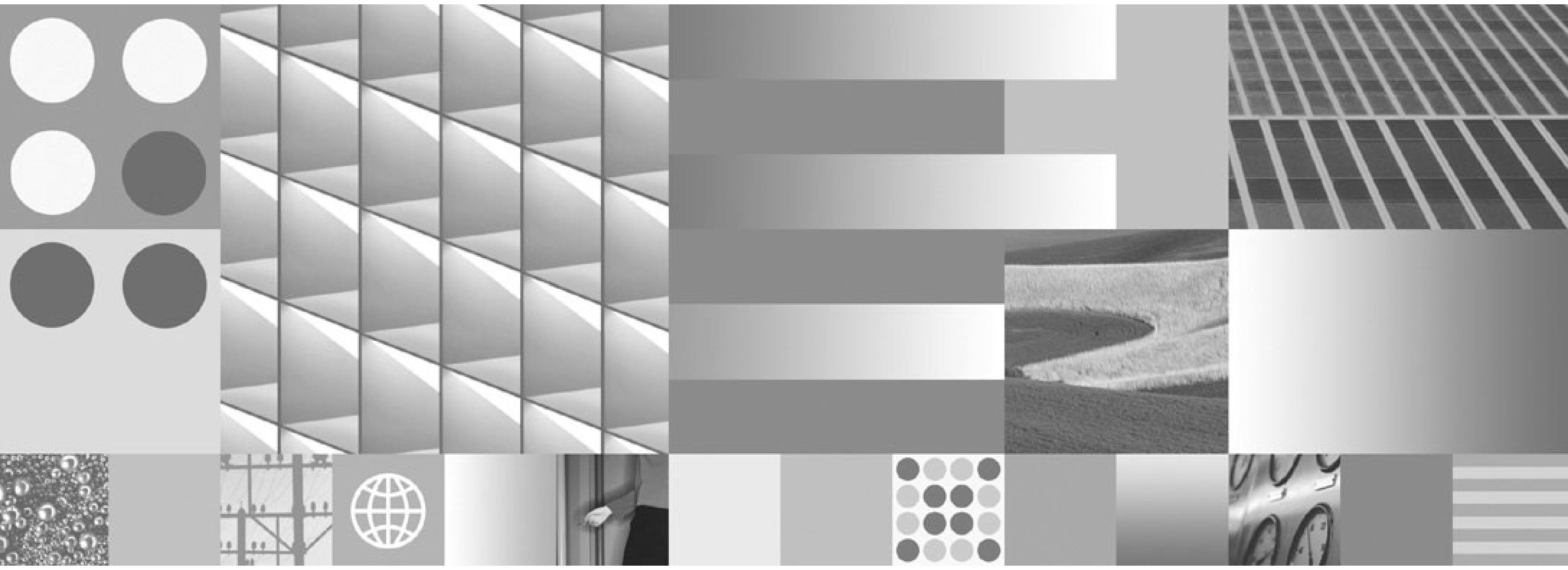
IBM Informix

Version 11.50

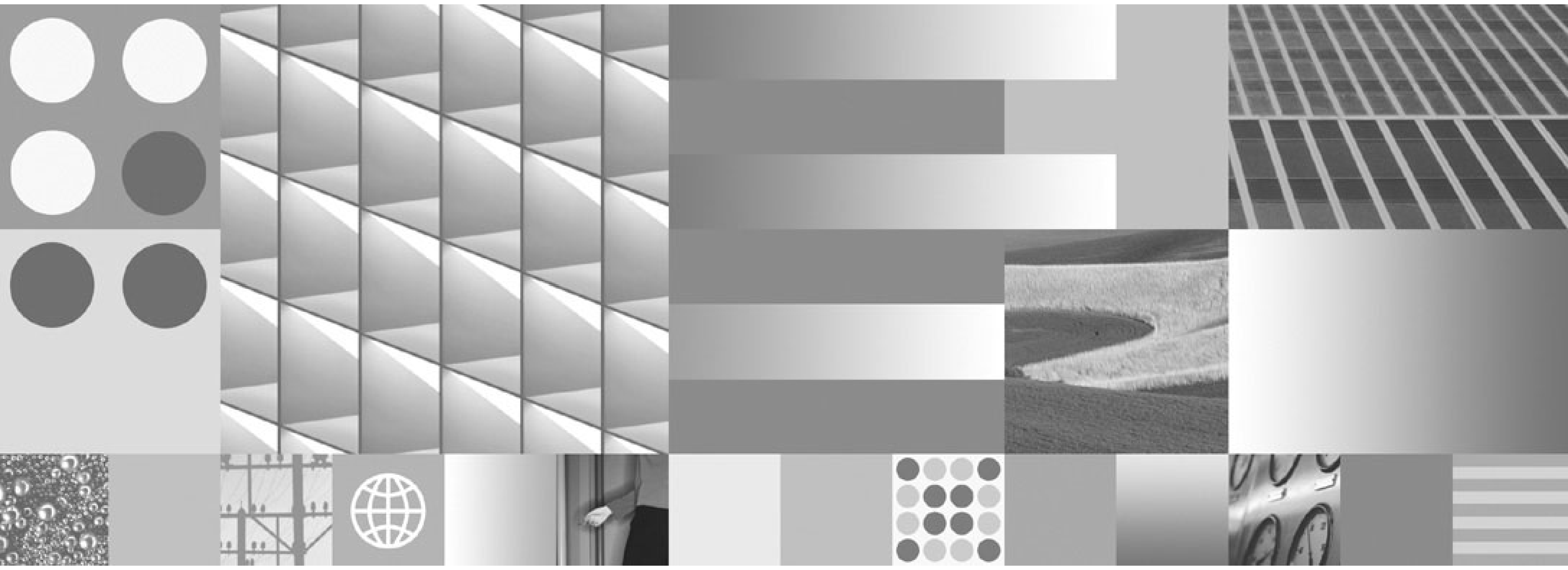


IBM Informix Virtual-Index Interface Programmer's Guide

SC23-9439-00

IBM Informix

Version 11.50



IBM Informix Virtual-Index Interface Programmer's Guide

SC23-9439-00

Note:
Before using this information and the product it supports, read the information in “Notices” on page B-1.

Contents

Introduction

In This Introduction

This introduction provides an overview of the information in this publication and describes the conventions it uses.

About This Publication

This publication explains how to create a secondary access method with the Virtual-Index Interface (VII) to extend the built-in indexing schemes of IBM Informix Dynamic Server, typically with a DataBlade® module.

Types of Users

This publication is written for experienced C programmers who develop secondary access methods, including:

- Partners and third-party programmers who have index requirements that the B-tree and R-tree indexes do not accommodate
- 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 publication assumes that you are using IBM Informix Dynamic Server, Version 11.50, as your database server.

Assumptions About Your Locale

IBM 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.

The examples in this publication are written with the assumption that you are using the default locale, **en_us.8859-1**. This locale supports U.S. English format conventions for date, time, and currency. In addition, this locale supports the ISO 8859-1 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 *IBM Informix GLS User's Guide*.

Demonstration Database

The DB–Access utility, which is provided with the IBM 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 IBM Informix manuals are based on the **stores_demo** database.
- The **sales_demo** database illustrates a dimensional schema for data-warehousing applications. For conceptual information about dimensional data modeling, see the *IBM Informix Database Design and Implementation Guide*.

For information about how to create and populate the demonstration databases, see the *IBM Informix DB–Access User's Guide*. For descriptions of the databases and their contents, see the *IBM 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.

Documentation Conventions

This section describes the following conventions, which are used in the product documentation for IBM® Informix Dynamic Server:

- Typographical conventions
- Feature, product, and platform conventions
- Syntax diagrams
- Command-line conventions
- Example code conventions

Typographical Conventions

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

Convention	Meaning
KEYWORD	Keywords of SQL, SPL, and some other programming languages appear in uppercase letters in a serif font.
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.

Convention	Meaning
boldface	Names of program entities (such as classes, events, and tables), environment variables, file names, path names, and interface elements (such as icons, menu items, and buttons) appear in boldface.
`monospace`	Information that the product displays and information that you enter appear in a monospace typeface.
KEYSTROKE	Keys that you are to press appear in uppercase letters in a sans serif font.
>	This symbol indicates a menu item. For example, "Choose **Tools > Options**" means choose the **Options** item from the **Tools** menu.

Feature, Product, and Platform Markup

Feature, product, and platform markup identifies paragraphs that contain feature-specific, product-specific, or platform-specific information. Some examples of this markup follow:

Dynamic Server

Identifies information that is specific to IBM Informix Dynamic Server

End of Dynamic Server

Windows Only

Identifies information that is specific to the Windows operating system

End of Windows Only

This markup can apply to one or more paragraphs within a section. When an entire section applies to a particular product or platform, this is noted as part of the heading text, for example:

Table Sorting (Windows)

