
describe_type	A selector indicating what property to set.
describe_buffer	A structure that holds the describe information for the specified property to set on the server. The specific structure or data type is indicated by the **describe_type** parameter.
describe_buffer_length	The length in bytes of the **describe_buffer**.

Returns
On success, returns 0 or the number of bytes written to the **describe_buffer**. A value of 0 indicates that the server was unable to set the attribute, but no error condition occurred. If an error occurred, or no property was retrieved, this function returns one of the generic **describe_parameter** errors.

Attributes for *describe_parameter_set

Code showing the attributes for `describe_parameter_set`.

```
typedef enum a_v4_extfn_describe_parm_type {
    EXTFNAPIV4_DESCRIBE_PARM_NAME,
    EXTFNAPIV4_DESCRIBE_PARM_TYPE,
    EXTFNAPIV4_DESCRIBE_PARM_WIDTH,
    EXTFNAPIV4_DESCRIBE_PARM_SCALE,
    EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL,
    EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES,
    EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT,
    EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE,

    EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS,

    } a_v4_extfn_describe_parm_type;
```

EXTFNAPIV4_DESCRIBE_PARM_NAME Attribute (Set)

The `EXTFNAPIV4_DESCRIBE_PARM_NAME` attribute indicates the parameter name. Used in a `describe_parameter_set` scenario.

Data Type
`char[]`

Description
The name of a parameter to a UDF.

Usage
If the UDF sets this property, the server compares the value with the name of the parameter supplied in the **CREATE PROCEDURE** statement. If the two values do not match, the server returns an error. This allows the UDF to ensure the **CREATE PROCEDURE** statement has the same parameter names as the UDF is expecting.

Returns
On success, returns the length of the parameter name.

On failure, returns one of the generic `describe_parameter` errors or:

- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – set error returned if the state is not equal to Annotation.
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – set error returned if the parameter is the result table.
- `EXTFNAPI4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` – set error returned if the UDF tries to reset the name.

Query Processing States
Valid in:

- Annotation state

See also
- *EXTFNAPIV4_DESCRIBE_PARM_NAME Attribute (Get)* on page 243
- *Generic describe_parameter Errors* on page 326
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_PARM_TYPE Attribute (Set)
The `EXTFNAPIV4_DESCRIBE_PARM_TYPE` attribute indicates the data type of the parameter. Used in a `describe_parameter_set` scenario.

Data Type
`a_sql_data_type`

Description
The data type of a parameter to a UDF.

Usage
When the UDF sets this property, the server compares the value to the parameter type supplied in the **CREATE PROCEDURE** statement. If the two values do not match, the server returns an error. This check ensures that the **CREATE PROCEDURE** statement has the same parameter data types that the UDF expects.

Returns
On success, returns `sizeof(a_sql_data_type)`.

On failure, returns one of the generic `describe_parameter` errors or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – set error returned if the **describe_buffer** is not the `sizeof(a_sql_data_type)`.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – set error returned if the query processing state is not equal to Annotation.
- `EXTFNAPI4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` – set error returned if the UDF tries to set the datatype of a parameter to something other than what it is already defined as.

Query Processing States
Valid in:

- Annotation state

See also
- *EXTFNAPIV4_DESCRIBE_PARM_TYPE Attribute (Get)* on page 244
- *Generic describe_parameter Errors* on page 326
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_PARM_WIDTH Attribute (Set)
The **EXTFNAPIV4_DESCRIBE_PARM_WIDTH** attribute indicates the width of a parameter. Used in a `describe_parameter_set` scenario.

Data Type
`a_sql_uint32`

Description
The width of a parameter to a UDF. `EXTFNAPIV4_DESCRIBE_PARM_WIDTH` applies only to scalar parameters. Parameter width is the amount of storage, in bytes, required to store a parameter of the associated data type.

- **Fixed length data types –** the bytes required to store the data.
- **Variable length data types –** the maximum length.
- **LOB data types –** the amount of storage required to store a handle to the data.
- **TIME data types –** the amount of storage required to store the encoded time.

Usage
This is a read-only property. The width is derived from the associated column data type. Once the data type is set, you cannot change the width.

Returns

On success, returns the `sizeof(a_sql_uint32)`.

On failure, returns one of the generic `describe_parameter` errors or:

- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – set error returned if the query processing state is not equal to Annotation.
- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – set error returned if the **describe_buffer** is not the size of `a_sql_uint32`.
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – set error returned if the specified parameter is a TABLE parameter. This includes parameter 0, or parameter *n*, where *n* is an input table.
- `EXTFNAPI4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` – set error returned if the UDF tries to reset the parameter width.

Query Processing States

Valid in:

- Annotation state

See also

- *EXTFNAPIV4_DESCRIBE_PARM_WIDTH Attribute (Get)* on page 245
- *Generic describe_parameter Errors* on page 326
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_PARM_SCALE Attribute (Set)

The **EXTFNAPIV4_DESCRIBE_PARM_SCALE** attribute indicates the scale of a parameter. Used in a `describe_parameter_set` scenario.

Data Type

`a_sql_uint32`

Description

The scale of a parameter to a UDF. For arithmetic data types, parameter scale is the number of digits to the right of the decimal point in a number.

This attribute is invalid for:

- Nonarithmetic data types
- TABLE parameters

Usage

This is a read-only property. The scale is derived from the associated column data type. Once the data type is set, you cannot change the scale.

Returns
On success, returns `sizeof(a_sql_uint32)`.

On failure, returns one of the generic `describe_parameter` errors or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – set error returned if the **describe_buffer** is not the size of `a_sql_uint32`.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – set error returned if the state is not Annotation.
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – set error returned if the specified parameter is a TABLE parameter. This includes parameter 0, or parameter *n*, where *n* is an input table.

Query Processing States
Valid in:

- Annotation state

See also
- *EXTFNAPIV4_DESCRIBE_PARM_SCALE Attribute (Get)* on page 246
- *Generic describe_parameter Errors* on page 326
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL Attribute (Set)
The **EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL** attribute returns whether or not the parameter is null. Using this attribute in a `describe_parameter_set` scenario returns an error.

Data Type
`a_sql_byte`

Description
True, if the value of a parameter can be NULL at the time of execution. For a TABLE parameter or parameter 0, the value is false.

Usage
This is a read-only property.

Returns
This is a read-only property, so all attempts to set result in an `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE` error.

Query Processing States
Not applicable.

EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES Attribute (Set)

The **EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES** attribute returns the number of distinct values. Using this attribute in a `describe_parameter_set` scenario returns an error.

Data Type

`a_v4_extfn_estimate`

Description

Returns the estimated number of distinct values across all invocations. valid only for scalar parameters.

Usage

This is a read-only property.

Returns

This is a read-only property; all attempts to set result in an `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE` error.

Query Processing States

Not applicable.

See also

- *EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_ VALUES Attribute (Get)* on page 250
- *EXTFNAPIV4_DESCRIBE_PARM_TYPE Attribute (Get)* on page 244
- *Generic describe_parameter Errors* on page 326
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT Attribute (Set)

The `EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES` attribute returns whether or not the parameter is constant. Using this attribute in a `describe_parameter_set` scenario returns an error.

Data Type

`a_sql_byte`

Description

True, if the parameter is a constant for the statement. Valid only for scalar parameters.

Usage

This is a read-only property.

Returns
This is a read-only property; all attempts to set result in an `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE` error.

Query Processing States
Not applicable.

See also
- *EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT Attribute (Get)* on page 251
- *EXTFNAPIV4_DESCRIBE_PARM_TYPE Attribute (Set)* on page 263
- *Generic describe_parameter Errors* on page 326
- *Query Processing States* on page 121
- *EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE Attribute (Get)* on page 252
- *EXTFNAPIV4_DESCRIBE_PARM_TYPE Attribute (Get)* on page 244

EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE Attribute (Set)
The **EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE** attribute indicates the value of the parameter. Used in a `describe_parameter_set` scenario.

Data Type
`an_extfn_value`

Description
The value of the parameter if it is known at describe time. Valid only for scalar parameters.

Usage
This is a read-only property.

Returns
This is a read-only property; all attempts to set result in an `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE` error.

Query Processing States
Not applicable.

EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS Attribute (Set)
The `EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS` attribute indicates the number of columns in the table. Used in a `describe_parameter_set` scenario.

Data Type
`a_sql_uint32`

Description
The number of columns in the table. Only valid for argument 0 and table arguments.

Usage
If the UDF sets this property, the server compares the value with the name of the parameter supplied in the **CREATE PROCEDURE** statement. If the two values do not match, the server returns an error. This allows the UDF to ensure the **CREATE PROCEDURE** statement has the same parameter names as the UDF is expecting.

Returns
On success, returns the `sizeof(a_sql_uint32)`.

On failure, returns one of the generic `describe_parameter` errors or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – set error returned if the **describe_buffer** is not the size of `size of a_sql_uint32`.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – set error returned if the state is not ANNOTATION.
- `EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER` – set error returned if the parameter is not a TABLE parameter.
- `EXTFNAPI4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` – set error returned if the UDF tries to reset the number of columns of the specified table.

Query Processing States
Valid in:

- Annotation state

See also

- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS Attribute (Get)* on page 253
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS Attribute (Set)
The `EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS` attribute indicates the number of rows in the table. Used in a `describe_parameter_set` scenario.

Data Type
`a_sql_a_v4_extfn_estimate`

Description
The estimated number of rows in the table. Only valid for argument 0 and table arguments.

Usage
The UDF sets this property for argument 0 if it estimates the number of rows in the result set. The server uses the estimate during optimization to make query processing decisions. You cannot set this value for an input table.

If you do not set a value, the server defaults to the number of rows specified by the **DEFAULT_TABLE_UDF_ROW_COUNT** option.

Returns
On success, returns `a_v4_extfn_estimate`.

On failure, returns one of the generic `describe_parameter` errors or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – set error returned if the **describe_buffer** is not the size of `a_v4_extfn_estimate`.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – set error returned if the state is not Optimization.
- `EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER` – get error returned if the parameter is not a TABLE parameter.
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – get error returned if the TABLE parameter is not the result table.
- `EXTFNAPI4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` – get error returned if the UDF tries to reset the number of columns of the specified table.

Query Processing States
Valid in:

- Query Optimization state

See also

- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS Attribute (Get)* on page 254
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY Attribute (Set)
The `EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY` attribute indicates the order of rows in the table. Used in a `describe_parameter_set` scenario.

Data Type
`a_v4_extfn_orderby_list`

Description
The order of rows in the table. This property is only valid for argument 0 and table arguments.

Usage

This attribute allows the UDF code to:

- Determine if the input **TABLE** parameter has been ordered
- Declare that the result set is ordered.

If the parameter number is 0, then the attribute refers to the outbound result set. If the parameter is > 0 and the parameter type is a table then the attribute refers to the input **TABLE** parameter.

The order is specified by the `a_v4_extfn_orderby_list`, which is a structure supporting a list of column ordinals and their associated ascending or descending property. If the UDF sets the order by property for the outbound result set, the server is then able to perform order by optimizations. For example, if the UDF produced ascending order on the first result set column, the server will eliminate a redundant order by request on the same column.

If the UDF does not set the orderby property on the outbound result set, the server assumes the data is not ordered.

If the UDF sets the orderby property on the input **TABLE** parameter, the server guarantees data ordering for the input data. In this scenario, the UDF describes to the server that the input data must be ordered. If the server detects a runtime conflict it raises a SQL exception. For example, when the UDF describes that the first column of the input **TABLE** parameter must have ascending order and the SQL statement contains a descending clause, the server raises a SQL exception.

In the event that the SQL did not contain an ordering clause, the server automatically adds the ordering to ensure that input **TABLE** parameter is ordered as required.

Returns

If successful, returns the number of bytes copied from `a_v4_extfn_orderby_list`.

Query Processing States

Valid in:

- Annotation state
- Query optimization state

See also

- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY Attribute (Get)* on page 255
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY (Set)

The `EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY` attribute indicates that the UDF requires partitioning. Used in a `describe_parameter_set` scenario.

Data Type

`a_v4_extfn_column_list`

Description

UDF developers use **EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY** to programmatically declare that the UDF requires partitioning before invocation can proceed.

Usage

The UDF can inquire to the partition to enforce it, or to dynamically adapt the partitioning. The UDF must allocate the **a_v4_extfn_column_list**, taking into consideration the total number of columns in the input table, and sending that data to the server.

Returns

On success, returns the size of `a_v4_extfn_column_list`. This value is equal to:

```
sizeof(a_v4_extfn_column_list) + sizeof(a_sql_uint32) *
number_of_partition_columns
```

On failure, returns one of the generic `describe_parameter` errors or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – Set error returned if the buffer length is less than the expected size.

Query Processing States

Valid in:

- Annotation state
- Query Optimization state

Example

```
void UDF_CALLBACK my_tpf_proc_describe( a_v4_extfn_proc_context
*ctx )
{
    if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {
        a_sql_int32    rc = 0;
        a_v4_extfn_column_list pbcol =
        { 1,        // 1 column in the partition by list
        2 };    // column index 2 requires partitioning

        // Describe partitioning for argument 1 (the table)
        rc = ctx->describe_parameter_set(
        ctx, 1,
        EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
        &pbcol,
        sizeof(pbcol) );
```

```
        if( rc == 0 ) {
        ctx->set_error( ctx, 17000,
           "Runtime error, unable set partitioning requirements for
column." );
        }
    }
}
```

See also

- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY (Get)* on page 256
- *Generic describe_parameter Errors* on page 326
- *V4 API describe_parameter and EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY* on page 141
- *Parallel TPF PARTITION BY Examples Using EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY* on page 143
- *Query Processing States* on page 121
- *Partitioning Input Data* on page 140

EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND Attribute (Set)

The `EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND` attribute indicates that the consumer requests rewind of an input table. Used in a `describe_parameter_set` scenario.

Data Type

`a_sql_byte`

Description

Indicates that the consumer wants to rewind an input table. Valid only for table input arguments. By default, this property is false.

Usage

If the UDF requires input table rewind capability, the UDF must set this property during Optimization.

Returns

On success, returns `sizeof(a_sql_byte)`.

On failure, returns one of the generic `describe_parameter` errors, or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – set error returned if the **describe_buffer** is not the size of `a_sql_byte`.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – set error returned if the state is not equal to Optimization.

- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – set error returned if the UDF attempts to set this attribute on parameter 0.
- `EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER` – set error returned if the UDF attempts to set this attribute on a parameter that is not a table.
- `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` – set error returned if the UDF attempts to set this attribute to a value other than 0 or 1.

Query Processing States
Valid in:

- Optimization state

Example
In this example, when the function **my_udf_describe** is called during the Optimization state, the call to `describe_parameter_set` informs the producer of the table input parameter 1 that a rewind may be required.

Sample procedure definition:

```
CREATE PROCEDURE my_udf(IN t TABLE(c1 INT))
RESULT (x INT)
EXTERNAL NAME 'my_udf@myudflib';
```

Sample `_describe_extfn` API function code fragment:

```
my_udf_describe(a_v4_extfn_proc_context *cntxt)
{
if( cntxt->current_state == EXTFNAPIV4_STATE_OPTIMIZATION ) {
a_sql_byte rewind_required = 1;
a_sql_int32 ret = 0;

ret = cntxt->describe_parameter_set( cntxt, 1,
EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND,
&rewind_required,
sizeof(a_sql_byte) );

if( ret <= 0 ) {
    // Handle the error.
}
}
}
```

See also

- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND Attribute (Get)* on page 258
- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND Attribute (Set)* on page 275
- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND Attribute (Get)* on page 259
- *_rewind_extfn* on page 323

- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND Attribute (Set)

The `EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND` attribute indicates if the parameter supports rewind. Used in a `describe_parameter_set` scenario.

Data Type

`a_sql_byte`

Description

Indicates whether a producer can support rewind. Valid only for table arguments.

You must also provide an implementation of the rewind table callback (**_rewind_extfn()**), if you plan on setting `DESCRIBE_PARM_TABLE_HAS_REWIND` to true. The server cannot execute the UDF if you do not provide the callback method.

Usage

A UDF sets this property during the Optimization state if it can provide rewind capability for its result table at no cost. If it is expensive for the UDF to provide rewind, do not set this property, or set it to 0. If set to 0, the server provides rewind support.

Returns

On success, returns `sizeof(a_sql_byte)`.

On failure, returns one of the generic `describe_parameter` errors, or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – set error returned if the **describe_buffer** is not the size of `a_sql_byte`.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – set error returned if the state is not equal to Optimization.
- `EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER` – set error returned if the UDF attempts to set this attribute on a parameter that is not a table.
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – set error returned if the specified argument is not the result table.
- `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` – set error returned if the UDF attempts to set this attribute to a value other than 0 or 1.

Query Processing States

Valid in:

- Optimization state

See also

- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND Attribute (Get)* on page 258

- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND Attribute (Set)* on page 273
- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND Attribute (Get)* on page 259
- *_rewind_extfn* on page 323
- *Query Processing States* on page 121

EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS Attribute (Set)

The `EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS` attribute lists unconsumed columns. Used in a `describe_parameter_set` scenario.

Data Type

`a_v4_extfn_column_list`

Description

The list of output table columns that are not going to be consumed by the server or the UDF.

For the output TABLE parameter, the UDF normally produces the data for all the columns, and the server consumes all the columns. The same holds true for the input TABLE parameter where the server normally produces the data for all the columns, and the UDF consumes all the columns.

However, in some cases the server, or the UDF, may not consume all the columns. The best practice in such a case is that the UDF performs a **GET** for the output table on the describe attribute **EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS**. This action queries the server for the list of output table columns which are not going to be consumed by the server. The list can then be used by the UDF when populating the column data for the output table; that is, the UDF skips populating data for unused columns.

In summary, for the output table the UDF polls the list of unused columns. For the input table, the UDF pushes the list of unused columns.

Usage

The UDF sets this property during Optimization if it is not going to use certain columns of the input TABLE parameter. The UDF must allocate a `a_v4_extfn_column_list` that includes all the columns of the output table, and then must pass it to the server. The server then marks all the un-projected column ordinals as 1. The server copies the list into its internal data structure.

Returns

On success, returns the size of the column list: `sizeof(a_v4_extfn_column_list) + sizeof(a_sql_uint32) * number result columns`.

On failure, returns one of the generic `describe_parameter` errors or:

- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – set error returned if the state is not Optimization.
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – set error returned if the UDF attempts to get this attribute on an input table.
- `EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER` – set error returned if the UDF attempts to set this attribute on a parameter that is not a table.

Query Processing States
Valid in:

- Optimization state

See also

- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS Attribute (Get)* on page 260

*describe_udf_get

The `describe_udf_get` v4 API method gets UDF properties from the server.

Declaration

```
a_sql_int32 (SQL_CALLBACK *describe_udf_get)(
    a_v4_extfn_proc_context    *cntxt,
    a_v4_extfn_describe_udf_type    describe_type,
    void    *describe_buffer,
    size_t    describe_buffer_len );
```

Parameters

Parameter	Description
cntxt	The procedure context object for this UDF.
describe_type	A selector indicating what property to retrieve.
describe_buffer	A structure that holds the describe information for the specified property to set on the server. The specific structure or data type is indicated by the **describe_type** parameter.
describe_buffer_length	The length in bytes of the **describe_buffer**.

Returns
On success, returns 0 or the number of bytes written to the **describe_buffer**. A value of 0 indicates that the server was unable to get the attribute but no error condition occurred. If an error occurred, or no property was retrieved, this function returns one of the generic `describe_udf` errors.

See also

- **describe_udf_set* on page 279
- *Generic describe_udf Errors* on page 326

Attributes for *describe_udf_get

Code showing the attributes for `describe_udf_get`.

```
typedef enum a_v4_extfn_describe_udf_type {
    EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS,
    EXTFNAPIV4_DESCRIBE_UDF_LAST
} a_v4_extGetfn_describe_udf_type;
```

EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS Attribute (Get)

The `EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS` attribute indicates the number of parameters. Used in a `describe_udf_get` scenario.

Data Type

`a_sql_uint32`

Description

The number of parameters supplied to the UDF.

Usage

Gets the number of parameters as defined in the **CREATE PROCEDURE** statement.

Returns

On success, returns the `sizeof(a_sql_uint32)`.

On failure, returns one of the generic `describe_udf` errors, or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – get error returned if the describe buffer is not the size of `a_sql_uint32`.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – get error returned if the phase is not greater than Initial.

Query Processing Phases

- Annotation phase
- Query optimization phase
- Plan building phase
- Execution phase

See also

- *EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS Attribute (Set)* on page 280
- *Generic describe_udf Errors* on page 326
- *Query Processing States* on page 121

*describe_udf_set

The `describe_udf_set` v4 API method sets UDF properties on the server.

