Virtual-Table Interface

Programmer's Manual

Version 9.2
September 1999
Part No. 000-6219

Published by Informix® Press

Informix Corporation
4100 Bohannon Drive
Menlo Park, CA 94025-1032

Documentation Team: Diana Chase, Kathy Eckardt, Abby Knott

Table of Contents

Chapter 4 Purpose-Function Reference

Chapter 5 Descriptor Function Reference

Chapter 6 SQL Statements for Access Methods

Index

Introduction

In This Introduction

This Introduction provides an overview of the information in this manual and describes the conventions it uses.

About This Manual

This manual explains how to use the Virtual-Table Interface (VTI) to create a primary access method so that users have a single SQL interface to Informix tables and to data that does not conform to Informix tables.

Types of Users

This manual is written for experienced C programmers who develop primary access methods, as follows:

- Informix partners who integrate data that does not conform to the built-in primary access method for relational tables or an industry-standard gateway API
- Informix engineers who support Informix customers, partners, and third-party developers

Before you develop an access method, you should be familiar with creating user-defined routines and programming with the DataBlade API.

Software Dependencies

This manual assumes that you are using Informix Dynamic Server 2000, Version 9.2, as your database server.

Assumptions About Your Locale

Informix products can support many languages, cultures, and code sets. All culture-specific information is brought together in a single environment, called a Global Language Support (GLS) locale.

This manual assumes that you use the U.S. 8859-1 English locale as the default locale. The default is **en_us.8859-1** (ISO 8859-1) on UNIX platforms or **en_us.1252** (Microsoft 1252) for Windows NT environments. This locale supports U.S. English format conventions for dates, times, and currency, and also supports the ISO 8859-1 or Microsoft 1252 code set, which includes the ASCII code set plus many 8-bit characters such as é, è, and ñ.

If you plan to use nondefault characters in your data or your SQL identifiers, or if you want to conform to the nondefault collation rules of character data, you need to specify the appropriate nondefault locale.

For instructions on how to specify a nondefault locale, additional syntax, and other considerations related to GLS locales, see the *Informix Guide to GLS Functionality*.

Demonstration Databases

The DB-Access utility, which is provided with your Informix database server products, includes one or more of the following demonstration databases:

- The **stores_demo** database illustrates a relational schema with information about a fictitious wholesale sporting-goods distributor. Many examples in Informix manuals are based on the **stores_demo** database.
- The **superstores_demo** database illustrates an object-relational schema. The **superstores_demo** database contains examples of extended data types, type and table inheritance, and user-defined routines.

For information about how to create and populate the demonstration databases, see the *DB-Access User's Manual*. For descriptions of the databases and their contents, see the *Informix Guide to SQL: Reference*.

The scripts that you use to install the demonstration databases reside in the **$INFORMIXDIR/bin** directory on UNIX platforms and in the **%INFORMIXDIR%\bin** directory in Windows environments.

New Features

For a comprehensive list of new database server features, see the release notes. This section lists new features relevant to this manual. These features fall into the following areas:

- Extensibility enhancements
- Performance improvements

Extensibility Enhancements

This manual describes the following extensibility enhancements to Version 9.2 of Dynamic Server:

- Ability to execute UDRs with the DataBlade API FastPath feature, including the new **mi_funcdesc_by_typeid()** function
- Ability to provide virtual-table information to the **oncheck** utility
- Ability to determine the level of distribution statistics desired: high, low, or medium

Performance Improvements

This manual describes the following performance improvements to Version 9.2 of Dynamic Server:

- Parallel processing of access-method functions
- Ability to scan multiple rows into memory

Documentation Conventions

This section describes the conventions that this manual uses. These conventions make it easier to gather information from this and other volumes in the documentation set.

The following conventions are discussed:

- Typographical conventions
- Icon conventions
- Syntax conventions
- Command-line conventions
- Sample-code conventions

Typographical Conventions

This manual uses the following conventions to introduce new terms, illustrate screen displays, describe command syntax, and so forth.

Convention	Meaning
KEYWORD	All primary elements in a programming language statement (keywords) appear in uppercase letters in a serif font.
italics ***italics*** *`italics`*	Within text, new terms and emphasized words appear in italics. Within syntax and code examples, variable values that you are to specify appear in italics.
boldface ***boldface***	Names of program entities (such as classes, events, and tables), environment variables, file and pathnames, and interface elements (such as icons, menu items, and buttons) appear in boldface.
`monospace` *`monospace`*	Information that the product displays and information that you enter appear in a monospace typeface.

(1 of 2)

Convention	Meaning
KEYSTROKE	Keys that you are to press appear in uppercase letters in a sans serif font.
♦	This symbol indicates the end of one or more product- or platform-specific paragraphs.
→	This symbol indicates a menu item. For example, “Choose **Tools→Options**” means choose the **Options** item from the **Tools** menu.

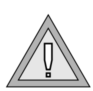
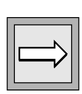
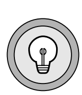
(2 of 2)

Icon Conventions

Throughout the documentation, you will find text that is identified by several different types of icons. This section describes these icons.

Comment Icons

Comment icons identify three types of information, as the following table describes. This information always appears in italics.

Icon	Label	Description
	***Warning:***	Identifies paragraphs that contain vital instructions, cautions, or critical information
	***Important:***	Identifies paragraphs that contain significant information about the feature or operation that is being described
	***Tip:***	Identifies paragraphs that offer additional details or shortcuts for the functionality that is being described

Feature, Product, and Platform Icons

Feature, product, and platform icons identify paragraphs that contain feature-specific, product-specific, or platform-specific information.

Icon	Description
GLS	Identifies information that relates to the Informix Global Language Support (GLS) feature
UNIX	Identifies information that is specific to UNIX platforms
WIN NT	Identifies information that is specific to the Windows NT environment

These icons can apply to an entire section or to one or more paragraphs within a section. If an icon appears next to a section heading, the information that applies to the indicated feature, product, or platform ends at the next heading at the same or higher level. A ♦ symbol indicates the end of feature-, product-, or platform-specific information that appears within one or more paragraphs within a section.

Compliance Icons

Compliance icons indicate paragraphs that provide guidelines for complying with a standard.

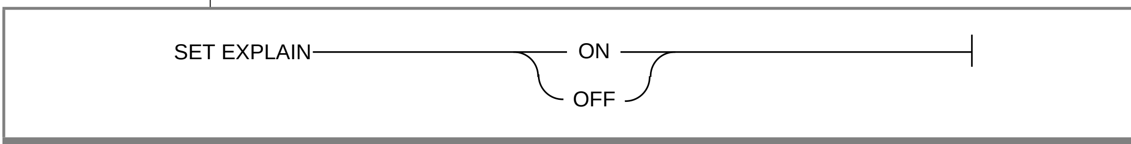
Icon	Description
+	Identifies information that is an Informix extension to ANSI SQL-92 entry-level standard SQL

This icon can apply to an entire section or to one or more paragraphs within a section. If an icon appears next to a section heading, the information that applies to the indicated feature, product, or platform ends at the next heading at the same or higher level. A ♦ symbol indicates the end of feature-, product-, or platform-specific information that appears within one or more paragraphs within a section.

Syntax Conventions

This section describes conventions for syntax diagrams. Each diagram displays the sequences of required and optional keywords, terms, and symbols that are valid in a given statement or segment, as Figure 1 shows.

Figure 1
Example of a Simple Syntax Diagram



Each syntax diagram begins at the upper-left corner and ends at the upper-right corner with a vertical terminator. Between these points, any path that does not stop or reverse direction describes a possible form of the statement.

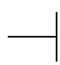
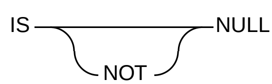
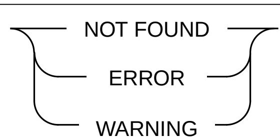
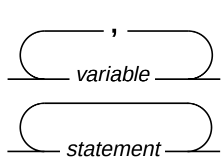
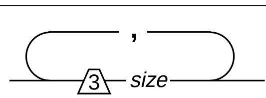
Syntax elements in a path represent terms, keywords, symbols, and segments that can appear in your statement. The path always approaches elements from the left and continues to the right, except in the case of separators in loops. For separators in loops, the path approaches counterclockwise. Unless otherwise noted, at least one blank character separates syntax elements.

Elements That Can Appear on the Path

You might encounter one or more of the following elements on a path.

Element	Description
KEYWORD	A word in UPPERCASE letters is a keyword. You must spell the word exactly as shown; however, you can use either uppercase or lowercase letters.
(. , ; @ + * - /)	Punctuation and other nonalphanumeric characters are literal symbols that you must enter exactly as shown.
' '	Single quotes are literal symbols that you must enter as shown.

(1 of 2)

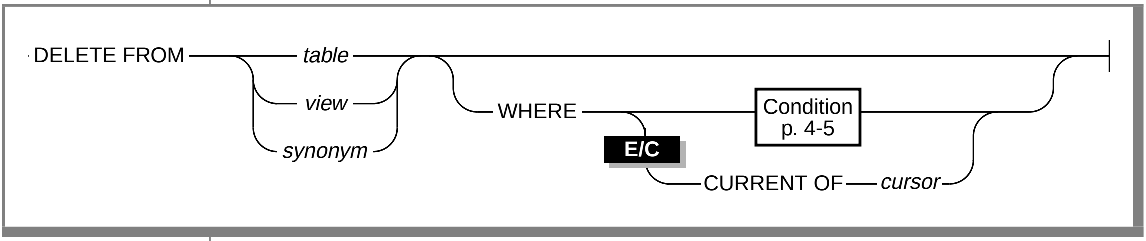
Element	Description
variable	A word in *italics* represents a value that you must supply. A table immediately following the diagram explains the value.
ADD Clause p. 3-288 ADD Clause	A reference in a box represents a subdiagram. Imagine that the subdiagram is spliced into the main diagram at this point. When a page number is not specified, the subdiagram appears on the same page.
Back to ADD Clause p. 1-14	A reference in a box in the upper-right corner of a subdiagram refers to the next higher-level diagram of which this subdiagram is a member.
— ALL -	A shaded option is the default action.
	Syntax within a pair of arrows is a subdiagram.
	The vertical line terminates the syntax diagram.
	A branch below the main path indicates an optional path. (Any term on the main path is required, unless a branch can circumvent it.)
	A set of multiple branches indicates that a choice among more than two different paths is available.
	A loop indicates a path that you can repeat. Punctuation along the top of the loop indicates the separator symbol for list items. If no symbol appears, a blank space is the separator.
	A gate (3) on a path indicates that you can only use that path the indicated number of times, even if it is part of a larger loop. You can specify *size* no more than three times within this statement segment.

(2 of 2)

How to Read a Syntax Diagram

Figure 2 shows a syntax diagram that uses most of the path elements that the previous table lists.

Figure 2
Example of a Syntax Diagram



To use this diagram to construct a statement, start at the top left with the keyword DELETE FROM. Then follow the diagram to the right, proceeding through the options that you want.

Figure 2 illustrates the following steps:

1. Type `DELETE FROM`.
2. You can delete a table, view, or synonym:
   - Type the table name, view name, or synonym, as you desire.
   - You can type `WHERE` to limit the rows to delete.
   - If you type `WHERE` and you are using DB-Access or the SQL Editor, you must include the Condition clause to specify a condition to delete. To find the syntax for specifying a condition, go to the "Condition" segment on the specified page.
   - If you are using Informix ESQL, you can include either the Condition clause to delete a specific condition or the CURRENT OF *cursor* clause to delete a row from the table.
3. Follow the diagram to the terminator.

   Your DELETE statement is complete.

Command-Line Conventions

This section defines and illustrates the format of commands that are available in Informix products. These commands have their own conventions, which might include alternative forms of a command, required and optional parts of the command, and so forth.

You might encounter one or more of the following elements on a command-line path.

Element	Description
command	This required element is usually the product name or other short word that invokes the product or calls the compiler or preprocessor script for a compiled Informix product. It might appear alone or precede one or more options. You must spell a command exactly as shown and use lowercase letters.
variable	A word in italics represents a value that you must supply, such as a database, file, or program name. A table following the diagram explains the value.
-flag	A flag is usually an abbreviation for a function, menu, or option name, or for a compiler or preprocessor argument. You must enter a flag exactly as shown, including the preceding hyphen.
.ext	A filename extension, such as **.sql** or **.cob**, might follow a variable that represents a filename. Type this extension exactly as shown, immediately after the name of the file. The extension might be optional in certain products.
(. , ; + * - /)	Punctuation and mathematical notations are literal symbols that you must enter exactly as shown.
' '	Single quotes are literal symbols that you must enter as shown.

How to Read a Command Line

The following command uses some of the elements that are listed in the previous table:

```
setenv INFORMIXC pathname
```

The elements in the diagram are case sensitive.

The previous example illustrates the following steps:

1. Type `setenv`.
2. Type `INFORMIXC`.
3. Supply a pathname.
4. Press RETURN to execute the command.

Sample-Code Conventions

Examples of the following code occur throughout this manual:

- SQL examples
- Access-method-module examples

Tip: *Ellipsis points in a code example indicate that more code would be added in a full application, but it is not necessary to show it to describe the concept being discussed.*

SQL Code Conventions

Except where noted, examples of SQL code are not specific to any single Informix application development tool. If only SQL statements are listed in the example, they are not delimited by semicolons. For instance, you might see the code in the following example:

```
CONNECT TO stores_demo
...

DELETE FROM customer
    WHERE customer_num = 121
...

COMMIT WORK
DISCONNECT CURRENT
```

Because each product requires different delimiters, the SQL code examples do show the delimiters. For example, in DB-Access, semicolons delimit multiple statements. In an SQL API, each statement starts with EXEC SQL and ends in a semicolon (or other appropriate delimiter).

For detailed information about the syntax rules for SQL statements that apply to a particular application development tool, SQL API, or DataBlade API routine, see the manual for your product.

Tip: *An access method does not typically run SQL code. The examples in this book show the code that a user or application creates and the access method interprets.*

Access-Method Code Conventions

This manual includes sample code for VTI modules. These samples:

- follow C-language coding conventions for indentation
- use C ANSI format for parameters in function declarations

Additional Documentation

For additional information, you might want to refer to the following types of documentation:

- On-line manuals
- Printed manuals
- Error message documentation
- Documentation notes, release notes, and machine notes
- Related reading

On-Line Manuals

An Answers OnLine CD that contains Informix manuals in electronic format is provided with your Informix products. You can install the documentation or access it directly from the CD. For information about how to install, read, and print on-line manuals, see the installation insert that accompanies Answers OnLine.

Informix on-line manuals are also available on the following Web site:

```
www.informix.com/answers
```

Printed Manuals

To order printed manuals, call 1-800-331-1763 or send email to moreinfo@informix.com. Please provide the following information when you place your order:

- The documentation that you need
- The quantity that you need
- Your name, address, and telephone number

Error Message Documentation

Informix software products provide ASCII files that contain all of the Informix error messages and their corrective actions.

UNIX

To read error messages and corrective actions on UNIX, use one of the following utilities.

Utility	Description
finderr	Displays error messages on line
rofferr	Formats error messages for printing

♦

WIN NT

To read error messages and corrective actions in Windows environments, use the **Informix Find Error** utility. To display this utility, choose **Start→Programs→Informix** from the Task Bar. ♦

Instructions for using the preceding utilities are available in Answers OnLine. Answers OnLine also provides a listing of error messages and corrective actions in HTML format.

Documentation Notes, Release Notes, Machine Notes

In addition to printed documentation, the following sections describe the on-line files that supplement the information in this manual. Please examine these files before you begin using your database server. They contain vital information about application and performance issues.

UNIX

On UNIX platforms, the following on-line files appear in the **$INFORMIXDIR/release/en_us/0333** directory.

On-Line File	Purpose
VTIDOC_9.2	The documentation notes file for your version of this manual describes topics that are not covered in the manual or that were modified since publication.
SERVERS_9.2	The release notes file describes feature differences from earlier versions of Informix products and how these differences might affect current products. This file also contains information about any known problems and their workarounds.
IDS_9.2	The machine notes file describes any special actions that you must take to configure and use Informix products on your computer. Machine notes are named for the product described.

♦

WIN NT

The following items appear in the **Informix** folder. To display this folder, choose **Start→Programs→Informix** from the Task Bar.

Program Group Item	Description
Documentation Notes	This item includes additions or corrections to manuals, along with information about features that might not be covered in the manuals or that have been modified since publication.
Release Notes	This item describes feature differences from earlier versions of Informix products and how these differences might affect current products. This file also contains information about any known problems and their workarounds.

Machine notes do not apply to Windows environments. ♦

Related Reading

The following publications provide additional information about the topics that this manual discusses. For a list of publications that provide an introduction to database servers and operating-system platforms, refer to your *Getting Started* manual.

- For information about qualifying access methods as Informix DataBlade modules, refer to the *DataBlade Developers Standards and Coding Guidelines* (Informix Press, January 1997).
- For articles, white papers, and examples, see *Dynamic Server with Universal Data Option: Best Practices (*Angela Sanchez, Editor; Prentice Hall PTR, Informix Press Books, 1999).

Compliance with Industry Standards

The American National Standards Institute (ANSI) has established a set of industry standards for SQL. Informix SQL-based products are fully compliant with SQL-92 Entry Level (published as ANSI X3.135-1992), which is identical to ISO 9075:1992. In addition, many features of Informix database servers comply with the SQL-92 Intermediate and Full Level and X/Open SQL CAE (common applications environment) standards.

Informix Welcomes Your Comments

Let us know what you like or dislike about our manuals. To help us with future versions of our manuals, we want to know about any corrections or clarifications that you would find useful. Include the following information:

- The name and version of the manual that you are using
- Any comments that you have about the manual
- Your name, address, and phone number

Send electronic mail to us at the following address:

doc@informix.com

The **doc** alias is reserved exclusively for reporting errors and omissions in our documentation.

We appreciate your suggestions.

Chapter

1

What Is a Virtual-Table Access Method?

In This Chapter

This chapter includes the following sections:

Warning: *This manual is specifically for customers and DataBlade partners developing alternative access methods for Informix Dynamic Server 2000. Informix continues to enhance and modify the interface described in this manual. Customers and partners who use this interface should work with an Informix representative to ensure that they continue to receive the latest information and that they are prepared to change their access method.*

What Is an Access Method?

An access method consists of software routines that open files, retrieve data into memory, and write data to permanent storage such as a disk.

A *primary* access method provides a relational-table interface for direct read and write access. A primary access method reads directly from and writes directly to source data. It provides a means to combine data from multiple sources in a common relational format that the database server, users, and application software can use.

A *secondary* access method provides a means to index data for alternate or accelerated access. An *index* consists of entries, each of which contains one or more key values and a pointer to the row in a table that contains the corresponding value or values. The secondary access method maintains the index to coincide with inserts, deletes, and updates to the primary data.

Dynamic Server recognizes both built-in and user-defined access methods. Although an index typically points to table rows, an index can point to values within smart large objects or to records from external data sources.

The database server provides the following built-in access methods:

- The built-in primary access method scans, retrieves, and alters rows in Informix relational tables.

  By default, tables that you create with the CREATE TABLE statement use the built-in primary access method.

- The built-in secondary access method is a generic B-tree index.

  By default, indexes that you create with the CREATE INDEX statement use this built-in secondary access method. For more information about the built-in B-tree index, refer to the *Informix Guide to SQL: Syntax*.

Tip: *Informix also provides the R-tree secondary access method. For more information, see the "Informix R-Tree Index User's Guide."*

Why Do You Provide Additional Access Methods?

This manual explains how to create primary access methods that provide SQL access to nonrelational and other data that does not conform to built-in access methods. For example, a user-defined access method might retrieve data from an external location or manipulate specific data within a smart large object.

An access method can make any data appear to the end user as rows from an internal relational table. With the help of an access method, the end user can apply SQL statements to retrieve nonstandard data. Because the access method creates rows from the data that it accesses, external or smart-large-object data can join with other data from an internal database.

This manual refers to the table that the access method presents to the end user as a *virtual table.*

Access to Storage Spaces

The database server allows a user-defined access-method access to either of the following types of storage spaces:

- A smart large object, which resides in an sbspace

  The database server can log, back up, and recover smart large objects.

- An external table, which resides in an extspace

  An extspace refers to a storage location that the Informix database server does not manage. For example, an extspace might refer to a path and filename that the operating system manages or another database that a different database manager controls.

  The database server does not provide transaction, backup, or recovery services for data that resides in an extspace.

For more information about how to choose the storage spaces that the user-defined access method will support, refer to "Managing Storage Spaces" on page 3-12.

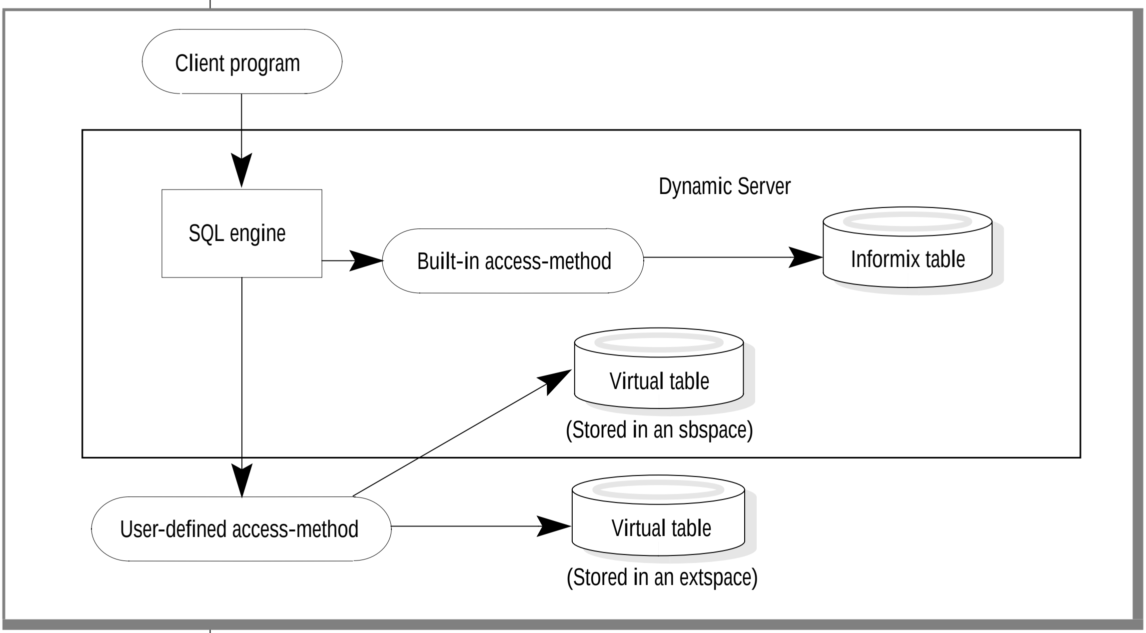
Seamless Use of SQL

With the aid of a user-defined primary access method, you can use a SELECT statement to access any of the following data as though the data resides in an Informix relational table:

- Legacy data such as flat-file records
- Mixed binary and text data such as a word-processor document
- Multiple-vendor data across the enterprise

The end user can use SQL to access both Informix data and *virtual* table data. A *virtual* table requires a user-defined access method to make the data in the table accessible to Dynamic Server. In Figure 1-1, a single application processes Informix data as well as virtual data in an external location and smart-large-object storage.

Figure 1-1
Using a Primary Access Method



What Components Define an Access Method?

When you add an access method to Dynamic Server, you add, or *register*, a collection of C user-defined routines (UDRs) in the system catalog. These UDRs take advantage of an Informix application programming interface, the Virtual-Table Interface (VTI).

What Does Informix Provide?

Informix provides the following application program interface support for the development of user-defined access methods:

- Virtual-Table Interface
- DataBlade API
- SQL extensions specific to the access method
- Additional Informix API libraries, as needed

Virtual-Table Interface

The Virtual-Table Interface (VTI) consists of the following items:

- Purpose functions
- Descriptors
- Accessor functions

Purpose Functions

The database server calls user-defined *purpose functions* to pass SQL statement specifications and state information to the access method. The following special traits distinguish purpose functions from other user-defined routines (UDRs):

- A purpose function conforms to a predefined syntax.

  The purpose-function syntax describes the parameters and valid return values but the access method developer chooses a unique function name.
- The database server calls a purpose function as the entry point into the access method for a specific access-method task.
- Each SQL statement results in specific purpose-function calls.
- The **sysams** system catalog table contains the unique function name for each purpose function.
- The database server substitutes calls to purpose functions for calls to built-in access-method modules.

For example, when the database server encounters a CREATE TABLE statement, it invokes an access-method function with the following required the parameter and return value types:

```
mi_integer am_create(MI_AM_TABLE_DESC *)
```

To determine which UDR provides the entry point for table creation in this example, the database server looks for the function identifier in the **am_create** column of the **sysams** system catalog. The database server then calls that UDR and passes by reference an MI_AM_TABLE_DESC structure that contains data-definition information.

The access-method developer provides the program code inside the purpose function to create the new table structure. When the purpose function exits, the access-method returns the value that VTI provides to indicate success or failure.

For information about the access-method developer's contribution to purpose functions, refer to "What Do You Provide?" on page 1-13. For the syntax and usage of each purpose function, refer to Chapter 4, "Purpose-Function Reference."

Descriptors

Descriptors are predefined opaque data types that the database server creates to exchange information with a Datablade module or an access method. The VTI provides several descriptors in addition to those that the DataBlade API provides. An access-method descriptor contains the specifications from an SQL statement or **oncheck** request as well as relevant information from the system catalog.

The database server passes descriptors by reference as arguments to purpose functions.The following list highlights only a few access-method descriptors to illustrate the type of information that the database server passes to an access method. For detailed information about all the VTI descriptors, refer to the "Descriptors" on page 5-6.

Descriptor Name and Structure	Database Server Entries in the Descriptor
table descriptor MI_AM_TABLE_DESC	The database server puts CREATE TABLE specifications in the table descriptor, including the following items: ■ Identification by table name, owner, table identifier, storage space, and current fragment ■ Structural details, such as the number of fragments in the whole table, column names, and data types ■ Optional user-supplied parameters ■ Constraints such as read / write mode
scan descriptor MI_AM_SCAN_DESC	The database server puts SELECT statement specifications in the scan descriptor, including the following items: ■ Columns to project ■ Lock type and isolation level ■ Pointers to the table descriptor and the qualification descriptor

(1 of 2)

Descriptor Name and Structure	Database Server Entries in the Descriptor

qualification descriptor
MI_AM_QUAL_DESC
	In the qualification descriptor, the database server describes the functions and Boolean operators that a WHERE clause specifies.
	A qualification *function* tests the value in a column against a constant or value that an application supplies. The following examples test the value in the `price` column against the constant value `80`:
	`WHERE lessthan(price,80)`
	`WHERE price < 80`
	The qualification descriptor for a function identifies the following items:
	■ Function name
	■ Arguments that the WHERE clause passes to the function
	■ Negation (NOT) operator, if any
	A complex qualification combines the results of two previous qualifications with an AND or OR operation, as the following example shows:
	`WHERE price < 80 AND cost > 60`
	A complex qualification descriptor contains each Boolean AND or OR operator from the WHERE clause.
	For examples, refer to “Interpreting the Qualification Descriptor” on page 3-21.

(2 of 2)

Descriptors reserve areas where the access method stores information. An access method can also allocate user-data memory of a specified duration and store a pointer to the user data in a descriptor, as the following list shows.

Descriptor Name and Structure	Access Method Entries in the Descriptor
table descriptor MI_AM_TABLE_DESC	To share state information among multiple purpose functions, the access method can allocate user-data memory with a PER_STATEMENT duration and store a pointer to the user data in the table descriptor. PER_STATEMENT memory lasts for the duration of an SQL statement, for as long as the accessed data source is open. For example, an access method might execute DataBlade API functions that open smart large objects or files and store the values, or handles, that the functions return in PER_STATEMENT memory.
scan descriptor MI_AM_SCAN_DESC	To maintain state information during a scan, an access method can allocate user-data memory with a PER_COMMAND duration and store a pointer to the user data in the scan descriptor. For example, as it scans a table, the access method can maintain a pointer in PER_COMMAND memory to the address of the current record.
qualification descriptor MI_AM_QUAL_DESC	As it processes each qualification against a single row, the access method can set the following items in the qualification descriptor: ■ A host-variable value for a function with an OUT argument ■ The MI_VALUE_TRUE or MI_VALUE_FALSE to indicate the result that each function or Boolean operator returns ■ An indicator that forces the database server to reoptimize between scans for a join or subquery

To allocate memory for a specific duration, the access method specifies a duration keyword. For example, the following command allocates PER_STATEMENT memory:

```
my_data = (my_data_t *) mi_dalloc(sizeof(my_data_t), PER_STATEMENT)
```

Accessor Functions

Unlike purpose functions, the VTI supplies the full code for each accessor function. Accessor functions obtain and set specific information in descriptors. For example, the access method can perform the following actions:

- Call the **mi_tab_name()** accessor function to obtain the name of the table from the table descriptor.
- Store state information, such as a file handle or LO handle, in shared memory, and then call the **mi_tab_setuserdata()** to place the pointer to the handle in the table descriptor so that subsequent purpose functions can retrieve the handle.

For the syntax and usage of each accessor function, refer to "Accessor Functions" on page 5-16.

DataBlade API

The DataBlade application programming interface includes functions and opaque data structures that enable an application to implement C UDRs. The access method uses functions from the DataBlade API that allocate shared memory, execute user-defined routines, handle exceptions, construct rows, and report whether a transaction commits or rolls back.

The remainder of this manual contains information about the specific DataBlade API functions that an access method calls. For more information about the DataBlade API, refer to the *DataBlade API Programmer's Manual*.

SQL Extensions

Informix extension to ANSI SQL-92 entry-level standard SQL includes statements and keywords that specifically refer to user-defined access methods.

Registering the Access Method in a Database

The CREATE PRIMARY ACCESS_METHOD statement registers a user-defined access method. When you register an access method, the database server puts information in the system catalog that identifies the purpose functions and other properties of the access method.

ALTER ACCESS_METHOD changes the registration information in the system catalog, and DROP ACCESS_METHOD removes the access-method entries from the system catalog.

For more information about the SQL statements that register, alter, or drop the access method, refer to Chapter 6, "SQL Statements for Access Methods."

Specifying an Access Method for a Virtual Table

The user needs a way to specify a virtual table in an SQL statement.

To create a virtual table with the CREATE TABLE statement, a user specifies the USING keyword followed by the access-method name and optionally with additional access-method-specific keywords.