Example Code Conventions

Examples of SQL code occur throughout this publication. Except as noted, the code is not specific to any single IBM 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
```

To use this SQL code for a specific product, you must apply the syntax rules for that product. For example, if you are using an SQL API, you must use EXEC SQL at the start of each statement and a semicolon (or other appropriate delimiter) at the end of the statement. If you are using DB–Access, you must delimit multiple statements with semicolons.

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.

For detailed directions on using SQL statements for a particular application development tool or SQL API, see the documentation for your product.

Additional Documentation

You can view, search, and print all of the product documentation from the IBM Informix Dynamic Server information center on the Web at http://publib.boulder.ibm.com/infocenter/idshelp/v115/index.jsp.

For additional documentation about IBM Informix Dynamic Server and related products, including release notes, machine notes, and documentation notes, go to the online product library page at http://www.ibm.com/software/data/informix/pubs/library/. Alternatively, you can access or install the product documentation from the Quick Start CD that is shipped with the product.

Compliance with Industry Standards

The American National Standards Institute (ANSI) and the International Organization of Standardization (ISO) have jointly established a set of industry standards for the Structured Query Language (SQL). IBM 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 IBM Informix database servers comply with the SQL-92 Intermediate and Full Level and X/Open SQL Common Applications Environment (CAE) standards.

Syntax Diagrams

This guide uses syntax diagrams built with the following components to describe the syntax for statements and all commands other than system-level commands.

Table 1. Syntax Diagram Components

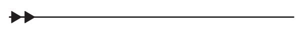
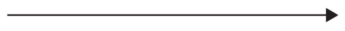
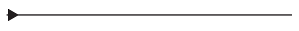
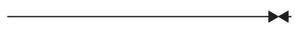
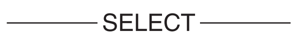
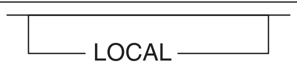
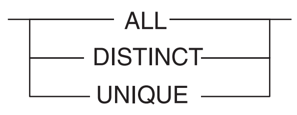
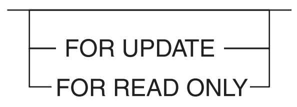
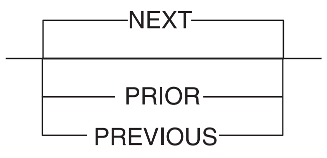
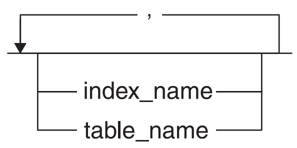
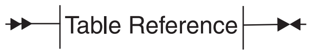
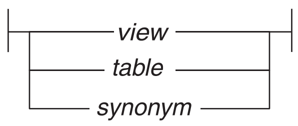
Component represented in PDF	Component represented in HTML	Meaning
	`>>----------------------`	Statement begins.
	`----------------------->`	Statement continues on next line.
	`>-----------------------`	Statement continues from previous line.
	`-----------------------><`	Statement ends.
	`--------SELECT----------`	Required item.
	`--+-----------------+---` `'------LOCAL------'`	Optional item.

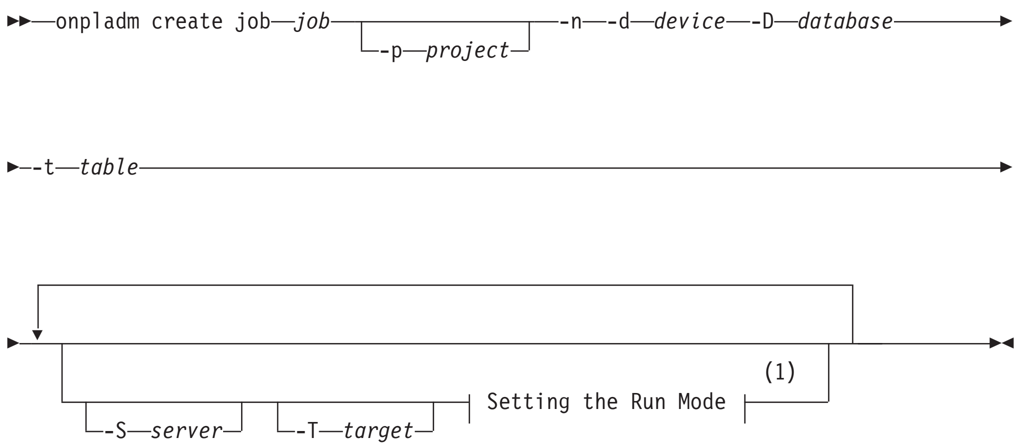
Table 1. Syntax Diagram Components (continued)

Component represented in PDF	Component represented in HTML	Meaning
	`---+-----ALL-------+---` `+--DISTINCT-----+` `'---UNIQUE------'`	Required item with choice. One and only one item must be present.
	`---+-----------------+---` `+--FOR UPDATE-----+` `'--FOR READ ONLY--'`	Optional items with choice are shown below the main line, one of which you might specify.
	`.---NEXT---------.` `----+----------------+---` `+---PRIOR--------+` `'---PREVIOUS-----'`	The values below the main line are optional, one of which you might specify. If you do not specify an item, the value above the line will be used as the default.
	`.-------,-----------.` `V                   |` `---+-----------------+---` `+---index_name---+` `'---table_name---'`	Optional items. Several items are allowed; a comma must precede each repetition.
	`>>-| Table Reference |-><`	Reference to a syntax segment.
Table Reference 	`Table Reference` `|--+-----view--------+--|` `+------table------+` `'----synonym------'`	Syntax segment.

How to Read a Command-Line Syntax Diagram

The following command-line syntax diagram uses some of the elements listed in the table in Syntax Diagrams.

Creating a No-Conversion Job

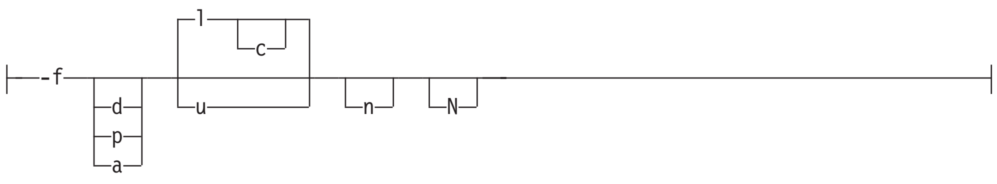


Notes:

1 See page Z-1

The second line in this diagram has a segment named "Setting the Run Mode," which according to the diagram footnote, is on page Z-1. If this was an actual cross-reference, you would find this segment in on the first page of Appendix Z. Instead, this segment is shown in the following segment diagram. Notice that the diagram uses segment start and end components.

Setting the Run Mode:



To see how to construct a command correctly, start at the top left of the main diagram. Follow the diagram to the right, including the elements that you want. The elements in this diagram are case sensitive because they illustrate utility syntax. Other types of syntax, such as SQL, are not case sensitive.

The Creating a No-Conversion Job diagram illustrates the following steps:

1. Type **onpladm create job** and then the name of the job.
2. Optionally, type **-p** and then the name of the project.
3. Type the following required elements:
   - **-n**
   - **-d** and the name of the device
   - **-D** and the name of the database
   - **-t** and the name of the table
4. Optionally, you can choose one or more of the following elements and repeat them an arbitrary number of times:
   - **-S** and the server name
   - **-T** and the target server name
   - The run mode. To set the run mode, follow the Setting the Run Mode segment diagram to type **-f**, optionally type **d**, **p**, or **a**, and then optionally type **l** or **u**.
5. Follow the diagram to the terminator.

Keywords and Punctuation

Keywords are words reserved for statements and all commands except system-level commands. When a keyword appears in a syntax diagram, it is shown in uppercase letters. When you use a keyword in a command, you can write it in uppercase or lowercase letters, but you must spell the keyword exactly as it appears in the syntax diagram.

You must also use any punctuation in your statements and commands exactly as shown in the syntax diagrams.

Identifiers and Names

Variables serve as placeholders for identifiers and names in the syntax diagrams and examples. You can replace a variable with an arbitrary name, identifier, or literal, depending on the context. Variables are also used to represent complex syntax elements that are expanded in additional syntax diagrams. When a variable appears in a syntax diagram, an example, or text, it is shown in *lowercase italic*.

The following syntax diagram uses variables to illustrate the general form of a simple SELECT statement.

```
►►──SELECT──column_name──FROM──table_name──────────────────────────────────►◄
```

When you write a SELECT statement of this form, you replace the variables *column_name* and *table_name* with the name of a specific column and table.

How to Provide Documentation Feedback

You are encouraged to send your comments about IBM Informix user documentation by using one of the following methods:

- Send e-mail to docinf@us.ibm.com.
- Go to the Information Center at http://publib.boulder.ibm.com/infocenter/idshelp/v115/index.jsp and open the topic that you want to comment on. Click **Feedback** at the bottom of the page, fill out the form, and submit your feedback.

Feedback from both methods is monitored by those who maintain the user documentation of Dynamic Server. The feedback methods are reserved for reporting errors and omissions in our documentation. For immediate help with a technical problem, contact IBM Technical Support. For instructions, see the IBM Informix Technical Support Web site at http://www.ibm.com/planetwide/.

We appreciate your suggestions.

Chapter 1. Access Methods

In This Chapter

This chapter explains the following subjects:
- The term *access method*
- Why you create user-defined access methods
- How you create user-defined access methods

Warning: This publication is specifically for customers and DataBlade partners developing alternative access methods for Dynamic Server. The interface described in this publication is being continually enhanced and modified. Customers and partners who use this interface should work with an technical support representative to ensure that they continue to receive the latest information and that they are prepared to change their access method.

Built-in Access Methods

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 of combining 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 of indexing 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 IBM 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 *IBM Informix Guide to SQL: Syntax*.

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

User-defined Access Methods

This publication explains how to create secondary access methods that provide SQL access to non-relational 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 or keys in an index. 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 publication refers to the index that the access method presents to the end user as a *virtual index.*

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 index, which resides in an *extspace*

  An extspace refers to a storage location that the IBM 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 3-9.

Seamless Use of SQL

With the aid of a user-defined secondary access method, an SQL statement can use one or more indexes.

Further, with the aid of a user-defined secondary access method, indexes can provide access to the following extended data:

- User-defined types
- Data inside a smart large object
- External data sources
- Nonrelational data

In addition, with the aid of a user-defined secondary access method, an index can contain any of the following key types:

- Return values from a user-defined function
- Approximate values such as stem words for a full-text search
- Attributes of data such as length
- Relative position to other data in a hierarchy or area of space

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

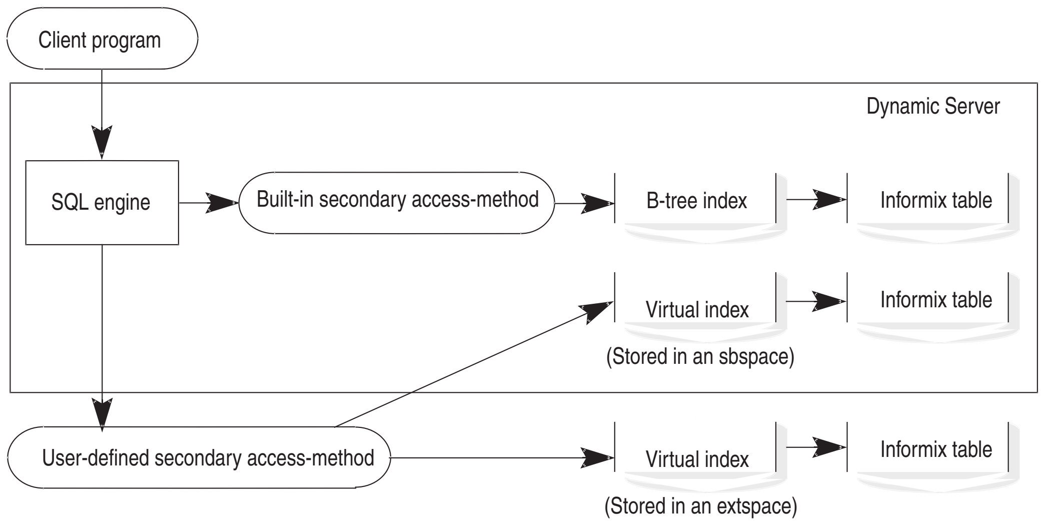


Figure 1-1. An Application Using a Secondary Access Method

Access-Method Components

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 IBM Informix application programming interface, the Virtual-Index Interface (VII).

Provided Components

The following application program interface support is provided for the development of user-defined access methods:

- Virtual-Index Interface
- DataBlade API
- Access-method specific SQL extensions
- Additional IBM Informix API libraries, as needed

Virtual-Index Interface

The Virtual-Index Interface (VII) 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 INDEX statement, it invokes an access-method function with the following required parameter and return value types:

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

To determine which UDR provides the entry point for index 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 index structure. When the purpose function exits, the access-method returns a prespecified value to indicate success or failure.

For information about the access-method developer's contribution to purpose functions, refer to "Components That You Provide" on page 1-7. For the syntax and usage of each purpose function, refer to Chapter 4, "Purpose-Function Reference," on page 4-1.

Descriptors: *Descriptors* are predefined opaque data types that the database server creates to exchange information with a Datablade module or an access method. The VII 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 VII descriptors, refer to the "Descriptors" on page 5-2.

Descriptor Name and Structure	**Database Server Entries in the Descriptor**
table descriptor MI_AM_TABLE_DESC	The database server puts CREATE INDEX specifications in the table descriptor, including the following items: • Identification by index name, owner, storage space, and current fragment • Structural details, such as the number of fragments in the whole index, column names, and data types • Optional user-supplied parameters • Constraints such as read/write mode and unique keys
scan descriptor MI_AM_SCAN_DESC	The database server puts SELECT statement specifications in the scan descriptor, including the following items: • Index-key columns • Lock type and isolation level • Pointers to the table descriptor and the qualification 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 3-20.

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 index 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 an index, the access method can maintain a pointer in PER_COMMAND memory to the address of the current index entry.
qualification descriptor MI_AM_QUAL_DESC	As it processes each qualification against a single index entry, 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 VII 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 index 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 5-8.

DataBlade API

The DataBlade application programming interface includes functions and opaque data structures that enable an application to implement C-language 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 publication contains information about the specific DataBlade API functions that an access method calls. For more information about the DataBlade API, refer to the *IBM Informix DataBlade API Programmer’s Guide*.

SQL Extensions

The IBM 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 SECONDARY 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," on page 6-1.

Specifying an Access Method for a Virtual Index: The user needs a way to specify a virtual index in an SQL statement.

To create a virtual index with the CREATE INDEX 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 index in an extspace or sbspace.

For more information about the SQL extensions specific to virtual indexes, refer to "Supporting Data Definition Statements" on page 3-8 and "Supporting Data Retrieval, Manipulation, and Return" on page 3-29.

API Libraries

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

Components That 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
- Operator-class functions
- User messages and documentation

Purpose Functions

A *purpose function* is a UDR that can interpret the user-defined structure of a virtual index. You implement purpose functions in C to build, connect, populate, query, and update indexes. 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 publication uses a column name from the **sysams** system catalog table as the generic purpose-function name. For example, this publication refers to the UDR that builds a new index 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**.

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

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

Table 1-1. Purpose Functions

Generic Name	Description	Invoking Statement or Command
am_create	Creates a new virtual index and registers it in the system catalog	CREATE INDEX ALTER FRAGMENT
am_drop	Drops an existing virtual index and removes it from the system catalog	DROP INDEX
am_open	Opens the file or smart large object that contains the virtual index Typically, **am_open** allocates memory to store handles and pointers.	CREATE INDEX DROP INDEX DROP DATABASE ALTER FRAGMENT DELETE, UPDATE, INSERT SELECT

am_close	Closes the file or smart large object that contains the virtual index and releases any remaining memory that the access method allocated	CREATE INDEX
		ALTER FRAGMENT
		DELETE, UPDATE, INSERT
		SELECT

am_insert	Inserts a new entry into a virtual index	CREATE INDEX
		ALTER FRAGMENT
		INSERT
		UPDATE *key*

am_delete	Deletes an existing entry from a virtual index	DELETE, ALTER FRAGMENT UPDATE *key*
am_update	Modifies an existing entry in a virtual index	UPDATE
am_stats	Builds statistics information about the virtual index	UPDATE STATISTICS

am_scancost	Calculates the cost of a scan for qualified data in a virtual index	SELECT
		INSERT, UPDATE, DELETE
		WHERE...

am_beginscan	Initializes pointers to a virtual index, and possibly parses the query statement, prior to a scan	SELECT
		INSERT, UPDATE, DELETE
		WHERE...

am_getnext	Scans for the next index entry 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...

am_endscan	Releases resources that am_beginscan allocates	SELECT
		INSERT, UPDATE, DELETE
		WHERE...

am_check	Performs a check on the physical integrity of a virtual index	**oncheck** utility

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

- Chapter 2, “Developing an Access Method,” on page 2-1, helps you decide which purpose functions to provide and explains how to register them in a database.
- Chapter 3, “Design Decisions,” on page 3-1, describes some of the functionality that you program and provides examples of program code.
- Chapter 4, “Purpose-Function Reference,” on page 4-1, 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 roll back transactions, and perform other tasks. You provide the additional UDRs and header files that complete the access method.

Operator Class

The functions that operate on index keys of a particular data type make up an *operator class*. The operator class has two types of functions:

- *Strategy* functions, which are operators that appear in SQL statements

  For example, the function `equal(`*column, constant*`)` or the operator expression `column = constant` appears in the WHERE clause of an SQL query.
- *Support* functions that the access method calls

  For example, the function `compare(`*column*`,` *constant*`)` might return a value that indicates whether each index key is less than, equal to, or greater than the specified constant.

The unique operator-class name provides a way to associate different kinds of operators with different secondary access methods.

You designate a default operator class for the access method. If a suitable operator class exists in the database server, you can assign it as the default. If not, you program and register your own strategy and support functions and then register an operator class.

For more information about operator classes, strategy functions, and support functions, refer to *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

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 IBM Informix implementation of SQL, configuration options, data types, error messages, backup procedures, and extended features that the IBM 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 3-32.

Access Method Flow

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

Locating 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 Only