Declaration

```
a_sql_int32 (SQL_CALLBACK *describe_udf_set)(
    a_v4_extfn_proc_context    *cntxt,
    a_v4_extfn_describe_udf_type    describe_type,
    const void    *describe_buffer,
    size_t    describe_buffer_len );
```

Parameters

Parameter	Description
cntxt	The procedure context object for this UDF.
describe_type	A selector indicating what property to set.
describe_buffer	A structure that holds the describe information for the specified property to set on the server. The specific structure or data-type is indicated by the **describe_type** parameter.
describe_buffer_length	The length, in bytes, of **describe_buffer**.

Returns

On success, returns the number of bytes written to the **describe_buffer**. If an error occurs, or no property is retrieved, this function returns one of the generic `describe_udf` errors.

If an error occurs, or no property is retrieved, this function returns one of the generic `describe_udf` errors, or:

- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – set error returned if any of the **cntxt** or **describe_buffer** arguments are NULL or if **describe_buffer_length** is 0.
- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – set error returned if there is a discrepancy between the requested attribute's size and the supplied **describe_buffer_length**.

See also

- **describe_udf_get* on page 277
- *Generic describe_udf Errors* on page 326

Attributes for *describe_udf_set

Code showing the attributes for `describe_udf_set`.

```
typedef enum a_v4_extfn_describe_udf_type {
    EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS,
```

```
    EXTFNAPIV4_DESCRIBE_UDF_LAST
} a_v4_extGetfn_describe_udf_type;
```

EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS Attribute (Set)

The `EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS` attribute indicates the number of parameters. Used in a `describe_udf_set` scenario.

Data Type

`a_sql_uint32`

Description

The number of parameters supplied to the UDF.

Usage

If the UDF sets this property, the server compares the value with the number of parameters supplied in the **CREATE PROCEDURE** statement. If the two values do not match, the server returns a SQL error. This allows the UDF to ensure the **CREATE PROCEDURE** statement has the same number of parameters expected by the UDF.

Returns

On success, returns the `sizeof(a_sql_uint32)`.

On failure, returns one of the generic `describe_udf` errors, or:

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – Set error returned if the describe buffer is not the size of `a_sql_uint32`.
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – Set error returned if the state is not equal to Annotation.
- `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` – set error returned if the UDF tries to reset the parameter datatype.

Query processing states

- Annotation state

See also

- *EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS Attribute (Get)* on page 278
- *Generic describe_udf Errors* on page 326
- *Query Processing States* on page 121

Describe Column Type (a_v4_extfn_describe_col_type)

The `a_v4_extfn_describe_col_type` enumerated type selects the column property retrieved or set by the UDF.

Implementation

```
typedef enum a_v4_extfn_describe_col_type {
    EXTFNAPIV4_DESCRIBE_COL_NAME,
    EXTFNAPIV4_DESCRIBE_COL_TYPE,
    EXTFNAPIV4_DESCRIBE_COL_WIDTH,
    EXTFNAPIV4_DESCRIBE_COL_SCALE,
    EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL,
    EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES,
    EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE,
    EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT,
    EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE,
    EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER,
    EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE,
    EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE,
    EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT,
    EXTFNAPIV4_DESCRIBE_COL_LAST
} a_v4_extfn_describe_col_type;
```

Members Summary

Member	Description
EXTFNAPIV4_DESCRIBE_COL_NAME	Column name (valid identifier).
EXTFNAPIV4_DESCRIBE_COL_TYPE	Column data type.
EXTFNAPIV4_DESCRIBE_COL_WIDTH	String width (precision for NUMERIC).
EXTFNAPIV4_DESCRIBE_COL_SCALE	Scale for NUMERIC.
EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL	True, if a column can be NULL.
EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES	Estimated number of distinct values in the column.
EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE	True, if column is unique within the table.
EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT	True, if column is constant for statement lifetime.
EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE	The value of a parameter, if known at describe time.

Member	Description
EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER	True, if column is needed by the consumer of the table.
EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE	The minimum value for the column (if known).
EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE	The maximum value for the column (if known).
EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT	The result column values are a subset of columns from an input table.
EXTFNAPIV4_DESCRIBE_COL_LAST	First illegal value for v4 API. Out-of-band value.

Describe Parameter Type (a_v4_extfn_describe_parm_type)

The `a_v4_extfn_describe_parm_type` enumerated type selects the parameter property retrieved or set by the UDF.

Implementation

```
typedef enum a_v4_extfn_describe_parm_type {
    EXTFNAPIV4_DESCRIBE_PARM_NAME,
    EXTFNAPIV4_DESCRIBE_PARM_TYPE,
    EXTFNAPIV4_DESCRIBE_PARM_WIDTH,
    EXTFNAPIV4_DESCRIBE_PARM_SCALE,
    EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL,
    EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES,
    EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT,
    EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE,

    EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS,

    EXTFNAPIV4_DESCRIBE_PARM_LAST
} a_v4_extfn_describe_parm_type;
```

Members Summary

Member	Description
EXTFNAPIV4_DESCRIBE_PARM_NAME	Parameter name (valid identifier).
EXTFNAPIV4_DESCRIBE_PARM_TYPE	Data type.
EXTFNAPIV4_DESCRIBE_PARM_WIDTH	String width (precision for NUMERIC).
EXTFNAPIV4_DESCRIBE_PARM_SCALE	Scale for NUMERIC.
EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL	True, if the value can be NULL.
EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES	Estimated number of distinct values across all invocations.
EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT	True, if parameter is a constant for the statement.
EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE	The value of a parameter, if known at describe time.
These selectors can retrieve or set properties of a TABLE parameter. These enumerator values cannot be used with scalar parameters:	
EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS	The number of columns in the table.
EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS	Estimated number of rows in the table.
EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY	The order of rows in a table.
EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY	The partitioning; use *number_of_columns*=0 for ANY.
EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND	True, if the consumer wants the ability rewind the input table.
EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND	Return true, if the producer supports rewind.

Member	Description
EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS	The list of output table columns that are not going to be consumed by the server or the UDF.
EXTFNAPIV4_DESCRIBE_PARM_LAST	First illegal value for v4 API. Out-of-band value.

Describe Return (a_v4_extfn_describe_return)

The `a_v4_extfn_describe_return` enumerated type provides a return value, when `a_v4_extfn_proc_context.describe_xxx_get()` or `a_v4_extfn_proc_context.describe_xxx_set()` does not succeed.

Implementation

```
typedef enum a_v4_extfn_describe_return {
    EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE          =  0,   // the specified operation has no
meaning either for this attribute or in

the current context.
    EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH        = -1,   // the provided buffer size
does not match the required length or the

length is insufficient.
    EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER       = -2,   // the provided parameter number
is invalid
    EXTFNAPIV4_DESCRIBE_INVALID_COLUMN           = -3,   // the column number is invalid
for this TABLE parameter
    EXTFNAPIV4_DESCRIBE_INVALID_STATE          = -4,   // the describe method call is not
valid in the present state
    EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE       = -5,   // the attribute is known but not
appropriate for this object
    EXTFNAPIV4_DESCRIBE_UNKNOWN_ATTRIBUTE        = -6,   // the identified attribute is
not known to this server version
    EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER        = -7,   // the specified parameter is
not a TABLE parameter (for describe_col_get()

or set())
    EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE       = -8,   // the specified attribute
value is illegal
    EXTFNAPIV4_DESCRIBE_LAST                = -9
} a_v4_extfn_describe_return;
```

Members Summary

Member	Return Value	Description
EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE	0	The specified operation has no meaning either for this attribute or in the current context.
EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH	-1	The provided buffer size does not match the required length, or the length is insufficient.
EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER	-2	The provided parameter number is invalid.
EXTFNAPIV4_DESCRIBE_INVALID_COLUMN	-3	The column number is invalid for this TABLE parameter.
EXTFNAPIV4_DESCRIBE_INVALID_STATE	-4	The describe method call is invalid in the present state.
EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE	-5	The attribute is known but not appropriate for this object.
EXTFNAPIV4_DESCRIBE_UNKNOWN_ATTRIBUTE	-6	The identified attribute is not known to this server version.
EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER	-7	The specified parameter is not a TABLE parameter (for `describe_col_get()` or `describe_col_set()`).
EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE	-8	The specified attribute value is illegal.

Member	Return Value	Description
EXTFNAPIV4_DESCRIBE_LAST	-9	First illegal value for v4 API.

Description

The return value of `a_v4_extfn_proc_context.describe_xxx_get()` and `a_v4_extfn_proc_context.describe_xxx_set()` is a signed integer. If the result is positive, the operation succeeds, and the value is the number of bytes copied. If the return value is less or equal to zero, the operation does not succeed, and the return value is one of the `a_v4_extfn_describe_return` values.

Describe UDF Type (a_v4_extfn_describe_udf_type)

Use the `a_v4_extfn_describe_udf_type` enumerated type to select the logical property the UDF retrieves or sets.

Implementation

```
typedef enum a_v4_extfn_describe_udf_type {
    EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS,
    EXTFNAPIV4_DESCRIBE_UDF_LAST
} a_v4_extfn_describe_udf_type;
```

Members Summary

Member	Description
EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS	The number of parameters supplied to the UDF.
EXTFNAPIV4_DESCRIBE_UDF_LAST	Out-of-band value.

Description

The `a_v4_extfn_proc_context.describe_udf_get()` method is used by the UDF to retrieve properties, and the `a_v4_extfn_proc_context.describe_udf_set()` method is used by the UDF to set properties about the UDF as a whole. The `a_v4_extfn_describe_udf_type` enumerator selects the logical property the UDF retrieves or sets.

See also

- *External Procedure Context (a_v4_extfn_proc_context)* on page 292

Execution State (a_v4_extfn_state)

The `a_v4_extfn_state` enumerated type represents the query processing phase of a UDF.

Implementation

```
typedef enum a_v4_extfn_state {
   EXTFNAPIV4_STATE_INITIAL,                // Server initial state,
not used by UDF
    EXTFNAPIV4_STATE_ANNOTATION,              // Annotating parse
tree with UDF reference
    EXTFNAPIV4_STATE_OPTIMIZATION,            // Optimizing
   EXTFNAPIV4_STATE_PLAN_BUILDING,           // Building execution
plan
    EXTFNAPIV4_STATE_EXECUTING,               // Executing UDF and
fetching results from UDF
    EXTFNAPIV4_STATE_LAST
} a_v4_extfn_state;
```

Members Summary

Member	Description
EXTFNAPIV4_STATE_INITIAL	Server initial phase. The only UDF method that is called during this query processing phase is `_start_extfn`.
EXTFNAPIV4_STATE_ANNOTATION	Annotating parse tree with UDF reference. The UDF is not invoked during this phase.
EXTFNAPIV4_STATE_OPTIMIZATION	Optimizing. The server calls the UDF's `_start_extfn` method, followed by the `_describe_extfn` function.
EXTFNAPIV4_STATE_PLAN_BUILDING	Building a query execution plan. The server calls the UDF's `_describe_extfn` function.
EXTFNAPIV4_STATE_EXECUTING	Executing UDF and fetching results from UDF. The server calls the `_describe_extfn` function before starting to fetch data from the UDF. The server then calls `_evaluate_extfn` to start the fetch cycle. During the fetch cycle, the server calls the functions defined in `a_v4_extfn_table_func`. When fetching finishes, the server calls the UDF's `_close_extfn` function.

Member	Description
EXTFNAPIV4_STATE_LAST	First illegal value for v4 API. Out-of-band value.

Description

The `a_v4_extfn_state` enumeration indicates which stage of UDF execution the server is in. When the server makes a transition from one phase to the next, the server informs the UDF it is leaving the previous phase by calling the UDF's `_leave_state_extfn` function. The server informs the UDF it is entering the new phase by calling the UDF's `_enter_state_extfn` function.

The query processing phase of a UDF restricts the operations that the UDF can perform. For example, in the Annotation phase, the UDF can retrieve the data types only for constant parameters.

See also

- *Query Processing States* on page 121
- *_start_extfn* on page 289
- *_evaluate_extfn* on page 290
- *_enter_state_extfn* on page 291
- *_leave_state_extfn* on page 291
- *Table Functions (a_v4_extfn_table_func)* on page 319

External Function (a_v4_extfn_proc)

The server uses the `a_v4_extfn_proc` structure to call into the various entry points in the UDF. The server passes an instance of `a_v4_extfn_proc_context` to each of the functions.

Method Summary

Method	Description
_start_extfn	Allocates a structure and stores its address in the **_user_data** field in the `a_v4_extfn_proc_context`.
_finish_extfn	Deallocates a structure whose address was stored in the **user_data** field in the `a_v4_extfn_proc_context`.

Method	Description
_evaluate_extfn	Required function pointer to be called for each invocation of the function on a new set of argument values.
_describe_extfn	See *Describe API* on page 208.
_enter_state_extfn	The UDF can use this function to allocate structures.
_leave_state_extfn	The UDF can use this function to release memory or resources needed for the state.

_start_extfn

Use the `_start_extfn` v4 API method as an optional pointer to an initializer function, for which the only argument is a pointer to `a_v4_extfn_proc_context` structure.

Declaration

```
_start_extfn(
a_v4_extfn_proc_context *
)
```

Usage

Use the `_start_extfn` method to allocate a structure and store its address in the `_user_data` field in the `a_v4_extfn_proc_context`. This function pointer must be set to the null pointer if there is no need for any initialization.

Parameters

Parameter	Description
cntxt	The procedure context object.

_finish_extfn

Use the `_finish_extfn` v4 API method as an optional pointer to a shutdown function, for which the only argument is a pointer to `a_v4_extfn_proc_context`.

Declaration

```
_finish_extfn(
     a_v4_extfn_proc_context *cntxt,
)
```

Usage

The `_finish_extfn` API deallocates a structure for which the address was stored in the `user_data` field in the `a_v4_extfn_proc_context`. This function pointer must be set to the null pointer if there is no need for any cleanup.

Parameters

Parameter	Description
cntxt	The procedure context object.

_evaluate_extfn

Use the `_evaluate_extfn` v4 API method as a required function pointer that is called for each invocation of the function on a new set of argument values.

Declaration

```
_evaluate_extfn(
     a_v4_extfn_proc_context *cntxt,
     void *args_handle
)
```

Usage

The `_evaluate_extfn` function must describe to the server how to fetch results by filling in the `a_v4_extfn_table_func` portion of the `a_v4_extfn_table` structure and use the `set_value` method on the context with argument zero to send this information to the server. This function must also inform the server of its output schema by filling in the `a_v4_extfn_value_schema` of the `a_v4_extfn_table` structure before calling `set_value` on argument 0. It can access its input argument values via the `get_value` callback function. Both constant and nonconstant arguments are available to the UDF at this time.

Parameters

Parameter	Description
cntxt	The procedure context object.
args_handle	Handle to the arguments in the server.

_describe_extfn

`_describe_extfn` is called at the beginning of each state to allow the server to get and set logical properties. The UDF can do this by using the six describe methods (`describe_parameter_get`, `describe_parameter_set`,

`describe_column_get`, `describe_column_set`, `describe_udf_get`, and `describe_udf_set`) in the `a_v4_proc_context object`.

See *Describe API* on page 208.

_enter_state_extfn

The UDF can implement the `_enter_state_extfn` v4 API method as an optional entry point to be notified whenever the UDF enters a new state.

Declaration

```
_enter_state_extfn(
     a_v4_extfn_proc_context *cntxt,
)
```

Usage

The UDF can use this notification to allocate structures.

Parameters

Parameter	Description
cntxt	The procedure context object.

_leave_state_extfn

The `_leave_state_extfn` v4 API method is an optional entry point the UDF can implement to receive a notification when the UDF moves out of a query processing state.

Declaration

```
_leave_state_extfn(
     a_v4_extfn_proc_context *cntxt,
)
```

Usage

The UDF can use this notification to release memory or resources needed for the state.

Parameters

Parameter	Description
cntxt	The procedure context object.

External Procedure Context (a_v4_extfn_proc_context)

Use the `a_v4_extfn_proc_context` structure to retain context information from the server and from the UDF.

Implementation

```
typedef struct a_v4_extfn_proc_context {
.
.
.
} a_v4_extfn_proc_context;
```

Method Summary

Return Type	Method	Description
short	**get_value**	Gets input arguments to the UDF.
short	**get_value_is_constant**	Allows the UDF to ask whether a given argument is a constant.
short	**set_value**	Used by the UDF in either the `_evaluate_extfn` or `_describe_extfn` functions to describe to the server what its output will look like and to inform the server how to fetch results from the UDF.
a_sql_uint32	**get_is_cancelled**	Call the **get_is_cancelled** callback every second or two to see if the user has interrupted the current statement.
short	**set_error**	Rolls back the current statement and generates an error.
void	**log_message**	Writes a message to the message log.
short	**convert_value**	Converts one data type to another.
short	**get_option**	Gets the value of a settable option.
void	**alloc**	Allocates a block of memory of length at least "len".
void	**free**	Free the memory allocated by **alloc()** for the specified lifetime.
a_sql_uint32	**describe_column_get**	See **describe_column_get* on page 209.

Return Type	Method	Description
a_sql_uint32	**describe_column_set**	See **describe_column_set* on page 225.
a_sql_uint32	**describe_parameter_get**	See **describe_parameter_get* on page 242.
a_sql_uint32	**describe_parameter_set**	See **describe_parameter_set* on page 261.
a_sql_uint32	**describe_udf_get**	See **describe_udf_get* on page 277.
a_sql_uint32	**describe_udf_set**	See **describe_udf_set* on page 279.
short	**open_result_set**	Opens a result set for a table value.
short	**close_result_set**	Closes an open result set.
short	**get_blob**	Retrieves an input parameter that is a blob.
short	**set_cannot_be_distributed**	Disables distribution at the UDF level even if the library is distributable.

Data Members and Data Types Summary

Data Member	Data Type	Description
_user_data	void *	This data pointer can be filled in by any usage with whatever context data the external routine requires.
_executionMode	a_sql_uint32	Indicates the debug/trace level requested via the **External_UDF_Execution_Mode** option. This is a read-only field.
current_state	a_sql_uint32	The *current_state* attribute reflects the current execution mode of the context. This can be queried from functions such as `_describe_extfn` to determine what course of action to take.

Description

In addition to retaining context information from the server and the UDF, the structure `a_v4_extfn_proc_context` allows the UDF to call back into the server to perform certain actions. The UDF can store private data in this structure in the `_user_data` member.

An instance of this structure gets passed to the functions in the `a_v4_extfn_proc` method by the server. User data is not maintained until after the server reaches the Annotation state.

get_value

Use the `get_value` v4 API method to obtain the values of input arguments sent to the UDF in a SQL query.

Declaration

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

Usage

The `get_value` API is used in an evaluation method to retrieve the value of each input argument to the UDF. For narrow argument data types (>32K), a call to `get_value` is sufficient to retrieve the entire argument value.

The `get_value` API can be called from any API that has access to the `arg_handle` pointer. This includes API functions that take `a_v4_table_context` as a parameter. The `a_v4_table_context` has an `args_handle` member variable that can be used for this purpose.

For all fixed-length data types, the data is available in the returned value and no further calls are necessary to obtain all of the data. The producer can decide what the maximum length is that is returned entirely in the call to `get_value` method. All fixed length data types should be guaranteed to fit in a single contiguous buffer. For variable-length data, the limit is producer-dependant.

For nonfixed-length data types, and depending on the length of the data, a blob may need to be created using the `get_blob` method to get the data. You can use the macro **EXTFN_IS_INCOMPLETE** on the value returned by `get_value` to determine whether a blob object is required. If **EXTFN_IS_INCOMPLETE** evaluates to true, a blob is required.

For input arguments that are tables, the type is **AN_EXTFN_TABLE**. For this type of argument, you must create a result set using the `open_result_set` method to read values in from the table.

If a UDF requires the value of an argument prior to the `_evaluate_extfn` API being called, then the UDF should implement the `_describe_extfn` API. From the `_describe_extfn` API, the UDF can obtain the value of constant expressions using the `describe_parameter_get` method.

Parameters

Parameter	Description
arg_handle	A context pointer provided by the consumer.
arg_num	The index of the argument to get a value for. The argument index starts at 1.
value	The value of the specified argument.

Returns
1 if successful, 0 otherwise.

an_extfn_value Structure
The **an_extfn_value** structure represents the value of an input argument returned by the `get_value` API.