With the IN clause, the user can place the virtual table in an extspace or sbspace.

For more information about the SQL extensions specific to virtual tables, refer to "Supporting Data Definition Statements" on page 3-11 and "Supporting Data Retrieval, Manipulation, and Return" on page 3-37.

API Libraries

GLS

Informix provides Global Language Support with the Informix GLS functions, which access Informix locales and support multibyte character sets. Use this API to allow the access method to interpret international alphabets. For more information, refer to the *Informix GLS Programmer's Manual*. ♦

For information about the complete set of APIs for Dynamic Server, refer to the *Getting Started* manual.

What Do You Provide?

As the developer of a user-defined access method, you design, write, and test the following components:

- Purpose functions
- Additional UDRs that the purpose functions call
- User messages and documentation

Purpose Functions

A *purpose function* is a UDR that can interpret the user-defined structure of a virtual table. You implement purpose functions in C to build, connect, populate, query, and update tables. The interface requires a specific purpose-function syntax for each of several specific tasks.

Tip: *To discuss the function call for a given task, this manual uses a column name from the* ***sysams*** *system catalog table as the generic purpose-function name. For example, this manual refers to the UDR that builds a new table as* ***am_create****. The* ***am_create*** *column in* ***sysams*** *contains the registered UDR name that the database server calls to perform the work of* ***am_create****.*

Figure 1-2 shows the task that each purpose function performs and the reasons that the database server invokes that purpose function. In Figure 1-2, the list groups the purpose functions as follows:

- Data definition
- File or smart-large-object access
- Data changes
- Scans
- Structure and data-integrity verification

Figure 1-2
Purpose Functions

Generic Name	Description	Invoking Statement or Command
am_create	Creates a new virtual table and registers it in the system catalog	CREATE TABLE, ALTER FRAGMENT
am_drop	Drops an existing virtual table and removes it from the system catalog	DROP TABLE
am_open	Opens the file or smart large object that contains the virtual table Typically, **am_open** allocates memory to store handles and pointers.	CREATE TABLE, DROP TABLE, DROP DATABASE, ALTER FRAGMENT, DELETE, UPDATE, INSERT, SELECT
am_close	Closes the file or smart large object that contains the virtual table and releases any remaining memory that the access method allocated	CREATE TABLE, ALTER FRAGMENT, DELETE, UPDATE, INSERT, SELECT
am_insert	Inserts a new row into a virtual table	ALTER FRAGMENT, INSERT
am_delete	Deletes an existing row from a virtual table	DELETE, ALTER FRAGMENT
am_update	Modifies an existing row in a virtual table	UPDATE
am_stats	Builds statistics information about the virtual table	UPDATE STATISTICS
am_scancost	Calculates the cost of a scan for qualified data in a virtual table	SELECT, INSERT, UPDATE, DELETE WHERE...
am_beginscan	Initializes pointers to a virtual table, and possibly parses the query statement, prior to a scan	SELECT, INSERT, UPDATE, DELETE WHERE...
am_getnext	Scans for the next row that satisfies a query	SELECT, INSERT, UPDATE, DELETE WHERE..., ALTER FRAGMENT
am_rescan	Scans for the next item from a previous scan to complete a join or subquery	SELECT, INSERT, UPDATE, DELETE WHERE...

(1 of 2)

Generic Name	Description	Invoking Statement or Command
am_endscan	Releases resources that am_beginscan allocates	SELECT, INSERT, UPDATE, DELETE WHERE...
am_getbyid	Uses a specific physical address to fetch a row	SELECT using an index, INSERT, UPDATE, DELETE
am_check	Performs a check on the physical integrity of a virtual table	**oncheck** utility

(2 of 2)

For more information about purpose functions, refer to the following chapters:

- Chapter 2, "Developing an Access Method," helps you decide which purpose functions to provide and explains how to register them in a database.
- Chapter 3, "Design Decisions," describes some of the functionality that you program and provides examples of program code.
- Chapter 4, "Purpose-Function Reference," specifies syntax and usage.

User-Defined Routines and Header Files

The database server calls a purpose function to initiate a specific task. Often, the purpose function calls other modules in the access-method library. For example, the scanning, insert, and update purpose functions might all call the same UDR to check for valid data type.

A complete access method provides modules that convert data formats, detect and recover from errors, commit and rollback transactions, and perform other tasks. You provide the additional UDRs and header files that complete the access method.

User Messages and Documentation

You provide messages and a user guide that help end users apply the access method in SQL statements and interpret the results of the **oncheck** utility.

A user-defined access method alters some of the functionality that the database server manuals describe. The documentation that you provide details storage-area constraints, deviations from the Informix implementation of SQL, configuration options, data types, error messages, backup procedures, and extended features that the Informix documentation library does not describe.

For samples of user documentation that you must provide, refer to "Supplying Error Messages and a User Guide" on page 3-41.

How Does an Access Method Work?

To apply a user-defined access method, the database server must locate the access-method components, particularly the purpose functions.

How Does the Database Server Locate Purpose Functions?

The SQL statements that register a purpose function and an access method create records in the system catalog, which the database server consults to locate a purpose function.

As the access-method developer, you write the purpose functions and register them with the CREATE FUNCTION statement. When you register a purpose function, the database server puts a description of it in the **sysprocedures** system catalog table.

For example, assume you write a **get_next_record()** function that performs the tasks of the **am_getnext** purpose function. Assume that as user **informix**, you register the **get_next_record()** function. Depending on the operating system, you use one of the following statements to register the function:

UNIX

```
CREATE FUNCTION get_next_record(pointer,pointer,pointer)
RETURNS int
WITH (NOT VARIANT)
EXTERNAL NAME "$INFORMIXDIR/extend/am_lib.bld(get_next_record)"
LANGUAGE C
```

♦

WIN NT

```
CREATE FUNCTION get_next_record (pointer,pointer,pointer)
RETURNS int
WITH (NOT VARIANT)
EXTERNAL NAME "%INFORMIXDIR%\extend\am_lib.bld(get_next_record)"
LANGUAGE C
```

♦

The **get_next_record()** declaration has three generic pointer arguments to conform with the prototype of the **am_getnext** purpose function. For a detailed explanation of the arguments and return value, refer to the description of **am_getnext** on page 4-27.

As a result of the CREATE FUNCTION statement, the **sysprocedures** system catalog table includes an entry with values that are similar to the example in Figure 1-3.

Figure 1-3
Partial sysprocedures Entry

Column Name	Value
procname	`get_next_record`
owner	`informix`
procid	`163`
numargs	`3`
externalname	`$INFORMIXDIR/extend/am_lib.bld(get_next_record)` (for UNIX)
langid	`1` (Identifies C in the **syslanguages** system catalog table)
paramtypes	`pointer,pointer,pointer`
variant	`f` (Indicates false or nonvariant)

You then register the access method with a CREATE PRIMARY ACCESS_METHOD statement to inform the database server what function from **sysprocedures** to execute for each purpose.

The following example registers the **super_access** access method and identifies **get_next_record()** as the **am_getnext** purpose function:

```
CREATE PRIMARY ACCESS_METHOD super_access
(AM_GETNEXT = get_next_record)
```

The **super_access** access method provides only one purpose function. If user **informix** executes the CREATE PRIMARY ACCESS_METHOD, the **sysams** system catalog table has an entry similar to Figure 1-4.

Figure 1-4
Partial sysams Entry

Column Name	Value
am_name	`super_access`
am_owner	`informix`
am_id	`100` (Unique identifier that the database server assigns)
am_type	`P`
am_sptype	`A`
am_getnext	`163` (Matches the **procid** value in the **sysprocedures** system catalog table entry for **get_next_record()**)

Which Purpose Functions Does the Database Server Use?

When an SQL statement or **oncheck** command specifies a virtual table, the database server executes one or more access-method purpose functions. A single SQL command might involve a combination of the following purposes:

- Open a connection, file, or smart large object
- Create a table
- Scan and select data
- Insert, delete, or update data
- Drop a table
- Close the connection, file, or smart large object

A single **oncheck** request requires at least the following actions:

- Open a connection, file, or smart large object
- Check the integrity of a table
- Close the connection, file, or smart large object

For information about which purpose functions the database server executes for specific commands, refer to "Purpose-Function Flow" on page 4-3.

The example in Figure 1-4 on page 1-20 specifies only the **am_getnext** purpose for the **super_access** access method. A SELECT statement on a virtual table that uses **super_access** initiates the following database server actions:

1. Get the function name for **am_getnext** that the **super_access** entry in **sysams** specifies; in this case **get_next_record()**.
2. Get the external file name of the executable from the **get_next_record()** entry in **sysprocedures** specifies.

   The CREATE FUNCTION statement on page 1-18 assigns the executable file as follows.

Operating System	Name of External Executable File
UNIX	$INFORMIXDIR/extend/am_lib.bld(get_next_record)
Windows NT	%INFORMIXDIR%\extend\am_lib.bld(get_next_record)

3. Allocate memory for the descriptors that the database server passes by reference through **get_next_record()** to the access method.
4. Execute the **am_getnext** purpose function, **get_next_record()**.

What Other Functions Does a Purpose Function Call?

A query might proceed as follows for the **super_access** access method, which has only an **am_getnext** purpose function:

1. The access method **am_getnext** purpose function, **get_next_record()**, uses DataBlade API functions to the initiate callback functions for error handling.
2. The database server prepares a table descriptor to identify the table that the query specifies, a scan descriptor to describe the query projection, and a qualification descriptor to describe the query selection criteria.

3. The database server passes a pointer to the scan descriptor through **get_next_record()** to the access method. The scan descriptor, in turn, points to the table descriptor and qualification descriptor in shared memory.
4. The access method **get_next_record()** function takes the following actions:
   a. Calls VTI accessor functions to retrieve the table description and then calls DataBlade API functions to open that table
   b. Calls accessor functions to retrieve the query projection and selection criteria from the scan and qualification descriptors
   c. Calls the DataBlade API function (usually **mi_dalloc()**) to allocate memory for a user-data structure to hold the current virtual-table data
   d. Begins its scan
5. Each time that the access method retrieves a qualifying record, it calls a DataBlade API function, **mi_row_create()**, to create an Informix-formatted row from the raw data.
6. The database server executes **get_next_record()** to continue scanning until **get_next_record()** returns MI_NO_MORE_RESULTS to indicate to the database server that the access method has returned every qualifying row.
7. The access method calls a DataBlade API function to close the table and release any allocated memory.
8. The database server reports the results to the user or application that initiated the query.

The steps in the preceding example illustrate the interaction between the database server, the access method, and the DataBlade API.

What Might a More Sophisticated Access Method Do?

The **super_access** access method in the example has no purpose functions to open or close files or smart large objects. The **get_next_record()** function must open and close any data as well as keep an indicator that notifies **get_next_record()** to open only at the start of the scan and close only after it completes the scan.

The incomplete **super_access** access method example does not create a virtual table because the example does not include an **am_create** purpose function. or add, delete, or update rows.

To enable INSERT, DELETE, and UPDATE statements to execute, the access method must provide registered UDRs for the **am_open**, **am_close**, **am_insert**, **am_delete**, and **am_update** purpose functions.

GLS

For the access method to support nondefault character sets, the purpose functions must also call the appropriate Informix GLS routines. For more information, refer to the *Informix GLS Programmer's Manual*. ♦

Chapter

Developing an Access Method

2

In This Chapter

This chapter describes the steps that you take to implement a user-defined access method with the Virtual-Table Interface (VTI).

To provide an access method

1. Choose the optional features that the access method supports.
2. Program and compile the C header files and purpose functions as well as the modules that the purpose functions call.
3. Execute the CREATE FUNCTION statement to register each purpose function in the **sysprocedures** system catalog table.
4. Execute the CREATE PRIMARY ACCESS_METHOD statement to register the user-defined access method in the **sysams** system catalog table.
5. Test the access method in an end-user environment.

The rest of this chapter describes the preceding steps in more detail.

Choosing Features

The VTI provides many optional features. Choose the features that you need to fulfill the access-method specifications.

The following optional features support data definition:

- Data in extspaces, sbspaces, or both
- Fragmentation
- User-configured keywords

Support for the following optional features can contribute to access-method performance:

- Clustered data
- Fetch by rowid for indexed tables
- Parallel-function execution
- More than one row returned per scan-function call
- Complex qualifications

For more information about any of these optional features, refer to Chapter 3, "Design Decisions."

Writing Purpose Functions

The VTI specifies the parameters and return values for a limited set of UDRs, called *purpose functions*, that correspond to one or more SQL statements. To process most SQL statements, the database server attempts to invoke a sequence of task-specific purpose functions. You choose the tasks and SQL statements that the access method supports and then write the appropriate purpose functions for those tasks. For more information about the specific purpose functions that the database server executes for specific statements, refer to "Purpose-Function Flow" on page 4-3.

Figure 2-1 shows purpose-function prototypes for access-method tasks and one or more corresponding SQL statement. Figure 2-1 includes the purpose function prototype that the database server calls to process the **oncheck** utility.

Figure 2-1
Statements and Their Purpose Functions

Invoking Statement or Command	Purpose-Function Prototype
All If you do not supply **am_open** and **am_close**, open and close the data source in **am_getnext**.	am_open(MI_AM_TABLE_DESC *) am_close(MI_AM_TABLE_DESC *)
CREATE TABLE	am_create(MI_AM_TABLE_DESC *)
DROP TABLE	am_drop(MI_AM_TABLE_DESC *)
INSERT	am_insert(MI_AM_TABLE_DESC *, MI_ROW *, mi_integer *)
DELETE	am_delete(MI_AM_TABLE_DESC *, mi_integer *)
SELECT INSERT, UPDATE, DELETE WHERE...	am_scancost(MI_AM_TABLE_DESC *, MI_AM_QUAL_DESC *) am_beginscan(MI_AM_SCAN_DESC *) am_getnext(MI_AM_SCAN_DESC *, MI_ROW **, mi_integer) am_endscan(MI_AM_SCAN_DESC *)
SELECT with join	am_rescan(MI_AM_SCAN_DESC *)
SELECT using an index	am_getbyid(MI_AM_TABLE_DESC *, MI_ROW **, mi_integer)
UPDATE	am_update(MI_AM_TABLE_DESC *, MI_ROW *, mi_integer *)
UPDATE STATISTICS	am_stats(MI_AM_TABLE_DESC *, MI_AM_TSTATS_DESC *)
oncheck utility	am_check(MI_AM_TABLE_DESC *, mi_integer *)

Important: *Do not use the purpose label (**am_open**, **am_create**, **am_getnext**) as the actual name of a user-defined purpose function. Do not use the prefix **vti_**, as in **vti_getnext**. Assign unique names, such as **image_open**, **docfile_open**, and **getnext_record**. To prevent potential name-space collision, follow the instructions for registering and using an object prefix in the "DataBlade Developers Kit User's Guide."*

When the database server calls a purpose function, it passes the appropriate parameters for the current database server activity. Most parameters reference the opaque *descriptor* data structures. The database server creates and passes descriptors to describe the state of the table and the current SQL statement or **oncheck** command. For an overview of descriptors, refer to "Descriptors" on page 1-9. For detailed information, refer to "Descriptors" on page 5-6.

As you write the purpose functions, adhere to the syntax provided for each in "Purpose-Function Syntax" on page 4-12.

At a minimum, you must supply one purpose function, the **am_getnext** purpose function, to scan data. To determine which other purpose functions to provide, decide if the access method should:

- open and initialize files or smart large objects, as well as close them again at the end of processing.
- create new tables.
- write and delete data.
- run the **oncheck** utility.
- influence query optimization.

Warning: *The database server issues an error if a user or application tries to execute an SQL statement but the access method does not include a purpose function to support that statement.*

The following sections name the functions that the database server calls for the specific purposes in the previous list. The access-method library might contain a separate function for each of several purpose-function prototypes or supply only an **am_getnext** purpose function as the entry point for all the essential access-method processing. For a detailed description of each purpose function, refer to Chapter 4, "Purpose-Function Reference."

Starting and Ending Processing

Most SQL statements cause the database server to execute the function that you register for **am_open**. To fulfill the **am_open** tasks, the function can open a connection, store file or smart-large-object handles, allocate user memory, and possibly set the number of entries that **am_getnext** returns.

At the end of processing, the database server calls the function that you register for **am_close**. This close of access-method processing reverses the actions of the **am_open** purpose function. It deallocates memory, possibly writes smart-large-object data to disk.

Creating and Dropping Database Objects

In response to a CREATE TABLE statement, the database server executes the function that you register for **am_create**. If the database server does not find a function name associated with **am_create**, it simply updates the appropriate system catalog tables to reflect the attributes of the table that CREATE TABLE specifies.

If you supply a function for **am_create**, consider the necessity of also providing a function to drop a table that the access method creates. The database server executes the function that you register for **am_drop** in response to a DROP TABLE or DROP DATABASE statement. If you do not provide a function to drop a virtual table, the database server simply deletes any system catalog information that describes the dropped object.

Optimizing Queries

To provide the optimum performance with an access method, perform the following actions:

- Provide **am_scancost** and **am_stats** purpose functions.
- Split scan processing into **am_beginscan**, **am_getnext**, **am_rescan**, and **am_endscan** purpose functions.
- Return more than one row from **am_getnext** or **am_rescan**, as "Buffering Multiple Results" on page 3-35 describes.
- Register purpose functions as parallelizable, as "Executing in Parallel" on page 3-33 describes.

Providing Optimizer Information

In response to a SELECT statement, the query optimizer compares the cost of alternative query paths. To determine the cost for the access method to scan the virtual table that it manages, the optimizer relies on two sources of information:

- The cost of a scan that the access method performs on its virtual table

  The **am_scancost** purpose function calculates and returns this cost to the optimizer. If you do not provide an **am_scancost** purpose function, the optimizer cannot analyze those query paths that involve a scan of data by the access method.
- The distribution statistics that the **am_stats** purpose function sets

  This purpose function takes the place of the type of distribution analysis that the database server performs for an UPDATE STATISTICS statement.

Splitting a Scan

The way in which you split a scan influences the ability of the access method to optimize performance during queries. You can choose to provide separate functions for each of the following purpose-function prototypes:

- **am_beginscan**

  In this purpose function, identify the columns to project and the function to execute for each WHERE clause qualification. The database server calls the function for **am_beginscan** only once per query.
- **am_getnext**

  In this purpose function, scan through the table to find a qualifying entry and return it. The database server calls this function as often as necessary to exhaust the qualified entries in the table.

- **am_rescan**

  In this purpose function, reuse the information from **am_beginscan** and possibly some data from **am_getnext** to perform any subsequent scans for a join or subquery.

- **am_endscan**

  In this purpose function, deallocate any memory that **am_beginscan** allocates. The database server calls this function only once.

If you provide only an **am_getnext** purpose function, that one purpose function (and any UDRs that it calls) analyzes the query, scans, rescans, and performs end-of-query cleanup.

Tip: *When the database server can scan an index to query a table, it does not need to call any of the functions in the previous list. Instead, the database server can pass the physical address (rowid) of each qualified row to an* ***am_getbyid*** *purpose function. The function for* ***am_getbyid*** *calls the appropriate DataBlade API or external routines to read or write disk data. It does not scan the table to find rows.*

If you supply a function for **am_getbyid**, you must also set the **am_rowids** purpose flag when you register the access method.

Inserting, Deleting, and Updating Data

The database server calls an **am_insert** purpose function in response to SQL statements such as INSERT or ALTER FRAGMENT. If you supply a function for **am_insert**, you must also set the **am_readwrite** purpose flag when you register the access method. (Refer to "Registering the Access Method" on page 2-11.)

You might also provide **am_delete** and **am_update** purpose functions to handle the SQL statements DELETE and UPDATE, respectively. If you provide an **am_delete** or **am_update** purpose function, you must also set the **am_readwrite** and **am_rowids** purpose flags when you register the access method.

Warning: *If you do not supply functions for* ***am_insert****,* ***am_update****, or* ***am_delete****, or you do not set the appropriate purpose flags, the database server cannot process the corresponding SQL statement and issues an error.*

Registering Purpose Functions

To register user-defined purpose functions with the database server, issue a CREATE FUNCTION statement for each one.

By convention, you package access-method functions in a DataBlade module. Install the software in **$INFORMIXDIR/extend/*DataBlade_name*** for UNIX or **%INFORMIXDIR%\extend*DataBlade_name*** for Windows NT.

For example, assume you create an **open_virtual** function that has a table descriptor as its only argument, as the following declaration shows:

```
mi_integer open_virtual(MI_AM_TAB_DESC *)
```

Because the database server always passes descriptors by reference as generic pointers to the access method, you register the purpose functions with an argument of type **pointer** for each descriptor. The following example registers the function **open_virtual()** function on a UNIX system. The path suggests that the function belongs to a DataBlade module named **amBlade**.

```
CREATE FUNCTION open_virtual(pointer)
RETURNING integer
[ WITH (PARALLELIZABLE)]
EXTERNAL NAME

'$INFORMIXDIR/extend/amBlade/my_virtual.bld(open_virtual)'
LANGUAGE C
```

The PARALLELIZABLE routine modifier indicates that you have designed the function to execute safely in parallel. Parallel execution can dramatically speed the throughput of data. By itself, the routine modifier does not guarantee parallel processing. For more information about parallel execution of functions that belong to an access method, refer to "Executing in Parallel" on page 3-33.

Important: *You must have the Resource or DBA privilege to use the CREATE FUNCTION statement and the Usage privilege on C to use the LANGUAGE C clause.*

For the complete syntax of the CREATE FUNCTION statement, refer to the *Informix Guide to SQL: Syntax*. For information about privileges, refer to the GRANT statement in the *Informix Guide to SQL: Syntax*.

Important: *The CREATE FUNCTION statement adds a function to a database but not to an access method. To enable the database server to recognize a registered function as a purpose function in an access method, you register the access method.*

Registering the Access Method

The CREATE FUNCTION statement identifies a function as part of a database, but not necessarily as part of an access method. To register the access method, issue the CREATE PRIMARY ACCESS_METHOD statement, which sets values in the **sysams** system catalog table, such as:

- the unique name of each purpose function.
- a storage-type (extspaces or sbspaces) indicator.
- flags that activate optional features, such as writable data or clustering.

The sample statement in Figure 2-2 assigns registered function names to some purpose functions. It specifies that the access method should use sbspaces, and it enables clustering.

```
CREATE PRIMARY ACCESS_METHOD my_virtual
(   AM_OPEN = open_virtual,
    AM_CLOSE = close_virtual,
    AM_CREATE = create_virtual,
    AM_DROP = drop_virtual,
    AM_BEGINSCAN = beginscan_virtual,
    AM_GETNEXT = getnext_virtual,
    AM_ENDSCAN = endscan_virtual,
    AM_INSERT = insert_virtual,
    AM_DELETE = delete_virtual,
    AM_UPDATE = update_virtual,
    AM_READWRITE
    AM_SPTYPE = S,
    AM_CLUSTER)
```

Figure 2-2
Registering an Access Method

Figure 2-3 shows the resulting **sysams** system catalog entry for the new access method.

```
am_name         my_virtual
am_owner        informix
am_id           101
am_type         P
am_sptype       S
am_cluster      1
am_rowids       0
am_readwrite     1
am_parallel     0
am_costfactor   1.000000000000
am_create       162
am_drop         163
am_open         164
am_close        165
am_insert       166
am_delete       167
am_update       168
am_stats        0
am_scancost     0
am_check        0
am_beginscan    169
am_endscan      170
am_rescan       0
am_getnext       171
```

Figure 2-3
sysams Entry

The statement in Figure 2-2 does not name a purpose function for **am_stats**, **am_scancost**, or **am_check**, as the 0 values in Figure 2-3 indicate. The database server sets a 0 value for **am_parallel** because none of the CREATE FUNCTION statements for the purpose functions included the PARALLELIZATION routine modifier.

Warning: *Even if you supply and register a purpose function with the CREATE FUNCTION statement, the database server assumes that a purpose function does not exist if the purpose-function name in the* ***sysams*** *system catalog table is missing or misspelled.*

For syntax and a list of available purpose settings, refer to Chapter 6, "SQL Statements for Access Methods."

Testing the Access Method

To test the access method, take the same actions that users of the access method take to create and access virtual data:

1. Create one or more storage spaces.
2. Use the access method to create tables in your storage spaces.
3. Run SQL statements to insert, query, and alter data.
4. Use the **oncheck** utility, which executes **am_check**, to check the integrity of the data structures that the access method writes to disk.

Typically, a database system administrator who is responsible for the configuration of the database server performs steps 1 and 4. A database administrator performs step 2. Anyone with the appropriate SQL privileges to access or update the table that uses the access method performs step 3.

Creating and Specifying Storage Spaces

A storage space is a physical area where the table data is stored. To test how the access method builds new tables, you create a new physical storage space before you create the table.

If the access method interfaces with legacy data, the storage spaces already exist, usually in external storage.

This section describes how to establish storage spaces.

Using Internal Storage

An sbspace holds smart large objects for the database server. This space is physically included in the database server configuration.

To test the access method with an sbspace

1. Create an sbspace with the **onspaces** utility.
2. Optionally, set the default sbspace for the database server.
3. Create a virtual table with the CREATE TABLE statement.

Creating an Sbspace

An sbspace must exist before you can create a virtual table in it. Before you can test the ability of the access method to create a table that does not yet exist, you must run the **onspaces** utility to create a smart-large-object storage space. The **onspaces** command associates a logical name with a physical area of a specified size in a database server partition.

The following **onspaces** command creates an sbspace named **vspace1**:

UNIX

```
onspaces -c -S vspace1 -g 2 -p /home/informix/chunk2
    -o 0 -s 20000
```

♦

WIN NT

```
onspaces -c -S vspace1 -g 2 -p \home\informix\chunk2
    -o 0 -s 20000
```

♦

Specifying the Logical Sbspace Name

The following example creates a virtual table in the previously created **vspace1**:

```
CREATE TABLE tab1(...)
    IN vspace1
    USING your_access_method
```

If you do not intend to specify an sbspace explicitly in the CREATE TABLE statement, specify a default sbspace. To find out how to create a default dbspace, see "Creating a Default Sbspace" on page 3-14.

The following example also creates a virtual table in the sbspace that SBSPACENAME specifies:

```
CREATE TABLE tab1(...)
    USING your_access_method
```

Using External Storage

An *extspace* lies outside the disk storage that is configured for the database server. To create a physical extspace, you might use an operating-system command or use a data management software system. An extspace can have a location other than a path or filename because the database server does not interpret the location. Only the access method uses the location information.

To store virtual data in an extspace, take one of the following actions:

- Create logical names for existing external storage with the **onspaces** utility, and then specify the reserved name or names when you create a virtual table with the CREATE TABLE statement.
- Directly specify an existing physical external storage location as a quoted string in the CREATE TABLE statement.
- Provide a default physical external storage location, such as a disk file, in the access-method code.

Specifying a Logical Name

The **onspaces** command creates an entry in the system catalog that associates a name with an existing extspace. To create a logical extspace name, use the following command-line syntax:

```
onspaces -c -x exspace_name -l "location_specifier"
```

UNIX

The following example assigns the logical name **disk_file** to a path and filename for a physical disk:

```
onspaces -c -x disk_file -l "/home/database/datacache"
```

The following example specifies a tape device:

```
onspaces -c -x tape_dev -l "/dev/rmt/0 "
```

♦

WIN NT/95

The following example assigns the logical name **disk_file** to a physical disk path and filename:

```
onspaces -c -x disk_file -l "\home\database\datacache"
```

♦

If you assign a name with **onspaces**, refer to it by its logical name in the SQL statement that creates the table, as in the following example:

```
CREATE TABLE tab1(
    col1 INTEGER,
    col2 INTEGER)
    IN disk_file
    USING your_access_method
```

Specifying the Physical Location

As an alternative to the extspace name, a CREATE TABLE statement can directly specify a quoted string that contains the external location.

```
CREATE TABLE tab1(
    col1 INTEGER,
    col2 INTEGER)
    IN "location_specifier"
    USING your_access_method
```

Providing a Default Extspace

If you do not intend to specify an extspace explicitly in the CREATE TABLE statement, the access method can create a default extspace. For an example that creates an extspace directly in the access-method code, refer to Figure 3-4 on page 3-15.

Using Fragments

If you want to test the access method for fragmentation support, specify a different storage space for each fragment.

The following example shows the creation of a table with two fragments. Each fragment corresponds to a separate extspace. The database server alternates between the fragments to store new data.

```
CREATE TABLE table_name(...)
    FRAGMENT BY ROUNDROBIN IN "location_specifier1",
"location_specifier2"
    USING access_method_name
```

To fragment a table in smart-large-object storage, create a separate sbspace for each fragment before you create the table. Use the **onspaces** command, as the following example shows:

```
onspaces -c -S fragspace1 -g 2 -p location_specifier1 -o 0 -s 20000
onspaces -c -S fragspace2 -g 2 -p location_specifier2 -o 0 -s 20000

CREATE TABLE catalog (status pages)
    USING catalog_am
    FRAGMENT BY EXPRESSION
        pages > 15 IN fragspace2,
        REMAINDER IN fragspace1
```

Avoiding Storage-Space Errors

An SQL error occurs if you include an IN clause with the CREATE TABLE statement and one of the following conditions is true:

- The IN clause specifies an extspace or sbspace that does not exist.
- The IN clause specifies an sbspace but the **am_sptype** purpose value is set to `'X'`.
- The IN clause specifies an extspace but the **am_sptype** purpose value is set to `'S'`.

An SQL error occurs if the CREATE TABLE statement contains no IN clause and one of the following conditions is true:

- The **am_sptype** purpose value is set to `'A'`, no default SBSPACENAME exists, and the access method does not create an extspace.
- The **am_sptype** purpose value is set to `'S'` , and no default SBSPACENAME exists.
- The **am_sptype** purpose value is set to `'X'` , and the access method does not create an extspace.

An SQL error occurs if of the following conditions is true:

- The **am_sptype** purpose value is set to `'D'`.
- The IN clause with the CREATE TABLE statement specifies a dbspace, even if the **am_sptype** purpose value is set to `‘A’`.

Inserting, Querying, and Updating Data

If you want to test fragmented tables, use the SQL syntax in "Supporting Fragmentation" on page 3-17. If you want to support user-configured options, use the SQL syntax in "Providing Configuration Keywords" on page 3-18.