```
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
```

End of UNIX Only

Windows Only

```
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
```

End of Windows Only

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 4-18**.

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 Table 1-2.

Table 1-2. Partial sysprocedures Entry

Column Name	Value
procname	`get_next_record`

Table 1-2. Partial sysprocedures Entry (continued)

Column Name	Value
owner	`informix`
procid	`163`
numargs	`3`
externalname	`$INFORMIXDIR/extend/am_lib.bld(get_next_record)`(on 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 SECONDARY 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 SECONDARY 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 SECONDARY ACCESS_METHOD, the **sysams** system catalog table has an entry similar to Table 1-3.

Table 1-3. 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	`S`
am_sptype	`A`
am_getnext	`163` (Matches the **procid** value in the **sysprocedures** system catalog table entry for **get_next_record()**)

Invoking Purpose Functions

When an SQL statement or **oncheck** command specifies a virtual index, 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 an index
- Scan and select data
- Insert, delete, or update data
- Drop an index
- 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 an index

- 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 4-1.

The example in Table 1-3 on page 1-11 specifies only the **am_getnext** purpose for the **super_access** access method. A SELECT statement on a virtual-index that uses **super_access** initiates the following database server actions:

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

   The CREATE FUNCTION statement assigns the executable file as follows:

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

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

Calling Functions From a Purpose Function

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 index 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 VII accessor functions to retrieve the index description and then calls DataBlade API functions to open that index
   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-index data
   d. Begins its scan
5. Each time that the access method retrieves a qualifying record, it stores the row and fragment identifiers in the row-id descriptor.
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 identified every qualifying row.
7. The access method calls a DataBlade API function to close the index 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.

Improving An Access Method

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 index because the example does not include an **am_create** purpose function or add, delete, or update index entries.

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.

Chapter 2. Developing an Access Method

In This Chapter

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

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 SECONDARY ACCESS_METHOD statement to register the user-defined access method in the **sysams** system catalog table.
5. If necessary, create support and strategy functions for an operator class and then execute the CREATE FUNCTION to register the functions in the **sysprocedures** system catalog table.
6. Execute the CREATE OPERATOR CLASS statement to register the operator class in the **sysopclasses** system catalog table.
7. Test the access method in an end-user environment.

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

Choosing Features

The VII 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
- Unique indexes
- Alternative indexes on the same columns
- Multiple-column index keys

Support for the following optional features can contribute to access-method performance:
- Clustered data
- Parallel-function execution
- More than one row returned per scan-function call
- More than one index entry inserted per insert-function call
- Key scan, which creates rows from index keys
- Complex qualifications

For more information about any of these optional features, refer to Chapter 3, "Design Decisions," on page 3-1.

Writing Purpose Functions

The VII specifies the parameters and return values for a limited set of UDRs, called *purpose functions*, that correspond to one or more SQL statements. For most SQL statements, the database server attempts to invoke a sequence of task-specific purpose functions to process the statement. 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 4-1.

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

Table 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 INDEX	`am_create(MI_AM_TABLE_DESC *)`
	`am_insert(MI_AM_TABLE_DESC *, MI_ROW *, MI_AM_ROWID_DESC *)`

DROP INDEX	`am_drop(MI_AM_TABLE_DESC *)`
INSERT	`am_insert(MI_AM_TABLE_DESC *, MI_ROW *, MI_AM_ROWID_DESC *)`
DELETE	`am_delete(MI_AM_TABLE_DESC *, MI_ROW *, MI_AM_ROWID_DESC *)`

Table 2-1. Statements and Their Purpose Functions (continued)

Invoking Statement or Command	Purpose-Function Prototype

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_AM_ROWID_DESC *)`
	`am_endscan(MI_AM_SCAN_DESC *)`

SELECT with join	`am_rescan(MI_AM_SCAN_DESC *)`
UPDATE	`am_update(MI_AM_TABLE_DESC *, MI_ROW *, MI_AM_ROWID_DESC *, MI_ROW *,MI_AM_ROWID_DESC *`
UPDATE STATISTICS	`am_stats(MI_AM_TABLE_DESC *,MI_AM_ISTATS_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. Avoid names such as **vii_open**, **vii_create**, **vii_***. 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 *IBM Informix 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 index and the current SQL statement or **oncheck** command. For an overview of descriptors, refer to "Descriptors" on page 1-4, and for detailed information, refer to "Descriptors" on page 5-2.

As you write the purpose functions, adhere to the syntax provided for each in "Purpose-Function Syntax" on page 4-7.

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 support the following tasks:

- Opening and initializing files or smart large objects, as well as closing them again at the end of processing
- Creating new indexes
- Inserting, updating, or deleting data
- Running the **oncheck** utility
- Optimizing queries

Warning: The database server issues an error if a user or application tries to execute an SQL statement and 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," on page 4-1.

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 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 and can write smart-large-object data to disk.

Creating and Dropping Database Objects

In response to a CREATE INDEX 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 index that CREATE INDEX specifies.

The **am_insert** purpose function also pertains to CREATE INDEX. The database server scans the table to read key values and then passes each key value to **am_insert**.

If you supply a function for **am_create**, consider the necessity of also providing a function to drop an index that the access method creates. The database server executes the function that you register for **am_drop** in response to a DROP TABLE, DROP INDEX, or DROP DATABASE statement. If you do not provide a function to drop a virtual index, 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 3-28 describes.
- Register purpose functions as parallelizable, as “Executing in Parallel” on page 3-27 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 index that it manages, the optimizer relies on two sources of information:

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

  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 strategy 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 index 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 index.
- **am_rescan**

  In this purpose function, reuse the information from **am_beginscan** and 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.

Inserting, Deleting, and Updating Data

The following optional purpose functions support the data-manipulation statements shown in the table.

Purpose Function	**Statement**
am_insert	INSERT
am_delete	DELETE
am_update	UPDATE

If you do support insert, delete, and update transactions for data in extspaces, you might need to write and call routines for transaction management from the purpose functions that create transactions. The database server has no mechanism to roll back external data if an error prevents the database server from committing a complete set of transactions to the corresponding virtual index. For more information, refer to “Determining Transaction Success or Failure” on page 3-31.

Warning: If you do not supply functions for **am_insert**, **am_update**, or **am_delete**, 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.

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 3-27.

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 *IBM Informix Guide to SQL: Syntax*. For information about privileges, refer to the GRANT statement in the *IBM 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 SECONDARY 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 key scans or clustering

The sample statement in Figure 2-1 assigns registered function names to some purpose functions. It specifies that the access method should use sbspaces, and it enables clustering.

```
CREATE SECONDARY 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_SPTYPE = S,
   AM_CLUSTER)
```

Figure 2-1. Statement That Assigns Registered Function Names to Some Purpose Functions

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

```
am_name        my_virtual
am_owner       informix
am_id          101
am_type        S
am_sptype      S
am_defopclass  0
am_keyscan     0
am_unique      0
am_cluster     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 2-2. Registering a Secondary Access Method

The statement in Figure 2-1 does not name a purpose function for **am_stats**, **am_scancost**, or **am_check**, or set the **am_keyscan** or **am_unique** flag, as the 0 values in Figure 2-2 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," on page 6-1.

Specifying an Operator Class

An *operator class* identifies the functions that a secondary access method needs to build, scan, and maintain the entries in an index.

You can associate an access method with multiple operator classes, particularly if the indexes that use the access method involve multiple data types. For example, the following indexes might require multiple operator classes:

```
CREATE TABLE sheet_music (col1 beat, col2 timbre, col3 chord)
CREATE INDEX tone ON music(timbre, chord) USING music_am
CREATE INDEX rhythm ON music(beat) USING music_am
```

Use a different function to compare values of data type **chord** from that which you use to compare values of data type **timbre**.

To supply an operator class for a secondary access method:

1. Write support and strategy functions for the operator class if no existing functions suit the data types that the access method indexes.
2. Register each new support and strategy function with the CREATE FUNCTION statement that includes the NONVARIANT modifier.
3. Assign the strategy and support functions to operator classes with the CREATE OPCLASS statement.
4. Assign an operator class as default to the secondary access method with the ALTER ACCESS_METHOD statement.

Writing or Choosing Strategy and Support Functions

In a query, the WHERE clause might specify a *strategy* function to qualify or *filter* rows. The following clauses represent the same strategy function, which compares the index key **cost** to a constant:

```
WHERE equal(cost, 100)
WHERE cost = 100
```

Support functions build and scan the index and can perform any of the following tasks for a secondary access method:

- Build an index
- Search for specific key values
- Add and delete index entries
- Reorganize the index to accommodate new entries

The access method can call the same support function to perform multiple tasks. For example, an access method might call a **between()** support function to retrieve keys for the WHERE clause to test and locate the entries immediately greater than and less than a new index entry for an INSERT command.

Tip: If possible, use the built-in B-tree operators or the operator class that a registered DataBlade module provides. Write new functions only if necessary to fit the data types that the secondary access method indexes.

Registering Strategy and Support Functions

Issue a separate CREATE FUNCTION statement for each operator-class function. Do not issue the CREATE FUNCTION statement for any built-in function or user-defined function that is already registered in the **sysprocedures** system catalog table.

Warning: Include the NOT VARIANT routine modifier for each operator-class function, or the optimizer might ignore the virtual index and scan the underlying table sequentially instead.

Making a Function Nonvariant

A nonvariant UDR exhibits the following characteristics:

- The function always returns the same result when invoked with the same arguments.
- In the **sysprocedures** system catalog table entry for the UDR, the **variant** column contains the value `f` (for false).

  The CREATE FUNCTION statement inserts a description of the strategy function in the **sysprocedures** system catalog table. By default, the **variant** column of the **sysprocedures** system catalog table contains a `t` (for true), even if that function invariably returns equivalent results. When you create a function with the NOT VARIANT routine modifier, the database server sets the sysprocedures variant indicator for that function too.

If you do write strategy or support functions, specify the NOT VARIANT routine modifier in the CREATE FUNCTION statement and ensure that the database server recognizes them as *nonvariant*.

Tip: Create the UDR as NOT VARIANT only if it really is not variant.

By contrast, a variant UDR exhibits the following characteristics:

- In the **sysprocedures** system catalog table entry for the UDR, the **variant** column contains the value `t` (for true).

  Because the CREATE FUNCTION statement for the function did not specify the NOT VARIANT routine modifier, the **variant** column contains the default value.
- Each execution of a *variant* function with the same arguments can return a different result.

Warning: Always specify the NOT VARIANT routine modifier in the CREATE function statement for an operator-class strategy function. If the **variant** column for a strategy function contains a `t`, the optimizer does not invoke the access method to scan the index keys. Instead, the database server performs a full table scan.

In the following example, the **FileToCLOB()** function returns variable results. Therefore, the optimizer examines every smart large object that the **reports** file references:

```
SELECT * FROM reports WHERE
   contains(abstract, ROW("IFX_CLOB",
   FileToCLOB("/data/clues/clue1.txt","server")
      ::lld_lob,NULL::LVARCHAR),
```

Granting Privileges

By default, the database server grants Execution privilege to the generic user **public** when you register a UDR. However, if the **NODEFAC** environment variable overrides default privileges in a database, you must explicitly grant Execution privilege to SQL users of that database. The following statement grants Execution privilege to all potential end users:

```
GRANT EXECUTE ON FUNCTION strategy_function TO PUBLIC
```

For more information, about Execution privileges, refer to the CREATE FUnCTION and GRANT statements in the *IBM Informix Guide to SQL: Syntax*. For more information about environment variables, refer to the *IBM Informix Guide to SQL: Reference*.