This code shows the declaration of the **an_extfn_value** structure:

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

This table describes what the returned values of **an_extfn_value** object look like after calling the `get_value` method:

Value Returned by get_value API	EXTFN_IS_INCOMPLETE	total_len	piece_len	data
null	FALSE	0	0	null
empty string	FALSE	0	0	non-null
Size < MAX_UINT32	FALSE	actual	actual	non-null
size < MAX_UINT32	TRUE	actual	0	non-null
size >= MAX_UINT32	TRUE	MAX_UINT32	0	non-null

The type field of **an_extfn_value** contains the data type of the value. For UDFs that have tables as input arguments, the data type of that argument is **DT_EXTFN_TABLE**. For v4 Table UDFs, the `remain_len` field is not used.

See also

- *_evaluate_extfn* on page 290
- *Table Context (a_v4_extfn_table_context)* on page 311
- *_describe_extfn* on page 290
- **describe_parameter_get* on page 242

get_value_is_constant

Use the `get_value_is_constant` v4 API method to determine whether the specified input argument value is a constant.

Declaration

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

Usage

The UDF can ask whether a given argument is a constant. This is 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.

Parameters

Parameter	Description
arg_handle	Handle the arguments in the server.
arg_num	The index value of the input argument being retrieved. Index values are 1..N.
value_is_constant	Out parameter for storing is constant.

Returns

1 if successful, 0 otherwise.

See also

- *_evaluate_extfn* on page 290

set_value

Use the `set_value` v4 API method to describe to the consumer how many columns the result set has and how data should be read.

Declaration

```
short set_value(
       void *          arg_handle,
       a_sql_uint32    arg_num,
       an_extfn_value *value
       )
```

Usage

This method is used by the UDF in the `_evaluate_extfn` API. The UDF must call the `set_value` method to tell the consumer how many columns are in the result set and what set of `a_v4_extfn_table_func` functions the UDF supports.

For the `set_value` API, the UDF provides an appropriate **arg_handle** pointer via the `_evaluate_extfn`API, or from the **args_handle** member of `a_v4_extfn_table_context` structure.

The **value** argument for the `set_value` method must be of type *DT_EXTFN_TABLE* for v4 Table UDFs.

Parameters

Parameter	Description
arg_handle	A context pointer provided by the consumer.
arg_num	The index of the argument to set a value for. The only supported argument is 0.
value	The value of the specified argument.

Returns

1 if successful, 0 otherwise.

See also

- *_evaluate_extfn* on page 290
- *Table Functions (a_v4_extfn_table_func)* on page 319
- *Table Context (a_v4_extfn_table_context)* on page 311

get_is_cancelled

Use the `get_is_cancelled` v4 API method to determine whether the statement has been cancelled.

Declaration

```
short get_is_cancelled(
       a_v4_extfn_proc_context *          cntxt,

       )
```

Usage

If a UDF entry point is performing work for an extended period of time (many seconds), 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 return. Call the `_finish_extfn` function to perform necessary cleanup. Do not subsequently call any other UDF entry points.

Parameters

Parameter	Description
cntxt	The procedure context object.

Returns

A nonzero value, if the statement is interrupted.

set_error

Use the `set_error` v4 API method to communicate an error back to the server and eventually to the user.

Declaration

```
void set_error(
       a_v4_extfn_proc_context *          cntxt,
       a_sql_uint32                       error_number,
       const char                         *error_desc_string
       )
```

Usage

Call the `set_error` API, if a UDF entry point encounters an error that should send an error message to the user and shut down the current statement. When called, `set_error` API rolls back the current statement and the user sees "`Error raised by user-defined function: <error_desc_string>`". The SQLCODE is the negated form of the supplied <error_number>.

To avoid collisions with existing error codes, UDFs should generate error numbers between 17000 and 99999. If a number outside this range is provided, the statement is still rolled back, but the error message is "`Invalid error raised by user-defined function: (<error_number>) <error_desc_string>`" with a SQLCODE of -1577. The maximum length of **error_desc_string** is 140 characters.

After a call to `set_error` is made, the UDF entry point should immediately perform a return; eventually the `_finish_extfn` function is called to perform necessary cleanup. Do not subsequently call any other UDF entry points.

Parameters

Parameter	Description
cntxt	The procedure context object
error_number	The error number to set
error_desc_string	The message string to use

See also

- *Scalar and Aggregate UDF Callback Functions* on page 82

log_message

Use the `log_message` v4 API method to to send a message to the server's message log.

Declaration

```
short log_message(
       const char         *msg,
       short              msg_length
       )
```

Usage

The `log_message` method writes a message to the message log. The message string must be a printable text string no longer than 255 bytes; longer messages may be truncated.

Parameters

Parameter	Description
msg	The message string to log
msg_length	The length of the message string

See also

- *Controlling Error Checking and Call Tracing* on page 27

convert_value

Use the `convert_value` v4 API method to convert data types.

Declaration

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

Usage

. The primary use of the `convert_value` API is the converting between `DT_DATE`, `DT_TIME`, and `DT_TIMESTAMP`, and `DT_TIMESTAMP_STRUCT`. An input and output `an_extfn_value` is passed to the function.

Input Parameters

Parameter	Description
an_extfn_value.data	Input data pointer
an_extfn_value.total_len	Length of input data
an_extfn_value.type	DT_ datatype of input

Output Parameters

Parameter	Description
an_extfn_value.data	UDF supplied output data point
an_extfn_value.piece_len	Maximum length of output data.
an_extfn_value.total_len	Server set length of converted
an_extfn_value.type	DT_ datatype of desired output

Returns

1 if successful, 0 otherwise.

See also

- *get_value* on page 294

get_option

The `get_option` v4 API method gets the value of a settable option.

Declaration

```
short get_option(
a_v4_extfn_proc_context * cntxt,
char *option_name,
an_extfn_value *output
)
```

Parameters

Parameter	Description
cntxt	The procedure context object
option_name	Name of the option to get
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` – server set data type of value

Returns

1 if successful, 0 otherwise.

See also

- *External Function Prototypes* on page 93
- *External Procedure Context (a_v4_extfn_proc_context)* on page 292

alloc

The `alloc` v4 API method allocates a block of memory.

Declaration

```
void*alloc(
     a_v4_extfn_proc_context *cntxt,
     size_t len
     )
```

Usage

Allocates a block of memory of length at least **len**. The returned memory is 8-byte aligned.

Tip: Use the `alloc()` method as your only means of memory allocation, which allows the server to keep track of how much memory is used by external routines. The server can adapt other memory users, track leaks, and provide improved diagnostics and monitoring.

Memory tracking is enabled only when **external_UDF_execution_mode** is set to a value of 1 or 2 (validation mode or tracing mode).

Parameters

Parameter	Description
cntxt	The procedure context object
len	The length, in bytes, to allocate

See also

- *free* on page 302
- *Enabling Memory Tracking* on page 135

free

The `free` v4 API method frees an allocated block of memory.

Declaration

```
void free(
     a_v4_extfn_proc_context *cntxt,
     void *mem
)
```

Usage

Frees the memory allocated by `alloc()` for the specified lifetime.

Memory tracking is enabled only when **external_UDF_execution_mode** is set to a value of 1 or 2 (validation mode or tracing mode).

Parameters

Parameter	Description
cntxt	The procedure context object
mem	Pointer to the memory allocated using the `alloc` method

See also

- *alloc* on page 301
- *Enabling Memory Tracking* on page 135

open_result_set

The `open_result_set` v4 API method opens a result set for a table value.

Declaration

```
short open_result_set(
a_v4_extfn_proc_context *cntxt,
a_v4_extfn_table *table,
a_v4_extfn_table_context **result_set
)
```

Usage

`open_result_set` opens a result set for a table value. A UDF can open a result set to read rows from an input parameter of type `DT_EXTFN_TABLE`. The server (or another UDF) can open a result set to read rows from the UDF.

Parameters

Parameter	Description
cntxt	The procedure context object
table	The table object on which to open a result set
result_set	An output parameter that is set to be an opened result set

Returns

1 if successful, 0 otherwise.

See the `fetch_block` and `fetch_into` v4 API method descriptions for examples of the use of `open_result_set`.

See also

- *External Procedure Context (a_v4_extfn_proc_context)* on page 292
- *fetch_into* on page 313
- *fetch_block* on page 316

close_result_set

The `close_result_set` v4 API method closes an open result set.

Declaration

```
short close_result_set(
    a_v4_extfn_proc_context *cntxt,
    a_v4_extfn_table_context *result_set
)
```

Usage

You can only use `close_result_set` once per result set.

Parameters

Parameter	Description
cntxt	The procedure context object
result_set	The result set to close

Returns

1 if successful, 0 otherwise.

get_blob

Use the `get_blob` v4 API method to retrieve an input blob parameter.

Declaration

```
short get_blob(
       void            *arg_handle,
       a_sql_uint32    arg_num,
       a_v4_extfn_blob **blob
       )
```

Usage

Use `get_blob` to retrieve a blob input parameter after calling `get_value()`. Use the macro `EXTFN_IS_INCOMPLETE` to determine if a blob object is required to read the data for the value returned from `get_value()`, if *piece_len* < *total_len*. The blob object is returned as an output parameter and is owned by the caller.

`get_blob` obtains a blob handle that can be used to read the contents of the blob. Call this method only on columns that contain blob objects.

Parameters

Parameter	Description
arg_handle	Handle to the arguments in the server
arg_num	The argument is a number 1...N
blob	Output argument containing the blob object

Returns
1 if successful, 0 otherwise.

See also
- *External Procedure Context (a_v4_extfn_proc_context)* on page 292
- *get_value* on page 294

set_cannot_be_distributed

The `set_cannot_be_distributed` v4 API method disables distributions at the UDF level, even if the distribution criteria are met at the library level.

Declaration

```
void set_cannot_be_distributed( a_v4_extfn_proc_context *cntxt)
```

Usage
In the default behavior, if the library is distributable, then the UDF is distributable. Use `set_cannot_be_distributed` in the UDF to push the decision to disable distribution to the server.

License Information (a_v4_extfn_license_info)

If you are a design partner, use the `a_v4_extfn_license_info` structure to define library-level license validations for your UDFs, including your company name, library version information, and an SAP-supplied license key.

Implementation

```
typedef struct an_extfn_license_info {
    short     version;
} an_extfn_license_info;

typedef struct a_v4_extfn_license_info {
    an_extfn_license_info version;

    const char        name[255];
```

```
    const char        info[255];
    void *        key;
} a_v4_extfn_license_info;
```

Data Member Summary

Data Member	Description
version	Internal use only. Must be set to 1.
name	Value the UDF sets as your company name.
info	Value the UDF sets for additional library information such as library version and build numbers.
key	(Design partners only) An SAP-supplied license key. The key is a 26-character array.

Optimizer Estimate (a_v4_extfn_estimate)

Use the `a_v4_extfn_estimate` structure to describe an estimate, which includes a value and a confidence level.

Implementation

```
typedef struct a_v4_extfn_estimate {
    double    value;
    double    confidence;
} a_v4_extfn_estimate;
```

Data Members and Data Types Summary

Data Member	Data Type	Description
value	double	The value for the estimate.
confidence	double	The confidence level associated with the estimate. The confidence varies from 0.0 to 1.0, with 0.0 meaning the estimate is invalid and 1.0 meaning the estimate is known to be true.

Order By List (a_v4_extfn_orderby_list)

Use the `a_v4_extfn_orderby_list` structure to describe the ORDER BY property of a table.

Implementation

```
typedef struct a_v4_extfn_orderby_list {
    a_sql_uint32        number_of_elements;
    a_v4_extfn_order_el order_elements[1];      // there are
number_of_elements entries
} a_v4_extfn_orderby_list;
```

Data Members and Data Types Summary

Data Member	Data Type	Description
number_of_elements	a_sql_uint32	The number of entries
order_elements[1]	a_v4_extfn_order_el	The order of the elements

Description

There are *number_of_elements* entries, each with a flag indicating whether the element is ascending or descending, and a column index indicating the appropriate column in the associated table.

See also

- *Column Order (a_v4_extfn_order_el)* on page 206

Partition By Column Number (a_v4_extfn_partitionby_col_num)

The `a_v4_extfn_partitionby_col_num` enumerated type represents the column number to allow the UDF to express **PARTITION BY** support similar to that of SQL support.

Implementation

```
typedef enum a_v4_extfn_partitionby_col_num {
    EXTFNAPIV4_PARTITION_BY_COLUMN_NONE = -1,        // NO PARTITION
BY
    EXTFNAPIV4_PARTITION_BY_COLUMN_ANY  = 0,         // PARTITION BY
ANY
                                // + INTEGER representing a specific
column ordinal
} a_v4_extfn_partitionby_col_num;
```

Members Summary

Member of a_v4_extfn_partition-by_col_num Enumerated Type	Value	Description
EXTFNAPIV4_PARTITION_BY_COLUMN_NONE	-1	NO PARTITION BY
EXTFNAPIV4_PARTITION_BY_COLUMN_ANY	0	PARTITION BY ANY positive integer representing a specific column ordinal
Column Ordinal Number	N > 0	Ordinal for the table column number to partition on

Description

This structure allows the UDF to programmatically describe the partitioning and the column to partition on.

Use this enumeration when populating the `a_v4_extfn_column_list number_of_columns` field. When describing partition by support to the server, the UDF sets the `number_of_columns` to one of the enumerated values, or to a positive integer representing the number of column ordinals listed. For example, to describe to the server that no partitioning is supported, create the structure as:

```
a_v4_extfn_column_list nopby = {
EXTFNAPIV4_PARTITION_BY_COLUMN_NONE,
0
};
```

The *EXTFNAPIV4_PARTITION_BY_COLUMN_ANY* member informs the server that the UDF supports any form of partitioning.

To describe a set of ordinals to partition on, create the structure as:

```
a_v4_extfn_column_list nopby = {
2,
3, 4
};
```

This describes a partition by over 2 columns whose ordinals are 3 and 4.

Note: This example is for illustrative purposes only and is not legal code. The caller must allocate the structure accordingly with room for 3 integers.

Row (a_v4_extfn_row)

Use the `a_v4_extfn_row` structure to represent the data in a single row.

Implementation

```
/* a_v4_extfn_row - */
typedef struct a_v4_extfn_row {
    a_sql_uint32          *row_status;
    a_v4_extfn_column_data    *column_data;
} a_v4_extfn_row;
```

Data Members and Data Types Summary

Data Member	Data Type	Description
row_status	a_sql_uint32 *	The status of the row. Set to 1 for existing rows and 0 otherwise.
column_data	a_v4_extfn_column_data *	An array of column data for the row.

Description

The row structure contains information for a specific row of columns. This structure defines the status of an individual row and includes a pointer to the individual columns within the row. The row status is a flag that indicates the existence of a row. The row status flag can be altered by nested fetch calls without requiring manipulation of the row block structure.

The *row_status* flag set as 1 indicates that the row is available and can be included in the result set. The *row_status* set as 0 means the row should be ignored. This is useful when the TPF is acting as a filter because TPF may pass through rows of an input table to the result set, but it may also want to skip certain rows, which it can do by setting a status of 0 for those rows.

See also

- *Column Data (a_v4_extfn_column_data)* on page 204

Row Block (a_v4_extfn_row_block)

Use the `a_v4_extfn_row_block` structure to represent the data in a block of rows.

Implementation

```
/* a_v4_extfn_row_block - */
typedef struct a_v4_extfn_row_block {
    a_sql_uint32        max_rows;
    a_sql_uint32        num_rows;
```

```
    a_v4_extfn_row  *row_data;
} a_v4_extfn_row_block;
```

Data Members and Data Types Summary

Data Member	Data Type	Description
max_rows	a_sql_uint32	The maximum number of rows this row block can handle
num_rows	a_sql_uint32	Must be less than or equal to the maximum of rows the row block contains
row_data	a_v4_extfn_row *	The row data vector

Description

The row block structure is utilized by the `fetch_into` and `fetch_block` methods to allow the production and consumption of data. The allocator sets the maximum number of rows. The producer icorrectly sets the number of rows. The data consumer should not attempt to read more than number of rows produced.

The owner of the `row_block` structure determines the value of *max_rows* data member. For example, when a table UDF is implementing `fetch_into`, the value of *max_rows* is determined by the server as the number of rows that can fit into 128K of memory. However, when a table UDF is implementing `fetch_block`, the table UDF itself determines the value of *max_rows*.

Restrictions and Limitations

The value for the both the *num_rows* and *max_rows* is > 0. The *num_rows* must be <= *max_rows*. The row_data field should not be NULL for a valid row block.

Table (a_v4_extfn_table)

Use the `a_v4_extfn_table` structure to represent how data is stored in a table and how the consumer fetches that data.

Implementation

```
typedef struct a_v4_extfn_table {
    a_v4_extfn_table_func       *func;
    a_sql_uint32                number_of_columns;
} a_v4_extfn_table;
```

Data Members and Data Types Summary

Data Member	Data Type	Description
func	a_v4_extfn_table_func *	This member holds a set of function pointers that the consumer uses to fetch result data
number_of_columns	a_sql_uint32 *	The number of columns in the table

Table Context (a_v4_extfn_table_context)

The `a_v4_extfn_table_context` structure represents an open result set over a table.

Implementation

```
typedef struct a_v4_extfn_table_context {

//    size_t struct_size;

    /* fetch_into() - fetch into a specified row_block. This entry point
        is used when the consumer has a transfer area with a specific format.
       The fetch_into() function will write the fetched rows into the provided row block.
    */
    short (UDF_CALLBACK *fetch_into)(a_v4_extfn_table_context *cntxt,
a_v4_extfn_row_block *);

    /* fetch_block() - fetch a block of rows. This entry point is used
        when the consumer does not need the data in a particular format. For example,
        if the consumer is reading a result set and formatting it as HTML, the consumer
       does not care how the transfer area is layed out. The fetch_block() entry point is
        more efficient if the consumer does not need a specific layout.

        The row_block parameter is in/out. The first call should point to a NULL row
block.
       The fetch_block() call sets row_block to a block that can be consumed, and this
block
        should be passed on the next fetch_block() call.
    */
    short (UDF_CALLBACK *fetch_block)(a_v4_extfn_table_context *cntxt,
a_v4_extfn_row_block **row_block);

    /*  rewind() - this is an optional entry point. If NULL, rewind is not supported.
Otherwise,
        the rewind() entry point restarts the result set at the beginning of the table.
    */
    short (UDF_CALLBACK *rewind)(a_v4_extfn_table_context *);

    /* get_blob() - If the specified column has a blob object, return it.  The blob
         is returned as an out parameter and is owned by the caller.  This method should
         only be called on a column that contains a blob.  The helper macro
EXTFN_COL_IS_BLOB can
        be used to determine whether a column contains a blob.
    */
    short (UDF_CALLBACK *get_blob)(a_v4_extfn_table_context *cntxt,
                a_v4_extfn_column_data *col,
                a_v4_extfn_blob **blob);

   /* The following fields are reserved for future use and 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_v4_extfn_proc_context     *proc_context;
    void                        *args_handle;           // use in
a_v4_extfn_proc_context::get_value() etc.
    a_v4_extfn_table            *table;
    void                        *user_data;
    void                        *server_internal_use;

   /* The following fields are reserved for future use and must be initialized to NULL.
*/
    void  *reserved6_must_be_null;
    void  *reserved7_must_be_null;
    void  *reserved8_must_be_null;
    void  *reserved9_must_be_null;
    void  *reserved10_must_be_null;

} a_v4_extfn_table_context;
```

Method Summary

Data Type	Method	Description
short	**fetch_into**	Fetch into a specified `row_block`
short	**fetch_block**	Fetch a block of rows
short	**rewind**	Restarts the result set at the beginning of the table
short	**get_blob**	Return a blob object, if the specified column has a blob object

Data Members and Data Types Summary

Data Member	Data Type	Description
proc_context	a_v4_extfn_proc_context *	A pointer to the procedure context object. The UDF can use this to set errors, log messages, cancel, and so on.
args_handle	void *	A handle to the arguments provided by the server.
table	a_v4_extfn_table *	Points to the open result set table. This is populated after `a_v4_extfn_proc_context open_result_set` has been called.
user_data	void *	This data pointer can be filled in by any usage with whatever context data the external routine requires.
server_internal_use	void *	Internal use only.

Description

The `a_v4_extfn_table_context` structure acts as a middle layer between the producer and the consumer to help manage the data, when the consumer and producer require separate formats.

A UDF can read rows from an input TABLE parameter using `a_v4_extfn_table_context`. The server or another UDF can read rows from the result table of a UDF using `a_v4_extfn_table_context`.

The server implements the methods of `a_v4_extfn_table_context`, which gives the server an opportunity to resolve impedance mismatches.

See also

- *fetch_into* on page 313
- *fetch_block* on page 316
- *rewind* on page 318

fetch_into

The `fetch_into` v4 API method fetches data into a specified row block.