You can provide support in the access method for CREATE TABLE statement keywords that effect transaction processing. For example, if a CREATE TABLE statement might specify the WITH ROWIDS keyword, the access method must include add a column of visible row identifiers to the table and allow queries on row identifiers. If a CREATE TABLE statement specifies the LOCK MODE clause, the access method must impose and manage locks during data retrieval and update. To determine the state of a table during transaction processing, the access method calls VTI functions to determine the lock mode, data-entry constraints, referential constraints, and other state information.

A user sets the *isolation level* with commands such as SET ISOLATION and SET TRANSACTION or with configuration settings in the ONCONFIG file. Informix recommends that you document the isolation levels that the access method supports, as "mi_scan_isolevel()" on page 5-45 describes. For information about setting isolation levels, refer to the *Informix Guide to SQL: Syntax* and the *Informix Guide to SQL: Tutorial*.

A database server administrator can use the ONCONFIG file to set defaults for such things as isolation level, locking, logging, and sbspace name. For information about defaults that you can set for the test-environment ONCONFIG file, refer to the *Administrator's Guide*.

For information about SQL statements and keywords that your access method can support, refer to the *Informix Guide to SQL: Syntax*. For information about the VTI functions that determine which statements and keywords the user specifies, refer to Chapter 5, "Descriptor Function Reference."

Checking Data Integrity

If you implement the **oncheck** command with the **am_check** access method, you can execute the **oncheck** command with appropriate options on a command line. The access method can issue messages that describe any problems in the test data.

For more information about how to implement the **oncheck** processing, refer to the description of **am_check** on page 4-15. For more information about how to specify options on the command line for **oncheck**, refer to the *Administrator's Reference*.

Dropping an Access Method

To drop an access method, execute the DROP ACCESS_METHOD statement, as the following example shows:

```
DROP ACCESS_METHOD my_virtual RESTRICT
```

For more information, refer to "DROP ACCESS_METHOD" on page 6-8.

Chapter

Design Decisions

3

In This Chapter

This chapter begins with several topics that discuss how the access method uses DataBlade API functions:

- "Storing Data in Shared Memory," which follows this section
- "Accessing Database and System Catalog Tables" on page 3-7
- "Handling the Unexpected" on page 3-8

The chapter continues with topics that discuss alternative ways to accomplish the following SQL tasks:

- "Supporting Data Definition Statements" on page 3-11
- "Processing Queries" on page 3-20
- "Enhancing Performance" on page 3-33
- "Supporting Data Retrieval, Manipulation, and Return" on page 3-37

The chapter ends with guidelines for helping end users and application developers to use the access method in "Supplying Error Messages and a User Guide" on page 3-41.

In particular, this chapter presents the choices that you make to optimize the performance and flexibility of your access method.

Storing Data in Shared Memory

The access method can allocate areas in shared memory to preserve information between purpose-function calls. To allocate memory, you make the following choices:

- Which function to call
- What duration to assign

Functions That Allocate and Free Memory

The DataBlade API provides two categories of memory-allocation functions:

- Public functions allocate memory that is local to one database server thread.
- Semipublic functions allocate named, global memory that multiple threads might share.

For either unnamed and named memory, you can specify a duration that reserves the memory for access method use beyond the life of a particular purpose function.

For most purposes, UDRs, including access methods, can allocate shared memory with the public DataBlade API memory-management functions, **mi_alloc()**, **mi_dalloc()**, or **mi_zalloc()**. UDRs share access to memory that a public function allocates with the pointer that the allocation function returns. For an example that allocates memory and stores a pointer, refer to "Persistent User Data" on page 3-6. The public **mi_free()** function frees the memory that a public function allocates.

The memory that you allocate with public functions is available only to UDRs that execute during a single-thread table operation. Access-method UDRs might execute across multiple threads to manipulate multiple fragments or span multiple queries. UDRs that execute in multiple threads can share named memory.

The semipublic DataBlade API **mi_named_alloc()** or **mi_named_zalloc()** memory-management functions allocate named memory, the **mi_named_get()** function retrieves named memory, and the **mi_named_free()** function releases the named memory. Related semipublic functions provide for locking on named memory.

Warning: *Do not call **malloc()** because the memory that **malloc()** allocates disappears after a virtual-processor (VP) switch. The access method might not properly deallocate memory that **malloc()** provides, especially during exception handling.*

Memory-Duration Options

When a UDR calls a DataBlade API memory-allocation function, the memory exists until the duration assigned to that memory expires. The database server stores memory in pools by duration. By default, memory-allocation functions assign a PER_ROUTINE duration to memory. The database server automatically frees PER_ROUTINE memory after the UDR that allocates the memory completes.

An SQL statement typically invokes many UDRs to perform a table task. Memory that stores state information must persist across all the UDR calls that the statement requires. The default PER_ROUTINE duration does not allow memory to persist for an entire SQL statement.

Use the **mi_dalloc()** function to specify a memory duration for a particular new memory allocation. If you do not specify a duration, the default duration applies. You can change the default from PER_ROUTINE to a different duration with the **mi_switch_mem_duration()** function. The following list describes memory durations that an access method typically specifies:

- Use PER_COMMAND for the memory that you allocate to scan-descriptor user data, which must persist from the **am_beginscan** thorough the **am_endscan** functions.
- Use PER_STATEMENT for the memory that you allocate for table-descriptor user data, which must persist from the **am_open** through the **am_close** functions.

You must store a pointer to the PER_COMMAND or PER_STATEMENT memory so that multiple UDRs that execute during the command or statement can retrieve and deference the pointer to access the memory.

For detailed information about the following items, refer to the *DataBlade API Programmer's Manual*:

- Functions that allocate public memory
- Duration keywords

For more information about semipublic functions and named memory, follow the link to the DataBlade Corner on the Informix Developer Network Web page, www.informix.com/idn.

Persistent User Data

The term *user data* refers to information that a purpose function saves in shared memory. The access method defines a user-data type and then allocates an area of memory with the appropriate size and duration. In the following example, the user data stores the information that the access method needs for a PER_STATEMENT duration.

```
MI_AM_TAB_DESC * tableDesc; /* Pointer to table descriptor */
typedef enum my_col_types
{
    MY_INT = 1,
    MY_CHAR
} my_col_type;

typedef struct my_row
{
    char           data[500];
    struct my_row *next;
} my_row_t;

typedef struct statement_data
{

    MI_DATUM   *retrow;/*Points to data in memory*/
    my_col_type  col_type[10]; /*Data types in the projected row*/
    mi_boolean    is_null[10]; /*Array of true and false indicators*/
    my_row_t           *current row;
    MI_CONNECTION      *conn;
    MI_CALLBACK_HANDLE *error_cback;
} statement_data_t;

/*Allocate memory*/
my_data = (statement_data_t *)
    mi_dalloc(sizeof(statement_data_t),PER_STATEMENT);

mi_tab_setuserdata(tableDesc, (void *) my_data); /*Store pointer*/
```

Figure 3-1
Allocating User-Data Memory

Figure 3-2 shows accessor functions that the VTI provides to store and retrieve user data.

Figure 3-2
Storing and Retrieving User-Data Pointers

Descriptor	User-Data Duration	Stores Pointer to User Data	Retrieves Pointer to User Data
Table descriptor	PER STATEMENT	**mi_tab_setuserdata()**	**mi_tab_userdata()**
Scan descriptor	PER COMMAND	**mi_scan_setuserdata()**	**mi_scan_userdata()**

The following example shows how to retrieve the pointer from the table descriptor that the **mi_tab_setuserdata()** function set in Figure 3-1:

```
my_data=(my_data_t *)mi_tab_userdata(tableDesc);
```

For more information about **mi_tab_setuserdata()**, **mi_tab_userdata()**, **mi_scan_setuserdata()**, and **mi_scan_userdata()**, refer to Chapter 5, "Descriptor Function Reference."

Accessing Database and System Catalog Tables

Although the VTI does not provide its own function for querying tables, you can execute an SQL statement with DataBlade API functions **mi_exec()**, **mi_prepare()**, or **mi_exec_prepared_statement()**. SQL provides data directly from the system catalog tables and enables the access method to create tables to hold user data on the database server.

The following example queries the system catalog table for previous statistics:

```
MI_CONNECTION *conn;
conn = mi_open(NULL, NULL, NULL);
/* Query system tables */
mi_exec(conn, "select tabname, nrows from systables ", MI_QUERY_NORMAL);
```

For more information on querying database tables, consult the *DataBlade API Programmer's Manual*.

***Warning:** A parallelizable UDR must not call **mi_exec()**, **mi_prepare()**, **mi_exec_prepared_statement()**, or a UDR that calls these functions. A database server exception results if a parallelizable UDR calls any UDR that prepares or executes SQL. For more information about parallelizable access-method functions, refer to* *"Executing in Parallel" on page 3-33*.

Handling the Unexpected

The access method can respond to events that the database server initiates, as well as to errors in requests for access-method features that the database server cannot detect.

Using Callback Functions

Database server events include the following types.

Event Type	Description
MI_Exception	Exceptions with the following severity: ■ Warnings ■ Runtime errors
MI_EVENT_END_XACT	End-of-transaction state transition
MI_EVENT_END_STMT	End-of-statement state transition
MI_EVENT_END_SESSION	End-of-session state transition

To have the access method handle an error or a transaction rollback, use the DataBlade API mechanism of *callback function*s. A callback function automatically executes when the database server indicates that the event of a particular type has occurred.

To register an access-method callback function, pass the function name and the type of event that invokes the function to **mi_register_callback()**, as the example in Figure 3-3 shows.

```
typedef struct statement_data
{
...
     MI_CALLBACK_HANDLE *error_cback;
} statement_data_t;

/*Allocate memory*/
my_data = (statement_data_t *)
     mi_dalloc(sizeof(statement_data_t),PER_STATEMENT);


my_data.error_cback=
     mi_register_callback(connection,
          MI_Exception, error_callback, NULL, NULL)
```

Figure 3-3
Registering a Callback Function

The example in Figure 3-3 accomplishes the following actions:

- Registers the **error_callback()** function as a callback function to handle the MI_Exception event
- Stores the callback handle that **mi_register_callback()** returns in **error_cback** field of the **my_data** memory

For more information about detecting if a transaction commits or rolls back, refer to "Checking Isolation Levels" on page 3-37.

By default, the database server aborts the execution of the access-method UDR if any of the following actions by the access method fail:

- Allocating memory
- Using the FastPath feature to execute a UDR
- Obtaining a handle for a file or smart large object
- Obtaining a connection
- Reading or writing to storage media, such as a disk

If you want to avoid an unexpected exit from the access method, register a callback function for any exception that you can anticipate. The callback function can rollback transactions and free memory before it returns control to the database server, or it can tell the database server to resume access-method processing.

For a complete discussion of callback processing and the DataBlade API **mi_register_callback()** function, refer to the *DataBlade API Programmer's Manual*. For code samples, follow the link to the DataBlade Corner on the Informix Developer Network Web page, www.informix.com/idn.

Using Error Messages

The database server cannot validate specifications for features that the access method adds. If the access method includes a feature that the database server cannot detect, the access method must explicitly handle syntax errors in requests for that feature. To handle errors that the database server cannot detect, call the DataBlade API **mi_db_error_raise()** function.

The following example shows how an access method might avoid an unexpected exit due to a user error that the database server cannot detect. The CREATE TABLE statement in this example specifies configuration parameters.

```
CREATE TABLE legacy
...
USING text_file_access(delimiter = '!')
```

The access method must notify a user if a statement specifies an invalid parameter. To determine the parameters that a CREATE TABLE statement specifies, the access method calls the accessor function **mi_tab_amparam()**. To notify a user of an invalid parameter, the access method raises an exception, as the following example shows:

```
if (mi_tab_amparam(tableDesc) != 'delimiter')
    mi_db_error_raise( connection, MI_EXCEPTION,
        "Invalid configuration keywordin the USING clause.");
```

The uppercase MI_EXCEPTION alerts the database server that an exception has occurred but does not necessarily halt execution. In contrast, the following call, which also raises an exception, assumes that a callback function exists for MI_Exception:

```
mi_db_error_raise( connection, MI_Exception, "Invalid...");
```

If the function that calls **mi_db_error_raise()** did not register a callback function for MI_Exception (upper and lowercase), execution aborts after the `Invalid...` error message appears.

The database server cannot always determine that the access method does not support a feature that a user specifies. The access method can test for the presence of specifications and either provide the feature or raise an exception for those features that it cannot provide.

For example, the database server does not know if the access method can handle lock types, isolation levels, referential constraints, or fragmentation that an SQL statement specifies. To retrieve the settings for mode, isolation level, and lock, the access method calls the following accessor functions.

Function	Purpose
mi_tab_mode()	The input/output mode (read only, read and write, write only, and log transactions)
mi_tab_isolevel()	The isolation level
mi_scan_locktype()	The lock type for the scan
mi_scan_isolevel()	The isolation level in force

For more information, refer to the following sections:

- "Checking Isolation Levels" on page 3-37
- "Notifying the User About Access-Method Constraints" on page 3-44
- "Accessor Functions" on page 5-16

Supporting Data Definition Statements

The *data definition* statement CREATE TABLE names the table and specifies the owner, column names and data types, fragmentation method, storage space, and other structural characteristics. Other data definition statements alter the structure from the original specifications in the CREATE TABLE statement. This section discusses design considerations for CREATE TABLE, ALTER TABLE, and ALTER FRAGMENT.

Interpreting the Table Descriptor

A *table descriptor* contains data definition specifications, such as owner, column names and data types, and storage space, that the CREATE TABLE, ALTER TABLE, and ALTER FRAGMENT statements specify for the virtual table. A table descriptor describes a single table fragment, so that the storage space and fragment identifier (part number) change in each of multiple table descriptors that the database server constructs for a fragmented table.

For a complete description, refer to "Table Descriptor" on page 5-13.

Managing Storage Spaces

A user-defined access method stores data in sbspaces, extspaces, or both. To access data in smart large objects, the access method must support sbspaces. To access legacy data in disk files or within another database management system, the access method supports extspaces.

Important: *Your access method cannot directly create, open, or manipulate a table in a dbspace.*

The following sections describe how the access method supports sbspaces, extspaces, or both:

- "Choosing DataBlade API Functions"
- "Setting the am_sptype Value" on page 3-13
- "Creating a Default Storage Space" on page 3-14
- "Ensuring Data Integrity" on page 3-15.
- "Checking Storage-Space Type" on page 3-17
- "Supporting Fragmentation" on page 3-17

Choosing DataBlade API Functions

The type of storage space determines whether you use **mi_file_*()** functions or **mi_lo_*()** functions to open, close, read from, and write to data.

To have the access method store data in an sbspace, use the smart-large-object interface of the DataBlade API. The names of most functions of the smart-large-object interface begin with the **mi_lo_** prefix. For example, you open a a smart large object in an sbspace with **mi_lo_open()** or one of the smart-large-object creation functions: **mi_lo_copy()**, **mi_lo_create()**, **mi_lo_expand()**, or **mi_lo_from_file()**.

If the access method stores data on devices that the operating system manages, use the DataBlade API file-access functions. Most file-access functions begin with the **mi_file_** prefix. For example, the **am_open** purpose function might open a disk file with **mi_file_open()**.

Important: *Do not use operating-system commands to access data in an extspace.*

For more information about smart-large-object functions and file-access functions, refer to the *DataBlade API Programmer's Manual*.

If another database manager reads and writes the data, pass input/output requests to the external database manager. For a demonstration primary access method that passes data requests to external processes, follow the link to the DataBlade Corner on the Informix Developer Network Web page, www.informix.com/idn.

Setting the am_sptype Value

Set the **am_sptype** value to `'S'` if the access method reads and writes to sbspaces but not to extspaces. Set the **am_sptype** value to `'X'` if the access method reads and writes only to extspaces but not to sbspaces.

To set the **am_sptype** purpose value, use the CREATE PRIMARY ACCESS_METHOD or ALTER ACCESS_METHOD statement, as Chapter 6, "SQL Statements for Access Methods," describes.

If you do not set the **am_sptype** storage option, the default value `'A'` means that a user can create a virtual table in either extspaces or sbspaces. The access method must be able to read and write to both types of storage spaces.

Warning: *In the access-method user guide, notify users whether the access method supports sbspaces, extspaces, or both and describe default behavior. The database server issues an SQL error if the user or application attempts to use a storage space that the access method does not support.*

Creating a Default Storage Space

A default storage space of the appropriate type prevents an exception from occurring if the user does not specify a storage-space name in the CREATE TABLE statement.

Creating a Default Sbspace

If the access method supports sbspaces, the user, typically the database server administrator, can create a default sbspace.

To create a default sbspace

1. Create a named sbspace with the **onspaces** utility.

   When you create the default sbspace, you can turn on transaction logging.
2. Assign that name as the default sbspace in the SBSPACENAME parameter of the ONCONFIG file.
3. Initialize the database server with the **oninit** utility.

For example, you create a default sbspace named **vspace** with the following steps.

1. From the command line, create the sbspace with logging turned on:
   ```
   onspaces -c -S vspace -p path -o offset -s size -Df "LOGGING=ON"
   ```
2. Edit the ONCONFIG file to insert the following line:
   ```
   SBSPACENAME vspace # Default sbspace name
   ```
3. Take the database server offline and then bring it online again to initialize memory with the updated configuration.
   ```
   onmode -ky
   oninit
   ```

For more information about the configuration file parameters, and the **onspaces**, **onmode**, and **oninit** utilities, refer to the *Administrator's Reference.*

Creating a Default Extspace

The ONCONFIG file does not provide a parameter that specifies a default extspace name. The access method might do one of the following things if the CREATE TABLE statement does not specify an extspace:

- Raise an error
- Specify an external storage space

  The example in Figure 3-4 specifies a directory path as the default extspace on a UNIX system.

```
mi_integer external_create(td)
MI_AM_TABLE_DESC *td;
{
...
/* Did the CREATE statement specify a named extspace? **/
dirname = mi_tab_spaceloc(td);
if (!dirname || !*dirname)
{
    /* No. Put the table in /tmp */
    dirname = (mi_string *)mi_alloc(5);
    strcpy(dirname, "/tmp");
}
sprintf(name,"%s/%s-%d", dirname, mi_tab_name(td),
        mi_tab_partnum(td));

out = mi_file_open(name,O_WRONLY|O_TRUNC|O_CREAT,0600);
```

Figure 3-4
Creating a Default Extspace

Ensuring Data Integrity

The access method might provide any of the following features to ensure that source data matches virtual data:

- Locks
- Logging
- Backup and recovery
- Transaction management

Activating Automatic Controls in Sbspaces

The following advantages apply to data that resides in sbspaces:

- A database server administrator can back up and restore sbspaces with standard Informix utilities.
- The database server automatically provides for locking.
- If a transaction fails, the database server automatically rolls back sbspace metadata activity.

If logging is turned on for the smart large object, the database server performs the following tasks:

- Logs transaction activity
- Rolls back uncommitted activity if a transaction fails

You can either advise the end user to set logging on with the **onspaces** utility or call the appropriate DataBlade API functions to set logging.

Important: *To provide transaction integrity, Informix recommends that the access method require transaction logging in sbspaces. Informix also recommends that the access method raise an error if an end user attempts to create a virtual table in an sbspace without logging.*

In the access-method user guide, alert the user to create sbspaces with transaction logging enabled. To create an sbspace with transaction logging, specify the LOGGING tag for the **onspaces -Df** option.

For more information about metadata logging and transaction logging, refer to the *Administrator's Guide*.

Adding Controls for Extspaces

Because the database server cannot safeguard operations on extspace data, include UDRs for any of the following features that you want the access method to provide:

- Locks
- Logging and recovery
- Transaction commit and rollback management (described in "Checking Isolation Levels" on page 3-37)

Checking Storage-Space Type

The database server issues an error if the CREATE TABLE statement specifies inappropriate storage type. To determine the storage space (if any) that the CREATE TABLE statement specifies, the access method calls the **mi_tab_spacetype()** function. For details, refer to **mi_tab_spacetype()** on page 5-80.

For more information about errors that occur from inappropriate storage-space type, refer to "Avoiding Storage-Space Errors" on page 2-17. For more information about documenting potential errors and intercepting error events, refer to "Supplying Error Messages and a User Guide" on page 3-41.

Supporting Fragmentation

A fragmented table has multiple physical locations, called *fragments*. The user specifies the criteria by which the database server distributes information into the available fragments. For examples of how a user creates fragments, refer to "Using Fragments" on page 2-16. For a detailed discussion about the benefits of and approaches to fragmentation, refer to the *Informix Guide to Database Design and Implementation*.

When the table is fragmented, each call to the access method involves a single fragment rather than the whole table. An SQL statement, such as CREATE TABLE, can result in a set of purpose-function calls from **am_open** through **am_close** for each fragment.

The database server can process fragments in parallel. For each fragment identifier, the database server starts a new access-method thread. To obtain the fragment identifier for the table, call the **mi_tab_partnum()** function.

An end user might change the way in which values are distributed among fragments after data already exists in the table. Because some rows might move to a different fragment, an ALTER FRAGMENT statement requires a scan, delete, and insert for each moved row. For information about how the database server uses the access method to redefine fragments, refer to "ALTER FRAGMENT Statement Interface" on page 4-8.

For information about the FRAGMENT BY clause, refer to the *Informix Guide to SQL: Syntax*.

Providing Configuration Keywords

You can provide configuration keywords that the access method interrogates to tailor its behavior. The user specifies one or more parameter choices in the USING clause of the CREATE TABLE statement. The access method calls the **mi_tab_amparam()** accessor function to retrieve the configuration keywords and values.

In the following example, the access method checks the keyword value to determine if the user wants mode set to the number of rows to store in a shared-memory buffer. The CREATE TABLE statement specifies the configuration keyword and value between parentheses.

```
CREATE TABLE ...
IN sbspace
USING sbspace_access_method ("setbuffer=10")
```

In the preceding statement, the **mi_tab_amparam()** function returns `setbuffer=10`. Figure 3-5 shows how the access method determines the value that the user specifies and applies it to create the sbspace.

```
mi_integer my_beginscan (sd)
     MI_AM_SCAN_DESC    *sd;
{
    MI_AM_TABLE_DESC    *td;
    mi_ineger             nrows;
    ...
    td=mi_scan_table(sd); /*Get table descriptor. */
    /*Check for parameter.
    ** Do what the user specifies.
    If (mi_tab_amparam(td) != NULL)
    {
        /* Extract number of rows from string.
        ** Set nrows to that number. (not shown.)
        */
        mi_tab_setniorows(nrows);
    }
    ...
}
```

Figure 3-5
Checking a Configuration Parameter Value

Important: *If the access method accepts parameters, describe them in the user guide for the access method. For example, a description of the action in Figure 3-5 would explain how to set a value in the parameter string* `"setbuffer="` *and describe how a buffer might improve performance.*

A user can specify multiple configuration parameters separated by commas, as the following syntax shows:

```
CREATE TABLE ...
...
USING access_method_name (keyword='string', keyword='string'
...)
```

Leveraging Indexes on Virtual Tables

The database server can quickly scan a B-tree the index for qualifying entries. For each qualifying entry, the database server takes one of the following actions:

- Invokes the access method to fetch a specific row from the base table
- Returns the index keys

  If the index keys contain all the columns that the query projects, the database server does not need to invoke the access method.

If the query requires data from the base table, the database server can pass row identifiers to the access method. With row identifiers, the access method retrieves data by address, which eliminates the need to scan the entire base table.

To enable an index on a virtual table, provide an **am_getbyid** purpose function to fetch data directly from a physical address and set the **am_rowids** purpose flag with the CREATE ACCESS_METHOD or ALTER ACCESS_METHOD statement.

If the database server can scan an index to locate rows in a virtual table, it executes **am_getbyid** instead of the **am_getnext** purpose function. The **am_getbyid** purpose function calls DataBlade API or external routines to access the row by its row identifier. Thus, if you provide **am_getbyid** and the appropriate index exists, the access method does not scan the table to find rows.

For more information about **am_getbyid**, refer to page 4-25. For more information about **am_rowids**, refer to Chapter 6, "SQL Statements for Access Methods."

Processing Queries

This section describes various options for processing a SELECT statement, or *query*, that involves a virtual table. An SQL query requests that the database server fetch and assemble stored data into rows. A SELECT statement often includes a WHERE clause that specifies the values that a row must have to qualify for selection.

Depending on the specifications in the query, the returned data might contain the entire stored table or particular columns and rows. The SELECT clause names the columns that make up the *projection*. An efficient access method returns values for projected columns only. The WHERE clause specifies the values that qualify a row for *selection*. An efficient access method formats and returns only those rows that contain the selected values.

The following query projects the values in the particular columns **name** and **department** and selects the particular rows that contain the value `Manager` in the **title** column. The query does not include **title** in the projection.

```
SELECT name, department FROM employee
    WHERE title = 'Manager'
```

Query processing involves the following actions:

- Interpreting the scan and qualification descriptors
- Scanning the table to select rows
- Returning rows that satisfy the query
- Maintaining cost and distribution information for the optimizer

Interpreting the Scan Descriptor

The database server constructs a *scan descriptor* to pass the contents of the SELECT clause to the access method. The scan descriptor specifies which columns the query projects and provides information about the locks and isolation levels that apply to the data that the query specifies.

For efficiency, the access method can format only the data that SELECT clause projects and place NULL values in the remaining columns. To determine which columns to project, call the **mi_scan_projs()** function.

As one of its primary functions, the scan descriptor stores a pointer to another opaque structure, the *qualification descriptor* that contains WHERE clause information. To access the qualification descriptor, use the pointer that the **mi_scan_quals()** function returns. A NULL-valued pointer indicates that the database server did not construct a qualification descriptor.

Important: *If **mi_scan_quals()** returns a NULL-valued pointer, the access method must format and return all possible rows.*

For more information about the information that the scan descriptor provides, refer to "Scan Descriptor" on page 5-11 and the scan-descriptor accessor functions that begin on page 5-44.

Interpreting the Qualification Descriptor

A qualification descriptor contains the individual qualifications that the WHERE clause specifies. A *qualification*, or *filter,* tests a value from a row against a constant value. Each branch or level of a WHERE clause specifies one of the following operations:

- A function
- A Boolean expression

The WHERE clause might include negation indicators, each of which reverses the result of a particular function.

The access method executes VTI accessor functions to extract individual qualifications from a qualification descriptor. The following table lists frequently used accessor functions.

Accessor Function	Purpose
mi_qual_nquals()	Determines the number of simple functions and Boolean operators in a complex qualification
mi_qual_qual()	Pointer to one qualification in a complex qualification descriptor or to the only qualification
mi_qual_issimple() **mi_qual_boolop()**	Determine which of the following qualifications the descriptor describes: ■ A simple function ■ A complex AND or OR expression

Accessor Function	Purpose
mi_qual_funcid() or **mi_qual_funcname()**	Identify a simple function by function identifier or function name
mi_qual_column()	Identifies the column argument of a function
mi_qual_constant()	Extracts the value from the constant argument of a function
mi_qual_negate()	MI_TRUE if the qualification includes the operator NOT
mi_qual_setvalue()	Sets an MI_VALUE_TRUE or MI_VALUE_FALSE indicator for one qualification in a complex qualification descriptor
mi_qual_value()	retrieves the results that **mi_qual_setvalue()** set for a previous qualification Until the qualification sets a result, this function returns the initial value, MI_VALUE_NOT_EVALUATED.

For a complete list of access functions for the qualification descriptor, refer to "Qualification Descriptor" on page 5-7.

Simple Functions

The smallest element of a qualification is a function that tests the contents of a column against a specified value. For example, in the following SELECT statement, the function tests whether value in the **lname** column is the character string SMITH:

```
SELECT lname, fname, customer_num from customer
WHERE lname = "SMITH"
```

In the preceding example, the equal operator (=) represents the function **equal()** and has two arguments, a column name and a string constant. The following formats apply to simple qualification functions.

Figure 3-6
Generic Function Prototypes

Generic Prototype	Description
function(*column_name*)	Evaluate the contents of the named column
function(*column_name*, *constant*) *function*(*constant*, *column_name*)	Evaluate the contents of the named column and the explicit value of the constant argument In a *commuted* argument list, the constant value precedes the column name.
function(*column* ?)	Evaluate the value in the specified column of the current row and a value, called a *host variable*, that a client program supplies.
function(*column*, *slv #*)	Evaluate the value in the specified column of the current row and a value, called a *statement-local variable* (SLV), that the UDR supplies.
function(*column*, *constant*, *slv #*) *function*(*constant*, *column*, *slv #*)	Evaluate the value in the specified column of the current row, an explicit constant argument, and an SLV.

Runtime Values as Arguments

The following types of arguments supply values as the function executes:

- A statement-local variable (SLV)
- A host variable

Statement-Local Variables

The parameter list of a UDR can include an OUT keyword that the UDR uses to pass information back to its caller. The following example shows a CREATE FUNCTION statement with an OUT the parameter:

```
CREATE FUNCTION stem(column LVARCHAR, OUT y CHAR)...
```

In an SQL statement, the argument that corresponds to the OUT parameter is called a *statement-local variable*, or SLV. The SLV argument appears as a variable name and pound sign (#), as the following example shows:

```
SELECT...WHERE stem(lname, y # CHAR)
```

The VTI includes functions to determine whether a qualification function includes an SLV argument and to manage its value. For more information about how the access method intercepts and sets SLVs, refer to the description of the **mi_qual_needoutput()** function on page 5-36 and the **mi_qual_setoutput()** function on page 5-40.

For more information about output parameters, the OUT keyword, and SLVs, refer to *Extending Informix Dynamic Server 2000*.

Host Variables

While a client application executes, it can calculate values and pass them to a function as an input parameter. Another name for the input parameter is *host variable*. In the SQL statement, a question mark (?) represents the host variable, as the following example shows:

```
SELECT...WHERE equal(lname, ?)
```

The SET parameter in following example contains both explicit values and a host variable:

```
SELECT...WHERE in(SET{'Smith', 'Smythe', ?}, lname)
```

Because the value of a host variable applies to every row in the table, the access method treats the host variable as a constant. However, the constant that the client application supplies might change during additional scans of the same table. The access method can request that the optimizer reevaluate the requirements of the qualification between scans.

For more information about how the access method provides for a host variable, refer to the description of **mi_qual_const_depends_hostvar()** and **mi_qual_setreopt()** in Chapter 5, "Descriptor Function Reference."

For more information about the following topics, refer to the manual that the table indicates.