Registering the Operator Class

The following statement syntax associates operators with an access method and places an entry in the **sysopclasses** system catalog table for the operator class:

```
CREATE OPCLASS music_ops FOR music_am
STRATEGIES(higher(note, note), lower(note, note))
SUPPORT(compare_octave(note, note), ...)
```

You must specify one or more strategy functions in the CREATE OPCLASS statement, but you can omit the support function if the access method includes code to build and maintain indexes. The following example specifies `none` instead of a support-function name:

```
CREATE OPCLASS special_operators FOR virtual_am
STRATEGIES (LessThan, LessThanOrEqual,
   Equal, GreaterThanOrEqual, GreaterThan)
SUPPORT (none)
```

Warning: When an SQL statement requires the access method to build or scan an index, the database server passes the support function names in the relative order in which you name them in the CREATE OPCLASS statement. List support functions in the correct order for the access method to retrieve and execute support tasks. For more information, refer to "Using FastPath" on page 3-19 and the description of accessor functions **mi_key_opclass_nsupt()** and **mi_key_opclass_supt()** in Chapter 5, "Descriptor Function Reference," on page 5-1.

Adding a Default Operator Class to the Access Method

Every access method must have at least one operator class so that the query optimizer knows which strategy and support functions apply to the index.

You assign a default operator class so that the database server can locate the strategy and support functions for an index if the CREATE INDEX statement does not specify them. To add an operator-class name as the default for the access method, set the **am_defopclass** purpose value in the **sysams** system catalog table. The following example shows how to set the **am_defopclass** purpose value:

```
ALTER ACCESS_METHOD my_virtual
   ADD AM_DEFOPCLASS = 'special_operators'
```

For more information, see "ALTER ACCESS_METHOD (+)" on page 6-2. For more information about operator classes, as well as strategy and support functions, refer to *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

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:

To test the access method:

1. Create one or more storage spaces.
2. Use the access method to create indexes 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 server 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 index that uses the access method performs step 3.

Creating and Specifying Storage Spaces

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

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. It is recommended that you store indexes in smart-large objects because the database server protects transaction integrity in sbspaces with rollback and recovery.

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 index with the CREATE INDEX statement.

Creating an Sbspace: An sbspace must exist before you can create a virtual index in it. Before you can test the ability of the access method to create an index 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 Only

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

End of UNIX Only

Windows Only

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

End of Windows Only

Specifying the Logical Sbspace Name: The following example creates a virtual index in the previously created **vspace1**:

```
CREATE INDEX ix1 ON tab1(col1)
   IN vspace1
   USING your_access_method
```

If you do not intend to specify an sbspace explicitly in the CREATE INDEX statement, specify a default sbspace. To find out how to create a default dbspace, see "Creating a Default Sbspace" on page 3-10.

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

```
CREATE INDEX ix1 ON tab1(col1)
   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.

Important: The use of external storage for secondary access methods is discouraged because you must provide transaction integrity, rollback, and recovery for indexes that reside in external storage spaces. If the access method requires external-space support, follow the guidelines in this section.

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 index with the CREATE INDEX statement.
- Directly specify an existing physical external storage location as a quoted string in the CREATE INDEX 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 Only

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"
```

End of UNIX Only

Windows Only

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"
```

End of Windows Only

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

```
CREATE INDEX ix1 ON tab1(col1)
   IN disk_file
   USING your_access_method
```

Specifying the Physical Location: As an alternative to the extspace name, a CREATE INDEX statement can directly specify a quoted string that contains the external location.

```
CREATE INDEX ix1 ON tab1(col1)
   IN "location_specifier"
   USING your_access_method
```

Providing a Default Extspace: If you do not intend to specify an extspace explicitly in the CREATE INDEX 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 3-3 on page 3-11.

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 an index with two fragments. Each fragment corresponds to a separate extspace. The database server alternates between the fragments to store new data.

```
CREATE INDEX index_name ON table(keys)
   FRAGMENT BY ROUNDROBIN IN "location_specifier1",
"location_specifier2"
   USING access_method_name
```

To fragment an index in smart-large-object storage, create a separate sbspace for each fragment before you create the index. 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 INDEX progress on 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 INDEX 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 INDEX 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 one of the following conditions is true:

- The **am_sptype** purpose value is set to `D`.
- The IN clause with the CREATE INDEX 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 indexes, use the SQL syntax in "Supporting Fragmentation" on page 3-12. You can provide support in the access method for CREATE INDEX statement keywords that effect transaction processing. If a CREATE INDEX statement specifies the LOCK MODE clause, the access method must impose and manage locks during data retrieval and update. To determine the state of an index during transaction processing, the access method calls VII 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. It is recommended that you document the isolation levels that the access method supports, as "mi_scan_isolevel()" on page 5-50 describes. For information about setting isolation levels, refer to the *IBM Informix Guide to SQL: Syntax* and the *IBM 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 *IBM Informix Administrator's Guide*.

For information about SQL statements and keywords that your access method can support, refer to the *IBM Informix Guide to SQL: Syntax*. For information about the VII functions that determine which statements and keywords the user specifies, refer to Chapter 5, "Descriptor Function Reference," on page 5-1.

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 4-10**. For more information about how to specify options on the command line for **oncheck**, refer to the *IBM Informix 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
```

Warning: Do not drop an access method if database objects exist that rely on the specified access method. For example, if you create an index using

my_virtual_am, you need to drop the index so **my_virtual_am** can process the DROP INDEX statement before you can execute DROP ACCESS_METHOD.

For more information, refer to "DROP ACCESS_METHOD (+)" on page 6-6.

Cannot Rename Databases That Have Virtual Indexes

You cannot rename a database if the database has any tables that were created using the secondary access method (also known as virtual index interface) or the primary access method (also known as virtual table interface).

Chapter 3. Design Decisions

In This Chapter

This chapter begins with several topics that discuss how the access method uses DataBlade API functions. It continues with topics that discuss alternative ways to accomplish SQL tasks. The chapter ends with guidelines for helping end users and application developers use the access method in “Supplying Error Messages and a User Guide” on page 3-32.

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 decide:

- 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 of a UDR that allocates memory and stores a pointer, refer to “Persistent User Data” on page 3-3. 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 index 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 an index 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 reference the pointer to access the memory.

For detailed information about the following, refer to the *IBM Informix DataBlade API Programmer's Guide*:

- Functions that allocate public memory
- Duration keywords

For more information about semipublic functions and named memory, see the indexing information on the IBM Informix Developer Zone at http://www.ibm.com/software/data/developer/informix. Look for the following titles from the list of tech notes:

- Memory Allocation for C UDRs
- Semi-Public Functions for DataBlade Module Development

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
{
   mi_integer     rowid;
   mi_integer     fragid;
   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 of index keys*/
   mi_boolean    is_null[10]; /*Array of true and false indicators*/
   my_row_t           *current index entry;
   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

Table 3-1 shows accessor functions that the VII provides to store and retrieve user data.

Table 3-1. 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," on page 5-1.

Accessing Database and System Catalog Tables

Although the VII 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_execute_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 *IBM Informix DataBlade API Programmer's Guide*.

Warning: A parallelizable UDR must not call **mi_exec()**, **mi_prepare()**, **mi_execute_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 3-27.

No Label-Based Access Control on Tables with Virtual Indexes

You cannot have label-based access control on virtual tables or tables with virtual indexes.

Executing a UDR Across Databases of the Same Database Server Instance

The database server supports built-in opaque parameters in functional indexes and Virtual Index Interfaces across multiple databases of the same database instance. You can implicitly and explicitly execute a UDR (written in SPL, C, or Java™) across databases with built-in data types and user-defined distinct types whose base types are built-in data type parameters and return types. These built-in data types include BOOLEAN, LVARCHAR, BLOB, and CLOB data types. User-defined opaque data types and distinct types whose base types are opaque data types must be explicitly cast to built-in data types if you want multiple databases on the same server instance to access them. All user-defined data types and casts must be defined in all of the participating databases of the same database server instance.

You can execute SQL statements, such as SELECT, INSERT, DELETE, UPDATE, and EXECUTE (implicit and explicit) involving the following data types across databases on the same server instance:

- Built-in data types
- User-defined distinct types whose base types are built-in data types
- Explicitly cast opaque data types
- Explicitly cast distinct types with opaque data-type columns

For example, if you use the SELECT statement in a query involving a user-defined opaque data type, be sure that the user-defined opaque data type is defined in all databases that you are using in the query. Then use the SELECT statement as follows:

```
SELECT coludt::lvarchar FROM db2:tab2 WHERE colint > 100;
SELECT loccolint, extcoludt::lvarchar FROM loctab, db2:exttab
   WHERE loctab.loccolint = exttab.extcolint;

SELECT coldistint, coldistudt::lvarchar FROM db2:tab2
   WHERE coldistint > 100;
SELECT loccoldistint, extcoludt::lvarchar FROM loctab, db2:exttab
       WHERE loctab.loccoldistint = exttab.extcoldistint;
```

For more information about the SQL to use in statements for more than one database in the same database server instance, see the *IBM Informix Guide to SQL: Syntax*.

Explicit execution occurs when the EXECUTE FUNCTION or EXECUTE PROCEDURE statement executes the UDR. Implicit execution occurs when the UDR appears in the projection list or predicate of a query, when the UDR is called to convert a function argument from one data type to another, or when an operator function for a user-defined data type is executed. The execution context of the UDR is the database in which the UDR is defined, not the local database.

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 3-2 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 3-2. Registering a Callback Function

The example in Figure 3-2 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 whether a transaction commits or rolls back, refer to "Checking Isolation Levels" on page 3-30.

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

- 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 roll back 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 *IBM Informix DataBlade API Programmer's Guide*. For code samples, see the indexing information on the IBM Informix Developer Zone at http://www.ibm.com/software/data/developer/informix.

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 INDEX statement in this example specifies configuration parameters.

```
CREATE INDEX fuzzy ON text(keywords)
   USING search_text(searchmode='string', wildcard='yes');
```

The access method must notify a user if a statement specifies an invalid parameter. To determine the parameters that a CREATE INDEX 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:

```
key_word = mi_tab_amparam (tableDesc);
if (strcmp (key_word, "searchmode") == 0)
{
          ...
}
else if (strcmp (key_word, "wildcard") == 0)
{
          ...
}
```

```
else
{
                    mi_db_error_raise (connection, MI_EXCEPTION,
                        "Invalid keyword in the USING clause.");
                    /* NOT REACHED */
}
```

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 3-30
- "Notifying the User About Access-Method Constraints" on page 3-34
- "Accessor Functions" on page 5-8

Supporting Data Definition Statements

The *data definition* statement CREATE INDEX names the index 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 INDEX statement. This section discusses design considerations for CREATE INDEX, ALTER INDEX, 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 INDEX, ALTER INDEX, and ALTER FRAGMENT statements specify for the virtual index. A table descriptor describes a single index 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 index.

For a complete description, refer to "Table Descriptor" on page 5-7.

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 an index 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
- Creating a Default Storage Space
- Ensuring Data Integrity
- Checking Storage-Space Type
- Supporting Fragmentation

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 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 *IBM Informix DataBlade API Programmer's Guide*.

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 SECONDARY ACCESS_METHOD or ALTER ACCESS_METHOD statement, as Chapter 6, "SQL Statements for Access Methods," on page 6-1 describes.

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

For an example of a demonstration secondary access method that provides for both extspaces and sbspaces, see the indexing information on the IBM Informix Developer Zone at http://www.ibm.com/software/data/developer/informix.

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 INDEX 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 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.

To create a default sbspace named vspace:

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 *IBM Informix Administrator's Reference*.

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

- Raise an error.
- Specify an external storage space.

  The example in Figure 3-3 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 3-3. 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 IBM 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 does the following:

- 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, it is recommended that the access method requires transaction logging in sbspaces. It is also recommended that the access method raises an error if an end user attempts to create a virtual index in an unlogged sbspace.

In the access-method user guide, provide the appropriate information to describe transaction logging using the access method. If the access method does not turn on transaction logging, the user guide should explain how to turn on logging for a virtual index in an sbspace.