Declaration

```
short fetch_into(
a_v4_extfn_table_context *cntxt,
a_v4_extfn_row_block *)
```

Usage

The `fetch_into` method is useful when the producer does not know how data should be arranged in memory. This method is used as an entry point when the consumer has a transfer area with a specific format. The `fetch_into()` function writes the fetched rows into the provided row block. This method is part of the `a_v4_extfn_table_context` structure.

Use `fetch_into` when the consumer owns the memory for the data transfer area and requests that the producer use this area. You use the `fetch_into` method when the consumer cares about how the data transfer area is set up and it is up to the producer to perform the necessary data copying into this area.

Parameters

Parameter	Description
cntxt	The table context object obtained from the `open_result_set` API
row_block	The row block object to fetch into

Returns
1 if successful, 0 otherwise.

If the UDF returns 1, the consumer knows that there are more rows left and the `fetch_into` method should be called again. However, a UDF returning a value of 0 indicates that there are no more rows and a call to the `fetch_into` method is unnecessary.

Consider the following procedure definition, which is an example of a TPF function that consumes an input parameter table and produces it as a result table. Both are instances of SQL values that are obtained and returned through the `get_value` and `set_value` v4 API methods, respectively.

```
CREATE PROCEDURE FETCH_EX( IN a INT, INT b TABLE( c1 INT ) )
    RESULT SET ( rc INT )
```

This procedure definition contains two table objects:

- The input TABLE parameter named `b`
- The return result set table

The following example shows how output tables are fetched from by the caller, in this case, the server. The server might decide to use the `fetch_into` method. Input tables are fetched from by the called entity, in this case the TPF. The TPF decides which fetch API to use.

```
SELECT rc from FETCH_EX( 1, TABLE( SELECT c1 from TABLE ) )
```

The example shows that prior to fetching/consuming from an input table, a table context must be established via the `open_result_set` API on the `a_v4_extfn_proc` structure. The `open_result_set` requires a table object, which can be obtained through the `get_value` API.

```
    an_extfn_value    arg;
    ctx->get_value( args_handle, 3, &arg );

    if( arg.type != DT_EXTFN_TABLE ) {
        // handle error
    }

    a_v4_extfn_table_context    *rs = NULL;
    a_v4_extfn_table         *inTable = arg.data;
    ctx->open_result_set( ctx, inTable, &rs );
```

After the table context is created, the `rs` structure executes the `fetch_into` API and fetches the rows.

```
a_v4_extfn_row_block    *rb = // get a row block to hold a series of
INT values.
rs->fetch_into( rs, &rb )  // fetch the rows.
```

Prior to producing rows to a result table, a table object must be created and returned to the caller via the `set_value` API on the `a_v4_extfn_proc_context` structure.

This example shows that a table UDF must create an instance of the `a_v4_extfn_table` structure. Each invocation of the table UDF should return a separate instance of the `a_v4_extfn_table` structure. The table contains the state fields to keep track of the current row and the number of rows to generate. State for a table can be stored as a field of the instance.

```
    typedef struct rg_table : a_v4_extfn_table {
        a_sql_uint32        rows_to_generate;
        a_sql_uint32        current_row;
    } my_table;
```

In the following example, each time a row is produced, **current_row** is incremented until the number of rows to be generated is reached, when `fetch_into` returns false to indicate end-of-file. The consumer executes the `fetch_into` API implemented by the table UDF. As part of the call to the `fetch_into` method, the consumer provides the table context, as well as the row block to fetch into.

```
     rs->fetch_into( rs, &rb )

short UDF_CALLBACK my_table_func_fetch_into(
    a_v4_extfn_table_context *tctx,
    a_v4_extfn_row_block *rb)
/*****************************************/
{
    my_table *myTable = tctx->table;

    if( rgTable->current_row < rgTable->rows_to_generate ) {
            // Produce the row...
    rgTable->current_row++;
    return 1;
    }

    return 0;
}
```

See also

- *The fetch_into Method* on page 130
- *Table Context (a_v4_extfn_table_context)* on page 311
- *Row Block (a_v4_extfn_row_block)* on page 309
- *External Procedure Context (a_v4_extfn_proc_context)* on page 292
- *get_value* on page 294
- *set_value* on page 297
- *Table (a_v4_extfn_table)* on page 310

fetch_block

The `fetch_block` v4 API method fetches a block of rows.

Declaration

```
short fetch_block(
a_v4_extfn_table_context *cntxt,
a_v4_extfn_row_block **row_block)
```

Usage

The `fetch_block` method is used as an entry point when the consumer does not need the data in a particular format. `fetch_block` requests that the producer create a data transfer area and provide a pointer to that area. The consumer owns the memory and takes responsibility for copying data from this area.

The `fetch_block` is more efficient if the consumer does not require a specific layout. The `fetch_block` call sets a `fetch_block` to a block that can be consumed, and this block should be passed on the next `fetch_block` call. This method is part of the `a_v4_extfn_table_context` structure.

Parameters

Parameter	Description
cntxt	The table context object.
row_block	An in/out parameter. The first call should always point to a NULL **row_block**.

When `fetch_block` is called and **row_block** points to NULL, the UDF must allocate a `a_v4_extfn_row_block` structure.

Returns

1 if successful, 0 otherwise.

If the UDF returns 1, the consumer knows that there are more rows left and calls the `fetch_block` method again. However, a UDF returning a value of 0 indicates that there are no more rows and a call to the `fetch_block` method is unnecessary.

Consider the following procedure definition, which is an example of a TPF function that consumes an input parameter table and produces it as a result table. Both are instances of SQL values that are obtained and returned through the `get_value` and `set_value` v4 API methods, respectively.

```
CREATE PROCEDURE FETCH_EX( IN a INT, INT b TABLE( c1 INT ) )
    RESULT SET ( rc INT )
```

This procedure definition contains two table objects:

- The input TABLE parameter named `b`
- The return result set table

The following example shows how output tables are fetched from by the caller, in this case, the server. The server might decide to use the `fetch_block` method. Input tables are fetched from by the called entity, in this case the TPF, which decides which fetch API to use.

```
SELECT rc from FETCH_EX( 1, TABLE( SELECT c1 from TABLE ) )
```

The example shows that prior to fetching/consuming from an input table, a table context must be established via the `open_result_set` API on the `a_v4_extfn_proc` structure. The `open_result_set` requires a table object, which can be obtained through the `get_value` API.

```
    an_extfn_value    arg;
    ctx->get_value( args_handle, 3, &arg );

    if( arg.type != DT_EXTFN_TABLE ) {
        // handle error
    }

    a_v4_extfn_table_context    *rs = NULL;
    a_v4_extfn_table                *inTable = arg.data;
    ctx->open_result_set( ctx, inTable, &rs );
```

After the table context is created, the `rs` structure executes the `fetch_block` API and fetches the rows.

```
a_v4_extfn_row_block    *rb =  // get a row block to hold a series of
INT values.
rs->fetch_block( rs, &rb )  // fetch the rows.
```

Prior to producing rows to a result table, a table object must be created and returned to the caller via the `set_value` API on the `a_v4_extfn_proc_context` structure.

This example shows that a table UDF must create an instance of the `a_v4_extfn_table` structure. Each invocation of the table UDF should return a separate instance of the `a_v4_extfn_table` structure. The table contains the state fields to keep track of the current row and the number of rows to generate. State for a table can be stored as a field of the instance.

```
    typedef struct rg_table : a_v4_extfn_table {
        a_sql_uint32        rows_to_generate;
        a_sql_uint32        current_row;
    } my_table;
```

See also

- *The fetch_block Method* on page 130
- *Table Context (a_v4_extfn_table_context)* on page 311
- *Row Block (a_v4_extfn_row_block)* on page 309
- *External Procedure Context (a_v4_extfn_proc_context)* on page 292

- *get_value* on page 294
- *set_value* on page 297
- *open_result_set* on page 303
- *Table (a_v4_extfn_table)* on page 310

rewind

Use the `rewind` v4 API method to restart a result set at the beginning of the table.

Declaration

```
short rewind(
       a_v4_extfn_table_context           *cntxt,
       )
```

Usage

Call the `rewind` method on an open result set to rewind the table to the beginning. If the UDF intends to rewind an input table, it must inform the producer during the state **EXTFNAPIV4_STATE_OPTIMIZATION** using the **EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND** parameter.

`rewind()` is an optional entry point. If NULL, rewind is not supported. Otherwise, the `rewind()` entry point restarts the result set at the beginning of the table.

Parameters

Parameter	Description
cntxt	The table context object

Returns

1 if successful, 0 otherwise.

See also

- *Query Optimization State* on page 124
- *Execution State* on page 128
- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND Attribute (Set)* on page 273

get_blob

Use the `get_blob` v4 API method to return a blob object from a specified column.

Declaration

```
 short get_blob(
a_v4_extfn_table_context *cntxt,
```

```
a_v4_extfn_column_data *col,
a_v4_extfn_blob **blob
)
```

Usage

The blob is returned as an output parameter and is owned by the caller. Call this method only on a column that contains a blob.

Use the helper macro `EXTFN_COL_IS_BLOB` to determine whether a column contains a blob. This is the declaration of `EXTFN_COL_IS_BLOB` in the header file `extfnapiv4.h`:

```
 #define EXTFN_COL_IS_BLOB(c, n)         (c[n].blob_handle != NULL)
```

Parameters

Parameter	Description
cntxt	The table context object
col	The column data pointer for which to get the blob
blob	On success, contains the blob object associated with the column

Returns

1 if successful, 0 otherwise.

See also

- *Table Context (a_v4_extfn_table_context)* on page 311

Table Functions (a_v4_extfn_table_func)

The consumer uses the `a_v4_extfn_table_func` structure to retrieve results from the producer.

Implementation

```
typedef struct a_v4_extfn_table_func {
//    size_t struct_size;

    /*  Open a result set. The UDF can allocate any resources needed
for the result set.
    */
    short (UDF_CALLBACK *_open_extfn)(a_v4_extfn_table_context *);

   /*  Fetch rows into a provided row block. The UDF should implement
this method if it does
        not have a preferred layout for its transfer area.
    */
```

```
   short (UDF_CALLBACK *_fetch_into_extfn)(a_v4_extfn_table_context
*, a_v4_extfn_row_block

*row_block);

   /*  Fetch a block that is allocated and configured by the UDF. The
UDF should implement this
        method if it has a preferred layout of the transfer area.
    */
    short (UDF_CALLBACK *_fetch_block_extfn)
(a_v4_extfn_table_context *, a_v4_extfn_row_block

**row_block);

   /*  Restart a result set at the beginning of the table. This is an
optional entry point.
    */
    short (UDF_CALLBACK *_rewind_extfn)(a_v4_extfn_table_context *);

    /*  Close a result set. The UDF can release any resources
allocated for the result set.
    */
    short (UDF_CALLBACK *_close_extfn)(a_v4_extfn_table_context *);

    /* The following fields are reserved for future use and must be
initialized to NULL. */
    void *_reserved1_must_be_null;
    void *_reserved2_must_be_null;

} a_v4_extfn_table_func;
```

Method Summary

Method	Data Type	Description
_open_extfn	void	Called by the server to initiate row fetching by opening a result set. The UDF can allocate any resources needed for the result set.
_fetch_into_extfn	short	Fetch rows into a provided row block. The UDF implements this method, if it does not have a preferred layout for its transfer area.
_fetch_block_extfn	short	Fetch a block that is allocated and configured by the UDF. The UDF implements this method, if it has a preferred layout of the transfer area.
_rewind_extfn	void	Optional function called by the server to restart the fetching from the beginning of the table.

Method	Data Type	Description
`_close_extfn`	void	Called by the server to terminate row fetching by closing the result set. The UDF can release any resources allocated for the result set.
`_reserved1_must_be_null`	void	Reserved for future use. Must be initialized to NULL.
`_reserved1_must_be_null`	void	Reserved for future use. Must be initialized to NULL.

Description

The `a_v4_extfn_table_func` structure defines the methods used to fetch results from a table.

See also

- *Table (a_v4_extfn_table)* on page 310
- *Table Context (a_v4_extfn_table_context)* on page 311
- *_open_extfn* on page 321
- *_fetch_into_extfn* on page 322
- *_fetch_block_extfn* on page 322
- *_rewind_extfn* on page 323
- *_close_extfn* on page 324

_open_extfn

The server calls the `_open_extfn` v4 API method to initiate fetching of rows.

Declaration

```
void _open_extfn(
     a_v4_extfn_table_context *cntxt,
)
```

Usage

The UDF uses this method to open a result set and allocate any resources (for example, streams) needed for sending results to the server.

Parameters

Parameter	Description
cntxt	The procedure context object

See also

- *Table Context (a_v4_extfn_table_context)* on page 311

_fetch_into_extfn

The `_fetch_into_extfn` v4 API method fetches rows into a provided row block.

Declaration

```
short _fetch_into_extfn(
    a_v4_extfn_table_context *cntxt,
    a_v4_extfn_row_block *row_block
    )
```

Usage

The UDF should implement this method, if it does not have a preferred layout for its transfer area.

Parameters

Parameter	Description
cntxt	The procedure context object
row_block	The row block object to fetch into.

Returns

1 if successful, 0 otherwise.

See also

- *Table Context (a_v4_extfn_table_context)* on page 311
- *Row Block (a_v4_extfn_row_block)* on page 309

_fetch_block_extfn

The `_fetch_block_extfn` v4 API method fetches a block that is allocated and configured by the UDF.

Declaration

```
short _fetch_block_extfn(
a_v4_extfn_table_context *cntxt,
```

```
a_v4_extfn_row_block **
)
```

Usage

The UDF should implement this method, if it has a preferred layout for its transfer area.

Parameters

Parameter	Description
cntxt	The procedure context object
row_block	The row block object to fetch into

Returns

1 if successful, 0 otherwise.

See also

- *Table Context (a_v4_extfn_table_context)* on page 311
- *Row Block (a_v4_extfn_row_block)* on page 309

_rewind_extfn

The _rewind_extfn v4 API method restarts a result set at the beginning of the table.

Declaration

```
void _rewind_extfn(
a_v4_extfn_table_context *cntxt,
)
```

Usage

This function is an optional entry point. The UDF implements the `_rewind_extfn` method when the result table is rewound to the beginning. The UDF should consider implementing this method only if it can provide the rewind functionality in an efficient and cost-effective manner.

If a UDF chooses to implement the `_rewind_extfn` method, it should tell the consumer during the state **EXTFNAPIV4_STATE_OPTIMIZATION** by setting the **EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND** parameter for argument 0.

The UDF may decide not to provide the rewind functionality, in which case the server compensates and provides the functionality.

Note: The server can choose not to call the `_rewind_extfn` method to perform the rewind.

Parameters

Parameter	Description
cntxt	The procedure context object

Returns
No return value.

See also

- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND Attribute (Get)* on page 258
- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND Attribute (Set)* on page 273
- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND Attribute (Set)* on page 275
- *EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND Attribute (Get)* on page 259
- *Query Processing States* on page 121
- *Execution State (a_v4_extfn_state)* on page 287
- *Table Context (a_v4_extfn_table_context)* on page 311

_close_extfn

The server calls the `_close_extfn` v4 API method to terminate fetching of rows.

Declaration

```
void _close_extfn(
     a_v4_extfn_table_context *cntxt,
)
```

Usage
The UDF uses this method when fetching is complete to close a result set and release any resources allocated for the result set.

Parameters

Parameter	Description
cntxt	The procedure context object

See also

- *Table Context (a_v4_extfn_table_context)* on page 311

API Troubleshooting for a_v4_extfn

The `describe_column`, `describe_parameter`, and `describe_udf` v4 API methods can return generic error messages. Executing a UDF that does not exist on the server returns a `Could not execute statement` error.

Generic describe_column Errors

Common error returns for **describe_column** get and set calls.

Get	Set
`EXTFNAPIV4_DESCRIBE_ BUFFER_SIZE_MISMATCH` – get error returned if the **cntxt** or **describe_buffer** are NULL, or if the **describe_buffer_length** is 0.	`EXTFNAPIV4_DESCRIBE_ BUFFER_SIZE_MISMATCH` – set error returned if the **cntxt** or **describe_buffer** are NULL, or if the **describe_buffer_length** is 0.
`EXTFNAPIV4_DESCRIBE_INVALID_STATE` – get error returned if the **cntxt** parameter is not a valid context.	`EXTFNAPIV4_DESCRIBE_INVALID_STATE` – set error returned if the **cntxt** parameter is not a valid context.
`EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – get error returned if the provided parameter number is outside legal range for the procedure: **< 0** or **>** the number of parameters for the procedure.	`EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – set error returned if the provided parameter number is outside legal range for the procedure: **< 0** or **>** the number of parameters for the procedure.
`EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER` – get error returned if **arg_num** is not a TABLE parameter.	`EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER` – set error returned if **arg_num** is not a TABLE parameter.
`EXTFNAPIV4_DESCRIBE_INVALID_COLUMN` – get error returned if the column number is not valid for the TABLE parameter.	`EXTFNAPIV4_DESCRIBE_INVALID_COLUMN` – set error returned if the column number is not valid for the TABLE parameter.
`EXTFNAPIV4_DESCRIBE_UNKNOWN_ATTRIBUTE` – get error returned if the value of **describe_type** is not one of the valid describe types from `a_v4_extfn_describe_parm_type`.	`EXTFNAPIV4_DESCRIBE_UNKNOWN_ATTRIBUTE` – set error returned if the value of **describe_type** is not one of the valid describe types from `a_v4_extfn_describe_parm_type`.

Generic describe_udf Errors

Common error returns for `describe_udf` get and set calls.

Get	Set
`EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – get error returned if any of the **cntxt** or **describe_buffer** arguments are NULL or if **describe_buffer_length** is 0.	`EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – set error returned if any of the **cntxt** or **describe_buffer** arguments are NULL or if **describe_buffer_length** is 0.
`EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – get error returned if the **cntxt** parameter is an invalid context.	`EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – set error returned if the **cntxt** parameter is an invalid context.
`EXTFNAPIV4_DESCRIBE_ UNKNOWN_ATTRIBUTE` – get error returned if the value of **describe_type** is not one of the `a_v4_extfn_describe_udf_type` describe types.	`EXTFNAPIV4_DESCRIBE_ UNKNOWN_ATTRIBUTE` – set error returned if the value of **describe_type** is not one of the `a_v4_extfn_describe_udf_type` describe types.

Generic describe_parameter Errors

Common error returns for `describe_parameter` get and set calls.

Get	Set
`EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – get error returned if the **cntxt** or **describe_buffer** is NULL, or if **describe_buffer_length** is 0.	`EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – set error returned if the **cntxt** or **describe_buffer** is NULL, or if **describe_buffer_length** is 0.
`EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – get error returned if the **cntxt** parameter is invalid.	`EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – set error returned if the **cntxt** parameter is invalid.
`EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – get error returned if the provided parameter number is outside the legal range for the procedure; that is, if the parameter number is < 0 or > the number of parameters for the procedure.	`EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – set error returned if the provided parameter number is outside the legal range for the procedure; that is, if the parameter number is < 0 or > the number of parameters for the procedure.

Get	Set
EXTFNAPIV4_DESCRIBE_ UNKNOWN_ATTRIBUTE – get error returned if the value of **describe_type** is an invalid a_v4_extfn_describe_parm_type describe type.	EXTFNAPIV4_DESCRIBE_ UNKNOWN_ATTRIBUTE – set error returned if the value of **describe_type** is an invalid a_v4_extfn_describe_parm_type describe type.
EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH – get error returned if there is a discrepancy between the requested attribute size and the supplied describe_buffer_length. For fixed-size attributes, such as an a_sql_byte data type, the sizes must match. For variable-length attribute data types, such as char[], the supplied buffer needs to be at least large enough to hold the value of the requested attribute.	EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH – set error returned if there is a discrepancy between the size of the requested attribute and the supplied **describe_buffer_length**. For fixed-size attributes, such as an a_sql_byte data type, the sizes must match.

See also

- *Query Processing States* on page 121

Missing UDF Returns an Error

An attempt to execute a UDF that does not exist on the server returns an error.

If you attempt to execute a query similar to:

```
select my_sum1(n_tabkey) from tabudf()
```

where:

- tabudf() is a table UDF, and
- the UDF my_sum1() does not exist on the server,

this error is returned:

```
Could not execute statement.
External procedures or functions are not allowed across server types.
SQLCODE=-1579, ODBC 3 State="HY000"
Line 1, column 1
```

External Environment for UDFs

An improperly defined UDF may cause memory violations or may lead to a database server failure. Running a UDF outside the database server, in an external environment, eliminates this risk to the server.

If a runtime exception occurs in the external environment, the server process is unaffected. The server issues an error to the UDF caller, and any subsequent calls to the UDF result in a restart of the external environment.

Note: The external runtime environments do not require the IQ_UDF or IQ_IDA license. The external runtime environments do not require the `a_v3_extfn` or `a_v4_extfn` APIs.