Topic	Manual
Setting values for host variables in client applications	*Informix ESQL/C Programmer's Manual*
Using DataBlade API functions from client applications	*DataBlade API Programmer's Manual*
Using host variables in SQL statements	*Informix Guide to SQL: Syntax*

Negation

The NOT operator reverses, or negates, the meaning of a qualification. In the following example, the access method returns only rows with an **lname** value other than `SMITH`:

```
WHERE NOT lname = "SMITH"
```

NOT can also reverse the result of a Boolean expression. In the next example, the access method rejects rows that have `southwest` or `northwest` in the **region** column:

```
WHERE NOT (region = "southwest" OR region = “northwest”)
```

Complex Boolean Expressions

In a complex WHERE clause, Boolean operators combine multiple conditions. The following example combines a function with a complex qualification:

```
WHERE year > 95 AND (quarter = 1 OR quarter = 3)
```

The OR operator combines two functions, `equal(quarter,1)` and `equal(quarter,3)`. If either is true, the combination is true. The AND operator combines the result of the `greaterthan(year,95)` with the result of the Boolean OR operator.

If a WHERE clause contains multiple conditions, the database server constructs a qualification descriptor that contains multiple, nested qualification descriptors.

Figure 3-7 shows a complex WHERE clause that contains multiple levels of qualifications. At each level, a Boolean operator combines results from two previous qualifications.

```
WHERE region = "southwest" AND
    (balance < 90 OR aged <= 30)
```

Figure 3-7
Complex WHERE Clause

Figure 3-8 and Figure 3-9 represent the structure of the qualification descriptor that corresponds to the WHERE clause in Figure 3-7.

```
AND((equal(region,'southwest'),
    OR(lessthan(balance,90), lessthanequal(aged,30)))
```

Figure 3-8
Function Nesting

The qualification descriptors for the preceding expression have a hierarchical relationship, as the following figure shows.

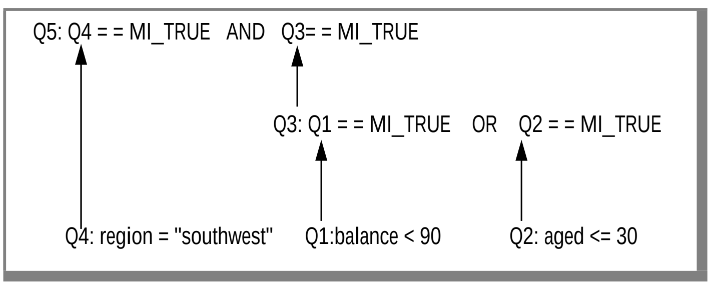


Figure 3-9
Qualification-Descriptor Hierarchy

For a detailed description of the functions that the access method uses to extract the WHERE clause conditions from the qualification descriptor, refer to "Qualification Descriptor" on page 5-7.

Qualifying Data

An access method can use one or more of the following resources to qualify or disqualify each source record or row:

- Pass the row to database server for evaluation
- Evaluate the source data inside the access-method
- Send part or all of the query to external software

Qualification by the Database Server

The optimizer does not create a qualification descriptor if the cost for the access method to qualify rows exceeds the cost for a full table scan. If the database serve does not construct a qualification descriptor, the **mi_scan_quals()** function returns a NULL-valued pointer.

Important: *The* ***mi_scan_quals()*** *function returns a NULL-valued pointer to indicate that* a *qualification descriptor does not exist. In response to the NULL-valued pointer, the access method creates a row from each source record.*

Qualification by the Access Method

An access method might perform all the qualification tests or it might examine some of the values that a WHERE clause specifies to partially qualify rows.

Guidelines for Implementation

An access method might create a row from each source record and pass the row to the database server for evaluation. However, each call to **mi_row_create()** to format a row or to **mi_eval_am_qual()** to have the database server evaluate the row can reduce performance. A developer might use this simple approach for low-volume data.

If possible, an access method evaluates the entire WHERE clause to eliminate unqualified source records. For each candidate record that it cannot disqualify, the access method calls **mi_row_create()** and **mi_eval_am_qual()** functions, which causes the database server to fill in any missing results in the qualification descriptor. For an example of this approach, refer to "Processing Complex Qualifications" on page 3-29.

Ideally, the access method only formats values that the query projects and fills the remaining columns with NULL values. To determine which columns contain the values that the query requires, the access method calls the **mi_scan_nprojs()** and **mi_scan_projs()** functions.

Executing Qualification Functions

This section describes the following alternative ways to process a simple function:

- To execute a function in a database server thread, use the routine identifier.
- To enable the access method or external software to execute an equivalent function, use the function name.

Using the Routine Identifier

The access method uses a DataBlade API facility called *FastPath* to execute registered UDRs that do not reside in the same shared-object module as the access-method functions. To use the FastPath facility, the access method performs the following general steps:

1. Call the **mi_qual_funcid()** accessor function to obtain the routine identifier
2. Pass the routine identifier to the DataBlade API **mi_func_desc_by_typeid()** function, which returns the function descriptor.
3. Pass the function descriptor to the DataBlade API **mi_routine_exec()** function.

For complete information about FastPath functions and the function descriptor (MI_FUNC_DESC), see the *DataBlade API Programmer's Manual*.

Tip: *You can obtain the function descriptor in the* ***am_beginscan*** *purpose function, store the function descriptor in the PER_COMMAND user data, and call* ***mi_scan_setuserdata()*** *to store a pointer to the user data. In the* ***am_getnext*** *purpose function, call* ***mi_scan_userdata()*** *to retrieve the pointer, access the function descriptor, and execute the function with* ***mi_routine_exec()****. For examples, follow the link to the DataBlade Corner on the Informix Developer Network Web page, www.informix.com/idn.*

Using the Function Name

To extract the function name from the qualification descriptor, the access method calls the **mi_qual_funcname()** accessor function.

You can use **mi_qual_funcname()** to identify the function in a qualification then directly call a local routine that implements it. For example, if access method contains a local **equal()** function, it might include the following condition:

```
/* Compare function name to string.*/
if (strcmp("equal", mi_qual_funcname(qd)) == 0)
{ /* Execute equal() locally. */ }
```

An access method can also use the **mi_qual_funcid()** function if external software controls the data. The access method uses this and other accessor functions to extract information from qualification descriptor into a form that the external software can interpret. For a demonstration access method that parses and passes a qualification to external software, follow the link to the DataBlade Corner on the Informix Developer Network Web page, www.informix.com/idn.

Processing Complex Qualifications

In Figure 3-10 on page 3-30, the **am_getnext** purpose function attempts to disqualify source records. It creates rows for fully qualified source records and for those that it cannot disqualify.

```
mi_integer sample_getnext(sd,retrow,retrowid)
    MI_AM_SCAN_DESC *sd;
    MI_ROW          **retrow
    mi_integer      retrowid;
{
    my_data_t    *my_data;
    MI_ROW_DESC    *rd;
    MI_AM_TABLE_DESC *td;
    MI_AM_QUAL_DESC  *qd;
    td = mi_scan_table(sd); /* Get table descriptor. */
    rd = mi_tab_rowdesc(td); /* Get column data types. */
    my_data = (my_data_t *)mi_tab_userdata(td); /* Get pointer to user data.*/
    MI_DATUM      qdvalue;
    /* Evaluate records until one qualifies for return to caller.. */
    for (;;)
    {
        if ( ! my_data ) return MI_NO_MORE_RESULTS;
        get_results(qd, mi_data);
        qdvalue = mi_qual_value(qd)
        if (qdvalue == MI_VALUE_TRUE)
        {
            /*Create MI_ROW and return it to the database server. */
            *retrow = mi_row_create(...);
            return MI_ROWS;
        }
        else if (qdvalue == MI_VALUE_NOT_EVALUATED)
         {
            /*Create MI_ROW and return it to the database server. */
            *retrow = mi_row_create(...);
            if (mi_eval_am_qual(retrow, qd);==MI_VALUE_TRUE
                return MI_ROWS;
        }
        /* Either get_result() or mi_eval_am_qual() returned MI_VALUE_FALSE. */
        mi_init_am_qual(qd); /* Reset qualification descriptor */
        my_data->rowptr++;
    } /*End loop.*/
}/* End getnext.*/
```

Figure 3-10
Sample am_getnext Purpose Function

In Figure 3-11, the **get_result()** function loops recursively through the qualification descriptor, looking for simple qualifications that the access method knows how to evaluate. It sets results for the simple qualifications and leaves MI_VALUE_NOT_EVALUATED in the Boolean-operator portions of the qualification descriptor.

Tip: *The examples in this section do not illustrate the code that the access method uses to execute functions. For information about executing functions, refer to* *"Executing Qualification Functions" on page 3-28*.

```
... get_result(qd, my_data)
    MI_AM_QUAL_DESC*qd;
    user_data_t    *my_data
{
    if (mi_qual_issimple(qd))
    {
        /* Execute simple, function. (Not shown.) */
        /* Test the result that the function returns. */
        if (result == MI_TRUE)
        {
            /* Set result in qualification descriptor.*/
            mi_qual_setvalue(qd,MI_VALUE_TRUE);
            return; ;
        }
        else
        {
            mi_qual_setvalue( qd,MI_VALUE_FALSE);
            return;;
        }
    } /* END: if (mi_qual_issimple(qd)) */
    else
    { /* Complex qualification (has AND or OR)..Loop until all functions execute.*/
        for (i = 0; i < mi_qual_nquals(qd); i++)
            get_result(mi_qual_qual(qd, i), my_data)
    } /* END: Complex qualification (has AND or OR) */
    return;;
```

Figure 3-11
Setting Results in the Qualification Descriptor

Qualification by External Software

If required, an access method can pass a qualification to external software. To exchange information with external software, the access method must manage communication. To obtain a demonstration access method that communicates with external software, follow the link to the DataBlade Corner on the Informix Developer Network Web page, www.informix.com/idn.

Supporting Query-Plan Evaluation

At the start of a SELECT statement, the database server initiates query planning. A *query plan* specifies the steps that the database server takes to fulfill a query with optimal efficiency. The database server includes an optimizer, which compares various combinations of operations and chooses the query plan from among alternative approaches. To help the optimizer select the best query plan, provide reliable information about the cost for using the access method to select data.

Calculating Statement-Specific Costs

The optimizer compares the cost in time and memory to perform such tasks as the following:

- Locating an index entry or table row on disk
- Retrieving the entry or row into memory
- Sorting and joining data
- Applying WHERE clause qualifications
- Retrieving rows from a primary table, if the optimizer uses an index

For more information about query plans, refer to the *Performance Guide*.

If the query involves a user-defined access method, the database server executes the **am_scancost** purpose function to request cost information from the access method. For a description of the factors that **am_scancost** calculates, refer to page 4-34.

To avoid error messages, the access method can use the **am_scancost** purpose function to notify the optimizer when it does not support all the requirements specified in a query. If necessary, **am_scancost** can return a negative cost so that the optimizer excludes this access method from the query plan.

Updating Statistics

The UPDATE STATISTICS statement stores statistics about the distribution of rows on physical storage media for use by the optimizer. The database server updates data-distribution statistics for internal, relational tables; the access method updates data-distribution statistics for virtual tables. When a user issues an UPDATE STATISTICS statement that requires the access method to determine the distribution of data in a table, the database server calls the **am_stats** purpose function.

The access method can call **mi_tab_update_stat_mode()** to determine if the UPDATE STATISTICS statement includes the keyword HIGH or MEDIUM, each of which influences the percentage of rows that the access method should sample and the particular statistics that it should supply.

To store statistics in the statistics descriptor, the **am_stats** purpose function calls the various accessor functions with the name prefix **mi_tstats_set**. The database server copies the information from the statistics descriptor in the appropriate system catalog tables. For information about these functions, refer to Chapter 5, "Descriptor Function Reference."

For information about how to access the system catalog tables or to maintain tables in an Informix database, refer to "Accessing Database and System Catalog Tables" on page 3-7. For information about the effects of query costs and distribution of data, refer to the *Performance Guide*.

Enhancing Performance

The access method can take advantage of the following performance enhancements:

- Executing parallel scans, inserts, deletes, and updates
- Buffering multiple rows

Executing in Parallel

Parallelizable routines can execute in parallel across multiple processors.

To make a UDR parallelizable, apply the following rules:

- Follow the guidelines for well-behaved user-defined routines.
- Avoid any DataBlade API routine that involves query processing (**mi_exec()**, **mi_exec_prepared_statement()**), collections (**mi_collection_***), row types, or save sets (**mi_save_set_***).
- Do not create rows that contain any complex types including another row type as one of the columns. Do not use the **mi_row_create()** or **mi_value()** functions with complex types or row types.
- Avoid DataBlade API FastPath functions (**mi_routine_***, **mi_func_desc_by_typeid()**) if the access method might pass them routine identifiers for nonparallelizable routines.
- Specify the PARALLELIZABLE routine modifier in the CREATE FUNCTION or CREATE PROCEDURE statement for the UDR.

For more information about the following topics, refer to the *DataBlade API Programmer's Manual*:

- Guidelines for well-behaved user-defined routines
- A complete list of nonparallelizable functions
- FastPath function syntax, usage, and examples

For more information about the PARALLELIZABLE (and other) routine modifiers, refer to the Routine Modifier segment in the *Informix Guide to SQL: Syntax*. For more information about parallelizable UDRs, refer to *Extending Informix Dynamic Server 2000*.

To make an access method parallelizable

1. Create a *basic set* of parallelizable purpose functions.

   The basic set, which enables a SELECT statement to execute in parallel, includes the following purpose functions: **am_open**, **am_close**, **am_getbyid**, **am_beginscan**, **am_endscan**, **am_getnext**, and **am_rescan**.

   An access method might not supply all of the purpose functions that define a basic parallelizable set. As long as you make all the basic purpose functions that you provide parallelizable, a SELECT statement that uses the access method can execute in parallel.

2. Add a parallelizable purpose function to the basic set for any of the following actions that you want the database server to execute in parallel.

Parallel SQL Statement	Parallelizable Purpose Function
INSERT (in a SELECT)	**am_insert**
SELECT INTO TEMP	**am_insert**
DELETE	**am_delete**
UPDATE	**am_update**

***Important:** A parallelizable purpose function must call only routines that are also parallelizable.*

The database server sets an **am_parallel** purpose value in the **sysams** system catalog table to indicate which access-method actions can occur in parallel. For more information, refer to "Purpose Options" on page 6-9.

Buffering Multiple Results

The **am_getnext** purpose function can find and store several qualified rows in shared memory before it returns control to the database server. The following steps set up and fill a multiple-row buffer in shared memory:

1. Call **mi_tab_setniorows()** in **am_open** or **am_beginscan** to set the number of rows that the access method can return in one scan.
2. Call **mi_tab_niorows()** at the start of **am_getnext** to find out how many rows to return.
3. Loop through **mi_tab_setnextrow()** in **am_getnext** until the number of qualifying rows matches the return value of **mi_tab_niorows()** or until no more qualifying rows remain.

Figure 3-12 shows the preceding steps. For more information about these functions, refer to Chapter 5, "Descriptor Function Reference."

```
mi_integer sample_beginscan(MI_AM_SCAN_DESC *sd)
{
    mi_integer     nrows = 512;
    MI_AM_TABLE_DESC *td=mi_scan_table(sd);
    mi_tab_setniorows(td, nrows);
}

mi_integer sample_getnext(MI_AM_SCAN_DESC  *sd, MI_ROW **retrow,
                   mi_integer *rowid
)
{
    mi_integer     nrows, row, nextrowid;
    MI_ROW        *nextrow; /* MI_ROW structure is not typically used.*/

    MI_AM_TABLE_DESC *td =mi_scan_table(sd);
    nrows = mi_tab_niorows(td);

    if (nrows > 0)
    {/*Store qualified results in shared-memory buffer.*/
        for (row = 0; row < nrows; ++row)
        {   /* Evaluate rows until we get one to return to caller. */
            find_good_row(sd, &nextrow, &nextrowid);
            *retrow=nextrow;
            mi_tab_setnextrow(td, nextrow, nextrowid);
        } /* End of loop for nrows times to fill shared memory.*/
    }/*End (nrows > 0). */
    else
    {/*Only one result per call to am_getnext. */
        find_good_row(sd, &nextrow, &nextrowid);
        *retrow = nextrow;

        *retrowid = nextrowid
    }
    /* When reach the end of data, return MI_NO_MORE_RESULTS, else return MI_ROWS. */
}
```

Figure 3-12
Storing Multiple Results In a Buffer

Supporting Data Retrieval, Manipulation, and Return

The following topics affect the design of **am_getnext**, **am_insert**, **am_delete**, and **am_update**:

- Checking isolation levels
- Converting data to and from Informix row format
- Detecting transaction success or failure

Checking Isolation Levels

The isolation level affects the concurrency between sessions that access the same set of data. The following tables show the types of phenomena that can occur without appropriate isolation-level controls.

- A *dirty read* occurs because transaction 2 sees the uncommitted results of transaction 1.

Transaction 1	Write (a)		Rollback
Transaction 2		Read (a)	

- A *nonrepeatable read* occurs if transaction 1 retrieves a different result from the each read.

Transaction 1	Read (a)			Read (a)
Transaction 2		Write or Delete (a)	Commit	

- A *phantom read* occurs if transaction 1 obtains a different result from each SELECT for the same criteria.

Transaction 1	Select (criteria)			Select (criteria)
Transaction 2		Update/Create (match to criteria)	Commit	

To determine which of the following isolation levels the user or application specifies, the access method can call either the **mi_tab_isolevel()** or **mi_scan_isolevel()** function.

Isolation Level	Type of Read Prevented
Serializable	Dirty read, nonrepeatable read, and phantom read
Repeatable Read or Cursor Stability	Dirty read and nonrepeatable read
Read Committed	Dirty read
Read Uncommitted	None

If an access method does not support Serializable isolation for data in an extspace, an update by another transaction can change data on disk after the access method sends the same row to the database server. The disk data no longer matches the data that the database server placed in shared memory.

For more information about how applications use isolation levels, consult the *Informix Guide to SQL: Reference*, *Informix Guide to SQL: Syntax*, and *Informix Guide to SQL: Tutorial*. For information about determining isolation level, refer to **mi_scan_isolevel()** or **mi_tab_isolevel()** in Chapter 5, "Descriptor Function Reference."

The database server automatically enforces Repeatable Read isolation under the following conditions:

- The virtual table resides in sbspaces.
- User-data logging is turned on for the smart large objects that contain the data.

  To find out how to turn on user-data logging with the access method, refer to "Activating Automatic Controls in Sbspaces" on page 3-16. To find out how to provide for logging with ONCONFIG parameters, refer to your *Administrator's Guide*.

The access method must provide the code to enforce isolation levels under the following circumstances:

- Users require Serializable isolation.

  The database server does not provide support for full Serializable isolation.
- Some or all of the data resides in extspaces.

Important: *You must document the isolation level that the access method supports in a user guide. For an example of how to word the isolation-level notice, refer to* *Figure 3-13 on page 3-44*.

Converting to and from Row Format

Before the access method can return row values to a query, the access method must convert source data to data types that the database server recognizes, native Informix data types and user-defined data types (UDTs). The database server can recognize a UDT because the application registers it in the database with a CREATE TYPE statement.

To create a row

1. Call **mi_tab_rowdesc()** to retrieve the row descriptor.
2. Call the appropriate DataBlade API row-descriptor accessor functions to obtain the information, such as data type, for each column.

   For a list of available row-descriptor accessor functions, refer to the description of MI_ROW_DESC in the *DataBlade API Programmer's Manual*.
3. Call **mi_scan_nprojs()** and **mi_scan_projs()** to determine which columns the query specifies.
4. If necessary, convert external data types to types that the database server recognizes.
5. Set the value of the columns that the query does not need to NULL.
6. Call the DataBlade API **mi_row_create()** function to create a row from the converted source data.

The database server passes an MI_ROW structure to the **am_insert** and **am_update** purpose functions. To extract the values to insert or update, call **mi_value()** or **mi_value_by_name()**. For more information about these functions, refer to the *DataBlade API Programmer's Manual*.

Determining Transaction Success or Failure

The access method can register an end-of-transaction callback function to handle the MI_EVENT_END_XACT event, which the database server raises at the end of a transaction. In that callback function, test the return value of the DataBlade API **mi_transition_type()** function to determine the state of the transaction, as follows.

Return Value for mi_transition_type()	Transaction State
MI_NORMAL_END	Successful transaction completion The database server can commit the data.
MI_ABORT_END	Unsuccessful transaction completion The database server must roll back the table to its state before the transaction began.

Warning: *Informix does not ensure uniform commit or rollback (called two-phase-commit protocol) with data in an external database server. If a transaction partially commits and then aborts, inconsistencies can occur between the database server and external data.*

As long as a transaction is in progress, the access method should save each original source record value before it executes a delete or update. For transactions that include both internal and external objects, the access method can include either an end-of-transaction or end-of-statement callback function to ensure the correct end-of-transaction action. Depending on the value that **mi_transition_type()** returns, the callback function either commits or rolls back (if possible) the operations on the external objects.

If an external transaction does not completely commit, the access method must notify the database server to roll back any effects of the transaction on state of the virtual table.

For detailed information about the following items, refer to the *DataBlade API Programmer's Manual*:

- Handling state-transitions in a UDR
- End-of-transaction callback functions
- End-of-statement callback functions

Supplying Error Messages and a User Guide

As you plan access-method purpose functions, familiarize yourself with the following information:

- The SQL statement syntax in the *Informix Guide to SQL: Syntax*
- The *Informix Guide to SQL: Tutorial* and the *Informix Guide to Database Design and Implementation*

These documents include examples of Informix SQL statements and expected results, which the SQL user consults.

The user of your access method will expect the SQL statements and keywords to behave as documented in the database server documentation. If the access method causes an SQL statement to behave differently, you must provide access-method documentation and messages to alert the user to these differences.

In the access-method user guide, list all SQL statements, keywords, and options that raise an exception if an end user attempts to execute them. Describe any features that the access method supports in addition to the standard SQL statements and keywords.

Create callback functions to respond to database server exceptions, as "Handling the Unexpected" on page 3-8 describes. Raise access-method exceptions for conditions that the database server cannot detect. Use the following sections as a checklist of items for which you supply user-guide information, callback functions, and messages.

Avoiding Database Server Exceptions

When an SQL statement involves the access method, the database server checks the purpose settings in the **sysams** system catalog table to determine whether the access method supports the statement and the keywords within that statement.

The database server issues an exception and an error message if the purpose settings indicate that the access method does not support a requested SQL statement or keyword. If a user inadvertently specifies a feature that the access-method design purposely omits and the SQL syntax conforms to the *Informix Guide to SQL: Syntax*, the documentation does not provide a solution.

Specify access-method support for the following items in the **sysams** system catalog table with a CREATE PRIMARY ACCESS_METHOD or ALTER ACCESS_METHOD statement:

- Statements
- Keywords
- Storage space type

Statements That the Access Method Does Not Support

For each SQL statement that the access method supports, **sysams** contains the name of the corresponding purpose function.

The user can receive an SQL error for statements that require a purpose function that you did not supply. For example, an UPDATE STATISTICS statement fails if the access method does not include an **am_stats** purpose function. The access-method user guide must advise users which statements to avoid.

For statements that alter data, you must also set the **am_readwrite** purpose flag. If the access method does not supply write support, the access-method user guide must advise users not to use any of the following statements: INSERT, UPDATE, DELETE, and ALTER FRAGMENT.

Keywords That the Access Method Does Not Support

You must set a purpose flag to indicate the existence of code within the access method to support certain keywords. If a purpose flag is not set, the database server assumes that the access method does not support the corresponding keyword and issues an error if an SQL statement specifies that keyword.

For example, unless the **am_cluster** purpose flag is set in the **sysams** system catalog table, an SQL statement with the CLUSTER keyword fails. If the access method does not provide for clustering, the access-method user guide must advise users not to use the CLUSTER keyword.

Storage Spaces and Fragmentation

An SQL statement fails if it specifies a storage space that does not agree with the **am_sptype** purpose value in the **sysams** system catalog table. In the user guide, specify whether the access method supports sbspaces, extspaces, or both. Advise the user how to do the following:

- Create sbspace or extspace names with the **onspaces** command
- Specify a default sbspace if the access method supports sbspaces
- Locate the default extspace if the access method creates one
- Specify an IN clause in a CREATE TABLE or ALTER FRAGMENT statement

For more information about specifying storage spaces, refer to "Creating and Specifying Storage Spaces" on page 2-13.

If the access method supports fragmentation in sbspaces, advise the user to create multiple sbspaces with **onspaces** before issuing an SQL statement that creates fragments. For an example, refer to "Using Fragments" on page 2-16.

Features That the Interface Does Not Support

The database server also raises exceptions due to restrictions that the VTI imposes on SQL. A user cannot specify a dbspace in a CREATE TABLE or ALTER FRAGMENT statement. The VTI does not support the following statements for virtual tables:

- An ALTER TABLE statement that adds, drops, or modifies a column
- A LOCK TABLE or UNLOCK TABLE statement
- An ATTACH or DETACH keyword in an ALTER FRAGMENT statement

Notifying the User About Access-Method Constraints

The database server cannot detect unsupported or restricted features for which the **sysams** system catalog table has no setting.

Specify any precautions that an application might require for isolation levels, lock types, and logging.

Advise users whether the access method handles logging and data recovery. Notify users about parameters that they might set to turn logging on. For an example, refer to Figure 3-5 on page 3-18.

Provide the precise wording for the isolation levels that the access method supports. Informix recommends that you use standard wording for isolation level. The following example shows the language to define the ways in which the qualifying data set might change in the transaction.

```
The access method fully supports the ANSI Repeatable Read level
of isolation. The user need not account for dirty reads or
nonrepeatable reads. Informix recommends precautions against
phantom reads.
```

Figure 3-13
Sample Language to Describe Isolation Level

Documenting Nonstandard Features

Provide instructions and examples for any feature that aids the user in applying the access method. For example, provide information and examples about the following items:

- Parameter keywords

  For examples, refer to "Providing Configuration Keywords" on page 3-18.

- Output from the **oncheck** utility

  For more information about the options that the **oncheck** provides, refer to the *Administrator's Reference*. For more information about providing **oncheck** functionality, refer to the description of the **am_check** purpose function on page 4-15.

Chapter

Purpose-Function Reference

4

In This Chapter

This chapter describes the purpose functions that the access-method developer provides. This chapter consists of two major parts:

- "Purpose-Function Flow" illustrates the sequence in which the database server calls purpose functions.
- "Purpose-Function Syntax" on page 4-12 specifies the predefined function-call syntax and suggests usage for each purpose function.

Purpose-Function Flow

The diagrams in this section show, for each SQL statement, which purpose functions the database server executes. Use the diagrams to determine which purpose functions to implement in the access method.

The complexity of the purpose-function flow for each statement determines the order in which the statement appears in this section. The following sections describe the purpose-function interface for SQL statements:

- "CREATE Statement Interface" on page 4-4
- "DROP Statement Interface" on page 4-5
- "SELECT...WHERE Statement Interface" on page 4-5
- "INSERT, DELETE, and UPDATE Statement Interface" on page 4-6
- "ALTER FRAGMENT Statement Interface" on page 4-8

Also see "oncheck Utility Interface" on page 4-12.

***Tip:** The database server invokes the **am_open** and **am_close** purpose functions once per fragment for first SQL statement that references a new virtual table. After the initial calls to **am_open** and **am_close**, the database server resumes the normal purpose function flow for the active SQL statement.*

The following statements result in an additional call to **am_open** and **am_close** before the INSERT statement:

```
CREATE TABLE newtab (...) USING myvti
INSERT INTO newtab VALUES (....)
```

CREATE Statement Interface

Figure 4-1 shows the order in which the database server executes purpose functions for a CREATE TABLE statement. If the IN clause specifies multiple storage spaces to fragment the table, the database server repeats the sequence of purpose functions that Figure 4-1 shows for each storage space.

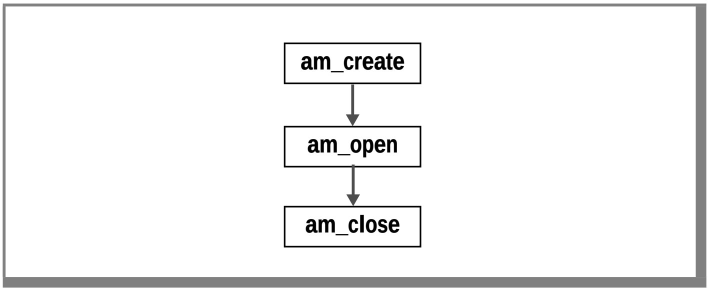


Figure 4-1
Processing a CREATE TABLE Statement

For more information about implementing the CREATE TABLE statement in the access method, refer to "Supporting Data Definition Statements" on page 3-11.

DROP Statement Interface

Figure 4-2 shows the processing for each fragment of a DROP TABLE or DROP DATABASE statement.

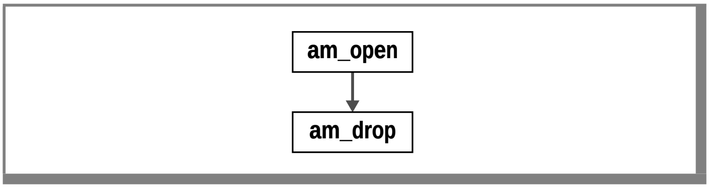


Figure 4-2
Processing a DROP Statement

SELECT...WHERE Statement Interface

Figure 4-3 shows the order in which the database server executes purpose functions for a SELECT statement with a WHERE clause. For information about how to process the scan and qualifications, refer to "Processing Queries" on page 3-20.

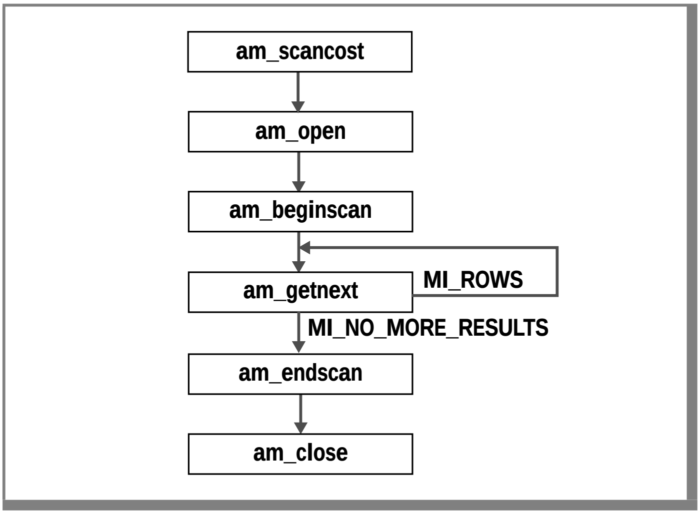


Figure 4-3
Processing a SELECT Statement Scan

INSERT, DELETE, and UPDATE Statement Interface

Figure 4-4 shows the order in which the database server executes purpose functions to insert, delete, or update a row at a specific physical address. The physical address consists of fragment identifier and row identifier.