To enable logging, the access method sets the MI_LO_ATTR_LOG create-time constant with the DataBlade API **mi_lo_create()** or **mi_lo_alter()** function. The following example attempts to set the constant that turns on logging and verifies that the setting succeeded:

```
mi_integer status;
status = mi_lo_specset_flags (lo_spec_p, MI_LO_ATTR_LOG);
if (status == MI_ERROR)
{
              mi_db_error_raise (NULL,MI_EXCEPTION,
                         "Unable to activate transaction
logging.");
              /* NOT REACHED */
              return MI_ERROR;
}
```

Tip: To save log space, temporarily turn off transaction logging at the start of the am_create purpose function. After the access method builds the new index, turn logging on. The following statement explicitly turns off transaction logging:

```
mi_lo_specset_flags(lo_spec_p, MI_LO_ATTR_NO_LOG)
```

For more information about metadata logging and transaction logging, refer to the *IBM Informix 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 3-30)

Checking Storage-Space Type

The database server issues an error if the CREATE INDEX statement specifies an inappropriate storage type. To determine the storage space (if any) that the CREATE INDEX statement specifies, the access method calls the **mi_tab_spacetype()** function. For details, refer to the description of **"mi_tab_spacetype()" on page 5-85**.

For more information about errors that occur from inappropriate storage-space type, refer to "Avoiding Storage-Space Errors" on page 2-13. For more information about documenting potential errors and intercepting error events, refer to "Supplying Error Messages and a User Guide" on page 3-32.

Supporting Fragmentation

A fragmented index 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 2-13. For a detailed discussion about the benefits of and approaches to fragmentation, refer to the *IBM Informix Database Design and Implementation Guide*.

When the secondary access method indexes a fragmented table, a single index might point to multiple table fragments. To obtain or set the fragment identifier for a row in an indexed table, the access method uses functions such as "Row-ID Descriptor" on page 5-5 describes.

When the index is fragmented, each call to the access method involves a single fragment rather than the whole index. An SQL statement such as CREATE INDEX 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 index, 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 index. Because some index entries might move to a different fragment, an ALTER FRAGMENT statement requires a scan, delete, and insert for each moved index entry. For information about how the database server uses the access method to redefine fragments, refer to "ALTER FRAGMENT Statement Interface" on page 4-2.

Tip: For an ALTER FRAGMENT statement, the database server creates a scan descriptor, but not a qualification descriptor. The **mi_scan_quals()** function returns a NULL-valued pointer to indicate that the secondary access method must return key values as well as the row identifier information for each index entry. For more information, refer to the description of **"mi_scan_quals()" on page 5-55**.

For information about the FRAGMENT BY clause, refer to the *IBM 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 INDEX 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 index entries to store in a shared memory buffer. The CREATE INDEX statement specifies the configuration keyword and value between parentheses.

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

In the preceding statement, the **mi_tab_amparam()** function returns `setbuffer=10`. Figure 3-4 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 3-4. 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 3-4 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 INDEX ...
USING access_method_name (keyword='string', keyword='string' ...)
```

Building New Indexes Efficiently

By default, the database server places one entry in shared memory per call to the **am_insert()** purpose function for a CREATE INDEX statement. The purpose function inserts the single entry and then returns control to the database server, which executes **am_insert** again until no more entries remain to insert.

Figure 3-5 shows how the **am_insert** purpose function writes multiple new index entries.

```
mi_integer my_am_open(MI_AM_TABLE_DESC *td)
{
...
   mi_tab_setniorows(td, 512);
}

mi_integer my_am_insert(MI_AM_TABLE_DESC *td, MI_ROW *newrow,
                  MI_AM_ROWID_DESC *rid)
{
   mi_integernrows;
   mi_integerrowid;
   mi_integerfragid;

   nrows = mi_tab_niorows(td);
   if (nrows > 0)
   {
      for (row = 0; row < nrows; ++row)
      {
          mi_tab_nextrow(td, &newrow, &rowid, &fragid)
          /*Write new entry. (Not shown.)*/
      } /* End get new entries from shared memory */
   }
   else
   {/* Shared memory contains only one entry per call to am_insert.*/
      rowid = mi_id_rowid(rid);
      fragid = mi_id_fragid(rid);
       /*Write new entry. (Not shown.)*/
   }/* End write one index entry. */
   /* Return either MI_OK or MI_ERROR, as required.
   ** (This example does not show error or exception-processing.) */
}
```

Figure 3-5. Processing Multiple Index Entries

In Figure 3-5, the access method performs the following steps:

1. The **am_open** purpose function calls **mi_tab_setniorows()** to specify the number of index entries that the database server can store in shared memory for **am_insert**.
2. At the start of **am_insert**, the purpose function calls **mi_tab_niorows()** to find out how many rows to retrieve from shared memory.

   The number of rows that shared memory actually contains might not equal the number of rows that **mi_tab_setniorows()** set.
3. The server loops through **mi_tab_setnextrow()** in **am_insert** to retrieve each new entry from shared memory.

For more information about **mi_tab_setniorows()**, **mi_tab_niorows()**, and **mi_tab_nextrow()**, refer to Chapter 5, “Descriptor Function Reference,” on page 5-1.

Enabling Alternative Indexes

A CREATE INDEX statement specifies one or more column names, or *keys*, from the table that the index references. A user-defined secondary access method can support alternative concurrent indexes that reference identical keys.

Typically, a user wants alternative indexes to provide a variety of search algorithms. The access method can test for predefined parameter values to determine how the user wants the index searched.

Consider the following example that enables two methods of search through a document for a character string:

- Look for whole words only.
- Use wildcard characters, such as *, to match any character.

The user specifies parameter keywords and values to distinguish between whole word and wildcard indexes on the same **keywords** column. This example uses a registered secondary access method named **search_text**.

```
CREATE TABLE text(keywords lvarchar, .....)
CREATE INDEX word ON text(keywords)
   USING search_text(searchmode='wholeword',wildcard='no');
CREATE INDEX pattern ON text(keywords)
   USING search_text(searchmode='string', wildcard='yes');
```

The access method allows both **word** and **pattern** indexes because they specify different parameter values. However, the access method issues an error for the following duplicate index:

```
CREATE INDEX fuzzy ON text(keywords)
   USING search_text(searchmode='string', wildcard='yes');
```

To determine if a user attempts to create a duplicate index, the **search_text** access method calls the following functions:

- The **mi_tab_amparam()** function returns the string `searchmode=string, wildcard=yes` from the CREATE INDEX statement.
- The **mi_tab_nparam_exist()** function indicates the number of indexes that already exist on column **keywords** (in this case, two).
- The **mi_tab_param_exist()** function returns the `searchmode=` and `wildcard=` values for each index on column **keywords**.

On the second call, **mi_tab_param_exist()** returns a string that matches the return string value from **mi_tab_amparam()**, so the access method alerts the user that it cannot create index **fuzzy**.

Figure 3-6 shows how the **am_create** purpose function tests for duplicate indexes.

```
MI_AM_TABLE_DESC *td;
mi_string *index_param, *other_param;
mi_integer i;

/* 1- Get user-defined parameters for the proposed index */
index_param = mi_tab_amparam(td);

/* 2- Get user-defined parameters for any other indexes
** that already exist on the same column(s).*/
for (i = 0; i < mi_tab_nparam_exist(td); i++)
   {
   other_param = mi_tab_param_exist(td,i);

   /* No configuration keywords distinguish the newindex
   ** from the existing index.
   ** Reject the request to create a new, duplicate index. */
   if ((index_param == NULL || index_param[0] == '\0')
      && (other_param == NULL || other_param[0] == '\0'))
      mi_db_error_raise(NULL, MI_EXCEPTION,
      "Duplicate index.");

   /* The user specifies identical keywords and values for a
   ** new index as those that apply to an existing index
   ** Reject the request to create a new, duplicate index.*/

   if (strcmp(index_param, other_param) == 0)
      mi_db_error_raise(NULL, MI_EXCEPTION,
      "Duplicate index.");
   }

/* The new index has unique keyword values.
** Extract them and create the new index. (Not shown) */
```

Figure 3-6. Avoiding Duplicate Indexes

For more information about **mi_tab_nparam_exist()**, **mi_tab_param_exist()**, and **mi_tab_amparam()**, refer to Chapter 5, "Descriptor Function Reference," on page 5-1.

Supporting Multiple-Column Index Keys

The key descriptor contains information about an index key. If the index contains more than one key column, the access method might provide for following operator-class considerations:

- The index might require multiple operator classes.

  Each key column corresponds to an operator class.
- The operator class for a particular key column determines the number and names of support functions for that single key column.
- The operator class determines the number and name of strategy functions for the single key column.

The key descriptor contains operator-class information on a per-column basis.

To access support functions for a multiple-column key:

1. Call the **mi_key_nkeys()** accessor function to determine the number of columns in the key.
2. Call the **mi_key_opclass_nsupt()** function to determine the number of support functions for a single key column.

If the access method needs every column in the key, use the return value from **mi_key_nkeys()** as the number of times to execute **mi_key_opclass_nsupt()**. For example, the **am_create** purpose function, which builds the index, might need support functions for every column.

3. Call the **mi_key_opclass_supt()** accessor function to extract one support function name.

   Use the return value from **mi_key_opclass_nsupt()** as the number of times to execute **mi_key_opclass_supt()**.

The sample syntax retrieves all the support functions.

```
MI_KEY_DESC * keyDesc;
mi_integer   keyNum;
mi_integer   sfunctNum;
mi_string    sfunctName;

keynum = mi_key_nkeys(keyDesc);

for (k=0; k<= keyNum; k++)
{
   sfunctNum = mi_key_opclass_nsupt(keyDesc, keyNum);