The database server includes support for these external runtime environments for UDFs:

- ESQL and ODBC (C/C++ Embedded SQL or ODBC server-side requests)
- Java
- Perl
- PHP

Each environment has its own set of APIs for processing arguments and returning values back to the server. The Java external environment, for example, uses the JDBC API.

System Tables

The system table `SYSEXTERNENV` stores the information needed to identify and launch each of the external environments.

The system table `SYSEXTERNENVOBJECT` stores non-Java external objects.

SQL Statements

The following SQL syntax allows you to set or modify the location of external environments in the SYSEXTERNENV table.

```
ALTER EXTERNAL ENVIRONMENT environment-name
    [ LOCATION location-string ]
```

Once an external environment is set up to be used on the database server, you can then install objects into the database and create stored procedures and functions that make use of these objects within the external environment. Installing, creating, and using these objects, stored procedures, and stored functions is similar to installing Java classes and creating and using Java stored procedures and functions.

To add a comment for an external environment, you can execute:

```
COMMENT ON EXTERNAL ENVIRONMENT environment-name
 IS comment-string
```

To install a Perl or PHP external object (for example, a Perl script) from a file or an expression into the database, execute an **INSTALL EXTERNAL OBJECT** statement similar to:

```
INSTALL EXTERNAL OBJECT object-name-string
  [ update-mode ]
  FROM { FILE file-path | VALUE expression }
  ENVIRONMENT environment-name
```

To add a comment for an installed Perl or PHP external object, you can execute:

```
COMMENT ON EXTERNAL [ENVIRONMENT] OBJECT object-name-string
  IS comment-string
```

To remove an installed Perl or PHP external object from the database, use a **REMOVE EXTERNAL OBJECT** statement:

```
REMOVE EXTERNAL OBJECT object-name-string
```

Once external objects are installed in the database, you can use them in external stored procedure and function definitions (similar to the current mechanism for creating Java stored procedures and functions).

```
CREATE PROCEDURE procedure-name(...)
  EXTERNAL NAME '...'
  LANGUAGE environment-name

CREATE FUNCTION function-name(...)
  RETURNS ...
  EXTERNAL NAME '...'
  LANGUAGE environment-name
```

Once these stored procedures and functions are created, you can use them like any other stored procedure or function in the database. The database server, when encountering an external environment stored procedure or function, automatically launches the external environment (if it has not already been started), and sends the necessary information to get the external environment to fetch the external object from the database and execute it. Any result sets or return values resulting from the execution are returned as needed.

To start or stop an external environment on demand, use the **START EXTERNAL ENVIRONMENT** and **STOP EXTERNAL ENVIRONMENT** statements:

```
 START EXTERNAL ENVIRONMENT environment-name
STOP EXTERNAL ENVIRONMENT environment-name
```

Executing UDFs from an External Environment

Execute UDFs in the ESQL, ODBC, Java, Perl, or PHP external environments.

Prerequisites

There are no licensing prerequisites. The external runtime environments do not require the IQ_IDA license. The external runtime environments do not require the `a_v3_extfn` or `a_v4_extfn` APIs.

Task

1. Set up the external environment to be used on the database server.
```
ALTER EXTERNAL ENVIRONMENT environment-name
    [ LOCATION location-string ]
```
2. Install external objects (CLR, ESQL and ODBC, Java, Perl, or PHP) into the database.
3. Use **CREATE PROCEDURE** and **CREATE FUNCTION** statements to create stored procedures and functions that make use of these objects within the external environment.
4. Reference the stored procedure or function. Reference stored procedures in the **FROM** clause of your query.

See also

- *The ESQL and ODBC External Environments* on page 331
- *The Java External Environment* on page 341
- *PERL External Environment* on page 369
- *PHP External Environment* on page 373
- *CREATE PROCEDURE Statement (Java UDF)* on page 361
- *CREATE FUNCTION Statement (Java UDF)* on page 363

External Environment Restrictions

Restrictions apply to all external environments for UDFs.

- The **NO RESULT SET** option is not supported.
- Only **IN** parameters are supported: **INOUT/OUT** are not supported.
- Functions with `LONG VARCHAR` or `LONG BINARY` result values are not permitted.

See also

- *Java External Environment Restrictions* on page 347

The ESQL and ODBC External Environments

To run a compiled native C function in an external environment instead of within the database server, the stored procedure or function is defined with the EXTERNAL NAME clause followed by the LANGUAGE attribute specifying one of C_ESQL32, C_ESQL64, C_ODBC32, or C_ODBC64.

Unlike the Perl, PHP, and Java external environments, you do not install any source code or compiled objects in the database. As a result, you do not need to execute any INSTALL statements before using the ESQL and ODBC external environments.

Here is an example of a function written in C++ that can be run within the database server or in an external environment.

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "extfnapi.h"

BOOL APIENTRY DllMain( HMODULE hModule,
    DWORD  ul_reason_for_call,
    LPVOID lpReserved
    )
{
    return TRUE;
}

// Note: extfn_use_new_api used only for
// execution in the database server

extern "C" __declspec( dllexport )
a_sql_uint32 extfn_use_new_api( void )
{
    return( EXTFN_API_VERSION );
}

extern "C" __declspec( dllexport )
void SimpleCFunction(
    an_extfn_api *api,
    void *arg_handle )
{
    short               result;
    an_extfn_value      arg;
    an_extfn_value      retval;
    int *               intptr;
    int                 i, j, k;

    j = 1000;
    k = 0;
    for( i = 1; i <= 4; i++ )
    {
        result = api->get_value( arg_handle, i, &arg );
        if( result == 0 || arg.data == NULL ) break;
        if( arg.type & DT_TYPES != DT_INT ) break;
        intptr = (int *) arg.data;
        k += *intptr * j;
        j = j / 10;
    }
    retval.type = DT_INT;
    retval.data = (void*)&k;
    retval.piece_len = retval.len.total_len =
       (a_sql_uint32) sizeof( int );
    api->set_value( arg_handle, 0, &retval, 0 );
    return;
}
```

When compiled into a dynamic link library or shared object, this function can be called from an external environment. An executable image called dbexternc12 is started by the database server and this executable image loads the dynamic link library or shared object for you.

Note that 32-bit or 64-bit versions of the database server can be used and either version can start 32-bit or 64-bit versions of dbexternc12. This is one of the advantages of using the external environment. Note that once dbexternc12 is started by the database server, it does not terminate until the connection has been terminated or a STOP EXTERNAL ENVIRONMENT statement (with the correct environment name) is executed. Each connection that does an external environment call will get its own copy of dbexternc12.

To call the compiled native function, SimpleCFunction, a wrapper is defined as follows:

```
CREATE FUNCTION SimpleCDemo(
  IN arg1 INT,
  IN arg2 INT,
  IN arg3 INT,
  IN arg4 INT )
RETURNS INT
EXTERNAL NAME 'SimpleCFunction@c:\\c\\extdemo.dll'
LANGUAGE C_ODBC32;
```

This is almost identical to the way a compiled native function is described when it is to be loaded into the database server's address space. The one difference is the use of the LANGUAGE C_ODBC32 clause. This clause indicates that SimpleCDemo is a function running in an external environment and that it is using 32-bit ODBC calls. The language specification of C_ESQL32, C_ESQL64, C_ODBC32, or C_ODBC64 tells the database server whether the external C function issues 32-bit or 64-bit ODBC, ESQL, or `a_v4_extfn` API calls when making server-side requests.

When the native function uses none of the ODBC, ESQL, or SQL Anywhere C API calls to make server-side requests, then either C_ODBC32 or C_ESQL32 can be used for 32-bit applications and either C_ODBC64 or C_ESQL64 can be used for 64-bit applications. This is the case in the external C function shown above. It does not use any of these APIs.

To execute the sample compiled native function, execute the following statement.

```
SELECT SimpleCDemo(1,2,3,4);
```

To use server-side ODBC, the C/C++ code must use the default database connection. To get a handle to the database connection, call get_value with an EXTFN_CONNECTION_HANDLE_ARG_NUM argument. The argument tells the database server to return the current external environment connection rather than opening a new one.

```
#include <windows.h>
#include <stdio.h>
#include "odbc.h"
#include "extfnapi.h"

BOOL APIENTRY DllMain( HMODULE hModule,
    DWORD  ul_reason_for_call,
```

```
    LPVOID lpReserved
    )
{
    return TRUE;
}

extern "C" __declspec( dllexport )
void ServerSideFunction( an_extfn_api *api, void *arg_handle )
{
    short               result;
    an_extfn_value      arg;
    an_extfn_value      retval;
    SQLRETURN           ret;

    ret = -1;
    // set up the return value struct
    retval.type = DT_INT;
    retval.data = (void*) &ret;
    retval.piece_len = retval.len.total_len =
        (a_sql_uint32) sizeof( int );

    result = api->get_value( arg_handle,
                    EXTFN_CONNECTION_HANDLE_ARG_NUM,
                    &arg );
    if( result == 0 || arg.data == NULL )
    {
        api->set_value( arg_handle, 0, &retval, 0 );
        return;
    }

    HDBC dbc = (HDBC)arg.data;
    HSTMT stmt = SQL_NULL_HSTMT;
    ret = SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
    if( ret != SQL_SUCCESS ) return;
    ret = SQLExecDirect( stmt,
            (SQLCHAR *) "INSERT INTO odbcTab "
                "SELECT table_id, table_name "
                "FROM SYS.SYSTAB", SQL_NTS );
    if( ret == SQL_SUCCESS )
    {
        SQLExecDirect( stmt,
            (SQLCHAR *) "COMMIT", SQL_NTS );
    }
    SQLFreeHandle( SQL_HANDLE_STMT, stmt );

    api->set_value( arg_handle, 0, &retval, 0 );
    return;
}
```

If the above ODBC code is stored in the file extodbc.cpp, it can be built for Windows using the following commands.

```
cl extodbc.cpp /LD /Ic:\sa12\sdk\include odbc32.lib
```

The following example creates a table, defines the stored procedure wrapper to call the compiled native function, and then calls the native function to populate the table.

```
CREATE TABLE odbcTab(c1 int, c2 char(128));

CREATE FUNCTION ServerSideODBC( )
RETURNS INT
EXTERNAL NAME 'ServerSideFunction@extodbc.dll'
LANGUAGE C_ODBC32;

SELECT ServerSideODBC();

// The following statement should return two identical rows
SELECT COUNT(*) FROM odbcTab
UNION ALL
SELECT COUNT(*) FROM SYS.SYSTAB;
```

Similarly, to use server-side ESQL, the C/C++ code must use the default database connection. To get a handle to the database connection, call get_value with an EXTFN_CONNECTION_HANDLE_ARG_NUM argument. The argument tells the database server to return the current external environment connection rather than opening a new one.

```
#include <windows.h>
#include <stdio.h>

#include "sqlca.h"
#include "sqlda.h"
#include "extfnapi.h"

BOOL APIENTRY DllMain( HMODULE hModule,
    DWORD  ul_reason_for_call,
    LPVOID lpReserved
    )
{
    return TRUE;
}

EXEC SQL INCLUDE SQLCA;
static SQLCA *_sqlc;
EXEC SQL SET SQLCA "_sqlc";
EXEC SQL WHENEVER SQLERROR { ret = _sqlc->sqlcode; };

extern "C" __declspec( dllexport )
void ServerSideFunction( an_extfn_api *api, void *arg_handle )
{
    short               result;
    an_extfn_value      arg;
    an_extfn_value      retval;

    EXEC SQL BEGIN DECLARE SECTION;
    char *stmt_text =
        "INSERT INTO esqlTab "
            "SELECT table_id, table_name "
            "FROM SYS.SYSTAB";
    char *stmt_commit =
        "COMMIT";
    EXEC SQL END DECLARE SECTION;
```

```
    int ret = -1;

    // set up the return value struct
    retval.type = DT_INT;
    retval.data = (void*) &ret;
    retval.piece_len = retval.len.total_len =
        (a_sql_uint32) sizeof( int );

    result = api->get_value( arg_handle,
                    EXTFN_CONNECTION_HANDLE_ARG_NUM,
                    &arg );
    if( result == 0 || arg.data == NULL )
    {
        api->set_value( arg_handle, 0, &retval, 0 );
        return;
    }
    ret = 0;
    _sqlc = (SQLCA *)arg.data;

    EXEC SQL EXECUTE IMMEDIATE :stmt_text;
    EXEC SQL EXECUTE IMMEDIATE :stmt_commit;

    api->set_value( arg_handle, 0, &retval, 0 );
}
```

If the above embedded SQL statements are stored in the file extesql.sqc, it can be built for Windows using the following commands.

```
sqlpp extesql.sqc extesql.cpp
cl extesql.cpp /LD /Ic:\sa12\sdk\include c:\sa12\sdk\lib
\x86\dblibtm.lib
```

The following example creates a table, defines the stored procedure wrapper to call the compiled native function, and then calls the native function to populate the table.

```
CREATE TABLE esqlTab(c1 int, c2 char(128));

CREATE FUNCTION ServerSideESQL( )
RETURNS INT
EXTERNAL NAME 'ServerSideFunction@extesql.dll'
LANGUAGE C_ESQL32;

SELECT ServerSideESQL();

// The following statement should return two identical rows
SELECT COUNT(*) FROM esqlTab
UNION ALL
SELECT COUNT(*) FROM SYS.SYSTAB;
```

As in the previous examples, to use server-side SAP Sybase IQ C API calls, the C/C++ code must use the default database connection. To get a handle to the database connection, call get_value with an EXTFN_CONNECTION_HANDLE_ARG_NUM argument. The argument tells the database server to return the current external environment connection rather than opening a new one. The following example shows the framework for obtaining the

connection handle, initializing the C API environment, and transforming the connection handle into a connection object (a_sqlany_connection) that can be used with the SAP Sybase IQ C API.

```
include <windows.h>
#include "sacapidll.h"
#include "extfnapi.h"

BOOL APIENTRY DllMain( HMODULE hModule,
    DWORD  ul_reason_for_call,
    LPVOID lpReserved
    )
{
    return TRUE;
}

extern "C" __declspec( dllexport )
void ServerSideFunction( an_extfn_api *extapi, void *arg_handle )
{
    short               result;
    an_extfn_value      arg;
    an_extfn_value      retval;
    unsigned            offset;
    char                *cmd;

    SQLAnywhereInterface  capi;
    a_sqlany_connection * sqlany_conn;
    unsigned int          max_api_ver;

    result = extapi->get_value( arg_handle,
                    EXTFN_CONNECTION_HANDLE_ARG_NUM,
                    &arg );

    if( result == 0 || arg.data == NULL )
    {
        return;
    }
    if( !sqlany_initialize_interface( &capi, NULL ) )
    {
        return;
    }
    if( !capi.sqlany_init( "MyApp",
            SQLANY_CURRENT_API_VERSION,
            &max_api_ver ) )
    {
        sqlany_finalize_interface( &capi );
        return;
    }
    sqlany_conn = sqlany_make_connection( arg.data );

    // processing code goes here

    capi.sqlany_fini();

    sqlany_finalize_interface( &capi );
```

```
    return;
}
```

If the above C code is stored in the file extcapi.c, it can be built for Windows using the following commands.

```
cl /LD /Tp extcapi.c /Tp c:\sa12\SDK\C\sacapidll.c
      /Ic:\sa12\SDK\Include c:\sa12\SDK\Lib\X86\dbcapi.lib
```

The following example defines the stored procedure wrapper to call the compiled native function, and then calls the native function.

```
CREATE FUNCTION ServerSideC()
RETURNS INT
EXTERNAL NAME 'ServerSideFunction@extcapi.dll'
LANGUAGE C_ESQL32;

SELECT ServerSideC();
```

The LANGUAGE attribute in the above example specifies C_ESQL32. For 64-bit applications, you would use C_ESQL64. You must use the embedded SQL language attribute since the SAP Sybase IQ C API is built on the same layer (library) as ESQL.

As mentioned earlier, each connection that does an external environment call will start its own copy of dbexternc12. This executable application is loaded automatically by the server the first time an external environment call is made. However, you can use the START EXTERNAL ENVIRONMENT statement to preload dbexternc12. This is useful if you want to avoid the slight delay that is incurred when an external environment call is executed for the first time. Here is an example of the statement.

```
START EXTERNAL ENVIRONMENT C_ESQL32
```

Another case where preloading dbexternc12 is useful is when you want to debug your external function. You can use the debugger to attach to the running dbexternc12 process and set breakpoints in your external function.

The STOP EXTERNAL ENVIRONMENT statement is useful when updating a dynamic link library or shared object. It will terminate the native library loader, dbexternc12, for the current connection thereby releasing access to the dynamic link library or shared object. If multiple connections are using the same dynamic link library or shared object then each of their copies of dbexternc12 must be terminated. The appropriate external environment name must be specified in the STOP EXTERNAL ENVIRONMENT statement. Here is an example of the statement.

```
STOP EXTERNAL ENVIRONMENT C_ESQL32
```

To return a result set from an external function, the compiled native function must use the native function call interface.

The following code fragment shows how to set up a result set information structure. It contains a column count, a pointer to an array of column information structures, and a pointer to an array of column data value structures. The example also uses the SAP Sybase IQ C API.

```
an_extfn_result_set_info    rs_info;

int columns = capi.sqlany_num_cols( sqlany_stmt );

an_extfn_result_set_column_info *col_info =
    (an_extfn_result_set_column_info *)
    malloc( columns * sizeof(an_extfn_result_set_column_info) );

an_extfn_result_set_column_data *col_data =
    (an_extfn_result_set_column_data *)
    malloc( columns * sizeof(an_extfn_result_set_column_data) );

rs_info.number_of_columns   = columns;
rs_info.column_infos        = col_info;
rs_info.column_data_values  = col_data;
```

The following code fragment shows how to describe the result set. It uses the SAP Sybase IQ C API to obtain column information for a SQL query that was executed previously by the C API. The information that is obtained from the SAP Sybase IQ C API for each column is transformed into a column name, type, width, index, and null value indicator that will be used to describe the result set.

```
a_sqlany_column_info        info;
for( int i = 0; i < columns; i++ )
{
    if( sqlany_get_column_info( sqlany_stmt, i, &info ) )
    {
        // set up a column  description
        col_info[i].column_name  = info.name;
        col_info[i].column_type  = info.native_type;
        switch( info.native_type )
        {
           case DT_DATE:      // DATE is converted to string by C API
           case DT_TIME:      // TIME is converted to string by C API
          case DT_TIMESTAMP:  // TIMESTAMP is converted to string by
C API
           case DT_DECIMAL:    // DECIMAL is converted to string by C
API
                col_info[i].column_type  = DT_FIXCHAR;
                break;
          case DT_FLOAT:     // FLOAT is converted to double by C API
                col_info[i].column_type  = DT_DOUBLE;
                break;
           case DT_BIT:       // BIT is converted to tinyint by C API
                col_info[i].column_type  = DT_TINYINT;
                break;
        }
        col_info[i].column_width = info.max_size;
      col_info[i].column_index = i + 1; // column indices are origin
1
        col_info[i].column_can_be_null = info.nullable;
    }
}
// send the result set description
if( extapi->set_value( arg_handle,
```

```
                    EXTFN_RESULT_SET_ARG_NUM,
                    (an_extfn_value *)&rs_info,
                    EXTFN_RESULT_SET_DESCRIBE ) == 0 )
{
    // failed
    free( col_info );
    free( col_data );
    return;
}
```

Once the result set has been described, the result set rows can be returned. The following code fragment shows how to return the rows of the result set. It uses the SAP Sybase IQ C API to fetch the rows for a SQL query that was executed previously by the C API. The rows returned by the SAP Sybase IQ C API are sent back, one at a time, to the calling environment. The array of column data value structures must be filled in before returning each row. The column data value structure consists of a column index, a pointer to a data value, a data length, and an append flag.