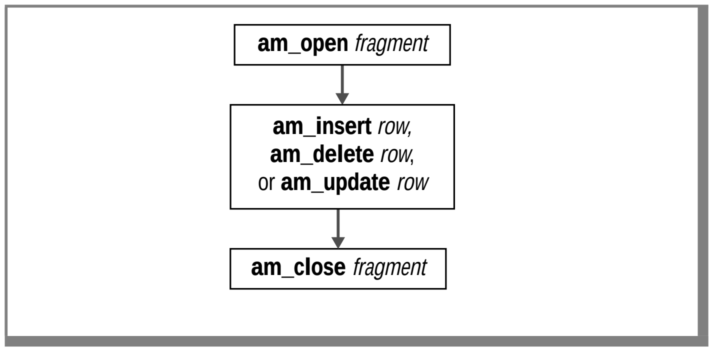


Figure 4-4
INSERT, DELETE, or UPDATE by Row Address

Figure 4-5 shows the order in which the database server executes purpose functions if the insert, delete, or in-place update has an associated WHERE clause.

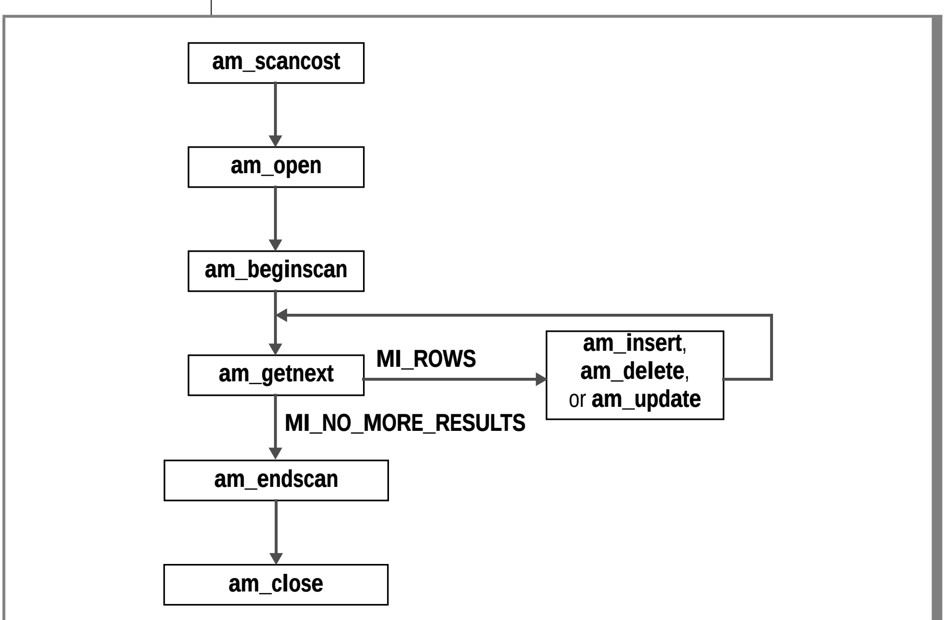


Figure 4-5
INSERT, DELETE, or UPDATE in a Subquery

Figure 4-6 shows the more complicated case in which **am_getnext** returns multiple rows to the database server. In either case, the database server calls **am_insert**, **am_delete**, or **am_update** once per row.

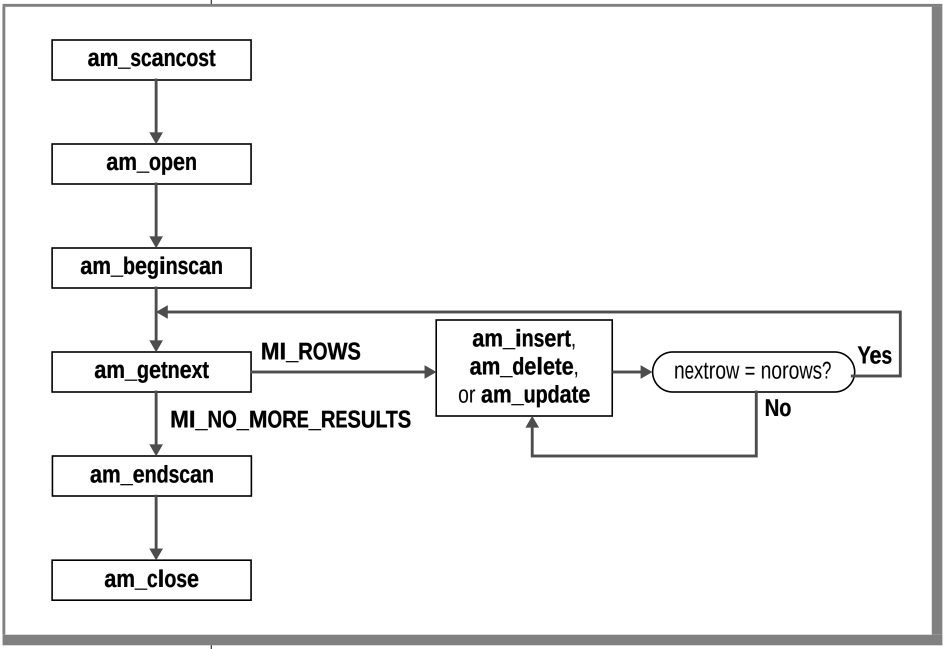


Figure 4-6
Returning Multiple Rows That Qualify for INSERT, DELETE, or UPDATE

For more information about implementing INSERT, DELETE, and UPDATE statements, refer to "Supporting Data Retrieval, Manipulation, and Return" on page 3-37.

ALTER FRAGMENT Statement Interface

When the database server executes an ALTER FRAGMENT statement, the database server moves data between existing fragments and also creates a new fragment.

The statement in Figure 4-7 creates and fragments a **jobs** table.

```
CREATE TABLE jobs (sstatus file_ops)
    FRAGMENT BY EXPRESSION
        sstatus > 15 IN fragspace2,
        REMAINDER IN fragspace1
    USING file_am
```

Figure 4-7
SQL to Create the Fragmented Jobs Table

The statement in Figure 4-8 changes the fragment expression for **jobs**, which redistributes the table entries.

```
ALTER FRAGMENT ON TABLE jobs
    MODIFY fragspace1 TO (sstatus <= 5) IN fragspace1,
    MODIFY fragspace2 TO
        (sstatus > 5 AND sstatus <= 10) IN fragspace2,
    REMAINDER IN fragspace3
```

Figure 4-8
SQL to Alter the Jobs Fragments

For each fragment that the ALTER FRAGMENT statement specifies, the database server performs the following actions:

1. Executes an access-method scan
2. Evaluates the returned rows to determine which ones must move to a different fragment
3. Executes the access method to create a new fragment for the target fragment that does not yet exist
4. Executes the access method to delete rows from one fragment and insert them in another

Figures 4-9 through Figure 4-12 show the separate sequences of purpose functions that create the fragments and distribute the data for the SQL ALTER FRAGMENT statement in Figure 4-8. The database server performs steps 1, 2, and 3 to move fragments from **fragspace1** to **fragspace2** and then performs steps 1 through 3 to move fragments from **fragspace2** to **fragspace3**.

Figure 4-9 shows the sequential scan in step 1, which returns all rows from the fragment because the scan descriptor contains a NULL-valued pointer instead of a pointer to a qualification descriptor.

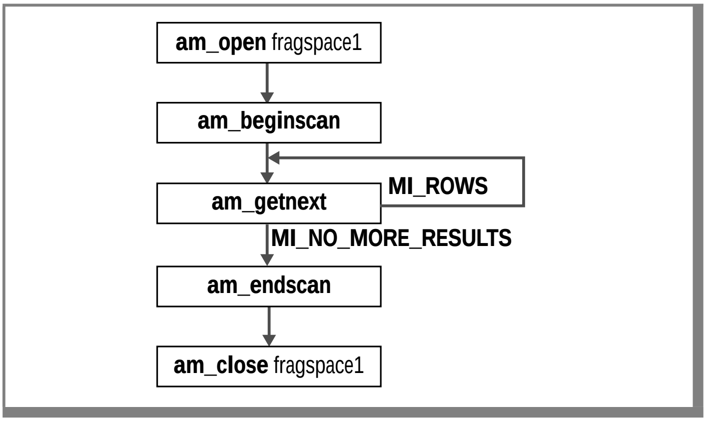


Figure 4-9
Getting All the Rows in Fragment 1

In Figure 4-10, the database server returns the row identifiers that the access method should delete from **fragspace1** and insert in **fragspace2**.

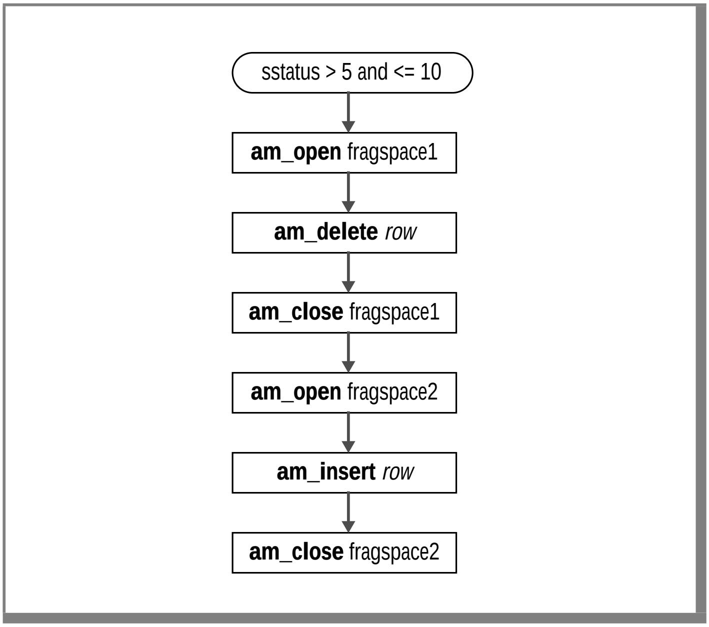


Figure 4-10
Moving Rows Between Fragments

Figure 4-11 again shows the sequential scan in step 1. This scan returns all the rows from **fragment2**.

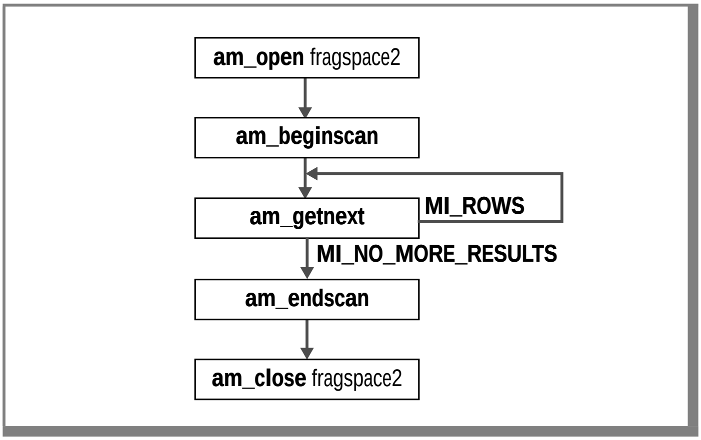


Figure 4-11
Getting All the Rows in Fragment 2

Figure 4-12 shows steps 3 and 4. The database server returns the row identifiers that the access method should delete from **fragspace2** and insert in **fragspace3**. The database server does not have **fragspace3**, so it executes **am_create** to have the access method create a fragment before it executes **am_insert**.

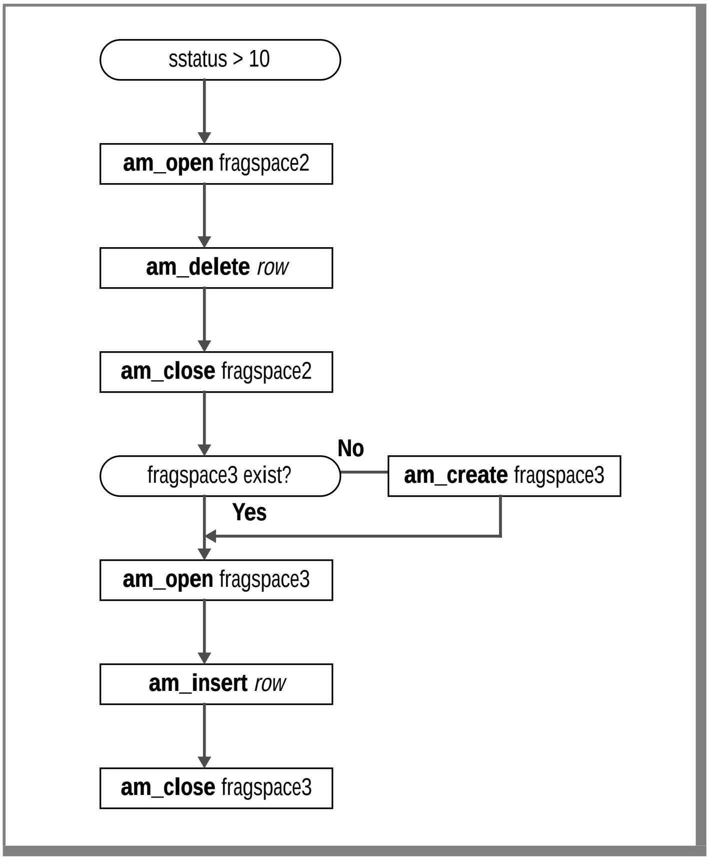


Figure 4-12
Adding and Filling a Fragment

For more information about fragments that a VTI-based access method manages, refer to "Supporting Fragmentation" on page 3-17.

oncheck Utility Interface

The **oncheck** utility reports on the state of a table and provides a means for a database system administrator to check on the state of objects in a database. You, as an access-method developer, can also use **oncheck** to verify that the access method creates and maintains appropriate tables.

As Figure 4-13 shows, the database server calls only one access-method function for the **oncheck** utility. If necessary, the **am_check** purpose function can call **am_open** and **am_close** or can itself contain the appropriate logic to obtain handles, allocate memory, and release memory.

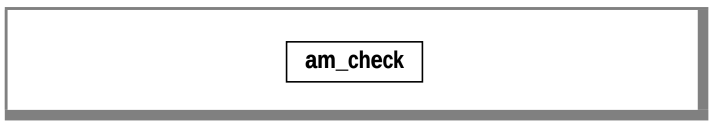


Figure 4-13
Processing the oncheck Utility

Purpose-Function Syntax

The database server expects a particular prototype for each purpose function. As the access-method developer, you program the actions of a purpose function but must use the parameters and return values that the VTI prototypes specify. This section lists purpose-function prototypes in alphabetical order.

For each purpose function that your access method provides, use the prototype that this chapter shows but change the prototype-function name to a unique name. For example, you might save your version of **am_open** with the name **vtable_open()**. To associate the unique purpose-function names to the corresponding prototype names, use the CREATE PRIMARY ACCESS_METHOD statement, as "CREATE ACCESS_METHOD" on page 6-6 specifies.

The parameter list for each purpose function includes (by reference) one or more *descriptor* data structures that describe the SQL statement keywords or **oncheck** options and the specified table that require the access method. For detailed information about each descriptor, refer to "Descriptors" on page 5-6.

Purpose functions are simply entry points from which the access method calls other routines from the access-method library, DataBlade API functions, and the VTI functions that "Accessor Functions" on page 5-16 describes.

am_beginscan

The database server calls **am_beginscan** to start a scan on a virtual table. This function initializes the scan.

Syntax

```
mi_integer am_beginscan(MI_AM_SCAN_DESC *scanDesc)
```

scanDesc points to the scan descriptor.

Usage

The functions that the access method supplies for **am_beginscan**, **am_getnext**, and **am_endscan** compose the main scan-management routines. In its turn, the **am_beginscan** purpose function might perform the following operations:

- Obtain the qualification descriptor from the scan descriptor
- Parse the criteria in the qualification descriptor

  For a more detailed discussion, refer to "Processing Queries" on page 3-20.
- Determine the need for data type conversions to process qualification expressions
- Based on the information in the qualification descriptor, initiate a search for data that fulfills the qualification
- Allocate PER_COMMAND memory to build user data and then store the user data in the scan descriptor for the **am_getnext** function

  For more information about memory allocation, refer to "Storing Data in Shared Memory" on page 3-4.

You can also choose to defer any processing of qualifications until the **am_getnext** function.

Return Values

MI_OK indicates success. MI_ERROR indicates failure.

Related Topics

See the descriptions of:

- purpose functions **am_endscan**, **am_getnext**, and **am_rescan.**
- “Optimizing Queries” on page 2-7.

am_check

If a user executes the **oncheck** utility for a virtual table, the database server calls **am_check**.

Syntax

```
mi_integer am_check(MI_AM_TABLE_DESC *tableDesc,
    mi_integer option)
```

tableDesc points to the table descriptor of the table that the current **oncheck** command specifies.

option contains an encoded version of the current command-line option string for the **oncheck** utility.

Usage

A user, generally a system administrator or operator, runs the **oncheck** utility to verify physical data structures. The options that follow the **oncheck** command indicate the kind of checking to perform. For information about **oncheck** options, refer to the *Administrator's Reference***.**

In response to an **oncheck** command, the database server calls the **am_check** purpose function, which checks the internal consistency of the table and returns a success or failure indicator. If appropriate, **am_check** can call the **am_open** and **am_close** purpose functions.

Interpreting Options

To determine the exact contents of the command line, pass the *option* argument to the following VTI macros. Each macro returns a value of MI_TRUE if the *option* includes the particular **-c** or **-p** qualifier that the following table shows.

Macro	Option	oncheck Action
MI_CHECK_DATA() MI_DISPLAY_DATA()	**-cd** **-pd**	Check and display data rows, but not simple or smart large objects
MI_CHECK_DATA_BLOBS() MI_DISPLAY_DATA_BLOBS()	**-cD** **-pD**	Check and display data rows, simple large objects, and smart-large-object metadata
MI_CHECK_EXTENTS() MI_DISPLAY_EXTENTS()	**-ce** **-pe**	Check and display chunks and extents, including sbspaces
MI_DISPLAY_TPAGES()	**-pp**	Check and display pages by table or fragment
MI_DISPLAY_CPAGES()	**-pP**	Check and display pages by chunk
MI_DISPLAY_SPACE()	**-pt**	Check and display space usage

The **am_check** purpose function executes each macro that it needs until one of them returns MI_TRUE. For example, the following syntax tests for **oncheck** option **-cD**:

```
if (MI_CHECK_EXTENTS(option) == MI_TRUE)
{
    /* Check rows and smart-large-object metadata
     * If problem exists, issue message.      */
}
```

Checking and Displaying Table State

The access method can call accessor function **mi_tab_spacetype()** to determine whether the specified table resides in an sbspace or extspace. If the data resides in an sbspace, the **am_check** purpose function can duplicate the expected behavior of the **oncheck** utility. For information about the behavior for each **oncheck** option, refer to the *Administrator's Reference*.

For an extspace, such as a file that the operating system manages, **am_check** performs tasks that correspond to the command-line option.

To provide detailed information about the state of the table, **am_check** can call the **mi_tab_check_msg()** function.

Return Values

MI_OK validates the table structure as error free.

MI_ERROR indicates the access method could not validate the table structure as error free.

Related Topics

See the descriptions of:

- purpose functions **am_open** and **am_close.**
- accessor function **mi_tab_check_msg()** in Chapter 5, "Descriptor Function Reference."

am_close

The database server calls **am_close** when the processing of a single SQL statement (SELECT, UPDATE, INSERT, DELETE) completes.

Syntax

```
mi_integer am_close(MI_AM_TABLE_DESC *tableDesc)
```

tableDesc points to the table descriptor.

Usage

The **am_close** function might:

- deallocate user-data memory that **am_open** allocated with a PER_STATEMENT duration.
- call **mi_file_close()**, **mi_lo_close()**, or one of the DataBlade API functions that copies smart-large-object data to a file.

Important: *Do not call the DataBlade API* ***mi_close()*** *function to free a database connection handle that you open (in the* ***am_open*** *purpose function) with* ***mi_open()****. Because the database connection has a PER_COMMAND duration, not a duration, the database server frees the handle before it calls the* ***am_close*** *purpose function.*

Return Values

MI_OK indicates success. MI_ERROR indicates failure.

Related Topics

See the description of:

- purpose function **am_open.**
- DataBlade API functions, such as **mi_file_close()** or **mi_lo_close()**, in the *DataBlade API Programmer's Manual*.
- "Starting and Ending Processing" on page 2-7.

am_create

The database server calls **am_create** to process a CREATE TABLE statement.

Syntax

```
mi_integer am_create(MI_AM_TABLE_DESC *tableDesc)
```

tableDesc points to the table descriptor.

Usage

Even if the access method does not provide an **am_create** function, the database server automatically adds the created object to the system catalog tables, such as **systables**. For example, a user might issue the CREATE TABLE statement to register an existing, external table in the database server system catalog.

The **am_create** function typically:

- calls accessor functions to extract table specifications from the table descriptor, including a pointer to the row descriptor.
- calls DataBlade API functions to extract column attributes from the row descriptor.
- verifies that the access method can provide all the requirements that CREATE TABLE specifies.
- calls the appropriate DataBlade API functions to create a smart large object or interact with the operating system for file creation, as described in "Managing Storage Spaces" on page 3-12.

Important: *By default, transaction logging is disabled in sbspaces. To find out how to turn logging on, refer to* *"Ensuring Data Integrity" on page 3-15*.

Return Values

MI_OK indicates success. MI_ERROR indicates failure.

Related Topics

In this manual, see the description of:

- purpose function **am_drop**.
- "Creating and Dropping Database Objects" on page 2-7.

In the *DataBlade API Programmer's Manual*, see the descriptions of:

- DataBlade API functions, such as **mi_lo_create()**, and create-time constants.
- DataBlade API accessor functions for the row descriptor.

am_delete

The database server calls **am_delete** for:

- a DELETE statement.
- an UPDATE statement that requires a change in physical location.
- an ALTER FRAGMENT statement that moves a row to a different fragment.

Syntax

```
mi_integer am_delete(MI_AM_TABLE_DESC *tableDesc,
    mi_integer rowID))
```

tableDesc points to the table descriptor.

rowID the identifier of the row to delete.

Usage

The **am_delete** purpose function deletes one row in the virtual table. In response to a DELETE statement, the database server first calls the appropriate purpose functions to scan for the table entry or entries that qualify for deletion and then executes **am_delete** separately for each qualifying entry.

Important: *The database server does not call the **am_delete** purpose function unless you set both the **am_rowids** and **am_readwrite** purpose flags. For more information about setting purpose flags, refer to* *Chapter 6, "SQL Statements for Access Methods."*

Warning: *If the access method does not supply an **am_delete** purpose function, but an SQL statement requires it, the database server raises an error. For more information on how to handle this error, refer to* *"Supplying Error Messages and a User Guide" on page 3-41.*

Return Values

MI_OK indicates success. MI_ERROR indicates failure.

Related Topics

See the descriptions of:

- purpose functions **am_insert** and **am_update**.
- purpose flags **am_rowids** and **am_readwrite** in "Setting Purpose Functions, Flags, and Values" on page 6-11.
- "Inserting, Deleting, and Updating Data" on page 2-9.

am_drop

The database server calls **am_drop** for a DROP TABLE or DROP DATABASE statement.

Syntax

```
mi_integer am_drop(MI_AM_TABLE_DESC *tableDesc)
```

tableDesc points to the table descriptor.

Usage

Even if the access method provides no **am_drop** purpose function, the database server automatically removes the dropped object from the system catalog tables. The database server no longer recognizes the name of the dropped object.

Return Values

MI_OK indicates success. MI_ERROR indicates failure.

Related Topic

See the descriptions of:

- purpose function **am_create**.
- "Creating and Dropping Database Objects" on page 2-7.

am_endscan

The database server calls **am_endscan** when **am_getnext** finds no more rows.

Syntax

```
mi_integer am_endscan(MI_AM_SCAN_DESC *scanDesc)
```

scanDesc points to the scan descriptor.

Usage

The **am_endscan** purpose function might:

- deallocate the PER_COMMAND user-data memory that the **am_beginscan** purpose function allocates and stores in the scan descriptor.

  For more information on the PER_COMMAND memory and memory deallocation, refer to "Storing Data in Shared Memory" on page 3-4.
- check for transaction commit or rollback.

  Call the appropriate DataBlade API functions to determine if the transaction succeeds. Disregard the copy of old values if the transaction commits, or reapply old values if the transaction rolls back.

  For more information about transaction processing, see "Determining Transaction Success or Failure" on page 3-40.

Return Values

MI_OK indicates success. MI_ERROR indicates failure.

Related Topics

See the descriptions of:

- purpose functions **am_beginscan**, **am_getnext**, and **am_rescan**.
- "Optimizing Queries" on page 2-7.

am_getbyid

The database server calls **am_getbyid** instead of **am_getnext** to pass the row identifier, rather than a scan descriptor. For example, the database server might obtain the row identifier from an index on the virtual table.

Syntax

```
mi_integer am_getbyid(MI_AM_TABLE_DESC *tableDesc,
                      MI_ROW **retrow, mi_integer rowID)
```

tableDesc	points to the table descriptor.
retrow	points to the location where the function should place a row structure that contains the fetched data.
rowID	is the row identifier or physical address of the row to fetch.

Usage

The **am_getbyid** purpose function does not scan a table to find a qualifying row.

Possible row identifiers that *rowID* might point to include:

- The sequence of this row within the fragment
- An offset to an LO handle
- A value that an external data manager assigns
- A value that the access method assigns

Like **am_getnext**, **am_getbyid** first fetches the specified row and then passes the *retrow* pointer to **mi_row_create()** to build the composite MI_ROW value from fetched data.

Important: *The database server does not call **am_getbyid** unless you set the **am_rowids** purpose flag. For more information about setting purpose flags, refer to Chapter 6, "SQL Statements for Access Methods."*

Return Values

MI_OK indicates success. MI_ERROR indicates failure.

Related Topics

See the descriptions of:

- purpose flag **am_rowids** in "Setting Purpose Functions, Flags, and Values" on page 6-11**.**
- DataBlade API function **mi_row_create()** in the *DataBlade API Programmer's Manual*.
- purpose function **am_getnext**.

am_getnext

The **am_getnext** purpose function identifies rows that meet query criteria.

Syntax

```
mi_integer
am_getnext(MI_AM_SCAN_DESC *scanDesc,
               MI_ROW **row, mi_integer *rowid);
```

scanDesc points to the scan descriptor.

row points to the location where the access method creates rows from source records that satisfy the query.

rowid points to the returned row identifier.

Usage

Every access method must provide an **am_getnext** purpose function. This required function typically reads source data and returns query results.

If a statement includes a WHERE clause, either **am_beginscan** or **am_getnext** can parse the qualification descriptor. For each row, an **am_getnext** purpose function might:

- read source data into user data.
- execute functions in the qualification descriptor.
- save the results in the qualification descriptor.
- call **mi_eval_am_qual()** to complete a complex qualification expression.
- build a row from the fetched data that matches the projection specifications in the query.

  To find out how to create a row, refer to "Converting to and from Row Format" on page 3-39.

The **am_getnext** purpose function can loop to fill a shared-memory buffer with multiple rows. For more information about buffering, see "Buffering Multiple Results" on page 3-35.

The database server calls the **am_getnext** purpose function until that function returns MI_NO_MORE_RESULTS. Then the database server calls the **am_endscan** purpose function, if any, that the access method supplies.

If the access method does not provide an **am_rescan** purpose function, **am_getnext** stores interim data for subsequent scans in memory that persists between executions of the access method. For more information on memory duration, refer to "Storing Data in Shared Memory" on page 3-4.

Return Values

MI_ROWS indicates the return of a qualified row.

MI_NO_MORE_RESULTS indicates the end of the scan.

MI_ERROR indicates failure.

Related Topics

See the descriptions of:

- purpose functions **am_getnext**, **am_endscan**, and **am_rescan**.
- accessor functions **mi_eval_am_qual()**, **mi_tab_niorows()**, and **mi_tab_setnextrow()** in Chapter 5, "Descriptor Function Reference."
- DataBlade API function **mi_row_create()** in the *DataBlade API Programmer's Manual*.
- "Executing Qualification Functions" on page 3-28.
- "Optimizing Queries" on page 2-7.

am_insert

The database server calls **am_insert** for:

- an INSERT or UPDATE statement.
- an ALTER FRAGMENT statement that moves a row to a different fragment.

Syntax

```
mi_integer
am_insert(MI_AM_TABLE_DESC *tableDesc,
          MI_ROW *row, mi_integer *rid)
```

tableDesc points to the table descriptor.

row points to a row structure in shared memory that contains the values for the access method to insert.

rid points to the row identifier of the new row.

Usage

Possible row identifiers include:

- the sequence of this row within the fragment.
- an offset to an LO handle.
- a value that an external data manager assigns.
- a value that the access method assigns.

For each new entry, **am_insert**:

- restructures and converts the data in the MI_ROW data structure as necessary to conform to the source table.
- stores the new data in the appropriate sbspace or extspace.

  If the data is in an extspace, the access method stores the *rowID* value for use in retrieving the new record in the future.

Important: *The database server does not call* ***am_insert*** *unless the* ***am_readwrite*** *purpose flag is set. If you do not set the* ***am_rowids*** *purpose flag, the database server ignores any row identifier that the access method provides. For more information about setting purpose flags, refer to* *Chapter 6, "SQL Statements for Access Methods."*

Warning: *If the access method does not supply* ***am_insert****, but an SQL statement requires it, the database server raises an error. For more information on how to handle this error, refer to* *"Supplying Error Messages and a User Guide" on page 3-41*.

Return Values

MI_OK indicates success. MI_ERROR indicates failure.

Related Topics

See the descriptions of:

- purpose functions **am_delete** and **am_update**.
- purpose flags **am_readwrite** and **am_rowids** in "Setting Purpose Functions, Flags, and Values" on page 6-11.
- "Inserting, Deleting, and Updating Data" on page 2-9.

am_open

The database server calls **am_open** to initialize input or output prior to processing an SQL statement.