      for (i=0; i<=sfunctNum; i++)
      {
         sfunctName =
         mi_key_opclass_supt(keyDesc,
                  keyNum, sfunctNum);
         /*
         ** Use the function name
         ** or store it in user data. (Not shown.)
         */
      } /* End get sfunctName */
   } /* End get sfunctNum */
} /* End get keynum */
```

Figure 3-7. Extracting Support Functions for a Multiple-Column Index Key

The access method might need information about all the strategy functions for a particular key. For example, the access method might use the key descriptor rather than the qualification descriptor to identify strategy functions.

To access strategy functions for a multiple-column key:

1. Call the **mi_key_nkeys()** accessor function to determine the number of columns in the key.
2. Call the **mi_key_opclass_nstrat()** function to determine the number of support functions for a single key column.

   If the access method needs every column in the key, use the return value from **mi_key_nkeys()** as the number of times to execute **mi_key_opclass_nstrat()**.
3. Call the **mi_key_opclass_strat()** accessor function to extract one support function name.

   Use the return value from **mi_key_opclass_nstrat()** as the number of times to execute **mi_key_opclass_strat()**.

To retrieve all the strategy functions, substitute **mi_key_opclass_nstrat()** for **mi_key_opclass_nsupt()** and **mi_key_opclass_strat()** for **mi_key_opclass_supt()** in Figure 3-7 on page 3-18.

Using FastPath

The access method can use 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. Obtains a routine identifier for the desired UDR.

   To find out how to obtain the routine identifier, refer to the section, "Obtaining the Routine Identifier" following.
2. Passes the routine identifier to the **DataBlade API mi_func_desc_by_typeid()** function, which returns the function descriptor.
3. Passes the function descriptor to the DataBlade API **mi_routine_exec()** function, which executes the function in a virtual processor.

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

Warning: A database server exception results if a parallelizable function attempts to execute a routine that is not parallelizable. Use **mi_func_desc_by_typeid()** and **mi_routine_exec()** from a parallelizable access method only if you can guarantee that these functions look up or execute a parallelizable routine.

Obtaining the Routine Identifier

You can obtain the routine identifier for a strategy function directly from the qualification descriptor that the database server passes to the access method. Call **mi_qual_funcid()**. Because the database server does not provide the routine identifier for a support function directly in a descriptor, use the following procedure to identify the support function for FastPath execution.

To obtain the routine identifier for a support function:

1. Use **mi_tab_keydesc()** to extract the key descriptor from the table descriptor.
2. Use **mi_key_opclass_nsupt()** to determine the number of support functions that the access method must look up.
3. Use **mi_key_opclass_supt()** to determine each support-function name and then assemble a function prototype with a statement similar to the following example:

   ```
   sprintf(prototype, "%s(%s,%s)",
         function_name, key_data_type, key_data_type);
   ```
4. Use DataBlade API FastPath function **mi_routine_get()** to look up the function descriptor.

For an example of a secondary access method that includes dynamic support-function execution, see the indexing information on the IBM Informix Developer Zone at http://www.ibm.com/software/data/developer/informix.

Reusing the Function Descriptor

The access method can store the function descriptor in user-data memory for use in multiple executions of the same UDR. For example, the access method stores the function descriptor so that it can repeat a WHERE-clause function on each index entry.

Important: The database server assigns a PER_COMMAND duration to the function descriptor. The access method cannot change the duration of the original function descriptor, but can store a copy of it as part of the PER_STATEMENT user data to which the table descriptor points. Any access-method purpose function can obtain the function descriptor because they all have access to the table descriptor.

If the access method uses FastPath to execute support functions, the **am_open** purpose function can store the function descriptor in PER_STATEMENT memory. For example, a CREATE INDEX statement causes the database server to call the **am_insert** purpose function iteratively. To execute the support function or functions that build an index, each iteration of **am_insert** can retrieve the support-function descriptor from the table descriptor.

For information about user data, refer to “Storing Data in Shared Memory” on page 3-2.

Processing Queries

This section describes various options for processing a SELECT statement, or *query*, that involves a virtual index. 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.

Query processing involves the following actions:
- Interpreting the scan and qualification descriptors
- Scanning the index to select index entries
- Optionally 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* in response to a SELECT statement. The scan descriptor provides information about the key data types, as well as the locks and isolation levels that apply to the data that the query specifies.

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 index keys.

For more information about the information that scan descriptor provides, refer to “Scan Descriptor” on page 5-6 and the scan-descriptor accessor functions that begin on page 5-49.

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 key 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 VII 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()	Points 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
mi_qual_funcid() or **mi_qual_funcname()**	Identifies 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

For a complete list of access functions for the qualification descriptor, refer to "Qualification Descriptor" on page 5-4.

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 the 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.

Table 3-2. Generic Function Prototypes

Generic Prototype	Description
function(*column_name*)	Evaluates the contents of the named column
function(*column_name*, *constant*) *function*(*constant*, *column_name*)	Evaluates 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* ?)	Evaluates 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* #)	Evaluates the value in the specified column of the current row and a value, called a *statement-local variable* (SLV), that the UDR supplies

Table 3-2. Generic Function Prototypes (continued)

Generic Prototype	Description
function(*column*, *constant*, *slv* #) *function*(*constant*, *column*, *slv* #)	Evaluates 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 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 VII 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 descriptions of the **"mi_qual_needoutput()" on page 5-42** function and the **"mi_qual_setoutput()" on page 5-46** function.

For more information about output parameters, the OUT keyword, and SLVs, refer to *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

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 the 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 entry in the index, 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 index. 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," on page 5-1.

For more information about the following topics, refer to the manual indicated in the table.

Topic	Manual
Setting values for host variables in client applications	*IBM Informix ESQL/C Programmer's Manual*
Using DataBlade API functions from client applications	*IBM Informix DataBlade API Programmer's Guide*
Using host variables in SQL statements	*IBM 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 3-8 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 3-8. Complex WHERE Clause

Figure 3-9 and Figure 3-10 represent the structure of the qualification descriptor that corresponds to the WHERE clause in Figure 3-8.

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

Figure 3-9. Function Nesting

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

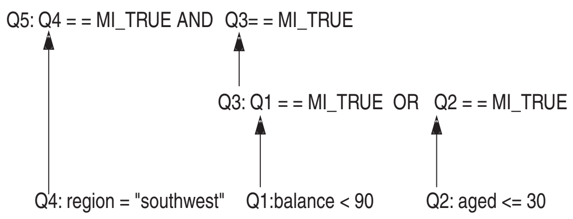

Q5: Q4 = = MI_TRUE AND Q3= = MI_TRUE
Q3: Q1 = = MI_TRUE OR Q2 = = MI_TRUE
Q4: region = "southwest"
Q1:balance < 90
Q2: aged <= 30

Figure 3-10. Qualification- Descriptor Hierarchy for a Three-Key Index

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 5-4.

Qualifying Data

To qualify table rows, a secondary access method applies the functions and Boolean operators from the qualification descriptor to key columns. The access method actually retrieves the contents of the keys from an index rather than from the table. If the index keys qualify, the secondary access method returns identifiers that enable the database server to locate the whole row that includes those key values.

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 routine identifier to execute a UDR with the DataBlade API FastPath facility. A qualification specifies a strategy UDR to evaluate index keys. To complete the qualification, the access method might also execute support UDRs. For information about FastPath and how to use it to execute strategy and support UDRs, refer to “Using FastPath” on page 3-19.

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, see the indexing information on the IBM Informix Developer Zone at http://www.ibm.com/software/data/developer/informix.

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 the 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. */ }
```

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.

Processing Complex Qualifications: In Figure 3-11 on page 3-25, the **am_getnext** purpose function attempts to disqualify index keys. It sets the row identifier and fragment identifier in the row-ID descriptor and signals the database server to retrieve the row information.

```
mi_integer sample_getnext(sd,retrow,retrowid)
   MI_AM_SCAN_DESC   *sd;
   MI_ROW                  **retrow
   MI_AM_ROWID_DESC  *retrowid;    /* Store rowid. */
{
   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 key column data types. */
   my_data = (my_data_t *)mi_tab_userdata(td); /* Get pointer to user
data.*/
   /* Evaluate keys until one qualifies for return to caller.. */
   for (;;)
   {
      if ( ! my_data ) return MI_NO_MORE_RESULTS;
      if ( eval_qual(sd, qd, my_data))== MI_TRUE)
      {
         mi_id_setrowid(retrowid, current->rowid);
         mi_id_setfragid(retrowid, current->fragid);
         return MI_ROWS;
      }

      my_data->rowptr++;
   } /*End loop.*/
}/* End getnext.*/
```

Figure 3-11. Sample am_getnext Purpose Function

For more examples, see the indexing information on the IBM Informix Developer Zone at http://www.ibm.com/software/data/developer/informix.

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 of 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 *IBM Informix 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 4-24.

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. For an example, refer to Figure 4-15 on page 4-25.

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 indexes; the access method updates data-distribution statistics for virtual indexes. When a user issues an UPDATE STATISTICS statement that requires the access method to determine the distribution of data in an index, 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_istats_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," on page 5-1.

The database server does not use the information in the statistics descriptor to evaluate query costs. The access method can, however, use these statistics during the **am_scancost** purpose function to compute the cost for a given query. For information about how to access the system catalog tables or to maintain tables in an IBM Informix database, refer to "Accessing Database and System Catalog Tables" on page 3-4.

Enhancing Performance

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

- Executing parallel scans, inserts, deletes, and updates
- Bypassing table scans
- 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 *IBM Informix DataBlade API Programmer's Guide*:

- 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 *IBM Informix Guide to SQL: Syntax*. For more information about parallelizable UDRs, refer to *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

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_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. All the strategy and support functions for the operator class that the index uses must also be 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 6-7.

Bypassing Table Scans

The secondary access method always returns row identifiers so that the database server can locate table rows. The access method can additionally format and return rows from the key columns that the scan descriptor specifies.

Set the **am_keyscan** purpose flag (with the CREATE SECONDARY ACCESS_METHOD or ALTER ACCESS_METHOD statement) to alert the database server that the **am_getnext** purpose function returns key values. When **am_keyscan** is set, the database server knows that **am_getnext** creates a row in shared memory from the key values in a qualified index entry. If the query selects only the columns in the key, the database server returns rows of index keys to the query. It does not retrieve the physical table row or extract the selected columns from the row.

Important: The access method cannot determine whether an individual query projects key columns. Before you decide to set the **am_keyscan** purpose flag, determine whether key columns satisfy queries with sufficient frequency for the access method to format rows, which requires a function call to the database server.

Warning: Do not set **am_keyscan** or format rows if users of the access method might index user-defined types (UDTs).

For more information about **am_keyscan**, refer to "Purpose Options" on page 6-7.

Buffering Multiple Results

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

To set up and fill a multiple-index entry buffer in shared memory:

1. Call **mi_tab_setniorows()** in **am_open** or **am_beginscan** to set the number of index entries 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 index entries to return.
3. Loop through **mi_tab_setnextrow()** in **am_getnext** until the number of qualifying index entries 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," on page 5-1.

```
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_AM_ROWID_DESC *ridDesc)
{
   mi_integer     nrows, row, nextrowid, nextfragid;
   MI_ROW        *nextrow=NULL; /* 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, &fragid);
         mi_tab_setnextrow(td, nextrow, nextrowid, nextfragid);
      } /* 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, &nextfragid);

      mi_id_setrowid(ridDesc, nextrowid);
      mi_id_setfragid(ridDesc, nextfragid);
   }
   /* When reach the end of data, return MI_NO_MORE_RESULTS, else return
MI_ROWS. */
}
```

Figure 3-12. Storing Multiple Results In a Buffer

Typically, a secondary access method does not create rows from key data. However, if you intend to set the **am_keyscan** purpose flag for a secondary access method, the access method must create an MI_ROW structure that contains key values in the appropriate order and of the appropriate data type to match the query specifications for a projected row.

Warning: Although a user can index UDTs, the database server issues an exception if the secondary access method creates and returns a row from index keys that contain UDTs.

For information about **am_keyscan**, refer to "Bypassing Table Scans" on page 3-28.

Supporting Data Retrieval, Manipulation, and Return

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

- Enforcing unique-index constraints
- Checking isolation levels
- Converting data to and from IBM Informix row format
- Detecting transaction success or failure

Enforcing Unique-Index Constraints

The UNIQUE or DISTINCT keyword in a CREATE INDEX or insert statement specifies that a secondary access method cannot insert multiple occurrences of a key value. The UNIQUE or DISTINCT keyword in a SELECT statement specifies that the access method must return only one occurrence of a key value.

To provide support for unique keys:

1. Program the **am_insert** purpose function to scan an index before it inserts each new entry and raise an exception for a key value that the index already contains.
2. Program the **am_getnext** to return only one occurrence of a key.
3. Set the **am_unique** purpose flag, as described in "Setting Purpose Functions, Flags, and Values" on page 6-8.

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)                    Roll Back
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/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   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, Phantom Read
Repeatable read or Cursor Stability	Dirty Read, Nonrepeatable Read
Read Committed	Dirty Read
Read Uncommitted	None

A virtual-index interface cannot use the COMMITTED READ LAST COMMITTED isolation level feature.

For more information about how applications use isolation levels, consult the *IBM Informix Guide to SQL: Reference*, *IBM Informix Guide to SQL: Syntax*, and *IBM 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," on page 5-1.

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

- The virtual index and all the table data that it accesses reside 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 3-11. To find out how to provide for logging with ONCONFIG parameters, refer to your *IBM Informix Administrator's Guide*.

The access method must provide the code to enforce isolation levels if users require Serializable isolation. The database server does not provide support for full Serializable isolation.

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 3-34.

Converting to and from Row Format

Before the access method can return key values to a query, the access method must convert source data to data types that the database server recognizes.

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 for each column.

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

Tip: The **mi_row_create()** function can affect performance because it requires database server resources. Use it only if you set the **am_keyscan** purpose flag for the access method.

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 *IBM Informix DataBlade API Programmer's Guide*.

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.

Return Value for mi_transition_type()	Transaction State
MI_ABORT_END	Unsuccessful transaction completion The database server must roll back the index to its state before the transaction began.

Warning: Uniform commit or rollback (called two-phase-commit protocol) with data in an external database server is not assured. 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 index.

For detailed information about the following subjects, refer to the *IBM Informix DataBlade API Programmer's Guide*:
- Handling state-transitions in a UDR
- End-of-transaction callback functions
- End-of-statement callback functions

For an example of a secondary access method that provides a state-transition callback function, see the indexing information on the IBM Informix Developer Zone at http://www.ibm.com/software/data/developer/informix.

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 *IBM Informix Guide to SQL: Syntax*
- The *IBM Informix Guide to SQL: Tutorial*
- The *IBM Informix Database Design and Implementation Guide*

These documents include examples of IBM 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 3-6 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 *IBM 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 SECONDARY ACCESS_METHOD or Alter ACCESS_METHOD statement:

- Statements
- Keywords
- Storage space type

Statements That the Access Method Does Not Support

The user can receive an SQL error for statements that require a purpose function that you did not supply. The access-method user guide must advise users which statements to avoid.

If the access method does not supply one or more of the following purpose functions, the access-method user guide must advise users not to use any of the following corresponding statements

Without this purpose function	Avoid this SQL statement
am_insert	INSERT, ALTER FRAGMENT
am_delete	DELETE, ALTER FRAGMENT
am_update	UPDATE
am_stats	UPDATE STATISTICS

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 you set the **am_unique** purpose flag in the **sysams** system catalog table, an SQL statement with the unique keyword fails. If the access method does not support unique indexes, the access-method user guide must advise users not to use the unique or DISTINCT 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 INDEX or ALTER FRAGMENT statement.

For more information about specifying storage spaces, refer to “Creating and Specifying Storage Spaces” on page 2-11.

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 2-13.

SQL Restrictions

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

- The FILLFACTOR clause in a CREATE INDEX statement
- ATTACH or DETACH in an ALTER FRAGMENT statement
- ASC or DESC keywords
- { CREATE | DROP } INDEX ONLINE operations

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.

Data Integrity Limitations

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 3-4 on page 3-14.