```
a_sqlany_data_value *value = (a_sqlany_data_value *)
    malloc( columns * sizeof(a_sqlany_data_value) );

while( capi.sqlany_fetch_next( sqlany_stmt ) )
{
    for( int i = 0; i < columns; i++ )
    {
        if( capi.sqlany_get_column( sqlany_stmt, i, &value[i] ) )
        {
            col_data[i].column_index = i + 1;
            col_data[i].column_data  = value[i].buffer;
            col_data[i].data_length  =
(a_sql_uint32)*(value[i].length);
            col_data[i].append  = 0;
            if( *(value[i].is_null) )
            {
                // Received a NULL value
                col_data[i].column_data = NULL;
            }
        }
    }
    if( extapi->set_value(  arg_handle,
                        EXTFN_RESULT_SET_ARG_NUM,
                        (an_extfn_value *)&rs_info,
                        EXTFN_RESULT_SET_NEW_ROW_FLUSH ) == 0 )
    {
        // failed
        free( value );
        free( col_data );
        free( col_data );
        extapi->set_value( arg_handle, 0, &retval, 0 );
        return;
    }
}
```

The Java External Environment

The database server includes support for Java stored procedures and functions. A Java stored procedure or function behaves the same as a SQL stored procedure or function except that the code for the procedure or function is written in Java and the execution of the procedure or function takes place outside the database server (that is, within a Java VM environment).

It should be noted that there is one instance of the Java VM for each database rather than one instance per connection. Java stored procedures can return result sets.

There are a few prerequisites to using Java in the database support:

1. A copy of the Java Runtime Environment must be installed on the database server computer.
2. The database server must be able to locate the Java executable (the Java VM).

To use Java in the database, make sure that the database server is able to locate and start the Java executable. Verify that this can be done by executing:

```
START EXTERNAL ENVIRONMENT JAVA;
```

If the database server fails to start Java then the problem probably occurs because the database server is not able to locate the Java executable. In this case, you should execute an ALTER EXTERNAL ENVIRONMENT statement to explicitly set the location of the Java executable. Make sure to include the executable file name.

```
ALTER EXTERNAL ENVIRONMENT JAVA
  LOCATION 'java-path';
```

For example:

```
ALTER EXTERNAL ENVIRONMENT JAVA
  LOCATION 'c:\\jdk1.6.0\\jre\\bin\\java.exe';
```

You can query the location of the Java VM that the database server will use by executing the following SQL query:

```
SELECT db_property('JAVAVM');
```

Note that the **START EXTERNAL ENVIRONMENT JAVA** statement is not necessary other than to verify that the database server can start the Java VM. In general, making a Java stored procedure or function call starts the Java VM automatically.

Similarly, the **STOP EXTERNAL ENVIRONMENT JAVA** statement is not necessary to stop an instance of Java since the instance automatically goes away when the all connections to the database have terminated. However, if you are completely done with Java and you want to make it possible to free up some resources, then the **STOP EXTERNAL ENVIRONMENT JAVA** statement decrements the usage count for the Java VM.

Once you have verified that the database server can start the Java VM executable, the next thing to do is to install the necessary Java class code into the database. Do this by using the

INSTALL JAVA statement. For example, you can execute the following statement to install a Java class from a file into the database.

```
INSTALL JAVA
NEW
FROM FILE 'java-class-file';
```

You can also install a Java JAR file into the database.

```
INSTALL JAVA
NEW
JAR 'jar-name'
FROM FILE 'jar-file';
```

Java classes can be installed from a variable, as follows:

```
CREATE VARIABLE JavaClass LONG VARCHAR;
SET JavaClass = xp_read_file('java-class-file')
INSTALL JAVA
NEW
FROM JavaClass;
```

To remove a Java class from the database, use the **REMOVE JAVA** statement, as follows:

```
REMOVE JAVA CLASS java-class
```

To remove a Java JAR from the database, use the **REMOVE JAVA** statement, as follows:

```
REMOVE JAVA JAR 'jar-name'
```

To modify existing Java classes, you can use the **UPDATE** clause of the **INSTALL JAVA** statement, as follows:

```
INSTALL JAVA
UPDATE
FROM FILE 'java-class-file'
```

You can also update existing Java JAR files in the database.

```
INSTALL JAVA
UPDATE
JAR 'jar-name'
FROM FILE 'jar-file';
```

Java classes can be updated from a variable, as follows:

```
CREATE VARIABLE JavaClass LONG VARCHAR;
SET JavaClass = xp_read_file('java-class-file')
INSTALL JAVA
UPDATE
FROM JavaClass;
```

Once the Java class is installed in the database, you can then create stored procedures and functions to interface to the Java methods. The **EXTERNAL NAME** string contains the information needed to call the Java method and to return `OUT` parameters and return values. The `LANGUAGE` attribute of the **EXTERNAL NAME** clause must specify **JAVA**. The format of the **EXTERNAL NAME** clause is:

EXTERNAL NAME '*java-call*' **LANGUAGE JAVA**

java-call :

```
[package-name.]class-name.method-name method-signature
```

method-signature :

```
( [ field-descriptor, ... ] ) return-descriptor
```

field-descriptor and return-descriptor :

- Z
- | B
- | S
- | I
- | J
- | F
- | D
- | C
- | V
- | [descriptor
- | Lclass-name;

A Java method signature is a compact character representation of the types of the parameters and the type of the return value. If the number of parameters is less than the number indicated in the method-signature, then the difference must equal the number specified in **DYNAMIC RESULT SETS**, and each parameter in the method signature that is more than those in the procedure parameter list must have a method signature of **[Ljava/SQL/ResultSet;**.

For Java UDFs, you do not need to set **DYNAMIC RESULT SETS**; **DYNAMIC RESULT SETS** equal to 1 is implied.

The *field-descriptor* and *return-descriptor* have the following meanings:

Field type	Java data type
B	byte
C	char
D	double
F	float

Field type	Java data type
I	int
J	long
L class-name;	an instance of the class class-name. The class name must be fully qualified, and any dot in the name must be replaced by a /. For example, java/lang/String
S	short
V	void
Z	Boolean
[	use one for each dimension of an array

For example,

```
double some_method(
  boolean a,
  int b,
  java.math.BigDecimal c,
  byte [][] d,
  java.sql.ResultSet[] rs ) {
}
```

would have the following signature:

```
'(ZILjava/math/BigDecimal;[[B[Ljava/SQL/ResultSet;)D'
```

The following procedure creates an interface to a Java method. The Java method does not return any value (V).

```
CREATE PROCEDURE insertfix()
EXTERNAL NAME 'JDBCExample.InsertFixed()V'
LANGUAGE JAVA;
```

The following procedure creates an interface to a Java method that has a String ([Ljava/lang/String;) input argument. The Java method does not return any value (V).

```
CREATE PROCEDURE InvoiceMain( IN arg1 CHAR(50) )
EXTERNAL NAME 'Invoice.main([Ljava/lang/String;)V'
LANGUAGE JAVA;
```

The following procedure creates an interface to a Java method `Invoice.init` which takes a string argument (Ljava/lang/String;), a double (D), another string argument (Ljava/lang/String;), and another double (D), and returns no value (V).

```
 CREATE PROCEDURE init( IN arg1 CHAR(50),
                       IN arg2 DOUBLE,
                       IN arg3 CHAR(50),
                       IN arg4 DOUBLE)
EXTERNAL NAME 'Invoice.init(Ljava/lang/String;DLjava/lang/
String;D)V'
LANGUAGE JAVA
```

The following Java example contains the function main which takes a string argument and writes it to the database server messages window. It also contains the function **whare** that returns a Java String.

```
import java.io.*;

public class Hello
{
    public static void main( String[] args )
    {
        System.out.print( "Hello" );
               for ( int i = 0; i  < args.length; i++ )
                   System.out.print( " " + args[i] );
        System.out.println();
    }
    public static String whare()
    {
        return( "I am SQL Anywhere." );
    }
}
```

The Java code above is placed in the file `Hello.java` and compiled using the Java compiler. The class file that results is loaded into the database as follows.

```
INSTALL JAVA
NEW
FROM FILE 'Hello.class';
```

Using Interactive SQL, the stored procedure that will interface to the method `main` in the class `Hello` is created as follows:

```
CREATE PROCEDURE HelloDemo( IN name LONG VARCHAR )
EXTERNAL NAME 'Hello.main([Ljava/lang/String;)V'
LANGUAGE JAVA;
```

Note that the argument to main is described as an array of `java.lang.String`. Using Interactive SQL, test the interface by executing the following SQL statement.

```
CALL HelloDemo('SQL Anywhere');
```

If you check the database server messages window, you will find the message written there. All output to `System.out` is redirected to the server messages window.

Using Interactive SQL, the function that will interface to the method `whare` in the class `Hello` is created as follows:

```
CREATE FUNCTION Whare()
RETURNS LONG VARCHAR
EXTERNAL NAME 'Hello.whoAreYou(V)Ljava/lang/String;'
LANGUAGE JAVA;
```

Note that the function `whare` is described as returning a `java.lang.String`. Using Interactive SQL, test the interface by executing the following SQL statement.

```
SELECT Whare();
```

You should see the response in the Interactive SQL Results window.

In attempting to troubleshoot why a Java external environment did not start, that is, if the application gets a "`main thread not found`" error when a Java call is made, the DBA should check the following:

- If the Java VM is a different bitness than the database server, then ensure that the client libraries with the same bitness as the VM are installed on the database server machine.
- Ensure that the `sajdbc.jar` and `dbjdbc12/libdbjdbc12` shared objects are from the same software build.
- If more than one `sajdbc.jar` are on the database server machine, make sure they are all synchronized to the same software version.
- If the database server machine is very busy, then there is a chance the error is being reported due to a timeout.

See also

- *INSTALL JAVA Statement* on page 358
- *CREATE PROCEDURE Statement (Java UDF)* on page 361
- *CREATE FUNCTION Statement (Java UDF)* on page 363
- *REMOVE Statement* on page 367
- *START JAVA Statement* on page 368
- *STOP JAVA Statement* on page 369

Java External Environment in a Multiplex

Before you can use Java external environment UDFs in a multiplex configuration, install the Java class file or JAR files on each node of the multiplex that requires the UDF.

Use Sybase Control Center or the Interactive SQL **INSTALL JAVA** statement to install the Java class file and JAR.

See also

- *INSTALL JAVA Statement* on page 358

Installing a Class Using Interactive SQL

To make your Java class available within the database, you install the class into the database using the INSTALL JAVA statement from Interactive SQL. You must know the path and file name of the class you want to install.

1. Connect to the database as a user with the MANAGE ANY EXTERNAL OBJECT system privilege.
2. Execute the following statement:

```
INSTALL JAVA NEW
FROM FILE 'path\\ClassName.class';
```

`path` is the directory where the class file is located, and `ClassName.class` is the name of the class file.

The double backslash ensures that the backslash is not treated as an escape character.

For example, to install a class in a file named `Utility.class`, held in the directory `c:\source`, you would execute the following statement:

```
INSTALL JAVA NEW
FROM FILE 'c:\\source\\Utility.class';
```

If you use a relative path, it must be relative to the current working directory of the database server.

Java External Environment Restrictions

Before developing Java UDFs and Java table UDFs, familiarize yourself with the restrictions specific to the Java external environment for UDFs.

- Aggregate Java functions are not supported.
- Query fragments involving Java UDFs are not eligable for DQP or SMP processing.
- You cannot **DROP** tables involved in the current query from within the Java external environment.
- You cannot **ALTER** tables involved in the current query from within the Java external environment.
- `UNSIGNED SMALLINT` datatype is not supported.
- Numeric functions are limited to a precision of 255 or less.
- Only one result set is permitted for Java table UDFs.

See also

- *External Environment Restrictions* on page 331

Java VM Memory Options

Use the **java_vm_options** option to specify any additional command line options that are required to start the Java virtual machine (VM).

Use this syntax:

```
SET OPTION PUBLIC.java_vm_options='java-options';
```

In the following example, you use **java_vm_options** to set the maximum heap size of the Java VM to 512 megabytes:

```
SET OPTION PUBLIC.java_vm_options='-Xmx512m';
```

In the following example, you set the initial heap size of the Java VM to 32 megabytes:

```
SET OPTION PUBLIC.java_vm_options='-Xms32m';
```

SQL Data Type Conversions for Java UDFs

SQL-to-Java and Java-to-SQL data type conversions are carried out according to the JDBC standard. LOB data types `LONG VARCHAR` and `LONG BINARY` are supported for input values but not for return values.

SQL to Java Data Type Conversion

The data type conversions used by the input values of Java Scalar UDFs and Java Table UDFs.

SQL type	Java Type
`BIGINT`	`long`
`BINARY`	`byte[ ]`
`BIT`	`boolean`
`CHAR`	`String`
`DATE`	`java.sql.Date`
`DECIMAL`	`java.math.BigDecimal`
`DOUBLE`	`double`
`IMAGE`	`byte[ ]`
`INTEGER`	`int`
`LONG BINARY`	`byte[ ]` **Note:** Large object data support requires a separately licensed SAP Sybase IQ option.
`LONG VARCHAR`	`String` **Note:** Large object data support requires a separately licensed SAP Sybase IQ option.
`MONEY`	`java.math.BigDecimal`
`NUMERIC`	`java.math.BigDecimal`
`REAL`	`float`
`SMALLINT`	`short`
`SMALLMONEY`	`java.math.BigDecimal`

SQL type	Java Type
TEXT	String
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
TINYINT	byte
UNSIGNED BIGINT	java.math.BigDecimal (with a precision of 20 and scale of 0)
UNSIGNED INT	java long
VARBINARY	byte[ ]
VARCHAR	String

Java to SQL Data Type Conversion

The return-value data types of Java scalar UDFs and Java Table UDFs.

Java Type	SQL Type
String	CHAR
String	VARCHAR
String	TEXT
java.math.BigDecimal	NUMERIC
java.math.BigDecimal	MONEY
java.math.BigDecimal	SMALLMONEY
boolean	BIT
byte	TINYINT
short	SMALLINT
int	INTEGER
long	BIGINT
float	REAL
double	DOUBLE

Java Type	SQL Type
byte[ ]	VARBINARY
byte[ ]	IMAGE
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	DATETIME/TIMESTAMP
java.lang.Double	DOUBLE
java.lang.Float	REAL
java.lang.Integer	INTEGER
java.lang.Long	BIGINT

Creating a Java Scalar UDF

Create and compile a Java class, install the class file onto the server, and create the function definition.

Prerequisites

- You are familiar with Java and can compile a `.java` file. You know where the resulting `.class` file will reside on the file system.
- You are familiar with Interactive SQL. You can connect to the `iqdemo` database from Interactive SQL, and can issue the **START EXTERNAL ENVIRONMENT JAVA** command from Interactive SQL.

Task

Use this task as template when creating your own Java UDFs.

1. Place this Java code in a file named `HelloJavaUDF.java`:

```
public class HelloJavaUDF
{
    public static String helloJava( String name )
    {
        // Simply return Hello and the name passed in.
        return "Hello " + name;
    }
}
```

   This creates the Java class `HelloJavaUDF` with a static method `helloJava`. The method takes a single string argument and returns a string.

2. Compile `HellowJavaUDF.java`:
   `javac <pathtojavafile>/HelloJavaUDF.java`
3. In Interactive SQL, connect to the `iqdemo` database.
4. In Interactive SQL, install the class file onto the server:

Using absolute path	`INSTALL JAVA NEW FROM FILE '<absolutepathtofile>/HelloJavaUDF.class'`
	Example:
	`INSTALL JAVA NEW FROM FILE 'd:/mydirectory/HelloJavaUDF.class'`

Using relative path	`INSTALL JAVA NEW FROM FILE '<pathrelativetocwd>/HelloJavaUDF.class'`
	Example:
	`INSTALL JAVA NEW FROM FILE 'myreldir/HelloJavaUDF.class'`

5. In Interactive SQL, create the function definition.
   Provide the following information:
   - The Java package, class, and method names
   - The Java data types of your function arguments, and their corresponding SQL data types
   - The SQL name assign to the Java UDF

```
CREATE FUNCTION my_helloJava(IN name VARCHAR(249) )
RETURNS VARCHAR(255)
EXTERNAL NAME 'example.HelloJavaUDF.helloJava(Ljava/lang/
String;)Ljava/lang/String;'
LANGUAGE JAVA
```

6. In Interactive SQL, use the Java UDF in a query against the `iqdemo` database:
   `SELECT my_helloJava( GivenName ) FROM Customers WHERE ID < 110`

See also

- *SQL to Java Data Type Conversion* on page 348
- *Java to SQL Data Type Conversion* on page 349

Example: Executing a Java Scalar UDF

Java scalar UDF code example.

1. Create the Java class.

```
public class Sample {
    public static int add( int a, int b ){
```

```
        return a + b;
    }
}
public class Sample {
    public static int add( int a, int b ){
        return new java.lang.Integer(a + b);
    }
}
```

2. Execute the SQL statement to deploy the Java class to the database.

```
INSTALL JAVA NEW FROM FILE 'd:\\java\\samples\\Sample.class'
```

3. Create the SQL function that maps to the Java method "`Sample.add( int, int)`".

```
CREATE FUNCTION sample_add_int (IN a int, IN b int)
RETURNS int
EXTERNAL NAME 'Sample.add(II)I'
LANGUAGE JAVA
```

4. Use the SQL function in a **SELECT** statement.

```
SELECT sample_add_int( ID, ID ) from Customers WHERE ID < 110
```

5. Remove a Java class from the database.

```
REMOVE JAVA CLASS 'Sample'
```

6. Update a java class in the database.

```
INSTALL JAVA UPDATE FROM FILE 'd:\\java\\samples\\Sample.class'
INSTALL JAVA JAR UPDATE FROM FILE 'd:\\java\\samples\\Sample.jar'
```

Creating a Java Scalar UDF Version of the SQL substr Function

Create a Java UDF deployment where the SQL function passes multiple arguments to the Java UDF.

Prerequisites

- You are familiar with Java and can compile a `.java` file. You know where the resulting `.class` file will reside on the file system.
- You are familiar with Interactive SQL. You can connect to the `iqdemo` database from Interactive SQL, and can issue the **START EXTERNAL ENVIRONMENT JAVA** command from Interactive SQL.

Task

1. Place this Java code in a file named `MyJavaSubstr`:

```
public class MyJavaSubstr
{
   public static String my_java_substr( String in, int start, int
length )
    {
    String rc = null;
```

```
    if ( start < 1 )
    {
        start = 1;
    }

    // Convert the SQL start, length to Java start, end.
    start --;  // Java is 0 based, but SQL is one based.
    int endindex = start+length;

    try {
        if ( in != null )
        {
            rc = in.substring( start, endindex );
        }
    } catch ( IndexOutOfBoundsException ex )
    {
        System.out.println("ScalarTestFunctions:
                my_java_substr("+in+","+start+","+length+")
failed");
        System.out.println(ex);
    }
    return rc;
 }
```

2. In Interactive SQL, connect to the `iqdemo` database.
3. In Interactive SQL, install the class file onto the server:
   `INSTALL JAVA NEW FROM FILE '<pathtofile>/MyJavaSubstr.class'`
4. In Interactive SQL, create the function definition:

```
CREATE or REPLACE FUNCTION java_substr(IN a VARCHAR(255), IN b
INT, IN c INT )
  RETURNS VARCHAR(255)
  EXTERNAL NAME
    'example.MyJavaSubstr.my_java_substr(Ljava/lang/
String;II)Ljava/lang/String;'
  LANGUAGE JAVA
```

   Notice the code snippet **Ljava/lang/String;II** indicating parameter types `String, int, int`.
5. In Interactive SQL, use the Java UDF in a query against the `iqdemo` database:
   select GivenName, java_substr(Surname,1,1) from Customers where lcase(java_substr(Surname,1,1)) = 'a';

Creating a Java Table UDF

Create, compile, and install a Java row generator and create the Java table UDF function definition.

Prerequisites

- You are familiar with Java and can compile a `.java` file. You know where the resulting `.class` file will reside on the file system.
- You are familiar with Interactive SQL. You can connect to the `iqdemo` database from Interactive SQL, and can issue the **START EXTERNAL ENVIRONMENT JAVA** command from Interactive SQL.

Task

This example executes a Java row generator (RowGenerator) that takes a single integer input and returns that number of rows in a result set. The result set has two columns: one `INTEGER` and one `VARCHAR`. The RowGenerator relies on two utility classes:

- `example.ResultSetImpl`
- `example.ResultSetMetaDataImpl`

These are simple implementations of the `java.sql.ResultSet` interface and `java.sql.ResultSetMetaData` interface.

1. Place this code in a file named `RowGenerator.java`:

```
package example;

import java.sql.*;

public class RowGenerator {

  public static void rowGenerator( int numRows, ResultSet rset[] ) {
    // Create the meta data needed for the result set
    ResultSetMetaDataImpl rsmd = new ResultSetMetaDataImpl(2);

    //The first column is the SQL type INTEGER.
    rsmd.setColumnType(1, Types.INTEGER);
    rsmd.setColumnName(1,"c1");
    rsmd.setColumnLabel(1,"c1");
    rsmd.setTableName(1,"MyTable");

    // The second column is the SQL type VARCHAR length 255
    rsmd.setColumnType(2, Types.VARCHAR);
    rsmd.setColumnName(2,"c2");
    rsmd.setColumnLabel(2,"c2");
    rsmd.setColumnDisplaySize(2, 255);
    rsmd.setTableName(2,"MyTable");