Syntax

```
mi_integer am_open(MI_AM_TABLE_DESC *tableDesc)
```

tableDesc points to the table descriptor

Usage

As part of the initialization, **am_open** might:

- determine the reason, or mode, for the open, as described in "mi_tab_mode()" on page 5-64.
- allocate PER_STATEMENT memory for a user-data structure as described in "Persistent User Data" on page 3-6.
- open a database connection with the DataBlade API **mi_open()** function.

  To enable subsequent purpose functions to use the database, **am_open** can copy the connection handle that **mi_open()** returns into the user-data structure.
- register callback functions to handle exceptions, as described in "Handling the Unexpected" on page 3-8.
- call the appropriate DataBlade API functions to obtain a file handle for an extspace or an LO handle for a smart large object.

Return Value

MI_OK indicates success. MI_ERROR indicates failure.

Related Topics

See the descriptions of:

- purpose function **am_close**.
- memory allocation, callback functions, and the functions to open files or smart large objects in the *DataBlade API Programmer's Manual*.
- **mi_tab_mode()** and **mi_tab_setniorows()** in Chapter 5, "Descriptor Function Reference."
- "Starting and Ending Processing" on page 2-7.

am_rescan

The database server typically calls **am_rescan** to process a join or subquery that requires multiple scans on the same table.

Syntax

```
mi_integer am_rescan(MI_AM_SCAN_DESC *scanDesc)
```

scanDesc points to the scan descriptor.

Usage

Although **am_rescan** is an optional purpose function, the access method can enhance efficiency by supplying **am_rescan** for applications that involve joins, subqueries, and other multiple-pass scan processes. The **am_rescan** purpose function ends the previous scan in an appropriate manner and begins a new scan on the same open table.

Without an **am_rescan** purpose function, the database server calls the **am_endscan** function and then **am_beginscan**, if the access method provides these functions.

Tip: *To determine if an outer join might cause a constant value to change, call* ***mi_qual_const_depends_outer()****. To determine the need to reevaluate the qualification descriptor, call* ***mi_scan_newquals()*** *from* ***am_rescan****.*

Return Values

MI_OK indicates success. MI_ERROR indicates failure.

Related Topics

See the descriptions of:

- purpose function **am_getnext**.
- accessor functions **mi_qual_const_depends_outer()** and **mi_scan_newquals()** in Chapter 5, “Descriptor Function Reference.”
- “Optimizing Queries” on page 2-7.

am_scancost

The query optimizer calls **am_scancost** during a SELECT statement, before it calls **am_open**.

Syntax

```
mi_real * am_scancost(MI_AM_TABLE_DESC *tableDesc,
                      MI_AM_QUAL_DESC *qualDesc)
```

tableDesc points to the table descriptor.

qualDesc points to the qualification descriptor, which specifies the criteria that a table row must satisfy to qualify for retrieval.

Usage

The **am_scancost** purpose function estimates the cost to fetch and qualify data for the current query. The optimizer relies on the **am_scancost** return value to evaluate a query path for a scan that involves the access method.

Warning: *If the access method does not have an* ***am_scancost*** *purpose function, the database server estimates the cost of a scan, which can diminish the optimal nature of the query plan.*

Calculating Cost

The following types of information influence cost:

- Distribution of values across storage media

  Is the data clustered? Are fragments spread across different physical volumes? Does any one fragment contain a large or a narrow range of values for a column that the query specifies?

- Information about the tables, columns, and indexes in the queried database

  Does the query contain a subquery? Does it require a place in memory to store aggregations? Does a qualification require casting or conversion of data types? Does the query involve multiple tables or inner joins? Do indexes exist for the appropriate key columns? Are keys unique?

To calculate a cost, **am_scancost** considers the following factors:

- Disk access

  Add 1 to the cost for every disk access required to access the data.
- Memory access

  Add .15 to the cost for every row accessed in memory.
- The cost of evaluating the qualification criteria

Compute the cost of retrieving only those table entries that qualify.

Important: *Because a function cannot return an **mi_real** data type by value, you must allocate memory to store the scan cost value and return a pointer to that memory from the **am_scancost** purpose function.*

Factoring Cost

To adjust the result of **am_scancost**, set the **am_costfactor** purpose value. The database server multiplies the cost that **am_scancost** returns by the value of **am_costfactor**, which defaults to 1 if you do not set it. To find out how to set purpose values, refer to Chapter 6, “SQL Statements for Access Methods.”

Forcing Reoptimization

The optimizer might need a new scan cost for subsequent scans of the same table; for example, due to a join. To execute **am_scancost** before each rescan, call the **mi_qual_setreopt()** function. For a list of VTI accessor functions that **am_scancost** can call to help evaluate cost and the need to reoptimize, refer to “Related Topics” on page 4-36.

Returning a Negative Cost

If the query specifies a feature that the access method does not support, return a value from **am_scancost** that forces the optimizer to pursue another path. In Figure 4-14, an access method that does not process Boolean operators checks the qualification descriptor for Boolean operators and returns a negative value if it finds one.

```
mi_real * my_scan_cost(td, qd)
    MI_AM_QUAL_DESC *qd;
    MI_AM_TABLE_DESC *td;
{......
    for (i = 0; i < mi_qual_nquals(qd); i++)
        if (mi_qual_issimple(qd, i) == MI_FALSE) /* Boolean operator found. */
            return -99;
    }
```

Figure 4-14
Forcing a Table Scan

Return Value

A pointer to an **mi_real** data type that contains the cost value

Related Topics

See the descriptions of:

- purpose function **am_stats**.
- purpose flag **am_scancost** in "Setting Purpose Functions, Flags, and Values" on page 6-11.
- accessor functions **mi_qual_const_depends_hostvar()**, **mi_qual_constisnull_nohostvar()**, **mi_qual_constant_nohostvar()**, **mi_qual_boolop()**, **mi_qual_issimple()**, and **mi_qual_setreopt()** in Chapter 5, "Descriptor Function Reference."

am_stats

The database server calls **am_stats** to process an UPDATE STATISTICS statement.

Syntax

```
mi_integer am_stats(MI_AM_TABLE_DESC *tableDesc,
        MI_AM_TSTATS_DESC *tstatsDesc);
```

tableDesc points to the table descriptor.

tstatsDesc points to the statistics descriptor.

Usage

To influence the **am_stats** sampling rate, an UPDATE STATISTICS statement might include an optional distribution-level keyword, LOW, MEDIUM, or HIGH. If the UPDATE STISTISTICS statement does not include one of these keywords, the default LOW distribution level applies.

Adjust the sampling rate in your version of the **am_stats** purpose function according to the distribution-level keyword that the user specifies in the UPDATE STATISTICS statement. To determine which keyword, LOW, MEDIUM, or HIGH, an UPDATE STATISTICS statement specifies, call the **mi_tab_update_stat_mode()** function. For detailed information about the sampling rates that each keyword implies, refer to the description of UPDATE STATISTICS in the *Informix Guide to SQL: Syntax*.

The **am_stats** purpose function calls the various VTI accessor functions that set values in statistics descriptor for the database server. The database server places the statistics descriptor results in the **systables** and **syscolumns** system catalog tables. The **am_stats** function can also save any additional values in a location that **am_scancost** can access, such as a file in the extspace or a table in sbspace.

Return Values

MI_OK indicates success. MI_ERROR indicates failure.

Related Topics

See the descriptions of:

- accessor functions **mi_tab_update_stat_mode()** and **mi_tstats_*** in Chapter 5, "Descriptor Function Reference."
- the "Statistics Descriptor" on page 5-12.
- "Updating Statistics" on page 3-32.

am_update

The database server calls **am_update** to process an UPDATE statement.

Syntax

```
mi_integer am_update(MI_AM_TABLE_DESC *tableDesc,
                     MI_ROW *row, mi_integer rowid);
```

tableDesc points to the table descriptor.

row points to the row structure that contains the updated values.

rowid indicates where to write the updated values.

Usage

The **am_update** function modifies the contents of an existing row.

If the access method needs to move the updated row, **am_update** can take the following actions:

- deletes the old row.
- adjusts the data format in *row* to conform to the source data.
- stores the updated source-data record.
- stores the updated row identifier.

Important: *The database server does not call* ***am_update*** *unless both the* ***am_rowids*** *and* ***am_readwrite*** *purpose flags are set. For more information about setting purpose flags, refer to* *Chapter 6, "SQL Statements for Access Methods."*

Warning: *If the access method does not supply* ***am_update****, but an SQL statement requires it, the database server raises an error. For more information on how to handle this error, refer to* *"Supplying Error Messages and a User Guide" on page 3-41*.

Return Values

MI_OK indicates success. MI_ERROR indicates failure.

Related Topics

See the descriptions of:

- purpose functions **am_delete** and **am_insert**.
- purpose flags **am_rowids** and **am_readwrite** in "Setting Purpose Functions, Flags, and Values" on page 6-11.
- "Inserting, Deleting, and Updating Data" on page 2-9.

Chapter

Descriptor Function Reference

5

In This Chapter

This chapter provides syntax and usage for the functions that Informix supplies to access-method developers. This chapter consists of the following information:

- "Descriptors" on this page describes the predefined data structures through which the database server and access method pass information.
- "Include Files" on page 5-15 lists the header files with descriptor and function declarations that the access method must include.
- "Accessor Functions" on page 5-16 lists every function that Informix provides specifically for use with the VTI.

The information in this chapter is organized in alphabetical order by descriptor and function name.

Purpose functions use the functions and data structures that this chapter describes to communicate with the database server. For details about the purpose function, refer to Chapter 4, "Purpose-Function Reference."

Descriptors

The application programming interface (API) that Informix provides with the VTI consists primarily of the following components:

- Opaque data structures, called *descriptors*, that the database server passes by reference to purpose functions
- *Accessor functions* that store and retrieve descriptor values

The Virtual-Table Interface (VTI) provides the following descriptors and accessor functions.

Descriptor	Describes	Accessor-Function Prefix	Reference
Qualification descriptor (MI_AM_QUAL_DESC)	WHERE clause criteria	mi_qual_	"Qualification Descriptor" on page 5-7
Row descriptor (MI_ROW)	Order and data types of projected columns	Various DataBlade API functions	*DataBlade API Programmer's Manual*
Scan descriptor (MI_AM_SCAN_DESC)	SELECT clause projection	mi_scan_	"Scan Descriptor" on page 5-11
Statistics descriptor (MI_AM_TSTATS_DESC)	Distribution of values	mi_tstats_	"Statistics Descriptor" on page 5-12
Table descriptor (MI_AM_TABLE_DESC)	Table attributes and fragment partition	mi_tab_	"Table Descriptor" on page 5-13

Each of the following sections describes the contents of a descriptor and the name of the accessor function that retrieves each descriptor field. For complete syntax, including the parameters and return type of each accessor function, refer to "Accessor Functions" on page 5-16.

Important: *Because the internal structure of any VTI descriptor might change, Informix declares them as opaque structures. To make a portable access method, always use the access functions to extract or set descriptor values. Do not access descriptor fields directly.*

Qualification Descriptor

A qualification descriptor, or MI_AM_QUAL_DESC structure, describes the conditions in the WHERE clause of an SQL statement. For a detailed description of qualification processing, including examples, refer to "Processing Queries" on page 3-20.

Use the VTI **mi_scan_quals()** function to obtain a pointer to the qualification descriptor from the scan descriptor.

The following accessor functions extract information from a qualification descriptor.

Accessor Function	Return Value
mi_qual_boolop()	The operator type (AND or OR) of a qualification that is a complex expression
mi_qual_column()	The position that the column argument to a qualification function occupies within a row
mi_qual_commuteargs()	MI_TRUE if the argument list begins with a constant rather than a column value
mi_qual_const_depends_hostvar()	MI_TRUE if a constant argument to a qualification function acquires a value at runtime from a host variable
mi_qual_const_depends_outer()	MI_TRUE if the value of a particular constant argument can change each rescan
mi_qual_constant()	The runtime value of the constant argument to a qualification function
mi_qual_constant_nohostvar()	The value specified in the WHERE clause for the constant argument to a qualification function
mi_qual_constisnull()	MI_ TRUE if the value of a constant argument to a qualification function is NULL
mi_qual_constisnull_nohostvar()	MI_ TRUE if the WHERE clause specifies a NULL value as the constant argument to a qualification function

(1 of 2)

Accessor Function	Return Value
mi_qual_funcid()	The routine identifier of a qualification function
mi_qual_funcname()	The name of a qualification function
mi_qual_handlenull()	MI_TRUE if the qualification function accepts NULL as an argument
mi_qual_issimple()	MI_TRUE if the qualification contains one function rather than a complex expression
mi_qual_needoutput()	MI_TRUE if the qualification function supplies an output parameter value Obtain and set a pointer to the output-parameter value with **mi_qual_setoutput().**
mi_qual_negate()	MI_TRUE if the qualification includes the operator NOT
mi_qual_nquals()	The number of nested qualifications in a complex expression, or 0 for a simple qualification that contains no Boolean operators
mi_qual_qual()	Pointer to one qualification in a complex qualification descriptor or to the only qualification
mi_qual_value()	One of the following possible values: ■ MI_VALUE_NOT_EVALUATED until the qualification returns a result ■ MI_VALUE_TRUE if the qualification returns MI_TRUE ■ MI_VALUE_FALSE if the qualification returns MI_FALSE Set the results in the qualification descriptor with **mi_qual_setvalue()**. Reset the qualification descriptor to MI_VALUE_NOT_EVALUATED with **mi_init_am_qual()**.

(2 of 2)

The following accessor functions set values in the descriptor.

Accessor Function	Value Set
mi_qual_setvalue()	The result from executing the qualification operator or function
mi_qual_setoutput()	A host-variable value
mi_qual_setreopt()	An indicator to force reoptimization between rescans
mi_eval_am_qual()	MI_TRUE if the current row satisfies the current qualification
mi_init_am_qual()	MI_VALUE_NOT_EVALUATED to reset all results fields in a qualification descriptor

Row Descriptor

A row descriptor, or MI_ROW_DESC structure, typically describes the columns that the CREATE TABLE statement establishes for a table. A row descriptor can also describe a single row-type column. The DataBlade API defines the row descriptor that the access-method API uses.

The table descriptor contains a pointer to the row descriptor.

The accessor functions for the row descriptor (**mi_column_***) provide information about each column, including the column name, floating-point precision and scale, alignment, and a pointer to a type descriptor. For information about the accessor functions for the row descriptor, refer to the *DataBlade API Programmer's Manual*.

Scan Descriptor

The scan descriptor, or MI_AM_SCAN_DESC structure, contains the specifications of an SQL query, including the following items:

- The columns to project
- A pointer to selection criteria from the WHERE clause
- Isolation and locking information
- A pointer to where the access method can store scanned data

The database server passes the scan descriptor to the access-method scanning purpose functions: **am_beginscan**, **am_endscan**, **am_rescan**, and **am_getnext**.

The following functions extract information from the scan descriptor.

Accessor Function	Return Value
mi_scan_forupdate()	MI_TRUE if a SELECT statement includes a FOR UPDATE clause
mi_scan_isolevel()	The isolation level for the table
mi_scan_locktype()	The lock type for the scan
mi_scan_newquals()	MI_TRUE if the qualification descriptor changes after the first scan for a join or subquery
mi_scan_nprojs()	The number of columns in the projected row that the access method returns to the query
mi_scan_projs()	A pointer to an array that identifies which columns from the row descriptor make up the projected row that the query returns
mi_scan_quals()	A pointer to the qualification descriptor or a NULL-valued pointer if the database server does not create a qualification descriptor
mi_scan_table()	A pointer to the table descriptor for the table that the access method scans
mi_scan_userdata()	A pointer to the user-data area of memory

The following accessor function sets data in the qualification descriptor.

Accessor Function	Value Set
mi_scan_setuserdata()	The pointer to user data that a subsequent function will need

Statistics Descriptor

An access method returns statistics to the UPDATE STATISTICS statement in a statistics descriptor, or MI_AM_TSTATS_DESC structure. The database server copies the separate values from the statistics descriptor to pertinent tables in the system catalog.

The following accessor functions set information in the statistics descriptor.

Accessor Function	Value Set
mi_tstats_setnpages()	The number of pages that the table uses
mi_tstats_setnrows()	The number of rows in the table

Table Descriptor

The table descriptor, or MI_AM_TABLE_DESC structure, provides information about the table, particularly the data definition from the CREATE TABLE statement that created the object.

The following accessor functions extract information from or set values in the table descriptor.

Accessor Function	Return Value
mi_tab_amparam()	Parameter values from the USING clause of the CREATE TABLE statement
mi_tab_createdate()	The date that the table was created
mi_tab_id()	The unique table identifier
mi_tab_isolevel()	The isolation level
mi_tab_istable()	MI_TRUE for a primary access method
mi_tab_mode()	The input/output mode (read-only, read and write, write-only, and log transactions)
mi_tab_name()	The table name
mi_tab_niorows()	The number of rows that **mi_tab_setniorows()** sets
mi_tab_numfrags()	The number of fragments in the table or 1 for a single-fragment table
mi_tab_owner()	The table owner
mi_tab_partnum()	The unique partition number, or fragment identifier, of this table or fragment
mi_tab_rowdesc()	A pointer to a row descriptor that describes the columns in the row
mi_tab_spaceloc()	The extspace location of the table fragment
mi_tab_spacename()	The storage space name for the fragment from the CREATE TABLE statement IN clause

(1 of 2)

Accessor Function	Return Value
mi_tab_spacetype()	The type of space used for the table: X for an extspace or S for an sbspace Any other value means that neither an IN clause nor the **sysams** system catalog table specifies the type of storage space.
mi_tab_update_stat_mode()	The level of statistics that an UPDATE STATISTICS statement generates: LOW, MEDIUM, or HIGH
mi_tab_userdata()	A pointer to the user-data area of memory

(2 of 2)

The following accessor functions set values in the table descriptor.

Accessor Function	Value Set
mi_tab_setniorows()	The number of rows that shared memory can store from a scan
mi_tab_setnextrow()	One row of the number that **mi_tab_setniorows()** allows
mi_tab_setuserdata()	A pointer in the user-data area of memory

Include Files

Several files contain definitions that the access method references. Include the following files in your access-method build:

- The **mi.h** file defines the DataBlade API descriptors, other opaque data structures, and function prototypes.
- The **miami.h** file defines the descriptors and prototypes for the VTI.
- If your access method alters the default memory duration, include the **memdur.h** and **minmdur.h** files.

GLS

- To call GLS routines for internationalization, include **ifxgls.h**. ♦

Accessor Functions

The VTI library contains functions that primarily access selected fields from the various descriptors.

For a description of any descriptor in this section, refer to "Descriptors" on page 5-6.

This chapter lists detailed information about specific VTI accessor functions in alphabetical order by function name. To find the accessor functions for a particular descriptor, look for the corresponding function-name prefix at the top of each page.

Descriptor	Accessor-Function Name or Prefix
Qualification	mi_qual_*(), mi_eval_am_qual(), mi_init_am_qual()
Scan	mi_scan_*()
Statistics	mi_tstats_*()
Table	mi_tab_*()

mi_eval_am_qual()

The **mi_eval_am_qual()** function evaluates parts of a qualification that the access method does not set to MI_VALUE_TRUE or MI_VALUE_FALSE.

Syntax

```
mi_boolean
mi_eval_am_qual(MI_ROW *row, MI_AM_QUAL_DESC *qualDesc);
```

row points to the row structure.

qualDesc points to the qualification descriptor.

Usage

The **am_getnext** purpose function can call **mi_eval_am_qual()** to obtain results for any qualifications that the access method cannot complete. Before the access method can call **mi_eval_am_qual(),** it must call **mi_row_create()** to assemble a row. For a detailed procedure and examples, refer to "Processing Complex Qualifications" on page 3-29.

Tip: *Both **mi_row_create()** and **mi_eval_am_qual()** can increase response time and CPU usage. Call them only if necessary.*

If **mi_eval_am_qual()** returns MI_TRUE, **am_getnext** returns MI_ROWS. If **mi_eval_am_qual()** returns MI_FALSE, **am_getnext** disregards the current row, does not return a value, and starts to evaluate the next row.

Return Values

MI_TRUE indicates that the row qualifies. MI_FALSE indicates that the row does not qualify.

Related Topic

See the description of function **mi_init_am_qual()**.

mi_init_am_qual()

The **mi_init_am_qual()** function reinitializes all parts of the qualification to MI_VALUE_NOT_EVALUATED.

Syntax

```
void mi_init_am_qual(MI_AM_QUAL_DESC *qualDesc);
```

qualDesc points to the qualification descriptor.

Usage

The database server does not initialize the results area of a qualification descriptor to MI_VALUE_NOT_EVALUATED after a call to **mi_eval_am_qual()** changes the results value to MI_VALUE_TRUE or MI_VALUE_FALSE. To initialize the qualification results for the next row, have **am_getnext** call **mi_init_am_qual()**.

Return Values

None

Related Topic

See the description of function **mi_eval_am_qual()**.

mi_qual_boolop()

The **mi_qual_boolop()** function retrieves the Boolean operator that combines two qualifications in a complex expression.

Syntax

```
MI_AM_BOOLOP mi_qual_boolop(MI_AM_QUAL_DESC *qualDesc);
```

qualDesc points to the qualification descriptor.

Usage

The access method first obtains results for the simple functions in a complex qualification. To determine how to combine the results that the access method has so far, it can call the **mi_qual_boolop()** function.

Return Values

MI_BOOLOP_NONE indicates that the current qualification does not contain a Boolean operator.

MI_BOOLOP_AND indicates that the current qualification contains a Boolean AND operator.

MI_BOOLOP_OR indicates that the current qualification contains a Boolean OR operator.

Related Topic

See the descriptions of:

- function **mi_qual_issimple()**.
- "Qualifying Data" on page 3-27.

mi_qual_column()

The **mi_qual_column()** function identifies the key-column argument to a qualification function.

Syntax

```
mi_smallint mi_qual_column(MI_AM_QUAL_DESC *qualDesc);
```

qualDesc points to the qualification descriptor.

Usage

A qualification identifies a column by a number that locates the column in the row descriptor. The **mi_qual_column()** function returns the number 0 for the first column specified in the row descriptor and adds 1 for each subsequent column.

For example, assume the WHERE clause contains the function `equal(name,'harry')` and that **name** is the second column in the row. The **mi_qual_column()** function returns the value 1.

The access method might need to identify the column by name, for example, to assemble a query for an external database manager. To retrieve the column name, pass the return value of **mi_qual_column()** and the row descriptor to the DataBlade API **mi_column_name()** function as in the following example:

```
rowDesc = mi_tab_rowdesc(tableDesc);
colnum=mi_qual_column(qualDesc);
colname=mi_column_name(rowDesc,colnum);
```

Return Values

The integer identifies the column argument by its position in the table row.

Related Topics

See the descriptions of:

- functions **mi_qual_constant()** and **mi_tab_rowdesc()**.
- DataBlade API row-descriptor accessor functions in the *DataBlade API Programmer's Manual*.

mi_qual_commuteargs()

The **mi_qual_commuteargs()** function determines if the constant precedes the column in a qualification-function argument list.

Syntax

```
mi_boolean mi_qual_commuteargs(MI_AM_QUAL_DESC *qualDesc);
```

qualDesc points to the qualification descriptor.

Return Values

MI_TRUE indicates that *constant* precedes *column* in the argument list; for example, ***function***(*constant*, *column*).

MI_FALSE indicates that *column* precedes *constant* in the argument list; for example, ***function***(*column*, *constant*).

Related Topics

See the description of accessor function **mi_qual_issimple()**.

mi_qual_constant()

The **mi_qual_constant()** function retrieves the constant value that the WHERE clause specifies as a qualification-function argument.

Syntax

```
MI_DATUM mi_qual_constant(MI_AM_QUAL_DESC *qualDesc);
```

qualDesc points to the qualification descriptor.

Usage

To retrieve the constant value from the argument lists of a qualification function, call **mi_qual_constant()** from the **am_beginscan** or **am_getnext** purpose function.

Qualification functions evaluate the contents of a column against some criteria, such as a supplied constant value.

If a qualification function does not involve a host variable, **mi_qual_constant()** retrieves the explicit constant argument. For example, **mi_qual_constant()** retrieves the string `harry` from the arguments to the following function:

```
WHERE equal(name,'harry')
```

If a qualification function involves a host variable but no explicit value, **mi_qual_constant()** retrieves the runtime constant value that is associated with the host variable. For example, **mi_qual_constant()** retrieves the runtime value that replaces ? in the following function:

```
WHERE equal(name,?)
```

Important: *Because the value that an application binds to host variables can change between scans, the results of* ***mi_qual_constant()*** *might change between calls to* ***am_getnext****.*

To determine if a function involves a host variable argument, execute **mi_qual_const_depends_hostvar()** in the **am_scancost** purpose function. If **mi_qual_const_depends_hostvar()** returns MI_TRUE, call **mi_qual_constant()** from **am_getnext** to retrieve the most recent value for the host variable and do not save the value from **mi_qual_constant()** in user data for subsequent scans.

Return Values

The MI_DATUM structure contains the value of the constant argument.

Related Topics

See the descriptions of:

- functions **mi_qual_column()**, **mi_qual_constisnull()**, and **mi_qual_const_depends_hostvar()**.
- generic functions in Figure 3-6 on page 3-23.
- MI_DATUM in the *DataBlade API Programmer's Manual*.

mi_qual_constant_nohostvar()

The **mi_qual_constant_nohostvar()** function returns an explicit constant value, if any, from the qualification-function arguments.

Syntax

```
MI_DATUM
mi_qual_constant_nohostvar(MI_AM_QUAL_DESC *qualDesc);
```

qualDesc points to the qualification descriptor.

Usage

To help calculate the cost of a qualification function, the **am_scancost** purpose function can extract the constant and column arguments and evaluate the distribution of the specified constant value in the specified column. Function arguments can include constants from two sources:

- A value that the WHERE clause explicitly supplies
- A dynamic value, or *host variable*, that the access method or a client application might supply

  In the WHERE clause, the function argument list contains a placeholder, such as a question mark (?), for the host variable.

The following function involves both an explicit value (`200`) and a host variable (?) as constant arguments, rather than an explicit value:

```
WHERE range(cost, 200, ?)
```

In the following example, a WHERE clause specifies two constant values in a row that holds three values. A client program supplies the remaining value.

```
WHERE equal(prices, row(10, ?, 20))
```

For the preceding qualification, the **mi_qual_constant_nohostvar()** function returns `row(10, NULL, 20)`.

Because the **am_scancost** purpose function cannot predict the value of a host variable, it can only evaluate the cost of scanning for constants that the WHERE clause explicitly specifies. Call the **mi_qual_constant_nohostvar()** function to obtain any argument value that is available to **am_scancost**. The **mi_qual_constant_nohostvar()** function ignores host variables if the qualification supplies an explicit constant value.

By the time the database server invokes the **am_beginscan** or **am_getnext** purpose function, the qualification descriptor contains a value for any host-variable argument. To execute the function, obtain the constant value with the **mi_qual_constant()** function.

Return Value

If the argument list of a function includes a specified constant value, **mi_qual_constant_nohostvar()** returns that value in an MI_DATUM structure.

If the specified constant contains multiple values, this function returns all provided values and substitutes NULL for each host variable.

If the function arguments do not explicitly specify a constant value, this function returns a NULL.

Related Topics

See the descriptions of:

- accessor functions **mi_qual_constisnull_nohostvar()** and **mi_qual_constant()**.
- "Runtime Values as Arguments" on page 3-23.
- MI_DATUM in the *DataBlade API Programmer's Manual*.
- host variables in the *DataBlade API Programmer's Manual*, *Extending Informix Dynamic Server 2000*, and the *Informix ESQL/C Programmer's Manual*.

mi_qual_constisnull()

The **mi_qual_constis**null**()** function determines if the arguments to a qualification function include a NULL constant.

Syntax

```
mi_boolean mi_qual_constisnull(MI_AM_QUAL_DESC *qualDesc);
```

qualDesc points to the qualification descriptor.

Usage

The **Return Value** column shows the results of the **mi_qual_constisnull()** function for various constant arguments.

Sample Function	Description	Return Value
function(*column*, 10)	The arguments specify the explicit non-NULL constant value 10.	MI_FALSE
function(*column*, NULL)	The arguments specify an explicit NULL value.	MI_TRUE

The form ***function***(*column*,?) should not occur because the qualification descriptor that the database server passes to the **am_beginscan** or **am_getnext** purpose function contains values for any host-variable argument.

Do not call this function from the **am_scancost** purpose function. Use **mi_qual_constisnull_nohostvar()** instead.

Return Values

MI_TRUE indicates that the arguments include an explicit NULL-valued constant.

mi_qual_constisnull_nohostvar()

The **mi_qual_constisnull_nohostvar()** function determines if a qualification-function argument list contains an explicit NULL value.

Syntax

```
mi_boolean
mi_qual_constisnull_nohostvar(MI_AM_QUAL_DESC *qualDesc);
```

qualDesc points to the qualification descriptor.

Usage

The **mi_qual_constisnull_nohostvar()** function evaluates the explicit value, if any, that the WHERE clause specifies in the function argument list. This function does not evaluate host variables. Call this function from the **am_scancost** purpose function.

The following functions compare a column that contains a row to a row constant. Each function depends on a client application to provide part or all of the constant value. The **Return Value** column shows the results of the **mi_qual_constisnull_nohostvar()** function.

Sample Function	Description	Return Value
function(*column*, ROW(10,?,20))	The row contains the explicit constant values `10` and `20`. The unknown value that replaces ? does not influence the return value of **mi_qual_constisnull_nohostvar()**.	MI_FALSE
function(*column*, ROW(NULL,?,20))	The first field in the row constant specifies an explicit NULL value.	MI_TRUE
function(*column*,?)	The arguments to the function contain no explicit values. The qualification descriptor contains a NULL in place of the missing explicit value.	MI_TRUE

Return Values

MI_TRUE indicates one of the following conditions in the argument list:

- An explicit NULL-valued constant
- No explicit values

MI_FALSE indicates that the constant argument is not NULL valued.

Related Topics

See the descriptions of:

- accessor function **mi_qual_constisnull()**.
- "Runtime Values as Arguments" on page 3-23.
- host variables in the *DataBlade API Programmer's Manual*, *Extending Informix Dynamic Server 2000*, and the *Informix ESQL/C Programmer's Manual*.

mi_qual_const_depends_hostvar()

The **mi_qual_const_depends_hostvar()** function indicates whether the value of a host variable influences the evaluation of a qualification.