Provide the precise wording for the isolation levels that the access method supports. It is recommended 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. It is recommended that you take precautions
against phantom reads.
```

Figure 3-13. Sample Language to Describe Isolation Level

WHERE Clause Limitations

The **sysams** system catalog table has no indicator to inform the database server that a secondary access method cannot process complex qualifications. If the access method does not process the Boolean operators in a WHERE clause, perform the following actions:

- Provide examples in the user guide of UNION and subqueries that replace AND or OR operators in a WHERE clause, as the following example demonstrates.

Query Using Boolean Operator	Query using UNION

`SELECT * FROM videos`
`  WHERE title = 'Hamlet'`
`        OR year > 1980;`
	`SELECT * FROM videos WHERE title = 'Hamlet`
	`UNION`
	`SELECT * FROM videos WHERE year > 1980;`

- In the **am_scancost** purpose function, call the **mi_qual_issimple()** or **mi_qual_boolop()** accessor function to detect a Boolean operator.

  If **mi_qual_issimple()** returns MI_FALSE, for example, return a value that forces the optimizer to ignore this access method for the particular query. For an example, refer to Figure 4-15 on page 4-25.
- Raise an error if **mi_qual_issimple()** returns MI_FALSE to the **am_getnext** purpose function.

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 more information, refer to “Enabling Alternative Indexes” on page 3-15.
- Output from the **oncheck** utility

  For more information about the options that the **oncheck** provides, refer to the *IBM Informix Administrator’s Reference*. For more information about providing **oncheck** functionality, refer to the description of the **“am_check” on page 4-10** purpose function.

Chapter 4. Purpose-Function Reference

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 4-7 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. This section describes the purpose-function interface for the following SQL statements:

- ALTER FRAGMENT Statement Interface
- CREATE Statement Interface
- DROP Statement Interface
- INSERT, DELETE, and UPDATE Statement Interface
- SELECT...WHERE Statement Interface

This section also describes the "oncheck Utility Interface" on page 4-7.

Tip: The database server invokes the **am_open** and **am_close** purpose functions once per fragment for the 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 (....)
```

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 4-1 creates and fragments a **jobsx** index.

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

Figure 4-1. SQL to Create the Fragmented Jobsx Index

The statement in Figure 4-2 changes the fragment expression for **jobsx**, which redistributes the index entries.

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

Figure 4-2. SQL to Alter the Jobsx 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 4-3 through Figure 4-6 show the separate sequences of purpose functions that create the fragments and distribute the data for the SQL ALTER FRAGMENT statement in Figure 4-2. 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 4-3 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.

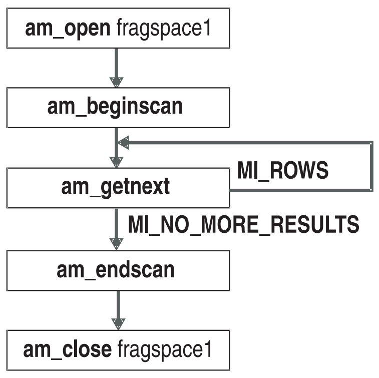


Figure 4-3. Getting All the Rows Entries in Fragment 1

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

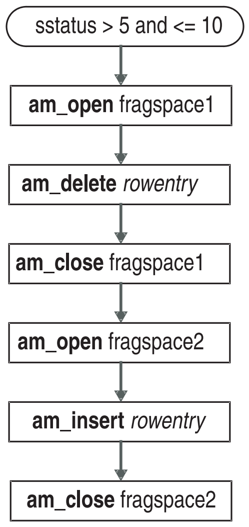


Figure 4-4. Moving Rows Entries Between Fragments

Figure 4-5 again shows the sequential scan in step 1. This scan returns all the rows from **fragment2**.

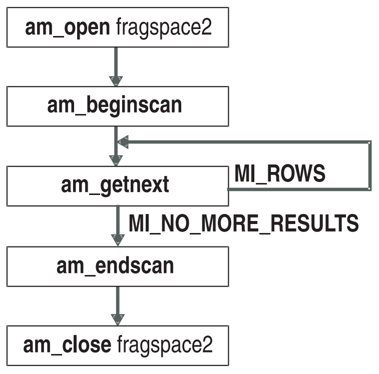


Figure 4-5. Getting All the Rows Entries in Fragment 2

Figure 4-6 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**.

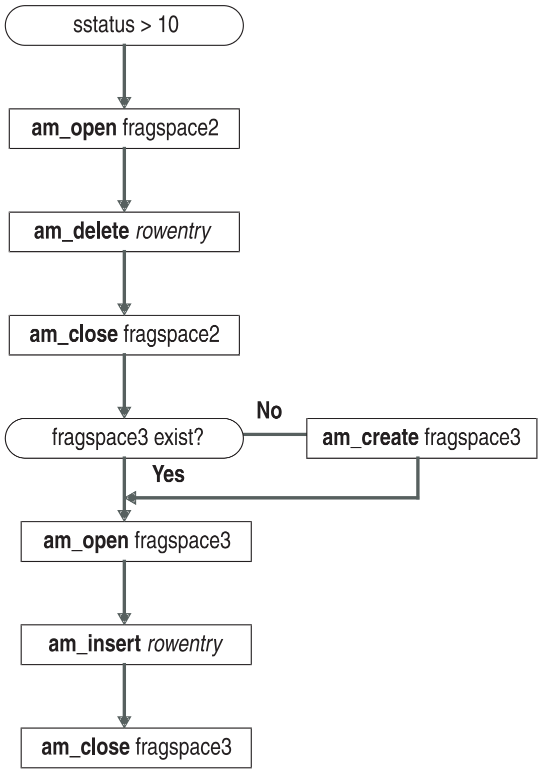


Figure 4-6. Adding and Filling a Fragment

For more information about fragments that a VII-based access method manages, refer to "Supporting Fragmentation" on page 3-12.

CREATE Statement Interface

Figure 4-7 and Figure 4-8 show the order in which the database server executes purpose functions for a CREATE TABLEINDEX statement. If the IN clause specifies multiple storage spaces to fragment the index, the database server repeats the sequence of purpose functions that Figure 4-7 and Figure 4-8 show for each storage space.

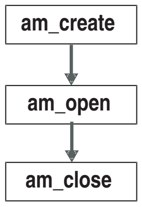


Figure 4-7. Processing a CREATE TABLE Statement

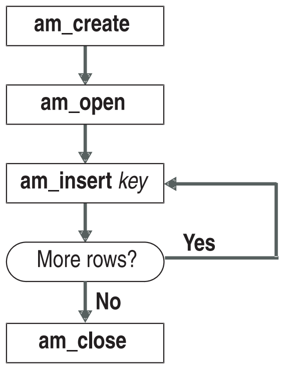


Figure 4-8. Processing a CREATE INDEX Statement

For more information about implementing the CREATE INDEX statement in the access method, refer to "Supporting Data Definition Statements" on page 3-8.

DROP Statement Interface

Figure 4-9 shows the processing for each fragment of a DROP INDEX or DROP DATABASE statement.

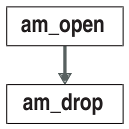


Figure 4-9. Processing a DROP Statement

INSERT, DELETE, and UPDATE Statement Interface

Figure 4-10 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 identifiers and row identifiers.

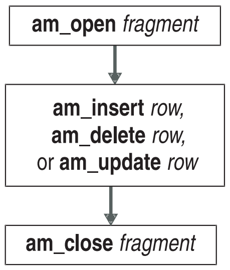


Figure 4-10. INSERT, DELETE, or UPDATE by Row Address

Figure 4-11 shows the order in which the database server executes purpose functions if the insert, delete, or in-place update has an associated WHERE clause.

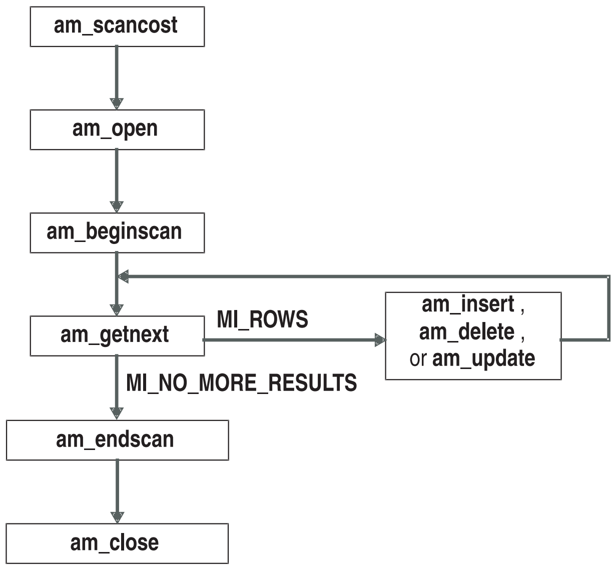


Figure 4-11. INSERT, DELETE, or UPDATE in a Subquery

Figure 4-12 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.

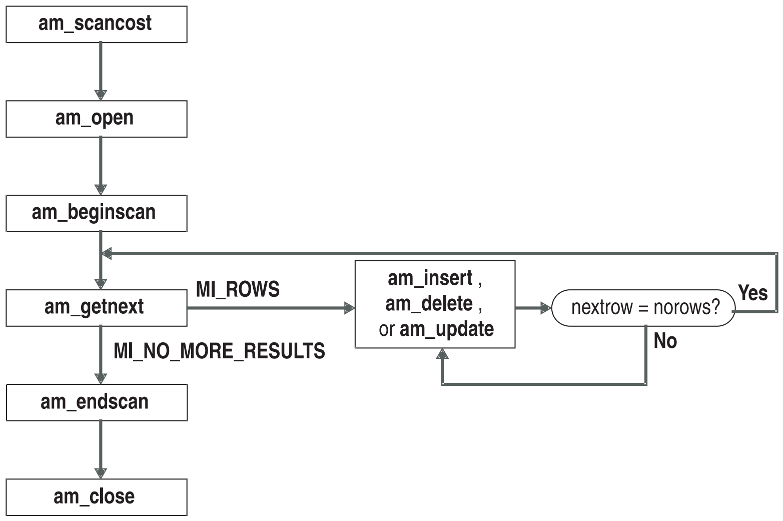


Figure 4-12. Returning Multiple Rows That Qualify for INSERT, DELETE, or UPDATE

For more information about implementing insert, delete, and update statements, refer to "Supporting Data Retrieval, Manipulation, and Return" on page 3-29.

SELECT...WHERE Statement Interface

Figure 4-13 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.

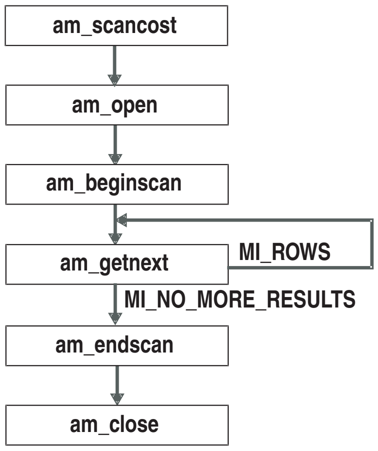


Figure 4-13. Processing a SELECT Statement Scan

oncheck Utility Interface

The **oncheck** utility reports on the state of an index and provides a means for a database server 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 indexes.

As Figure 4-14 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 4-14. 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 VII 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()**vindex_open(). To associate the unique purpose-function names to the corresponding prototype names, use the CREATE PRIMARYSECONDARY ACCESS_METHOD statement, as “CREATE ACCESS_METHOD (+)” on page 6-4 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 index that requires the access method. For detailed information about each descriptor, refer to “Descriptors” on page 5-2.

Purpose functions are simply entry points from which the access method calls other routines from the access-method library, DataBlade API functions, and the VII functions that “Accessor Functions” on page 5-8 describes.

am_beginscan

The database server calls **am_beginscan** to start a scan on a virtual index. 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:

- Obtains the qualification descriptor from the scan descriptor
- Parses the criteria in the qualification descriptor

  For a more detailed discussion, refer to “Processing Queries” on page 3-20.
- Determines the need for data type conversions to process qualification expressions
- Calls the necessary accessor functions to retrieve the index operator class from the system catalog

  The **am_beginscan** purpose function can obtain and store the function descriptor for strategy and support functions. For more information, refer to “Executing Qualification Functions” on page 3-24 and “Using FastPath” on page 3-19.
- Initiates a search for data that fulfills the qualification, based on the information in the qualification descriptor
- Allocates PER_COMMAND memory to build user data and then stores 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 3-2.