    // Create result set using the ResultSetMetaData
```

```
    ResultSetImpl rs = null;
    try {
      rs = new ResultSetImpl( (ResultSetMetaData)rsmd );
      rs.beforeFirst();  // Make sure we are at the beginning.
    } catch( Exception e ) {
      System.out.println( "Error: couldn't create result set." );
      System.out.println( e.toString() );
    }

    // Add the rows to the result set and populate them
    for( int i = 0; i < numRows; i++ ) {
      try {
        rs.insertRow();   // insert a new row.
        rs.updateInt( 1, i );   // put the integer value in the first
column
       rs.updateString( 2, ("Str" + i) );  // put the VARCHAR/String value
in the second column
      } catch( Exception e ) {
        System.out.println( "Error: couldn't insert row/data on row " +
i );
        System.out.println( e.toString() );
      }
    }

    try {
       rs.beforeFirst();  // rewind the result set so that the server gets
it from the beginning.
    } catch( Exception e ) {
       System.out.println( e.toString() );
    }
    rset[0] = rs;  // assign the result set to the 1st of the passed in
array.
  }
}
```

2. Compile `RowGenerator.java`, `ResultSetImpl.java`, and `ResultSetMetaData.java`. The Windows directory `%ALLUSERSPROFILE%\samples\java` (`$IQDIR15/samples/java` on UNIX) contains `ResultSetImpl.java` and `ResultSetMetaData.java`.

   ```
   javac <pathtojavafile>/ResultSetMetaDataImpl.java
   ```

   ```
   javac <pathtojavafile>/ResultSetImpl.java
   ```

   ```
   javac <pathtojavafile>/RowGenerator.java
   ```

3. In Interactive SQL, connect to the `iqdemo` database.
4. In Interactive SQL, install the three class files:

   ```
   INSTALL JAVA NEW FROM FILE '<pathtofile>/
   ResultSetMataDataImpl.class'
   ```

   ```
   INSTALL JAVA NEW FROM FILE '<pathtofile>/
   ResultSetImpl.class'
   ```

```
INSTALL JAVA NEW FROM FILE '<pathtofile>/
RowGenerator.class'
```

5. In Interactive SQL, create the Java Table function definition.
   Be ready to provide this information:
   - The Java package, class, and method names
   - The Java data types of your function arguments, and their corresponding SQL data types
   - The SQL name to assign to the Java UDF

```
CREATE or REPLACE PROCEDURE rowgenerator( IN numRows INTEGER )
    RESULT ( c1 INTEGER , c2 VARCHAR(255) )
    EXTERNAL NAME
        'example.RowGenerator.rowGenerator(I[Ljava/sql/
ResultSet;)V'
    LANGUAGE JAVA
```

Note: The RESULT set has two columns; one `INTEGER` and the other `VARCHAR(255)`. The Java prototype has two arguments; one INT (I) and the other an array of `java.sql.ResultSets ([Ljava/sql/ResultSet;)`. The Java prototype shows the function returning `Void(V)`.

6. In Interactive SQL, use the Java table UDF in a query against the `iqdemo` database:

```
SELECT * from rowGenerator(5);
```

   The query returns five rows of two columns.

See also

- *SQL to Java Data Type Conversion* on page 348
- *Java to SQL Data Type Conversion* on page 349

Example: Executing a Java Table UDF

Java Table UDF code example.

1. Java code for a simple `return_rset` method. Compile into `Sample.class`.

```
public class Sample {
    public static void return_rset( ResultSet[] rset1 ) throws
SQLException {
        // Creates new connection back to same db.
        Connection conn = DriverManager.getConnection(
            "jdbc:ianywhere:driver=Sybase
IQ;UID=DBA;PWD=sql" );
        Statement stmt = conn.createStatement();
        ResultSet rset = stmt.executeQuery (
            "SELECT ID " +
            "FROM Customers" );
            rset1[0] = rset;
        }
```

```
    }
}
```

2. SQL statement deploying the Java class to the database:

```
INSTALL JAVA NEW FROM FILE 'd:\\java\\samples\\Sample.class'
```

3. SQL procedure mapping to the Java method `Sample.return_rset( java.sql.ResultSet):`

```
CREATE PROCEDURE sample_result_set()
RESULT ( ID int )
DYNAMIC RESULT SETS 1
EXTERNAL NAME 'Sample.return_rset([Ljava/sql/ResultSet;)V'
LANGUAGE JAVA
```

4. SQL procedure in a **SELECT** statement:

```
SELECT * from sample_result_set( ) where ID < 110
```

Example: Executing a Java Table UDF with Java Result Set Construction

Java Table UDF code example. This example creates a result set.

1. Java code for Java creation of a `return_rset` method, for numeric values:

```
public static void rowgenerator( int a, int b,  ResultSet rset[] )
{
    int result = a + b;
    // Create the meta data needed for the result set
    ResultSetMetaDataImpl rsmd = new ResultSetMetaDataImpl(1);
    rsmd.setColumnType(1, Types.INTEGER);
    rsmd.setColumnName(1,"sum");
    rsmd.setColumnLabel(1,"sum");
    rsmd.setTableName(1,"my_sum");

    // Create result set
    ResultSetImpl rs = null;
    try {
        rs = new ResultSetImpl( (ResultSetMetaData)rsmd );
        rs.beforeFirst();
    } catch( Exception e ) {
       System.out.println( "Error: couldn't create result set." );
        System.out.println( e.toString() );
    }

    // Add the rows to the result set and populate them
    try {
        rs.insertRow();
        rs.updateInt( 1, result );
    } catch( Exception e ) {
       System.out.println( "Error: couldn't insert row/data on row
1" );
        System.out.println( e.toString() );
    }
    try {
        rs.beforeFirst();
    } catch( Exception e ) {
```

```
        System.out.println( e.toString() );
    }
    rset[0] = rs;
}
```

2. Java code for java creation of a `return_rset method`, for non-numerical values:

```
public static void char_result_udf( java.lang.String s, ResultSet
rset[]) {
    // Create the meta data needed for the result set
    ResultSetMetaDataImpl rsmd = new ResultSetMetaDataImpl(1);
    rsmd.setColumnType(1, Types.CHAR);
    if(s.length()==0){
        rsmd.setColumnDisplaySize(1, 1);
    } else {
        rsmd.setColumnDisplaySize(1,s.length() );
    }
    rsmd.setColumnName(1,"c1");
    rsmd.setColumnLabel(1,"c1");
    rsmd.setTableName(1,"my_string");

    // Create result set
    ResultSetImpl rs = null;
    try {
        rs = new ResultSetImpl( (ResultSetMetaData)rsmd );
        rs.beforeFirst();
        //Insert some values into the result set
        rs.insertRow();
        rs.updateString(1, c);
    } catch( Exception e ) {
       System.out.println( "Error: couldn't create result set." );
        System.out.println( e.toString() );
    try {
        rs.beforeFirst();
    } catch( Exception e ) {
       System.out.println( "Error: couldn't insert row/data on row
1" );
        System.out.println( e.toString() );
    }
    rset[0] = rs;
```

Java External Environment SQL Statement Reference

Use these SQL statements when developing Java stored procedures and functions.

INSTALL JAVA Statement

Makes Java classes available for use within a database.

Quick Links:

Go to Parameters on page 359

Go to Examples on page 360

Go to Usage on page 360

Go to Standards on page 361

Go to Permissions on page 361

Syntax

```
INSTALL JAVA [ install-mode ] [ JAR jar-name ]
    FROM source

install-mode – (back to Syntax)
    { NEW | UPDATE }

source – (back to Syntax)
    { FILE file-name | URL url-value }
```

Parameters

(back to top) on page 358

- **NEW –** (default) requires that the referenced Java classes be new classes, rather than updates of currently installed classes. An error occurs if a class with the same name exists in the database and the NEW install mode clause is used
- **UPDATE –** an install mode of specifies that the referenced Java classes may include replacements for Java classes already installed in the given database.
- **JAR –** a character string value of up to 255 bytes that is used to identify the retained JAR in subsequent **INSTALL**, **UPDATE**, and **REMOVE** statements. *jar-name* or text-pointer must designate a JAR file or a column containing a JAR. JAR files typically have extensions of `.jar` or `.zip`.

  Installed JAR and zip files can be compressed or uncompressed. However, JAR files produced by the Sun JDK **jar** utility are not supported. Files produced by other zip utilities are supported.

  If the JAR option is specified, then the JAR is retained as a JAR after the classes that it contains have been installed. That JAR is the associated JAR of each of those classes. The set of JARs installed in a database with the JAR clause are called the retained JARs of the database.

  Retained JARs are referenced in **INSTALL** and **REMOVE** statements. Retained JARs have no effect on other uses of Java-SQL classes. Retained JARs are used by the SQL system for requests by other systems for the class associated with given data. If a requested class has an associated JAR, the SQL system can supply that JAR, rather than the individual class.
- **source –** specifies the location of the Java classes to be installed and must identify either a class file or a JAR file.

  The formats supported for *file-name* include fully qualified file names, such as `'c:\libs\jarname.jar'` and `'/usr/u/libs/jarname.jar'`, and relative file names, which are relative to the current working directory of the database server.

The class definition for each class is loaded by the VM of each connection the first time that class is used. When you **INSTALL** a class, the VM on your connection is implicitly restarted. Therefore, you have immediate access to the new class, whether the **INSTALL** uses an install-mode clause of NEW or UPDATE.

For other connections, the new class is loaded the next time a VM accesses the class for the first time. If the class is already loaded by a VM, that connection does not see the new class until the VM is restarted for that connection (for example, with a **STOP JAVA** and **START JAVA**).

Examples

(back to top) on page 358

- **Example 1 –** install the user-created Java class named “Demo” by providing the file name and location of the class:

```
INSTALL JAVA NEW
FROM FILE 'D:\JavaClass\Demo.class'
```

After installation, the class is referenced using its name. Its original file path location is no longer used. For example, this statement uses the class installed in the previous statement:

```
CREATE VARIABLE d Demo
```

If the Demo class was a member of the package sybase.work, the fully qualified name of the class must be used:

```
CREATE VARIABLE d sybase.work.Demo
```

- **Example 2 –** install all the classes contained in a zip file and associate them within the database with a JAR file name:

```
INSTALL JAVA
JAR 'Widgets'
FROM FILE 'C:\Jars\Widget.zip'
```

The location of the zip file is not retained and classes must be referenced using the fully qualified class name (package name and class name).

Usage

(back to top) on page 358

Only new connections established after installing the class, or that use the class for the first time after installing the class, use the new definition. Once the Java VM loads a class definition, it stays in memory until the connection closes.

If you have been using a Java class or objects based on a class in the current connection, you need to disconnect and reconnect to use the new class definition.

Standards

(back to top) on page 358

- SQL—Vendor extension to ISO/ANSI SQL grammar.
- SAP Sybase Database product—Not supported by Adaptive Server.

Permissions

(back to top) on page 358

- Requires the MANAGE ANY EXTERNAL OBJECT system privilege and a newer version of the compiled class file or JAR file available in a file on disk.
- All installed classes can be referenced in any way by any user.

CREATE PROCEDURE Statement (Java UDF)

Creates an interface to an external Java table UDF.

For **CREATE PROCEDURE** reference information for external procedures, see *CREATE PROCEDURE Statement (External Procedures)*. For **CREATE PROCEDURE** reference information for table UDFs, see *CREATE PROCEDURE Statement (Table UDF)*

Quick Links:

Go to Parameters on page 362

Go to Usage on page 362

Go to Standards on page 362

Go to Permissions on page 363

Syntax

Syntax 1 – For a query referencing at least one SAP Sybase IQ table:

```
CREATE [ OR REPLACE ] PROCEDURE
    [ owner.]procedure-name ( [ parameter, …] )
    [ RESULT (result-column, ...)]
    [ SQL SECURITY { INVOKER | DEFINER } ]
    EXTERNAL NAME 'java-call' [ LANGUAGE java ] }
```

Syntax 2 – For a query referencing catalog store tables only:

```
CREATE [ OR REPLACE ] PROCEDURE
    [ owner.]procedure-name ( [ parameter, …] )
    [ RESULT (result-column, ...)]
    | NO RESULT SET
    [ DYNAMIC RESULT SETS integer-expression ]
    [ SQL SECURITY { INVOKER | DEFINER } ]
    EXTERNAL NAME 'java-call' [ LANGUAGE java ] }


parameter - (back to Syntax 1) or (back to Syntax 2)
```

```
    [ IN parameter_mode parameter-name data-type
    [ DEFAULT expression ]

result-column – (back to Syntax 1) or (back to Syntax 2)
    column-name data-type

java-call – (back to Syntax 1) or (back to Syntax 2)
    '[ package-name.]class-name.method-name method-signature'

java – (back to Syntax 1) or (back to Syntax 2)
    [ ALLOW | DISALLOW SERVER SIDE REQUESTS ]
```

Parameters

(back to top) on page 361

- **java –** DISALLOW is the default. ALLOW indicates that server-side connections are allowed.

  Note: Do not specify ALLOW unless necessary. A setting of ALLOW slows down certain types of SAP Sybase IQ table joins. If you change a procedure definition from ALLOW to DISALLOW, or vice-versa, the change will not be recognized until you make a new connection.

  Do not use UDFs with both ALLOW SERVER SIDE REQUESTS and DISALLOW SERVER SIDE REQUESTS in the same query.

Usage

(back to top) on page 361

If your query references SAP Sybase IQ tables, note that different syntax and parameters apply compared to a query that references only catalog store tables.

Java table UDFs are only supported in the FROM clause.

For Java table functions, exactly one result set is allowed. If the Java table functions are joined with an SAP Sybase IQ table or if a column from an SAP Sybase IQ table is an argument to the Java table function then only one result set is supported.

If the Java table function is the only item in the FROM clause then N number of result sets are allowed.

Standards

(back to top) on page 361

- SQL—ISO/ANSI SQL compliant.
- SAP Sybase Database product—The Transact-SQL **CREATE PROCEDURE** statement is different.

- SQLJ—The syntax extensions for Java result sets are as specified in the proposed SQLJ1 standard.

Permissions

(back to top) on page 361

Unless creating a temporary procedure, a user must have the CREATE PROCEDURE system privilege to create a procedure for themselves. To create UDF procedure for others, a user must specify an owner and have either the CREATE ANY PROCEDURES or CREATE ANY OBJECT system privilege. If a procedure has an external reference, a user must also have the CREATE EXTERNAL REFERENCE system privilege, in addition to the previously mentioned system privileges, regardless of whether or not they are the owner of procedure.

Referencing Temporary Tables Within Procedures

Sharing a temporary table between procedures can cause problems if the table definitions are inconsistent.

For example, suppose you have two procedures `procA` and `procB`, both of which define a temporary table, `temp_table`, and call another procedure called `sharedProc`. Neither `procA` nor `procB` has been called yet, so the temporary table does not yet exist.

Now, suppose that the `procA` definition for `temp_table` is slightly different than the definition in `procB`—while both used the same column names and types, the column order is different.

When you call `procA`, it returns the expected result. However, when you call `procB`, it returns a different result.

This is because when `procA` was called, it created `temp_table`, and then called `sharedProc`. When `sharedProc` was called, the **SELECT** statement inside of it was parsed and validated, and then a parsed representation of the statement is cached so that it can be used again when another **SELECT** statement is executed. The cached version reflects the column ordering from the table definition in `procA`.

Calling `procB` causes the temp_table to be recreated, but with different column ordering. When `procB` calls `sharedProc`, the database server uses the cached representation of the **SELECT** statement. So, the results are different.

You can avoid this from happening by doing one of the following:

- ensure that temporary tables used in this way are defined consistently
- consider using a global temporary table instead

CREATE FUNCTION Statement (Java UDF)

Creates a new external Java table UDF function in the database.

Quick Links:

Go to Parameters on page 364

Go to Examples on page 366

Go to Usage on page 366

Go to Standards on page 366

Go to Permissions on page 366

Syntax

```
CREATE [ OR REPLACE | TEMPORARY ] FUNCTION [ owner.]function-name
    ( [ parameter on page 364, ...] )
    [ SQL SECURITY { INVOKER | DEFINER } ]
    RETURNS data-type
    ON EXCEPTION RESUME
    | [ NOT ] DETERMINISTIC
    { compound-statement | AS tsql-compound-statement on page 364
    | EXTERNAL NAME 'java-call on page 364' LANGUAGE JAVA [ ALLOW | DISALLOW
SERVER SIDE REQUESTS ] environment-name}

parameter – (back to Syntax) on page 364
    IN parameter-name data-type [ DEFAULT expression ]

tsql-compound-statement – (back to Syntax) on page 364
    sql-statement
    sql-statement …

java-call – (back to Syntax) on page 364
    '[ package-name.]class-name.method-name method-signature on page
364'

method-signature – (back to java-call) on page 364
    ( [ field-descriptor on page 364, ...] ) return-descriptor on page 364

field-descriptor and return-descriptor – (back to method-signature) on page 364
    Z | B | S | I | J | F | D | C | V | [ descriptor | L class-name;
```

Parameters

(back to top) on page 363

- **CREATE [OR REPLACE] –** parameter names must conform to the rules for database identifiers. They must have a valid SQL data type and be prefixed by the keyword IN, signifying that the argument is an expression that provides a value to the function.

  The CREATE clause creates a new function, while the OR REPLACE clause replaces an existing function with the same name. When a function is replaced, the definition of the function is changed but the existing permissions are preserved. You cannot use the OR REPLACE clause with temporary functions.

- **TEMPORARY –** the function is visible only by the connection that created it, and that it is automatically dropped when the connection is dropped. Temporary functions can also be explicitly dropped. You cannot perform **ALTER**, **GRANT**, or **REVOKE** operations on them,

and unlike other functions, temporary functions are not recorded in the catalog or transaction log.

Temporary functions execute with the permissions of their creator (current user), and can only be owned by their creator. Therefore, do not specify owner when creating a temporary function. They can be created and dropped when connected to a read-only database.

- **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 INVOKER is specified, more memory is used because annotation must be done for each user that calls the procedure. Also, name resolution is done as the invoker as well. Therefore, take care to qualify all object names (tables, procedures, and so on) with their appropriate owner.

- **data-type –** LONG BINARY and LONG VARCHAR are not permitted as return-value data types.

- **compound-statement –** a set of SQL statements bracketed by **BEGIN** and **END**, and separated by semicolons. See *BEGIN … END Statement*.

- **tsql-compound-statement –** a batch of Transact-SQL statements.

- **[NOT] DETERMINISTIC –** function is re-evaluated each time it is called in a query. The results of functions not specified in this manner may be cached for better performance, and re-used each time the function is called with the same parameters during query evaluation.

  Functions that have side effects, such as modifying the underlying data, should be declared as NOT DETERMINISTIC. For example, a function that generates primary key values and is used in an **INSERT … SELECT** statement should be declared NOT DETERMINISTIC:

```
CREATE FUNCTION keygen( increment INTEGER )
RETURNS INTEGER
NOT DETERMINISTIC
BEGIN
  DECLARE keyval INTEGER;
  UPDATE counter SET x = x + increment;
  SELECT counter.x INTO keyval FROM counter;
  RETURN keyval
END
INSERT INTO new_table
SELECT keygen(1), ...
FROM old_table
```

  Functions may be declared as DETERMINISTIC if they always return the same value for given input parameters. All user-defined functions are treated as deterministic unless they are declared NOT DETERMINISTIC. Deterministic functions return a consistent result for the same parameters and are free of side effects. That is, the database server assumes

that two successive calls to the same function with the same parameters will return the same result without unwanted side-effects on the semantics of the query.

- **LANGUAGE JAVA –** a wrapper around a Java method. For information on calling Java procedures, see *CREATE PROCEDURE Statement*.
- **environment-name –** a wrapper around a Java method.

  The DISALLOW clause is the default. The ALLOW clause indicates that server-side connections are allowed.

  Note: Do not specify the ALLOW clause unless necessary. ALLOW slows down certain types of SAP Sybase IQ table joins. Do not use UDFs with both the ALLOW and DISALLOW SERVER SIDE REQUESTS clauses in the same query.

Examples

(back to top) on page 363

- **Example 1 –** creates an external function written in Java:

```
CREATE FUNCTION dba.encrypt( IN name char(254) )
RETURNS VARCHAR
EXTERNAL NAME
'Scramble.encrypt (Ljava/lang/String;)Ljava/lang/String;'
LANGUAGE JAVA
```

Usage

(back to top) on page 363

When functions are executed, not all parameters need to be specified. If a default value is provided in the **CREATE FUNCTION** statement, missing parameters are assigned the default values. If an argument is not provided by the caller and no default is set, an error is given.

Standards

(back to top) on page 363

- SQL—ISO/ANSI SQL compliant.
- SAP Sybase Database product—Not supported by Adaptive Server.

Permissions

(back to top) on page 363

For function to be owned by self – Requires the CREATE PROCEDURE system privilege

For function to be owned by any user – Requires one of:

- CREATE ANY PROCEDURES system privilege.

- CREATE ANY OBJECT system privilege.

To create a function containing an external reference, regardless of whether or not they are the owner of the function, also requires the CREATE EXTERNAL REFERENCE system privilege.

REMOVE Statement

Removes a class, a package, or a JAR file from a database. Removed classes are no longer available for use as a variable type. Any class, package, or JAR to be removed must already be installed.

Quick Links:

Go to Parameters on page 367

Go to Examples on page 367

Go to Standards on page 368

Go to Permissions on page 368

Syntax