Syntax

```
mi_boolean
mi_qual_const_depends_hostvar(MI_AM_QUAL_DESC *qualDesc)
```

qualDesc points to the qualification descriptor.

Usage

Call **mi_qual_const_depends_hostvar()** in the **am_scancost** purpose function to determine whether a qualification function contains a host variable but no explicit constant value.

Because the database server executes **am_scancost** before the application binds the host variable to a value, the qualification descriptor cannot provide a value in time to evaluate the cost of the scan.

If **mi_qual_const_depends_hostvar()** returns MI_TRUE, **am_scancost** can call **mi_qual_setreopt(),** which tells the database server to reoptimize before it executes the scan.

Return Values

MI_TRUE indicates that a host variable provides values when the function executes. MI_FALSE indicates that the qualification descriptor supplies the constant value.

Related Topics

See the descriptions of:

- accessor functions **mi_qual_needoutput()** and **mi_qual_setreopt()**.
- "Runtime Values as Arguments" on page 3-23.
- host variables in the *DataBlade API Programmer's Manual*, *Extending Informix Dynamic Server 2000*, and the *Informix ESQL/C Programmer's Manual*.

mi_qual_const_depends_outer()

The **mi_qual_const_depends_outer()** function indicates that an outer join provides the constant in a qualification.

Syntax

```
mi_boolean
mi_qual_const_depends_outer(MI_AM_QUAL_DESC *qualDesc)
```

qualDesc points to the qualification descriptor.

Usage

If this **mi_qual_const_depends_outer()** evaluates to MI_TRUE, the join or subquery can produce a different constant value for each rescan.

Call **mi_qual_const_depends_outer()** in **am_rescan**. If your access method has no **am_rescan** purpose function, call **mi_qual_const_depends_outer()** in **am_beginscan**.

Return Values

MI_TRUE indicates that the constant depends on an outer join. MI_FALSE indicates that the constant remains the same on a rescan.

Related Topics

See the description of accessor function **mi_qual_constant()**.

mi_qual_funcid()

The **mi_qual_funcid()** function returns the routine identifier of a qualification function.

Syntax

```
mi_integer mi_qual_funcid(MI_AM_QUAL_DESC *qualDesc);
```

qualDesc points to the qualification descriptor.

Usage

To execute a registered UDR or an internal function with the DataBlade API FastPath facility, the access method needs a valid routine identifier. The **mi_qual_funcid()** function provides a routine identifier, if available, for the qualification function.

If **mi_qual_funcid()** returns a positive number, the routine identifier exists in the **sysprocedures** system catalog table, and the database server can execute the function. A negative return value from the **mi_qual_funcid()** function can indicate a valid function if the database server loads an internal function in shared memory but does not describe the function in **sysprocedures**.

Warning: *A negative return value might indicate that the SQL WHERE clause specified an invalid function.*

Return Values

A positive integer is the routine identifier by which the database server recognizes a function.

A negative return value indicates that the **sysprocedures** system catalog table does not have a routine identifier for the function.

Related Topics

In this book, see the descriptions of:

- accessor function **mi_qual_funcname()**.
- "Using the Routine Identifier" on page 3-28.

In the *DataBlade API Programmer's Manual*, see the descriptions of:

- the function descriptor (MI_FUNC_DESC data structure) and its accessor functions.
- FastPath function execution, including DataBlade API functions **mi_func_desc_by_typeid()** and **mi_routine_exec().**

mi_qual_funcname()

The mi_qual_funcname() function returns the name of a qualification function.

Syntax

```
mi_string * mi_qual_funcname(MI_AM_QUAL_DESC *qualDesc)
```

qualDesc points to the qualification descriptor.

Usage

If **mi_qual_funcid()** returns a negative value instead of a valid routine identifier, the qualification function is not registered in the database. The access method might call the qualification function by name from the access-method library or send the function name and arguments to external software. For examples, refer to "Using the Function Name" on page 3-29.

Return Value

The return string contains the name of a simple function in the qualification.

mi_qual_handlenull()

The **mi_qual_handlenull()** function determines if the qualification function can accept NULL arguments.

Syntax

```
mi_boolean mi_qual_handlenull(MI_AM_QUAL_DESC *qualDesc)
```

qualDesc points to the qualification descriptor.

Usage

The database server indicates that a UDR can accept NULL-valued arguments if the CREATE FUNCTION statement specified the HANDLESNULLS routine modifier.

Return Values

MI_TRUE indicates that the function handles NULL values. MI_FALSE indicates that the function does not handle NULL values.

mi_qual_issimple()

The **mi_qual_issimple()** function determines if a qualification is a function. A function has one of the formats that Figure 3-6 on page 3-23 shows, with no AND or OR operators.

Syntax

```
mi_boolean mi_qual_issimple(MI_AM_QUAL_DESC *qualDesc);
```

qualDesc points to the qualification descriptor.

Usage

Call **mi_qual_issimple()** to determine where to process the current qualification. If **mi_qual_issimple()** returns MI_TRUE, call the access method routine that executes the qualification-function execution.

For an example that uses **mi_qual_issimple()** to find the functions in a complex WHERE clause, refer to "Processing Complex Qualifications" on page 3-29.

If **mi_qual_issimple()** returns MI_FALSE, the current qualification is a Boolean operator rather than a function. For more information about the Boolean operator, call the **mi_qual_boolop()** accessor function.

Return Values

MI_TRUE indicates that the qualification is a function. MI_FALSE indicates that the qualification is not a function.

Related Topic

See the description of:

- accessor function **mi_qual_boolop()**.
- "Simple Functions" on page 3-22.

mi_qual_needoutput()

The **mi_qual_needoutput()** function determines if the access method must set the value for an OUT argument in a UDR.

Syntax

```
mi_boolean mi_qual_needoutput(MI_AM_QUAL_DESC *qualDesc,
                    mi_integer n);
```

qualDesc points to the qualification descriptor.

n is always set to `0` to indicate the first and only argument that needs a value.

Usage

If a UDR declaration includes an OUT parameter, the function call in the WHERE clause includes a corresponding placeholder, called a *statement-local variable* (*SLV*). If the **mi_qual_needoutput()** function detects the presence of an SLV, the access method calls the **mi_qual_setoutput()** function to set a constant value for that SLV.

For examples of OUT parameters and SLVs, refer to "Runtime Values as Arguments" on page 3-23.

Return Values

MI_TRUE indicates that the qualification function involves an SLV argument. MI_FALSE indicates that the qualification function does not specify an SLV argument.

Related Topic

See the description of accessor function **mi_qual_setoutput()**.

mi_qual_negate()

The **mi_qual_negate()** function indicates whether the NOT Boolean operator applies to the results of the specified qualification. The NOT operator can negate the return value of a function or a Boolean expression.

Syntax

```
mi_boolean mi_qual_negate(MI_AM_QUAL_DESC *qualDesc);
```

qualDesc points to the qualification descriptor.

Return Values

MI_TRUE indicates that the qualification function should be negated.
MI_FALSE indicates that the qualification function should not be negated.

Related Topic

See the description of "Negation" on page 3-25.

mi_qual_nquals()

The **mi_qual_nquals()** function retrieves the number of qualifications in an AND or OR qualification expression.

Syntax

```
mi_integer mi_qual_nquals(MI_AM_QUAL_DESC *qualDesc);
```

qualDesc points to the qualification descriptor.

Return Values

The return integer indicates the number of qualifications in an AND or OR qualification expression. A return value of 0 indicates that the qualification contains one simple function and no Boolean operators.

Related Topic

See the description of "Complex Boolean Expressions" on page 3-25.

mi_qual_qual()

The **mi_qual_qual()** function points to one function or Boolean expression in a complex qualification.

Syntax

```
MI_AM_QUAL_DESC* mi_qual_qual(MI_AM_QUAL_DESC *qualDesc,
                              mi_integer n);
```

qualDesc points to the qualification descriptor.

n identifies which qualification to retrieve in the expression.

Set *n* to `0` to retrieve the first qualification descriptor in the array of qualification descriptors. Set *n* to `1` to retrieve the second qualification descriptor in the array. Increment *n* by `1` to retrieve each subsequent qualification.

Usage

To determine the number of qualifications in an expression and thus the number of iterations through **mi_qual_qual()**, first call the **mi_qual_nquals()** accessor function. If **mi_qual_nquals()** returns `0`, the access method does not call **mi_qual_qual()** because the access method already knows the address of the qualification descriptor. For a simple qualification, **mi_qual_qual()** points to the same qualification descriptor as **mi_scan_quals()**.

If **mi_qual_nquals()** returns a nonzero value, the qualification descriptor combines nested qualifications in a complex expression. The access method can loop through **mi_qual_qual()** to process each qualification from those that AND or OR combine. For an example, refer to "Processing Complex Qualifications" on page 3-29.

Return Values

The pointer that this function returns provides the beginning address of the next qualification from a complex WHERE clause.

mi_qual_setoutput()

The **mi_qual_setoutput()** function sets a constant-argument value for a UDR.

Syntax

```
void
mi_qual_setoutput(MI_AM_QUAL_DESC *qualDesc, mi_integer n,
                MI_DATUM value, mi_boolean nullflag);
```

qualDesc points to the qualification descriptor.

n is always set to `0` to indicate the first and only argument that needs a value.

value passes the output value in a MI_DATUM data structure.

null_flag is MI_TRUE if *value* is NULL.

Usage

If a function declaration includes an OUT parameter, the function call in the WHERE clause includes a corresponding placeholder, called a *statement-local variable* (*SLV*). If the **mi_qual_needoutput()** function detects the presence of an SLV, the access method calls the **mi_qual_setoutput()** function to set a constant value for that SLV.

For examples of OUT parameters and SLVs, refer to "Runtime Values as Arguments" on page 3-23.

Return Values

None

Related Topic

See the description of accessor function **mi_qual_needoutput()**.

mi_qual_setreopt()

The **mi_qual_setreopt()** function sets an indicator in the qualification descriptor to force reoptimization.

Syntax

```
void  mi_qual_setreopt(MI_AM_QUAL_DESC *qualDesc)
```

qualDesc points to the qualification descriptor.

Usage

The **am_scancost** purpose function can call the **mi_qual_setreopt()** to indicate that the optimizer should reevaluate the query path between scans. For example, if either the **mi_qual_const_depends_hostvar()** or **mi_qual_const_depends_outer()** function returns MI_TRUE, the access method can call **mi_qual_setreopt()** to alert the optimizer that the constant-argument value in a qualification descriptor might change between scans on the same table.

If the access method sets **mi_qual_setreopt()**, the database server invokes the **am_scancost** purpose function before the next scan.

Return Values

None

Related Topics

See the descriptions of:

- accessor functions **mi_qual_const_depends_hostvar()** and **mi_qual_const_depends_outer()**.
- purpose function **am_scancost**.

mi_qual_setvalue()

The **mi_qual_setvalue()** function sets a qualification result.

Syntax

```
void mi_qual_setvalue(MI_AM_QUAL_DESC *qualDesc,
                      MI_AM_VALUE result_value);
```

qualDesc points to the qualification descriptor.

result_value indicates the result from executing the qualification.

MI_VALUE_TRUE indicates that the qualification is true.

MI_VALUE_FALSE indicates that the qualification is false.

MI_VALUE_NOT_EVALUATED indicates a pending evaluation.

Usage

The database server initializes all results in a qualification descriptor to MI_VALUE_NOT_EVALUATED. Typically, **am_getnext** makes a qualification test and then calls the **mi_qual_setvalue()** function to change *result_value* from MI_VALUE_NOT_EVALUATED to the test results (MI_VALUE_TRUE or MI_VALUE_FALSE).

When **am_getnext** sets all the qualifications that it can for a row, it calls the **mi_eval_am_qual()** function to handle any qualifications that it has not set. For an example, refer to “Processing Complex Qualifications” on page 3-29.

Return Values

None

Related Topics

See the descriptions of accessor functions **mi_eval_am_qual(),** **mi_init_am_qual()**, **mi_qual_boolop()**, and **mi_qual_qual().**

mi_qual_value()

The **mi_qual_value()** function retrieves the result of a qualification.

Syntax

```
MI_AM_VALUE mi_qual_value(MI_AM_QUAL_DESC *qualDesc);
```

qualDesc points to the qualification descriptor.

Usage

To evaluate a nested qualification, the access method can use a recursive function. If a previous recursion set a value for the qualification with the **mi_qual_setvalue()** or **mi_eval_am_qual()** function, **mi_qual_value()** returns MI_TRUE or MI_FALSE.

The access method can use the **mi_qual_value()** to obtain the MI_TRUE or MI_FALSE value for each argument to a Boolean expression. If **mi_qual_value** returns MI_VALUE_NOT_EVALUATED, evaluate the corresponding qualification or pass it to **mi_eval_am_qual()**.

For examples recursive evaluation, refer to "Processing Complex Qualifications" on page 3-29.

Return Values

MI_VALUE_TRUE indicates a satisfied qualification.

MI_VALUE_FALSE indicates one of the following:

- A previous function disqualified a column-argument value.
- A previous Boolean operation returned MI_FALSE.

MI_VALUE_NOT_EVALUATED indicates a qualification for which no results exist.

mi_scan_forupdate()

The **mi_scan_forupdate()** function determines if the SELECT query includes a FOR UPDATE clause.

Syntax

```
mi_boolean mi_scan_forupdate(MI_AM_SCAN_DESC *scanDesc);
```

scanDesc points to the scan descriptor.

Usage

The access method should protect data with the appropriate lock level for update transactions and possibly store user data for the **am_update** or **am_delete** purpose function.

To determine the lock level, call the **mi_scan_locktype()** access function.

Return Values

MI_TRUE indicates that the query includes a FOR UPDATE clause.

MI_FALSE indicates that the query does not include a FOR UPDATE clause.

Related Topic

See the description of accessor functions **mi_scan_locktype()** and **mi_tab_mode()**.

mi_scan_isolevel()

The **mi_scan_isolevel()** function retrieves the isolation level that the database server expects for the table that **am_getnext** scans.

Syntax

```
MI_ISOLATION_LEVEL mi_scan_isolevel(MI_AM_SCAN_DESC
*scanDesc);
```

scanDesc points to the scan descriptor.

Usage

If the access method supports isolation levels, it can call **mi_scan_isolevel()** from **am_beginscan** to determine the appropriate isolation level. For a detailed description of isolation levels, see "Checking Isolation Levels" on page 3-37.

Call **mi_scan_isolevel()** to validate that the isolation level requested by the application does not surpass the isolation level that the access method supports. If the access method supports Serializable, it does not call **mi_scan_isolevel()** because Serializable includes the capabilities of all the other levels.

Return Values

MI_ISO_NOTRANSACTION indicates that no transaction is in progress.

MI_ISO_READUNCOMMITTED indicates Dirty Read.

MI_ISO_READCOMMITTED indicates Read Committed.

MI_ISO_CURSORSTABILITY indicates Cursor Stability.

MI_ISO_REPEATABLEREAD indicates Repeatable Read.

MI_ISO_SERIALIZABLE indicates Serializable.

Related Topics

See the descriptions of:

- functions **mi_scan_locktype()** and **mi_tab_isolevel()**.
- isolation levels in "Checking Isolation Levels" on page 3-37.
- sample isolation-level language for access-method documentation in Figure 3-13 on page 3-44.

mi_scan_locktype()

The **mi_scan_locktype()** function retrieves the lock type that the database server expects for the table that **am_getnext** scans.

Syntax

```
MI_LOCK_TYPE mi_scan_locktype(MI_AM_SCAN_DESC *scanDesc);
```

scanDesc points to the scan descriptor.

Usage

If the access method supports locking, use the return value from this function to determine whether you need to lock an object during **am_getnext**.

Return Values

MI_LCK_S indicates a shared lock on the table.

MI_LCK_X indicates an exclusive lock on the table.

MI_LCK_IS_S indicates an intent-shared lock on the table and shared lock on the row.

MI_LCK_IX_X indicates intent-exclusive lock on the table and exclusive lock on the row.

MI_LCK_SIX_X indicates an intent-shared exclusive lock on the table and an exclusive lock on the row.

Related Topics

See the descriptions of:

- functions **mi_scan_isolevel()** and **mi_scan_forupdate()**.
- locks in the *Performance Guide*.

mi_scan_nprojs()

The **mi_scan_nprojs()** function returns a value that is 1 less than the number of columns in a query projection.

Syntax

```
mi_integer  mi_scan_nprojs(MI_AM_SCAN_DESC *scanDesc)
```

scanDesc points to the scan descriptor.

Usage

Use the return value from this function to determine the number times to loop through the related **mi_scan_projs()** function.

The **mi_scan_nprojs()** function returns 2 for to indicate that the following SELECT statement projects 3 columns:

```
SELECT column1, column2, column3 FROM table
```

Return Values

The integer return value indicates the number of columns that the SELECT clause of a query specifies.

Related Topic

See the description of accessor function **mi_scan_projs()**.

mi_scan_newquals()

The **mi_scan_newquals()** function indicates whether the qualification descriptor includes changes between multiple scans for the same query statement.

Syntax

```
mi_boolean mi_scan_newquals(MI_AM_SCAN_DESC *scanDesc);
```

scanDesc points to the scan descriptor.

Usage

This function pertains to multiple-scan queries, such as a join or subquery. If the access method provides a function for the **am_rescan** purpose, that rescan function calls **mi_scan_newquals()**.

If this function returns MI_TRUE, retrieve information from the qualification descriptor and obtain function descriptors. If it returns MI_FALSE, retrieve state information that the previous scan stored in user data.

Return Values

MI_TRUE indicates that the qualifications have changed since the start of the scan (**am_beginscan**). MI_FALSE indicates that the qualifications have not changed.

mi_scan_projs()

The **mi_scan_projs()** function identifies each column that the SELECT clause of a query specifies.

Syntax

```
mi_smallint * mi_scan_projs(MI_AM_SCAN_DESC *scanDesc)
```

scanDesc points to the scan descriptor.

Usage

Use the return value from **mi_scan_nprojs()** to determine the number of times to execute **mi_scan_projs()**. Then use **mi_scan_projs()** to identify columns that the return row must contain.

A qualification identifies a column by a number that locates the column in the row descriptor. The number 0 indicates the first column in the row descriptor. In the following example, **mi_scan_projs()** points to the values 1, 5, and 4:

```
SELECT column1, column5, column4 FROM table
```

The row descriptor describes the columns in the order that they appear in the CREATE TABLE statement. The following example shows how to determine the data type of each projected column:

```
MI_TYPE_DESC    *typedesc;
MI_AM_TABLE_DESC*td;
MI_ROW_DESC     *rd;
MI_AM_SCAN_DESC *sd;
mi_integer      n;
mi_smallint     c, *projcols;   /* column identifiers */
rd = mi_tab_rowdesc(td);    /* describes a table row*/
n = mi_scan_nprojs(sd); /*How many columns are projected?*/
projcols=mi_scan_projs(sd);/* identifies projected columns*/

for (int i = 0; i < n; i++)
{
    c = projcols[i];       /* Get offset to row descriptor.*/
    /* Get data type for projected column. For example
    ** my_data->col_type[c] = mi_column_typedesc(rd, c) */
}
```

Tip: *Because the access method needs to return data for only the columns that make up the projection, the access method can put a NULL value in the remaining columns. Eliminate unnecessary column data to improve performance and reduce the resources that the database server allocates to format and store the returned rows.*

Return Values

Each of the small integers in the array that this function returns identifies a column by the position of that column in the row descriptor.

Related Topics

See the descriptions of:

- accessor functions **mi_scan_nprojs()**, **mi_scan_table()**, and **mi_tab_rowdesc()**.
- the **mi_column_*** group of DataBlade API functions and the row descriptor (MI_ROW_DESC data structure) in the *DataBlade API Programmer's Manual*.

mi_scan_quals()

The **mi_scan_quals()** function returns the qualification descriptor, which describes the conditions that an entry must satisfy to qualify for selection.

Syntax

```
MI_AM_QUAL_DESC* mi_scan_quals(MI_AM_SCAN_DESC *scanDesc);
```

scanDesc points to the scan descriptor.

Usage

The **am_getnext** purpose function calls **mi_scan_quals()** to obtain the starting point from which it evaluates a row and then passes the return value (a pointer) from this function to all the qualification-descriptor accessor functions.

Important: *If this function returns a null-valued pointer, the access method sequentially scans the table and returns all rows.*

Return Values

A valid pointer indicates the start of the qualification descriptor for this scan. A NULL-valued pointer indicates that the access method should return all row.

Related Topics

See the description of the accessor functions in "Qualification Descriptor" on page 5-7.

mi_scan_setuserdata()

The **mi_scan_setuserdata()** function stores a pointer to user data in the scan descriptor.

Syntax

```
void mi_scan_setuserdata(MI_AM_SCAN_DESC *scanDesc, void
*user_data);
```

scanDesc points to the scan descriptor.

user_data points to the user data.

Usage

The access method can create a user-data structure in shared memory to store reusable information, such as function descriptors for qualifications and to maintain a row pointer for each execution of the **am_getnext** purpose function. To retain user data in memory during the scan (starting when **am_beginscan** is called and ending when **am_endscan** is called), follow these steps:

1. In the **am_beginscan** purpose function, call the appropriate DataBlade API function to allocate memory for the user-data structure.

   Allocate the user-data memory with a duration of PER_COMMAND.
2. In **am_getnext**, populate the user-data structure with scan-state information.
3. Before **am_getnext** exits, call **mi_scan_setuserdata()** to store a pointer to the user-data structure in the scan descriptor.
4. In the **am_endscan** purpose function, call the appropriate DataBlade API function to deallocate the user-data memory.

Return Values

None

Related Topics

See the descriptions of:

- function **mi_scan_userdata()**.
- DataBlade API functions for memory allocation and duration in "Storing Data in Shared Memory" on page 3-4.

mi_scan_table()

The **mi_scan_table()** function retrieves a pointer to the table descriptor for the table that the access method scans.

Syntax

```
MI_AM_TABLE_DESC* mi_scan_table(MI_AM_SCAN_DESC *scanDesc);
```

scanDesc points to the scan descriptor.

Usage

The table descriptor points to the row descriptor. The row descriptor contains the column data types that define a row.

The table descriptor also typically contains PER_STATEMENT user data that remains in memory until the completion of the current SQL statement.

Return Values

This function returns a pointer to the table descriptor that is associated with this scan.

Related Topics

See the descriptions of:

- accessor functions in "Table Descriptor" on page 5-13.
- accessor functions for the row descriptor in the *DataBlade API Programmer's Manual*.

mi_scan_userdata()

The **mi_scan_userdata()** function retrieves the pointer from the scan descriptor that points to a user data structure.

Syntax

```
void* mi_scan_userdata(MI_AM_SCAN_DESC *scanDesc);
```

scanDesc points to the scan descriptor.

Usage

If the access method allocates user-data memory to hold scan-state information, it places a pointer to that user data in the scan descriptor. Use the **mi_scan_userdata()** function to retrieve the pointer for access to the user data.

For example, the **am_getnext** might maintain a row pointer to keep track of its progress through the table during a scan. Each time **am_getnext** prepares to exit, it stores the address or row identifier of the row that it just processed. The next execution of **am_getnext** retrieves and increments the address to fetch the next row in the table.

Return Values

This function returns a pointer to a user-data structure that the access method creates during the scan.

Related Topic

See the description of:

- function **mi_scan_setuserdata()**.
- "Storing Data in Shared Memory" on page 3-4.

mi_tab_amparam()

The **mi_tab_amparam()** function retrieves any user-defined configuration values for the table.

Syntax

```
mi_string* mi_tab_amparam(MI_AM_TABLE_DESC *tableDesc);
```

tableDesc points to the table descriptor.

Usage

If the access method supports configuration keywords, the USING *access-method* clause of the CREATE TABLE statement can specify values for those keywords. A user or application can apply values to adjust the way in which the access method behaves.

Return Values

The pointer accesses a string that contains user-specified keywords and values. A NULL-valued pointer indicates that the CREATE TABLE statement specified no configuration keywords.

Related Topics

See the descriptions of:

- "Providing Configuration Keywords" on page 3-18.
- the USING clause of the CREATE TABLE statement in the *Informix Guide to SQL: Syntax*.

mi_tab_check_msg()

The **mi_tab_check_msg()** function sends messages to the **oncheck** utility.

Syntax

```
mi_integer mi_tab_check_msg(MI_AM_TABLE_DESC *tableDesc,
    mi_integer msg_type,
    char *msg[, marker_1, ..., marker_n])
```

tableDesc points to the descriptor for the table that the **oncheck** command line specifies.

msg_type indicates where **oncheck** should look for the message.

If *msg_type* is MI_SQL, an error occurred. The **syserrors** system catalog table contains the message.

If *msg_type* is MI_MESSAGE, the pointer in the *msg* argument contains the address of an information-only message string.

msg points to a message string of up to 400 bytes if *msg_type* is MI_MESSAGE.

If *msg_type* is MI_SQL, *msg* points to a 5-character **SQLSTATE** value. The value identifies an error or warning in the **syserrors** system catalog table.

marker_n specifies a marker name in the **syserrors** system catalog table and a value to substitute for that marker.

When a user initiates the **oncheck** utility, the database server invokes the **am_check** purpose function, which checks the structure and integrity of virtual tables. To report state information to the **oncheck** utility, **am_check** can call the **mi_tab_check_msg()** function.

The **syserrors** system catalog table can contain user-defined error and warning messages. A five-character **SQLSTATE** value identifies each message.

The text of an error or warning message can include markers that the access method replaces with state-specific information. To insert state-specific information in the message, the access method passes values for each marker to **mi_tab_check_msg()**.

To raise a exception whose message text is stored in **syserrors**, provide the following information to the **mi_tab_check_msg()** function:

- A message type of MI_SQL
- The value of the **SQLSTATE** variable that identifies the custom exception
- Optionally, values specified in parameter pairs that replace markers in the custom exception message

The access method can allocate memory for messages or create automatic variables that keep their values for the duration of the **mi_tab_check_msg()** function.

The DataBlade API **mi_db_error_raise()** function works similarly to **mi_tab_check_msg()**. For examples that show how to create messages, refer to the description of **mi_db_error_raise()** in the *DataBlade API Programmer's Manual*.

Important: *Do not use msg_type values MI_FATAL or MI_EXCEPTION with* ***mi_tab_check_msg()****. These message types are reserved for the DataBlade API function* ***mi_db_error_raise()****.*

Return Values

None

Related Topics

See the descriptions of:

- purpose function **am_check** on page 4-15.
- DataBlade API function **mi_db_error_raise()** in the *DataBlade API Programmer's Manual*, particularly the information about raising custom messages.
- **oncheck** in the *Administrator's Reference*.

mi_tab_createdate()

The **mi_tab_createdate()** function returns the date that the table was created.

Syntax

```
mi_date *mi_tab_createdate(MI_AM_TABLE_DESC *tableDesc);
```

tableDesc points to the table descriptor.

Return Value

The date indicates when the CREATE TABLE statement was issued.

mi_tab_id()

The **mi_tab_id()** function retrieves the table identifier from the table descriptor.

Syntax

```
mi_integer mi_tab_id(MI_AM_TABLE_DESC *tableDesc)
```

tableDesc points to the table descriptor.

Usage

The access method can call the **mi_tab_id()** function to determine the unique identifier that the **systables** system catalog table associates with the virtual table.

Return Values

The return value identifies the table to the database server in the **tabid** column of the **systables** or **sysfragments** system catalog table.

The table identifier is identical for each fragment in the table.

mi_tab_isolevel()

The **mi_tab_isolevel()** function retrieves the isolation level that the SET ISOLATION or SET TRANSACTION statement applies.

Syntax

```
MI_ISOLATION_LEVEL mi_tab_isolevel(MI_AM_TAB_DESC
*tableDesc);
```

tableDesc points to the table descriptor.

Usage

If the access method supports isolation levels, it can call **mi_tab_isolevel()** to validate that the isolation level requested by the application does not surpass the isolation level that the access method supports. If the access method supports Serializable, it does not call **mi_tab_isolevel()** because Serializable includes the capabilities of all the other levels.

Return Values

MI_ISO_NOTRANSACTION indicates that no transaction is in progress.
MI_ISO_READUNCOMMITTED indicates Dirty Read.
MI_ISO_READCOMMITTED indicates Read Committed.
MI_ISO_CURSORSTABILITY indicates Cursor Stability.
MI_ISO_REPEATABLEREAD indicates Repeatable Read.
MI_ISO_SERIALIZABLE indicates Serializable.

Related Topics

See the descriptions of:

- functions **mi_scan_locktype()** and **mi_scan_isolevel()**.
- isolation levels in "Checking Isolation Levels" on page 3-37.
- sample isolation-level language for access-method documentation in Figure 3-13 on page 3-44.

mi_tab_istable()

The **mi_tab_istable()** function indicates whether the table descriptor describes a table.

Syntax

```
mi_boolean mi_tab_istable(MI_AM_TABLE_DESC *tableDesc)
```

tableDesc points to the table descriptor.

Usage

If the access method shares source files with a secondary access method, use this function to verify that the table descriptor belongs to the primary access method.

Return Values

MI_TRUE indicates that the table descriptor pertains to a table. MI_FALSE indicates that it describes an index.

mi_tab_mode()

The **mi_tab_mode()** function retrieves the I/O mode of the table from the table descriptor.

Syntax

```
mi_unsigned_integer
mi_tab_tab_mode(MI_AM_TABLE_DESC *tableDesc)
```

tableDesc points to the table descriptor.

Usage

The I/O mode refers to the operations expected subsequent to the opening of a table. To determine the input and output requirements of the current statement:

1. Call **mi_tab_mode()** to obtain an input-output indicator.
2. Pass the value that **mi_tab_mode()** returns to the macros in Figure 5-1 for interpretation.

   Each macro returns either MI_TRUE or MI_FALSE.

Figure 5-1
Macro Modes

Macro	Mode Verified
MI_INPUT()	Open for input only, usually in the case of a SELECT statement
MI_OUTPUT()	Open for output only, usually in the case of an INSERT statement
MI_INOUT()	Open for input and output, usually in the case of an UPDATE statement
MI_NOLOG()	No logging required

In the following example, the access method calls **mi_tab_mode()** to verify that a query is read-only. If **MI_INOUT()** returns MI_FALSE, the access method requests a multiple-row buffer because the access method can return several rows without interruption by an update:

```
if (MI_INOUT(tableDesc) == MI_FALSE)
    mi_tab_setniorows(tableDesc, 10);
```

If **MI_INOUT()** returns MI_TRUE, the access method can process only one row identifier with each call to **am_getnext**.

Return Values

The integer indicates whether an input or output request is active.

To interpret the returned integer, use the macros that Figure 5-1 on page 5-64 describes.

Related Topics

See the descriptions of

- "Buffering Multiple Results" on page 3-35.
- purpose functions **am_beginscan** and **am_getnext**.
- setting logging preferences in Figure 3-5 on page 3-18.

mi_tab_name()

The **mi_tab_name()** function retrieves the table name that the active SQL statement or **oncheck** command specifies.

Syntax

```
mi_string* mi_tab_name(MI_AM_TABLE_DESC *tableDesc)
```

tableDesc points to the table descriptor.

Return Values

The name specifies the table to access. The table name is identical for each fragment in the table.

mi_tab_niorows()

The **mi_tab_niorows()** function retrieves the number of rows that the database server expects to process in **am_getnext**.

Syntax

```
mi_integer
mi_tab_niorows(MI_AM_TABLE_DESC *tableDesc)
```

tableDesc points to the table descriptor.

Usage

Call this function from **am_getnext** and then loop through the scan as often as necessary to fill the reserved number of rows or until no more rows qualify. See **mi_tab_setnextrow()** for an example.

Return Values

The maximum number of rows that **am_getnext** can place in shared memory.

A return value of 0 indicates that **am_open** or **am_beginscan** did not call the **mi_tab_setniorows()** function or that **mi_tab_setniorows()** returned an error. Thus, the database server did not reserve memory for multiple rows, and the access method must process only one row.

A negative return value indicates an error.

Related Topics

See the descriptions of functions **mi_tab_setniorows()** and **mi_tab_setnextrow()**.

mi_tab_numfrags()

The **mi_tab_numfrags()** function retrieves the number of fragments in the table.

Syntax

```
mi_integer mi_tab_numfrags(MI_AM_TABLE_DESC *tableDesc)
```

tableDesc points to the table descriptor.

Return Values

The integer specifies the number of fragments in the table from the table descriptor. If the table is not fragmented, **mi_tab_numfrags()** returns `1`.

mi_tab_owner()

The **mi_tab_owner()** function retrieves the owner of the table.

Syntax

```
mi_string* mi_tab_owner(MI_AM_TABLE_DESC *tableDesc)
```

tableDesc points to the table descriptor.

Usage

The user who creates a table owns that table. The database server identifies the owner by user ID, which it stores in the **systables** system catalog table. In some environments, user ID of the table owner must precede the table name as follows:

```
SELECT * from owner.table_name
```

Return Values

The string contains the user ID of the table owner.

Related Topic

See the description of the Owner Name segment in the *Informix Guide to SQL: Syntax*.

mi_tab_partnum()

The **mi_tab_partnum()** function retrieves the fragment identifier for the table.

Syntax

```
mi_integer mi_tab_partnum(MI_AM_TABLE_DESC *tableDesc)
```

tableDesc points to the table descriptor.

Usage

If a CREATE TABLE or ALTER FRAGMENT statement specifies fragmentation, use this function to determine the current fragment identifier (also called a partition number). Each fragment occupies one named sbspace or extspace.

Return Values

The integer specifies physical address of the fragment. If the table is not fragmented, the return value corresponds to the **partnum** value for this table in the **systables** system catalog table.

For a fragmented table, the return value corresponds to the fragment identifier and the **partn** value in the **sysfragments** system catalog table.

mi_tab_rowdesc()

The **mi_tab_rowdesc()** function retrieves the row descriptor, which describes the columns that belong to the table that the table descriptor identifies.

Syntax

```
MI_ROW_DESC* mi_tab_rowdesc(MI_AM_TABLE_DESC *tableDesc)
```

tableDesc points to the table descriptor.

Usage

To access information in the row descriptor, pass the pointer in this column to the DataBlade API row-descriptor accessor functions. A row descriptor describes the columns from that make up the table.

The order of the columns in the row descriptor corresponds to the order of the columns in the CREATE TABLE statement. Another accessor function, such as **mi_scan_projs()**, can obtain information about a specific column by passing the position of the column in the row descriptor.

Return Values

The pointer enables the access method to locate the row descriptor, which describes the columns in this table.

Related Topics

Refer to the *DataBlade API Programmer's Manual* for the descriptions of:

- DataBlade API row-descriptor accessor functions **mi_column_bound()**, **mi_column_count()**, **mi_column_id()**, **mi_column_name()**, **mi_column_nullable()**, **mi_column_scale()**, **mi_column_type_id()**, and **mi_column_typedesc()**.
- the row descriptor (MI_ROW_DESC data structure).

mi_tab_setnextrow()

The **am_getnext** purpose function calls **mi_tab_setnextrow()** to store the next entry that qualifies for selection.

Syntax

```
mi_integer
mi_tab_setnextrow(MI_AM_TABLE_DESC  *tableDesc,
            MI_ROW **row,
            mi_integer *rowid,
            mi_integer *fragid)
```

tableDesc points to the table descriptor.

row points to the address of a row structure that contains fetched data.

rowid points to the row identifier of the fetched values.

fragid is always `0` because the table descriptor identifies the fragment.

Usage

Use this function in the **am_getnext** purpose function if the access method can fetch multiple rows into shared memory. The values in *row* and *rowid* replace arguments that the database server passes to **am_getnext** if shared memory accommodates only one fetched row.

The **mi_tab_setnextrow()** function works together with the following other accessor functions:

- The **mi_tab_setniorows()** function sets a number of rows to pass to **am_getnext.**
- The **mi_tab_niorows()** function gets the number of rows to expect.

For an example that shows how these three functions work together, refer to Figure 3-12 on page 3-36.

Return Values

The integer indicates which row in shared memory to fill. The first call to **mi_tab_setnextrow()** returns 0. Each subsequent call adds 1 to the previous return value. The maximum rows available depends on the value that **mi_tab_niorows()** returns.

A negative return value indicates an error.

Related Topics

See the descriptions of:

- functions **mi_tab_setniorows()** and **mi_tab_niorows()**.
- "Buffering Multiple Results" on page 3-35.

mi_tab_setniorows()

The **mi_tab_setniorows()** function indicates:

- that the access method can handle more than one row per call.
- the number of rows for which the database server should allocate memory.

Syntax

```
mi_integer mi_tab_setniorows(MI_AM_TABLE_DESC *tableDesc,
                             mi_integer nrows)
```

tableDesc points to the table descriptor.

nrows specifies the maximum number of rows that **am_getnext** processes.

Usage

The access method must call this function in either **am_open** or **am_beginscan**. Multiple calls to **mi_tab_setniorows()** during the execution of a single statement cause an error.

Return Values

The integer indicates the actual number of rows for which the database server allocates memory. Currently, the return value equals *nrows*. A zero or negative return value indicates an error.

Related Topics

See the descriptions of functions **mi_tab_niorows()** and **mi_tab_setnextrow()**.

mi_tab_setuserdata()

The **mi_tab_setuserdata()** function stores a pointer to user data in the table descriptor.

Syntax

```
void mi_tab_setuserdata(MI_AM_TABLE_DESC *tableDesc,
                        void *user_data)
```

tableDesc points to the table descriptor.

user_data points to a data structure that the access method creates.

Usage

The access method stores state information from one purpose function so that another purpose function can use it.

To save table-state information as user data

1. Call the appropriate DataBlade API memory-management function to allocate PER_STATEMENT memory for the user-data structure.
2. Populate the user-data structure with the state information.
3. Call the **mi_tab_setuserdata()** function to store the pointer that the memory-allocation function returns in the table descriptor.
4. Pass the pointer as the *user_data* argument.

Typically, an access method performs the preceding procedure in the **am_open** purpose function and deallocates the user-data memory in the **am_close** purpose function. To have the table descriptor retain the pointer to the user data as long as the table remains open, specify a memory duration of PER_STATEMENT, as "Memory-Duration Options" on page 3-5 and "Persistent User Data" on page 3-6 describe.

To retrieve the pointer from the table descriptor to access the table-state user data, call the **mi_tab_userdata()** function in any purpose function between **am_open** and **am_close**.

Return Values

None

Related Topics

See the descriptions of:

- function **mi_tab_userdata()**.
- purpose functions **am_open** and **am_close**.
- DataBlade API functions for memory allocation and duration in "Storing Data in Shared Memory" on page 3-4.

mi_tab_spaceloc()

The **mi_tab_spaceloc()** function retrieves the location of the extspace in which the table resides.

Syntax

```
mi_string* mi_tab_spaceloc(MI_AM_TABLE_DESC *tableDesc)
```

tableDesc points to the table descriptor.

Usage

A user, usually a database system administrator, can assign a short name to an extspace with the **onspaces** utility. When a user creates a table, the CREATE TABLE statement can include an IN clause to specify one of the following:

- The name that is assigned with the **onspaces** utility
- A string that contains the actual location

To find out the string that the user specifies as the storage space, call the **mi_tab_spaceloc()** function.

For example, the **mi_tab_spaceloc()** function returns the string `host=dcserver,port=39` for a storage space that the following commands specify:

```
onspaces -c -x dc39 -l "host=dcserver,port=39"
CREATE TABLE remote...
    IN dc39
    USING access_method
```

Return Values

A string identifies the extspace.

If the table resides in an sbspace, this function returns a NULL-valued pointer.

mi_tab_spacename()

The **mi_tab_spacename()** function retrieves the name of the storage space where the virtual table resides.

Syntax

```
mi_string* mi_tab_spacename(MI_AM_TABLE_DESC *tableDesc)
```

tableDesc points to the table descriptor.

Usage

Call the **mi_tab_spacename()** function to determine the storage space identifier from one of the following sources:

- An IN clause specification
- The SBSPACENAME value in the database ONCONFIG file

IN Clause

When a user creates a table, the CREATE TABLE statement can include an IN clause that specifies one of the following:

- The name that is assigned with the **onspaces** utility
- A string that contains the actual location

For example, the **mi_tab_spacename()** function returns the string `dc39` for a storage space that the following commands specify:

```
onspaces -c -x dc39 -l "host=dcserver,port=39"
CREATE TABLE remote...
    IN dc39
    USING access_method
```

The statement that creates the table can specify the physical storage location rather than a logical name that the **onspaces** utility associates with the storage space. In the following UNIX example, **mi_tab_spacename()** returns the physical path, /tmp:

```
CREATE TABLE remote...
    IN '/tmp'
    USING access_method
```

If the IN clause specifies multiple storage spaces, each makes up a fragment of the table, and the table descriptor pertains to only the fragment that the return value for the **mi_tab_spacename()** function names.

SBSPACENAME Value

An optional SBSPACENAME parameter in the ONCONFIG file indicates the name of an existing sbspace as the default location to create new smart large objects or virtual tables. The database server assigns the default sbspace to a virtual table under the following circumstances:

- A CREATE TABLE statement does not include an IN clause.
- The database server determines (from the **am_sptype** purpose value in the **sysams** system catalog table) that the access method supports sbspaces.
- The ONCONFIG file contains a value for the SBSPACENAME parameter.
- The **onspaces** command created an sbspace with the name that SBSPACENAME specifies.
- The default sbspace does not contain a table due to a previous SQL statement.

For more information, refer to "Creating a Default Storage Space" on page 3-14.

Return Values

A string identifies the sbspace or extspace that the CREATE TABLE statement associates with the table. A NULL-valued pointer indicates that the table does not reside in a named storage space.

mi_tab_spacetype()

The **mi_tab_spacetype()** function retrieves the type of storage space in which the virtual table resides.

Syntax

```
mi_char1 mi_tab_spacetype(MI_AM_TABLE_DESC *tableDesc)
```

tableDesc points to the table descriptor.

Return Values

The letter S indicates that the table resides in an sbspace. The letter X indicates that the table resides in an extspace. The letter D indicates that the table resides in a dbspace and is reserved for Informix use only.

Important: *A user-defined access method cannot create tables in dbspaces.*

mi_tab_update_stat_mode()

The **mi_tab_update_stat_mode()** function indicates whether an UPDATE STATISTICS function includes a LOW, MEDIUM, or HIGH mode keyword.

Syntax

```
MI_UPDATE_STAT_MODE
mi_tab_update_stat_mode(MI_AM_TABLE_DESC *tableDesc))
```

tableDesc points to the table descriptor.

Usage

To extract the distribution-level keyword that an UPDATE STATISTICS statement specifies, the **am_stats** purpose function calls the **mi_tab_update_stat_mode()** function. Three keywords describe distribution level: HIGH, MEDIUM, and the default LOW.

If a purpose function other than **am_stats** calls **mi_tab_update_stat_mode()**, the return value indicates that UPDATE STATISTICS is not running.

Return Values

MI_US_LOW indicates that the UPDATE STATISTICS statement specifies the LOW keyword or that LOW is in effect by default. MI_US_MED or MI_US_HIGH indicates that the UPDATE STATISTICS specifies the MEDIUM or the HIGH keyword, respectively. MI_US_NOT_RUNNING indicates that no UPDATE STATISTICS statement is executing. MI_US_ERROR indicates an error.

Related Topics

See the descriptions of:

- purpose function **am_stats** on page 4-37.
- UPDATE STATISTICS in the *Informix Guide to SQL: Syntax* and the *Performance Guide*.

mi_tab_userdata()

The **mi_tab_userdata()** function retrieves, from the table descriptor, a pointer to a user-data structure that the access method maintains in shared memory.

Syntax

```
void* mi_tab_userdata(MI_AM_TABLE_DESC *tableDesc)
```

tableDesc points to the table descriptor.

Usage

During the **am_open** purpose function, the access method can create and populate a user-data structure in shared memory. The table descriptor user data generally holds state information about the table for use by other purpose functions. To ensure that the user data remains in memory until **am_close** executes, the access method allocates the memory with a duration of PER_STATEMENT.

To store the pointer in that structure in the table descriptor, **am_open** calls **mi_tab_setuserdata()**. Any other purpose function can call **mi_tab_userdata()** to retrieve the pointer for access to the state information.

Return Values

The pointer indicates the location of a user-data structure in shared memory.

Related Topic

See the descriptions of:

- function **mi_tab_setuserdata()**.
- "Storing Data in Shared Memory" on page 3-4.

mi_tstats_setnpages()

The **mi_tstats_setnpages()** function stores the number of table pages in the statistics descriptor.

Syntax

```
void mi_tstats_setnpages(MI_AM_TSTATS_DESC *tstatsDesc,
                         mi_integer npages)
```

tstatsDesc points to the statistics descriptor.

npages provides the number of pages in the table.

Usage

The **am_stats** purpose function sets the number of data pages, which the database server stores in the **npused** column of the **systables** system catalog table. The optimizer uses the number of pages in a table to choose an optimal query path.

Return Values

None

mi_tstats_setnrows()

The **mi_tstats_setnrows()** function stores the number of table rows in the statistics descriptor.

Syntax

```
void mi_tstats_setnrows(MI_AM_TSTATS_DESC *tstatsDesc,
                        mi_integer nrows)
```

tstatsDesc points to the statistics descriptor.

nrows provides the number of rows in the table.

Usage

The **am_stats** purpose function sets the number of rows in the table, which the database server stores in the **nrows** column of the **systables** system catalog table. The optimizer uses it to choose an optimal query path.

Return Values

None

Chapter

6

SQL Statements for Access Methods

In This Chapter

This chapter describes the syntax and usage of the following SQL statements, which insert, change, or delete entries in the **sysams** system catalog table:

- ALTER ACCESS_METHOD
- CREATE PRIMARY ACCESS_METHOD
- DROP ACCESS_METHOD

For information about how to interpret the syntax diagrams in this chapter, refer to "Syntax Conventions" on page 9 of the Introduction.

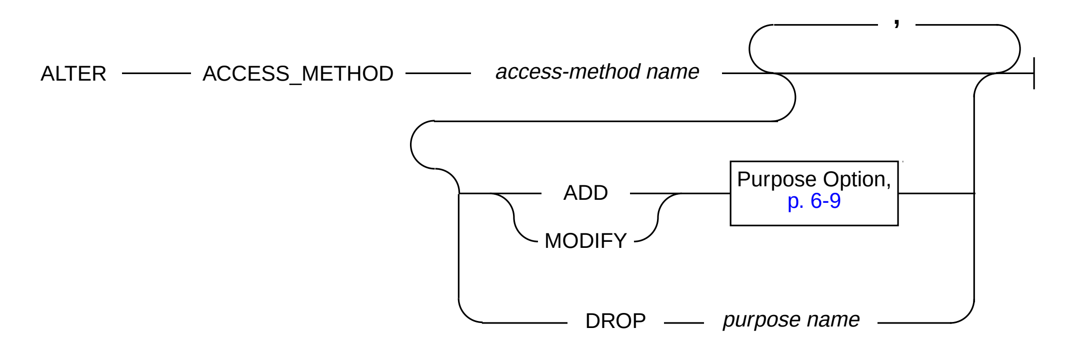
This chapter also provides the valid purpose-function, purpose-flag, and purpose-value settings.

+

ALTER ACCESS_METHOD

The ALTER ACCESS_METHOD statement changes the attributes of a user-defined access method in the **sysams** system catalog table.

Syntax



Element	Purpose	Restrictions	Syntax
access-method name	The access method to alter	A previous CREATE PRIMARY ACCESS_METHOD statement must register the access method in the database.	Database Object Name segment; see *Informix Guide to SQL: Syntax*.
purpose name	A keyword that indicates which purpose function, purpose value, or purpose flag to drop	A previous statement must associate the purpose name with this access method.	"Purpose-Name Keyword" on page 6-12.

Usage

Use ALTER ACCESS_METHOD to modify the definition of a user-defined access-method. You must be the owner of the access method or have DBA privileges to alter an access method.

When you alter an access method, you change the purpose-option specifications (purpose functions, purpose flags, or purpose values) that define the access method. For example, you alter an access method to assign a new purpose-function name or provide a multiplier for the scan cost. For detailed information about how to set purpose-option specifications, refer to "Purpose Options" on page 6-9.

If a transaction is in progress, the database server waits to alter the access method until the transaction is committed or rolled back. No other users can execute the access method until the transaction has completed.

Sample Statements

The following statement alters the **remote** access method.

```
ALTER ACCESS_METHOD remote
ADD AM_INSERT=ins_remote,
ADD AM_READWRITE,
DROP AM_CHECK,
MODIFY AM_SPTYPE = 'X' );
```

Figure 6-1
Sample ALTER ACCESS_METHOD Statement

The preceding example:

- adds an **am_insert** purpose function.
- drops the **am_check** purpose function.
- sets (adds) the **am_readwrite** flag.
- modifies the **am_sptype** purpose value.

References

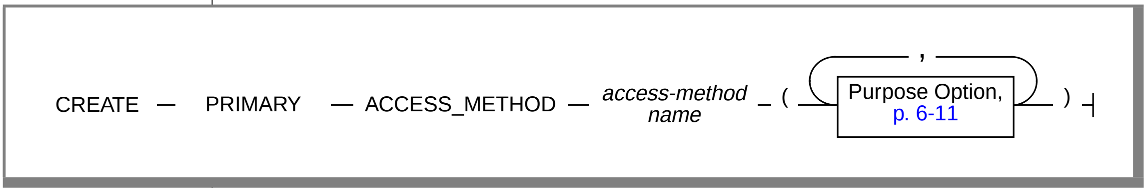
See the descriptions of:

- CREATE ACCESS_METHOD statement and Purpose Options in this chapter.
- privileges in the *Informix Guide to Database Design and Implementation* or the GRANT statement in the *Informix Guide to SQL: Syntax*.

CREATE ACCESS_METHOD

Use the CREATE PRIMARY ACCESS_METHOD statement to register a new primary access method. When you register an access method, the database server places an entry in the **sysams** system catalog table.

Syntax



Element	Purpose	Restrictions	Syntax
access-method name	The access method to add	The access method must have a unique name in the **sysams** system catalog table.	Database Object Name segment; see *Informix Guide to SQL: Syntax*.

Usage

The CREATE PRIMARY ACCESS_METHOD statement adds a user-defined access method to a database. When you create an access method, you specify purpose functions, purpose flags, or purpose values as attributes of the access method. To set purpose options, refer to "Purpose Options" on page 6-9.

You must have the DBA or Resource privilege to create an access method. For information about privileges, refer to the *Informix Guide to Database Design and Implementation* or the GRANT statement in the *Informix Guide to SQL: Syntax*.

Sample Statements

The following statement creates a primary access method named **textfile** that resides in an extspace. The **am_getnext** purpose function is assigned to a function name that already exists. The **textfile** access method supports clustering.

```
CREATE PRIMARY ACCESS_METHOD textfile(
AM_GETNEXT = textfile_getnext,
AM_CLUSTER,
AM_SPTYPE = 'X' );
```

Figure 6-2
Sample CREATE PRIMARY ACCESS_METHOD Statement

References

See the descriptions of:

- ALTER ACCESS_METHOD and DROP ACCESS_METHOD statements, as well as Purpose Options, in this chapter.
- privileges in the *Informix Guide to Database Design and Implementation* or the GRANT statement in the *Informix Guide to SQL: Syntax*.

DROP ACCESS_METHOD

Use the DROP ACCESS_METHOD statement to remove a previously defined access method from the database.

Syntax

DROP ——— ACCESS_METHOD ——— *access-method name* ——— RESTRICT ——|

Element	Purpose	Restrictions	Syntax
access-method name	The access method to drop	The access method must be registered in the **sysams** system catalog table with a previous CREATE ACCESS_METHOD statement.	Database Object Name segment; see *Informix Guide to SQL: Syntax*.

Usage

The RESTRICT keyword is required. You cannot drop an access method if tables exist that use that access method.

If a transaction is in progress, the database server waits to drop the access method until the transaction is committed or rolled back. No other users can execute the access method until the transaction has completed.

You must own the access method or have the DBA privilege to use the DROP ACCESS_METHOD statement.

References

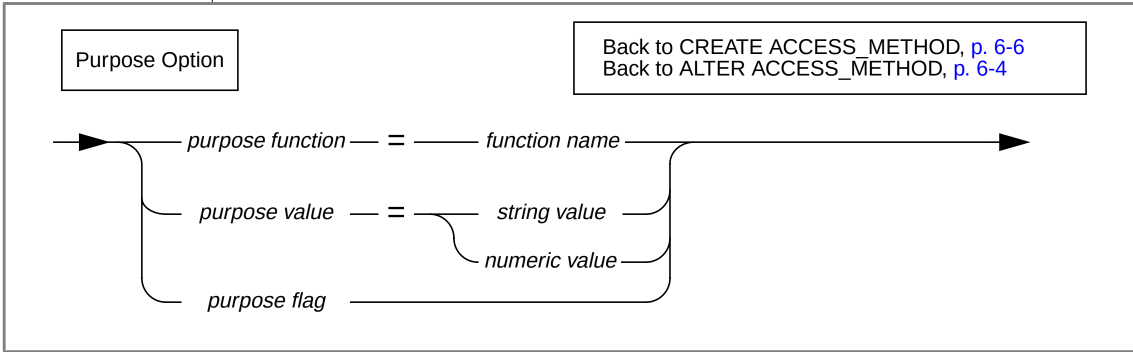
See the descriptions of:

- CREATE ACCESS_METHOD and ALTER ACCESS_METHOD statements in this chapter.
- keyword RESTRICT in the *Informix Guide to SQL: Syntax*.
- privileges in the *Informix Guide to Database Design and Implementation* or the GRANT statement in the *Informix Guide to SQL: Syntax*.

Purpose Options

The database server recognizes a registered access method as a set of attributes, including the access-method name and options called *purposes*. The CREATE PRIMARY ACCESS_METHOD and ALTER ACCESS_METHOD statements specify purpose attributes with the following syntax.

Syntax



Element	Purpose	Restrictions	Syntax
purpose function	A keyword that specifies a task and the corresponding access-method function	The interface specifies the predefined purpose-function keywords to which you can assign UDR names. You cannot name a UDR with the same name as the keyword.	Function purpose category; see Figure 6-3 on page 6-12.
purpose value	A keyword that identifies configuration information	The interface specifies the predefined configuration keywords to which you can assign values.	Value purpose category; see Figure 6-3 on page 6-12.
purpose flag	A keyword that indicates which feature a flag enables	The interface specifies flag names.	Flag purpose category; see Figure 6-3 on page 6-12.

(1 of 2)

Element	Purpose	Restrictions	Syntax
function name	The user-defined function that performs the tasks of the specified purpose function	A CREATE FUNCTION statement must register the function in the database.	Database Object Name segment; see *Informix Guide to SQL: Syntax*.
string value	An indicator that is expressed as one or more characters	None.	Quoted String segment; see *Informix Guide to SQL: Syntax*.
numeric value	A value that can be used in computations	None.	A numeric literal.

(2 of 2)

Usage

Each purpose-name keyword corresponds to a column name in the **sysams** system catalog table. The database server uses the following types of purpose attributes:

- Purpose functions

  A purpose-function attribute maps the name of a user-defined function to one of the prototype purpose functions that Figure 1-2 on page 1-15 describes.

- Purpose flags

  Each flag indicates whether an access method supports a particular SQL statement or keyword.

- Purpose values

  These string, character, or numeric values provide configuration information that a flag cannot supply.

You specify purpose options when you create an access method with the CREATE PRIMARY ACCESS_METHOD statement. To change the purpose options of an access method, use the ALTER ACCESS_METHOD statement.

To enable a purpose function

1. Register the access-method function that performs the appropriate tasks with a CREATE FUNCTION statement.
2. Set the purpose-function name equal to a registered UDR name.

   For example, Figure 6-2 on page 6-7 sets the **am_getnext** purpose-function name to the UDR name **textfile_getnext**. This example creates a new access method.

   The example in Figure 6-1 on page 6-5 adds a purpose function to an existing access method.

To enable a purpose flag, specify the purpose name without a corresponding value.

To clear a purpose-option setting in the **sysams** system catalog table, use the DROP clause of the ALTER ACCESS_METHOD statement.

Setting Purpose Functions, Flags, and Values

Figure 6-3 describes the possible settings for the **sysams** columns that contain purpose-function names, purpose flags, and purpose values. The items in Figure 6-3 appear in the same order as the corresponding **sysams** columns.

Figure 6-3
Purpose Functions, Purpose Flags, and Purpose Values

Purpose-Name Keyword	Explanation	Purpose category	Default Setting
am_sptype	A character that specifies what type of storage space the access method supports For a user-defined access method, **am_sptype** can have any of the following settings: ■ 'X' indicates that the access method accesses only extspaces ■ 'S' indicates that the access method accesses only sbspaces ■ 'A' indicates that the access method can provide data from extspaces and sbspaces You can specify **am_sptype** only for a new access method. You cannot change or add an **am_sptype** value with ALTER ACCESS_METHOD. Do not set **am_sptype** to 'D' or attempt to store a virtual table in a dbspace.	Value	'A'
am_cluster	A flag that you set if the access method supports clustering of tables	Flag	Not set
am_rowids	A flag that you set if the primary access method can retrieve a row from a specified address	Flag	Not set
am_readwrite	A flag that you set if the access method supports data changes The default setting for this flag, not set, indicates that the virtual data is read-only. Unless you set this flag, an attempt to write data can cause the following problems: ■ An INSERT, DELETE, UPDATE, or ALTER FRAGMENT statement causes an SQL error. ■ The database server does not execute **am_insert**, **am_delete**, or **am_update**.	Flag	Not set

(1 of 3)

Purpose-Name Keyword	Explanation	Purpose category	Default Setting
am_parallel	A flag that the database server sets to indicate which purpose functions can execute in parallel If set, the hexadecimal **am_parallel** flag contains one or more of the following bit settings: ■ The `1` bit is set for parallelizable scan. ■ The `2` bit is set for parallelizable delete. ■ The `4` bit is set for parallelizable update. ■ The `8` bit is set for parallelizable insert.	Flag	Not set
am_costfactor	A value by which the database server multiplies the cost that the **am_scancost** purpose function returns An **am_costfactor** value from `0.1` to `0.9` reduces the cost to a fraction of the value that **am_scancost** calculates. An **am_costfactor** value of `1.1` or greater increases the **am_scancost** value.	Value	1.0
am_create	The name of a user-defined function that adds a virtual table to the database	Function	None
am_drop	The name of a user-defined function that drops a virtual table	Function	None
am_open	The name of a user-defined function that makes a fragment, extspace, or sbspace available	Function	None
am_close	The name of a user-defined function that reverses the initialization that **am_open** performs	Function	None
am_insert	The name of a user-defined function that inserts a row	Function	None
am_delete	The name of a user-defined function that deletes a row	Function	None
am_update	The name of a user-defined function that changes the values in a row	Function	None
am_stats	The name of a user-defined function that builds statistics based on the distribution of values in storage spaces	Function	None

(2 of 3)

Purpose-Name Keyword	**Explanation**	**Purpose category**	**Default Setting**
am_scancost	The name of a user-defined function that calculates the cost of qualifying and retrieving data	Function	None
am_check	The name of a user-defined function that tests the physical structure of a table	Function	None
am_beginscan	The name of a user-defined function that sets up a scan	Function	None
am_endscan	The name of a user-defined function that reverses the setup that AM_BEGINSCAN initializes	Function	None
am_rescan	The name of a user-defined function that scans for the next item from a previous scan to complete a join or subquery	Function	None
am_getbyid	The name of a user-defined function that fetches data from a specific physical address	Function	None
am_getnext	The name of the required user-defined function that scans for the next item that satisfies the query	Function	None

(3 of 3)

The following rules apply to the purpose-option specifications in the CREATE PRIMARY ACCESS_METHOD and ALTER ACCESS_METHOD statements:

- To specify multiple purpose options in one statement, separate them with commas.
- The CREATE PRIMARY ACCESS_METHOD statement must specify a routine name for the **am_getnext** purpose function.

  The ALTER ACCESS_METHOD statement cannot drop **am_getnext** but can modify it.
- The ALTER ACCESS_METHOD statement cannot add, drop, or modify the **am_sptype** value.

References

In this manual, see the following topics:

- "Managing Storage Spaces" on page 3-12.
- "Executing in Parallel" on page 3-33.
- "Registering Purpose Functions" on page 2-10 and "Registering the Access Method" on page 2-11.
- "Calculating Statement-Specific Costs" on page 3-32.
- Chapter 4, "Purpose-Function Reference."

In the *Informix Guide to SQL: Syntax*, see the descriptions of:

- Database Object Name segment (for a routine name), Quoted String segment, and Literal Number segment.
- CREATE FUNCTION statement.

Index

A

B

C

D

E

F

G

I

J

L

M

S

T

U

W

X