```
REMOVE JAVA  classes_to_remove

classes_to_remove
    { CLASS java_class_name [, java_class_name ]…
    | PACKAGE java_package_name [, java_package_name ]…
    | JAR jar_name [, jar_name ]… [ RETAIN CLASSES ] }
```

Parameters

(back to top) on page 367

- **java_class_name –** he name of one or more Java classes to be removed. Those classes must be installed classes in the current database.
- **java_package_name –** he name of one or more Java packages to be removed. Those packages must be the name of packages in the current database.
- **jar_name –** a character string value of maximum length 255. Each *jar_name* must be equal to the *jar_name* of a retained JAR in the current database. Equality of *jar_name* is determined by the character string comparison rules of the SQL system.
- **RETAIN CLASSES –** the specified JARs are no longer retained in the database, and the retained classes have no associated JAR. If RETAIN CLASSES is specified, this is the only action of the **REMOVE** statement.

Examples

(back to top) on page 367

- **Example 1 –** remove a Java class named “Demo” from the current database:

```
REMOVE JAVA CLASS Demo
```

Standards

(back to top) on page 367

- SQL—Vendor extension to ISO/ANSI SQL grammar.
- SAP Sybase Database product—Not supported by Adaptive Server. A similar feature is available in an Adaptive Server-compatible manner using nested transactions.

Permissions

(back to top) on page 367

Requires one of:

- MANAGE ANY EXTERNAL OBJECT system privilege.
- You own the object.

START JAVA Statement

Loads the Java VM at a convenient time, so that when the user starts to use Java functionality, there is no initial pause while the Java VM is loaded.

Quick Links:

Go to Examples on page 368

Go to Standards on page 368

Go to Permissions on page 369

Syntax

```
START EXTERNAL ENVIRONMENT JAVA
```

Examples

(back to top) on page 368

- **Example 1 –** start the Java VM:

```
START EXTERNAL ENVIRONMENT JAVA
```

Standards

(back to top) on page 368

- SQL—Vendor extension to ISO/ANSI SQL grammar.
- SAP Sybase Database product—Not applicable.

Permissions

(back to top) on page 368

None

STOP JAVA Statement

Releases resources associated with the Java VM to economize on the use of system resources.

Quick Links:

Go to Standards on page 369

Go to Permissions on page 369

Syntax

```
STOP EXTERNAL ENVIRONMENT JAVA
```

Standards

(back to top) on page 369

- SQL—Vendor extension to ISO/ANSI SQL grammar.
- SAP Sybase Database product—Not applicable.

Permissions

(back to top) on page 369

None

PERL External Environment

A Perl stored procedure or function behaves the same as a SQL stored procedure or function except that the code for the procedure or function is written in Perl and the execution of the procedure or function takes place outside the database server (that is, within a Perl executable instance).

It should be noted that there is a separate instance of the Perl executable for each connection that uses Perl stored procedures and functions. This behavior is different from Java stored procedures and functions. For Java, there is one instance of the Java VM for each database rather than one instance per connection. The other major difference between Perl and Java is that Perl stored procedures do not return result sets, whereas Java stored procedures can return result sets.

There are a few prerequisites to using Perl in the database support:

1. Perl must be installed on the database server computer and the database server must be able to locate the Perl executable.

1. The DBD::SQLAnywhere driver must be installed on the database server computer.
2. On Windows, Microsoft Visual Studio must also be installed. This is a prerequisite since it is necessary for installing the DBD::SQLAnywhere driver.

In addition to the above prerequisites, the database administrator must also install the Perl External Environment module.

To install the external environment module (Windows):

- Run the following commands from the `SDK\PerlEnv` subdirectory:

```
perl Makefile.PL
nmake
nmake install
```

To install the external environment module (UNIX):

- Run the following commands from the `sdk/perlenv` subdirectory:

```
perl Makefile.PL
make
make install
```

Once the Perl external environment module has been built and installed, the Perl in the database support can be used.

To use Perl in the database, make sure that the database server is able to locate and start the Perl executable. Verify that this can be done by executing:

```
START EXTERNAL ENVIRONMENT PERL;
```

If the database server fails to start Perl, then the problem probably occurs because the database server is not able to locate the Perl executable. In this case, you should execute an **ALTER EXTERNAL ENVIRONMENT** statement to explicitly set the location of the Perl executable. Make sure to include the executable file name.

```
ALTER EXTERNAL ENVIRONMENT PERL
  LOCATION 'perl-path';
```

For example:

```
ALTER EXTERNAL ENVIRONMENT PERL
  LOCATION 'c:\\Perl\\bin\\perl.exe';
```

Note that the **START EXTERNAL ENVIRONMENT PERL** statement is not necessary other than to verify that the database server can start Perl. In general, making a Perl stored procedure or function call starts Perl automatically.

Similarly, the **STOP EXTERNAL ENVIRONMENT PERL** statement is not necessary to stop an instance of Perl since the instance automatically goes away when the connection terminates. However, if you are completely done with Perl and you want to free up some resources, then the **STOP EXTERNAL ENVIRONMENT PERL** statement releases the Perl instance for your connection.

Once you have verified that the database server can start the Perl executable, the next thing to do is to install the necessary Perl code into the database. Do this by using the **INSTALL** statement. For example, you can execute the following statement to install a Perl script from a file into the database.

```
INSTALL EXTERNAL OBJECT 'perl-script'
  NEW
  FROM FILE 'perl-file'
  ENVIRONMENT PERL;
```

Perl code also can be built and installed from an expression, as follows:

```
INSTALL EXTERNAL OBJECT 'perl-script'
  NEW
  FROM VALUE 'perl-statements'
  ENVIRONMENT PERL;
```

Perl code also can be built and installed from a variable, as follows:

```
CREATE VARIABLE PerlVariable LONG VARCHAR;
SET PerlVariable = 'perl-statements';
INSTALL EXTERNAL OBJECT 'perl-script'
  NEW
  FROM VALUE PerlVariable
  ENVIRONMENT PERL;
```

To remove Perl code from the database, use the **REMOVE** statement, as follows:

```
REMOVE EXTERNAL OBJECT 'perl-script'
```

To modify existing Perl code, you can use the **UPDATE** clause of the **INSTALL EXTERNAL OBJECT** statement, as follows:

```
INSTALL EXTERNAL OBJECT 'perl-script'
  UPDATE
  FROM FILE 'perl-file'
  ENVIRONMENT PERL
 INSTALL EXTERNAL OBJECT 'perl-script'
  UPDATE
  FROM VALUE 'perl-statements'
  ENVIRONMENT PERL
 SET PerlVariable = 'perl-statements';
INSTALL EXTERNAL OBJECT 'perl-script'
  UPDATE
  FROM VALUE PerlVariable
  ENVIRONMENT PERL
```

Once the Perl code is installed in the database, you can then create the necessary Perl stored procedures and functions. When creating Perl stored procedures and functions, the LANGUAGE is always PERL and the EXTERNAL NAME string contains the information needed to call the Perl subroutines and to return OUT parameters and return values. The following global variables are available to the Perl code on each call:

- **$sa_perl_return –** This is used to set the return value for a function call.

- **$sa_perl_argN –** where N is a positive integer [0 .. n]. This is used for passing the SQL arguments down to the Perl code. For example, `$sa_perl_arg0` refers to argument 0, `$sa_perl_arg1` refers to argument 1, and so on.
- **$sa_perl_default_connection –** This is used for making server-side Perl calls.
- **$sa_output_handle –** This is used for sending output from the Perl code to the database server messages window.

A Perl stored procedure can be created with any set of data types for input and output arguments, and for the return value. However, all non-binary data types are mapped to strings when making the Perl call while binary data is mapped to an array of numbers. A simple Perl example follows:

```
INSTALL EXTERNAL OBJECT 'SimplePerlExample'
  NEW
  FROM VALUE 'sub SimplePerlSub{
    return( ($_[0] * 1000) +
            ($_[1] * 100) +
            ($_[2] * 10) +
            $_[3] );
  }'
  ENVIRONMENT PERL;

CREATE FUNCTION SimplePerlDemo(
    IN thousands INT,
    IN hundreds INT,
    IN tens INT,
    IN ones INT)
  RETURNS INT
  EXTERNAL NAME '<file=SimplePerlExample>
    $sa_perl_return = SimplePerlSub(
      $sa_perl_arg0,
      $sa_perl_arg1,
      $sa_perl_arg2,
      $sa_perl_arg3)'
  LANGUAGE PERL;

// The number 1234 should appear
SELECT SimplePerlDemo(1,2,3,4);
```

The following Perl example takes a string and writes it to the database server messages window:

```
INSTALL EXTERNAL OBJECT 'PerlConsoleExample'
  NEW
  FROM VALUE 'sub WriteToServerConsole { print $sa_output_handle
$_[0]; }'
  ENVIRONMENT PERL;

CREATE PROCEDURE PerlWriteToConsole( IN str LONG VARCHAR)
  EXTERNAL NAME '<file=PerlConsoleExample>
    WriteToServerConsole( $sa_perl_arg0 )'
  LANGUAGE PERL;
```

```
// 'Hello world' should appear in the database server messages window
CALL PerlWriteToConsole( 'Hello world' );
```

To use server-side Perl, the Perl code must use the *$sa_perl_default_connection* variable. The following example creates a table and then calls a Perl stored procedure to populate the table:

```
CREATE TABLE perlTab(c1 int, c2 char(128));

INSTALL EXTERNAL OBJECT 'ServerSidePerlExample'
  NEW
  FROM VALUE 'sub ServerSidePerlSub
    { $sa_perl_default_connection->do(
        "INSERT INTO perlTab SELECT table_id, table_name FROM
SYS.SYSTAB" );
      $sa_perl_default_connection->do(
        "COMMIT" );
    }'
  ENVIRONMENT PERL;

CREATE PROCEDURE PerlPopulateTable()
  EXTERNAL NAME '<file=ServerSidePerlExample> ServerSidePerlSub()'
  LANGUAGE PERL;

CALL PerlPopulateTable();

// The following should return 2 identical rows
SELECT count(*) FROM perlTab
UNION ALL
SELECT count(*) FROM SYS.SYSTAB;
```

PHP External Environment

A PHP stored procedure or function behaves the same as a SQL stored procedure or function except that the code for the procedure or function is written in PHP and the execution of the procedure or function takes place outside the database server (that is, within a PHP executable instance).

There is a separate instance of the PHP executable for each connection that uses PHP stored procedures and functions. This behavior is quite different from Java stored procedures and functions. For Java, there is one instance of the Java VM for each database rather than one instance per connection. The other major difference between PHP and Java is that PHP stored procedures do not return result sets, whereas Java stored procedures can return result sets. PHP only returns an object of type LONG VARCHAR, which is the output of the PHP script.

There are two prerequisites to using PHP in the database support:

1. A copy of PHP must be installed on the database server computer and the database server must be able to locate the PHP executable.
2. The PHP extension must be installed on the database server computer.

In addition to the above two prerequisites, the database administrator must also install the PHP External Environment module. Prebuilt modules for several versions of PHP are included. To

install prebuilt modules, copy the appropriate driver module to your PHP extensions directory (which can be found in `php.ini`). On UNIX, you can also use a symbolic link.

To install the external environment module (Windows):

1. Locate the `php.ini` file for your PHP installation, and open it in a text editor. Locate the line that specifies the location of the `extension_dir` directory. If `extension_dir` is not set to any specific directory, it is a good idea to set it to point to an isolated directory for better system security.
2. Copy the desired external environment PHP module from the installation directory to your PHP extensions directory. Change the x.y to reflect the version you have selected.

   ```
   copy "%SQLANY12%\Bin32\php-5.x.y_sqlanywhere_extenv12.dll"
     php-dir\ext
   ```
3. Add the following line to the Dynamic Extensions section of the `php.ini` file to load the external environment PHP module automatically. Change the x.y to reflect the version you have selected.

   ```
   extension=php-5.x.y_sqlanywhere_extenv12.dll
   ```

   Save and close `php.ini`.
4. Make sure that you have also installed the PHP driver from the installation directory into your PHP extensions directory. This file name follows the pattern `php-5.x.y_sqlanywhere.dll` where x and y are the version numbers. It should match the version numbers of the file that you copied in step 2.

To install the external environment module (UNIX):

1. Locate the `php.ini` file for your PHP installation, and open it in a text editor. Locate the line that specifies the location of the `extension_dir` directory. If `extension_dir` is not set to any specific directory, it is a good idea to set it to point to an isolated directory for better system security.
2. Copy the desired external environment PHP module from the installation directory to your PHP installation directory. Change the x.y to reflect the version you have selected.

   ```
   cp $SQLANY12/bin32/php-5.x.y_sqlanywhere_extenv12.so
     php-dir/ext
   ```
3. Add the following line to the Dynamic Extensions section of the `php.ini` file to load the external environment PHP module automatically. Change the x.y to reflect the version you have selected.

   ```
   extension=php-5.x.y_sqlanywhere_extenv12.so
   ```

   Save and close `php.ini`.
4. Make sure that you have also installed the PHP driver from the installation directory into your PHP extensions directory. This file name follows the pattern `php-5.x.y_sqlanywhere.so` where x and y are the version numbers. It should match the version numbers of the file that you copied in step 2.

To use PHP in the database, the database server must be able to locate and start the PHP executable. You can verify if the database server is able to locate and start the PHP executable by executing the following statement:

```
START EXTERNAL ENVIRONMENT PHP;
```

If you see a message that states that 'external executable' could not be found, then the problem is that the database server is not able to locate the PHP executable. In this case, you should execute an **ALTER EXTERNAL ENVIRONMENT** statement to explicitly set the location of the PHP executable including the executable name or you should ensure that the PATH environment variable includes the directory containing the PHP executable.

```
ALTER EXTERNAL ENVIRONMENT PHP
  LOCATION 'php-path';
```

For example:

```
ALTER EXTERNAL ENVIRONMENT PHP
  LOCATION 'c:\\php\\php-5.2.6-win32\\php.exe';
```

To restore the default setting, execute the following statement:

```
ALTER EXTERNAL ENVIRONMENT PHP
  LOCATION 'php';
```

The **START EXTERNAL ENVIRONMENT PHP** statement is not necessary other than to verify that the database server can start PHP. In general, making a PHP stored procedure or function call starts PHP automatically.

Similarly, the **STOP EXTERNAL ENVIRONMENT PHP** statement is not necessary to stop an instance of PHP since the instance automatically goes away when the connection terminates. However, if you are completely done with PHP and you want to free up some resources, then the **STOP EXTERNAL ENVIRONMENT PHP** statement releases the PHP instance for your connection.

Once you have verified that the database server can start the PHP executable, the next thing to do is to install the necessary PHP code into the database. Do this by using the **INSTALL** statement. For example, you can execute the following statement to install a particular PHP script into the database.

```
INSTALL EXTERNAL OBJECT 'php-script'
  NEW
  FROM FILE 'php-file'
  ENVIRONMENT PHP;
```

PHP code can also be built and installed from an expression as follows:

```
INSTALL EXTERNAL OBJECT 'php-script'
  NEW
  FROM VALUE 'php-statements'
  ENVIRONMENT PHP;
```

PHP code can also be built and installed from a variable as follows:

```
CREATE VARIABLE PHPVariable LONG VARCHAR;
SET PHPVariable = 'php-statements';
```

```
INSTALL EXTERNAL OBJECT 'php-script'
  NEW
  FROM VALUE PHPVariable
  ENVIRONMENT PHP;
```

To remove PHP code from the database, use the **REMOVE** statement as follows:

```
REMOVE EXTERNAL OBJECT 'php-script';
```

To modify existing PHP code, you can use the **UPDATE** clause of the **INSTALL** statement as follows:

```
INSTALL EXTERNAL OBJECT 'php-script'
  UPDATE
  FROM FILE 'php-file'
  ENVIRONMENT PHP;
 INSTALL EXTERNAL OBJECT 'php-script'
  UPDATE
  FROM VALUE 'php-statements'
  ENVIRONMENT PHP;
 SET PHPVariable = 'php-statements';
INSTALL EXTERNAL OBJECT 'php-script'
  UPDATE
  FROM VALUE PHPVariable
  ENVIRONMENT PHP;
```

Once the PHP code is installed in the database, you can then go ahead and create the necessary PHP stored procedures and functions. When creating PHP stored procedures and functions, the LANGUAGE is always PHP and the EXTERNAL NAME string contains the information needed to call the PHP subroutines and for returning OUT parameters.

The arguments are passed to the PHP script in the $argv array, similar to the way PHP would take arguments from the command line (that is, $argv[1] is the first argument). To set an output parameter, assign it to the appropriate $argv element. The return value is always the output from the script (as a LONG VARCHAR).

A PHP stored procedure can be created with any set of data types for input or output arguments. However, the parameters are converted to and from a boolean, integer, double, or string for use inside the PHP script. The return value is always an object of type LONG VARCHAR. A simple PHP example follows:

```
INSTALL EXTERNAL OBJECT 'SimplePHPExample'
  NEW
  FROM VALUE '<?php function SimplePHPFunction(
    $arg1, $arg2, $arg3, $arg4 )
    { return ($arg1 * 1000) +
      ($arg2 * 100) +
      ($arg3 * 10) +
      $arg4;
    } ?>'
  ENVIRONMENT PHP;

CREATE FUNCTION SimplePHPDemo(
  IN thousands INT,
  IN hundreds INT,
```

```
  IN tens INT,
  IN ones INT)
RETURNS LONG VARCHAR
EXTERNAL NAME '<file=SimplePHPExample> print SimplePHPFunction(
    $argv[1], $argv[2], $argv[3], $argv[4]);'
LANGUAGE PHP;

// The number 1234 should appear
SELECT SimplePHPDemo(1,2,3,4);
```

For PHP, the **EXTERNAL NAME** string is specified in a single line of SQL.

To use server-side PHP, the PHP code can use the default database connection. To get a handle to the database connection, call sasql_pconnect with an empty string argument (" or ""). The empty string argument tells the PHP driver to return the current external environment connection rather than opening a new one. The following example creates a table and then calls a PHP stored procedure to populate the table:

```
CREATE TABLE phpTab(c1 int, c2 char(128));

INSTALL EXTERNAL OBJECT 'ServerSidePHPExample'
  NEW
  FROM VALUE '<?php function ServerSidePHPSub() {
    $conn = sasql_pconnect( '''' );
    sasql_query( $conn,
    "INSERT INTO phpTab
       SELECT table_id, table_name FROM SYS.SYSTAB" );
    sasql_commit( $conn );
  } ?>'
  ENVIRONMENT PHP;

CREATE PROCEDURE PHPPopulateTable()
EXTERNAL NAME '<file=ServerSidePHPExample> ServerSidePHPSub()'
LANGUAGE PHP;

CALL PHPPopulateTable();

// The following should return 2 identical rows
SELECT count(*) FROM phpTab
UNION ALL
SELECT count(*) FROM SYS.SYSTAB;
```

For PHP, the EXTERNAL NAME string is specified in a single line of SQL. In the above example, note that the single quotes are doubled-up because of the way quotes are parsed in SQL. If the PHP source code was in a file, then the single quotes would not be doubled-up.

To return an error back to the database server, throw a PHP exception. The following example shows how to do this.

```
CREATE TABLE phpTab(c1 int, c2 char(128));

INSTALL EXTERNAL OBJECT 'ServerSidePHPExample'
  NEW
  FROM VALUE '<?php function ServerSidePHPSub() {
    $conn = sasql_pconnect( '''' );
```

```
    if( !sasql_query( $conn,
      "INSERT INTO phpTabNoExist
         SELECT table_id, table_name FROM SYS.SYSTAB" )
    ) throw new Exception(
      sasql_error( $conn ),
      sasql_errorcode( $conn )
    );
    sasql_commit( $conn );
  } ?>'
  ENVIRONMENT PHP;

CREATE PROCEDURE PHPPopulateTable()
  EXTERNAL NAME
    '<file=ServerSidePHPExample> ServerSidePHPSub()'
  LANGUAGE PHP;

CALL PHPPopulateTable();
```

The above example should terminate with error SQLE_UNHANDLED_EXTENV_EXCEPTION indicating that the table phpTabNoExist could not be found.

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X