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 2-4

am_check

If a user executes the oncheck utility for a virtual index, 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 index 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. The additional **-y** or **-n** option specifies that the user wants **oncheck** to repair any damage to an index. For information about **oncheck** options, refer to the *IBM Informix 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 index 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 VII 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

MI_CHECK_IDXKEYS()
MI_DISPLAY_IDXKEYS()
	-ci
	-pk
		Check and display index key values

MI_CHECK_IDXKEYS_ROWIDS()
MI_DISPLAY_IDXKEYS_ROWIDS()
	-cI
	-pK
		Check and display index keys and rowids

MI_DISPLAY_IDXKEYLEAVES()	**-pl**	Check and display leaf key values
MI_DISPLAY_IDXKEYLEAVES_ROWIDS ()	**-pL**	Check and display leaf key values and row identifiers

Macro	Option	oncheck Action
MI_DISPLAY_IDXSPACE()	**-pT**	Check and display index space usage
MI_CHECK_NO_TO_ALL	**-n**	Do not attempt to repair inconsistencies
MI_CHECK_YES_TO_ALL	**-y**	Automatically repair an index

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** demonstrate:

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

Checking and Displaying Table Index State: The access method can call accessor function **mi_tab_spacetype()** to determine whether the specified index 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 *IBM Informix 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 index, **am_check** can call the **mi_tab_check_msg()** function.

Handling Index Problems: An access method can contain the logic to repair an index and execute additional macros to determine whether it should repair a problem that **am_check** detects. The following table shows the **oncheck** options that enable or disable repair and the **am_check** macro that detects each option.

Option	Meaning	Macro
-y	Automatically repair any problem.	MI_CHECK_YES_TO_ALL
-n	Do not repair any problem.	MI_CHECK_NO_TO_ALL

If a user does not specify **-y** or **-n** with an **oncheck** command, the database server displays a prompt that asks whether the user wants the index repaired. Similarly, when both **MI_CHECK_YES_TO_ALL()** and **MI_CHECK_NO_TO_ALL()** return MI_FALSE, **am_check** can call accessor function **mi_tab_check_set_ask()**, which causes the database server to ask if the user wants the index repaired. If the user answers `yes` or `y`, the database server adds **-y** to the *option* argument and executes **am_check** a second time.

Tip: Store any information that **am_check** needs to repair the index in PER_STATEMENT memory. Call **mi_tab_check_is_recheck()** to determine if the **am_check** can use previous PER_STATEMENT information that it stored in the preceding execution. If **mi_tab_check_is_recheck()** returns MI_TRUE, call **mi_tab_userdata()** to access the problem description.

If either the **MI_CHECK_YES_TO_ALL()** macro or **mi_tab_check_is_recheck()** accessor function returns MI_TRUE, **am_check** should attempt to repair an index.

Important: Indicate in the access-method user guide whether the access method supports index repair. Issue an exception if the user specifies a repair that **am_check** cannot make.

Return Values

MI_OK validates the index structure as error free.

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

Related Topics

See the descriptions of:

- Purpose functions **am_open** and **am_close**
- Accessor functions **mi_tab_check_msg(), mi_tab_check_set_ask(),** and **mi_tab_check_is_recheck()** in Chapter 5, "Descriptor Function Reference," on page 5-1

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 index descriptor.

Usage

The **am_close** function:

- Deallocates user-data memory that **am_open** allocated with a PER_STATEMENT duration
- Calls **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, 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 *IBM Informix DataBlade API Programmer's Guide*
- "Starting and Ending Processing" on page 2-4

am_create

The database server calls **am_create** to process a CREATE INDEX statement. The **am_create** function creates the index, based on the information in the table descriptor, which describes the keys in an index.

Syntax

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

tableDesc	points to the index 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 **systablessysindexes**. For example, a user might issue the CREATE INDEX command to register an existing, external index in the database server system catalog.

The **am_create** function typically:

- Calls accessor functions to extract index 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 the CREATE INDEX specifies
- Validates CREATE INDEX statements that specify identical keys, as described in "Enabling Alternative Indexes" on page 3-15
- 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 3-9
- Executes support functions that build the index

  The access method might supply the support functions or execute UDRs from outside the access-method shared-object library. For more information, refer to "Using FastPath" on page 3-19.

Important: By default, transaction logging is disabled in sbspaces. To find out how to turn logging on, refer to "Ensuring Data Integrity" on page 3-11.

Return Values

MI_OK indicates success. MI_ERROR indicates failure.

Related Topics

In this publication, see the description of:

- Purpose function **am_drop**
- "Creating and Dropping Database Objects" on page 2-4

In the *IBM Informix DataBlade API Programmer's Guide*, 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_ROW *row,
   MI_AM_ROWID_DESC *ridDesc)
```

tableDesc	points to the index descriptor.
row	points to a row structure that contains the key value to delete.
ridDesc	points to the row-ID descriptor.

Usage

The **am_delete** purpose function deletes one rowindex key in the virtual index. Additionally, the function passes (by reference) the row-ID descriptor, which contains the location of the underlying table row to delete.

In response to a DELETE statement, the database server first calls the appropriate purpose functions to scan for the index entry or entries that qualify for deletion and then executes **am_delete** separately for each qualifying entry.

The access method identifies and executes support functions to adjust the index structure after the delete. For more information, refer to "Using FastPath" on page 3-19.

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," on page 6-1.

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 3-32.

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 6-8
- "Inserting, Deleting, and Updating Data" on page 2-5

am_drop

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

Syntax

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

tableDesc	points to the index 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 Topics

See the descriptions of:

- Purpose function **am_create**
- "Creating and Dropping Database Objects" on page 2-4

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:

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

  For more information on PER_COMMAND memory and memory deallocation, refer to "Storing Data in Shared Memory" on page 3-2.
- Checks 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 3-31.

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 2-4

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_AM_ROWID_DESC *ridDesc)
```

scanDesc	points to the scan descriptor.
row	points to the location where an access method can create a row structure that contains the index keys. Most secondary access methods fill the *row* location with NULL values and do not create rows. Create a row only if the access method supports the **am_keyscan** purpose flag.
ridDesc	points to the returned row-ID descriptor.

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 rowindex entry, an **am_getnext** purpose function can:

- Read source index data into user data
- Execute strategy 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 3-31.
- Call **mi_id_setrowid()** and **mi_id_setfragid()** to give the location of the table row to the database server

Typically, the database server uses the information that the access method sets in the row-id descriptor to access a row from the indexed table. The access method can build a row from the key values if you set the **am_keyscan** purpose flag to indicate that the access method returns keys to the query, as "Bypassing Table Scans" on page 3-28 describes.

To find out how to create a row, refer to "Converting to and from Row Format" on page 3-31.

The **am_getnext** purpose function can loop to fill a shared-memory buffer with multiple rowsindex entries. For more information about buffering, see "Buffering Multiple Results" on page 3-28 and the example of an **am_getnext** loop in "Buffering Multiple Results" on page 3-28.

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 3-2.

Return Values

MI_ROWS indicates the return of a row-ID descriptor for 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_scan_quals(), mi_tab_niorows(),** and **mi_tab_setnextrow()** in Chapter 5, "Descriptor Function Reference," on page 5-1
- The **am_keyscan** purpose flag in "Purpose Options" on page 6-7
- DataBlade API function mi_row_create() in the *IBM Informix DataBlade API Programmer's Guide*
- "Executing Qualification Functions" on page 3-24 and "Using FastPath" on page 3-19
- "Optimizing Queries" on page 2-4

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
- A CREATE INDEX statement that builds an index on preexisting data

Syntax

```
mi_integer am_insert(MI_AM_TABLE_DESC *tableDesc,
        MI_ROW *row,
        MI_AM_ROWID_DESC *ridDesc)
```

tableDesc	points to the index descriptor.
row	points to a row structure in shared memory that contains the values for the access method to insert.
ridDesc	points to the row-ID descriptor, which contains the row identifier and fragment identifier for the new row that corresponds to the new index entry.

Usage

If *row* and *ridDesc* are `0`, **am_insert** calls **mi_tab_niorows()** to determine the maximum number of new index entries to expect. For each entry up to the maximum number passed, the **am_insert** function calls **mi_tab_nextrow()**. For a complete example, see "mi_tab_nextrow()" on page 5-71.

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 index
- Manipulates the index structure to make room for the new entry
- Stores the new data in the appropriate sbspace or extspace

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

To manipulate the index structure, **am_insert** executes support functions, either with a call to an access-method function or with the DataBlade API FastPath facility. For more information, refer to "Using FastPath" on page 3-19. Call **mi_tab_userdata()** to retrieve the pointer to PER_STATEMENT user data. Call **mi_routine_exec()** to execute the support function.

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," on page 6-1.

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 3-32.

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_rowid** in "Setting Purpose Functions, Flags, and Values" on page 6-8
- "Using FastPath" on page 3-19 and information about the DataBlade API FastPath facility in the *IBM Informix DataBlade API Programmer's Guide*
- "Inserting, Deleting, and Updating Data" on page 2-5

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 index descriptor.

Usage

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

- Determines the reason, or mode, for the open, as described in "mi_tab_mode()" on page 5-68
- Allocates PER_STATEMENT memory for a user-data structure as described in "Persistent User Data" on page 3-3
- Opens 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.
- Registers callback functions to handle exceptions, as described in "Handling the Unexpected" on page 3-6
- Calls the appropriate DataBlade API functions to obtain a file handle for an extspace or an LO handle for a smart large object
- Calls **mi_setniorows()** to set the number of entries for which the database server should allocate memory

  For more information, refer to "Building New Indexes Efficiently" on page 3-14.

Return Values

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 *IBM Informix DataBlade API Programmer's Guide*
- **mi_tab_mode()** and **mi_tab_setniorows()** in Chapter 5, "Descriptor Function Reference," on page 5-1
- "Starting and Ending Processing" on page 2-4

am_rescan

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

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 index.

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," on page 5-1
- "Optimizing Queries" on page 2-4

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 index descriptor.
qualDesc	points to the qualification descriptor, which specifies the criteria that a table rowindex key 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 or bypasses the virtual index, 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 index entries that qualify. If retrieving an index entry does not supply the columns that the SELECT statement projects, the scan cost includes both of the following:

- Number of disk accesses to fetch the entry from the index
- Number of disk accesses to fetch the entry from the table

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,” on page 6-1.

Forcing Reoptimization: The optimizer might need a new scan cost for subsequent scans of the same index, for example because of a join. To execute **am_scancost** before each rescan, call the **mi_qual_setreopt()** function.

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 4-15, 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 -1;
   }
```

Figure 4-15. Forcing a Table Scan

The database server might respond to a negative scan-cost value in one of the following ways:

- Use another index, if available
- Perform a sequential table scan

Warning: The database server has no means to detect if a secondary access method does not set values for complex expressions. If an access method has no code to evaluate AND or OR, call accessor function **mi_qual_boolop()** or **mi_qual_issimple()** to determine if the qualification descriptor contains a Boolean operator.

Return Values

This function returns a pointer to an **mi_real** data type that contains the cost value.

Related Topics

See the descriptions of:

- Purpose functions **am_stats** and **am_getnext**
- Purpose flags **am_scancost** and **am_rowids** in “Setting Purpose Functions, Flags, and Values” on page 6-8
- 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,” on page 5-1
- DataBlade API function **mi_row_create()** in the *IBM Informix DataBlade API Programmer’s Guide*

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_ISTATS_DESC *istatsDesc)
```

tableDesc	points to the index 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 STATISTICS 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 *IBM Informix Guide to SQL: Syntax*.

The **am_stats** purpose function calls the various VII accessor functions that set values in the statistics descriptors for the database server. The database server places the statistics descriptor results in the **systables**, **syscolumns**, and **sysindexes** 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:

- The **am_scancost** purpose function
- Accessor functions **mi_tab_update_stat_mode()** and **mi_tstats_*mi_istats_*** in Chapter 5, "Descriptor Function Reference," on page 5-1
- The "Statistics Descriptor" on page 5-6
- "Updating Statistics" on page 3-26

am_truncate

Dynamic Server provides built-in **am_truncate** purpose functions for its primary access methods that support TRUNCATE operations on columns of permanent and temporary tables. Dynamic Server also provides a built-in **am_truncate** purpose function for its secondary access method for TRUNCATE operations on B-tree indexes.

Usage

You must use the **am_truncate()** access method with the TRUNCATE statement to operate on virtual tables or on tables with virtual indexes. You use TRUNCATE to depopulate a local table and free the storage space that formerly held its data rows and B-tree structures.

For more information **am_truncate()**, see the *IBM Informix Guide to SQL: Syntax*.

am_update

The database server calls **am_update** to process an UPDATE statement if the update affects the key rows or results in changing the physical location of the row.

Syntax

```
mi_integer am_update(MI_AM_TABLE_DESC *tableDesc, MI_ROW *oldrow,
       MI_AM_ROWID_DESC *oldridDesc,
       MI_ROW *newrow,
       MI_AM_ROWID_DESC *newridDesc)
```

tableDesc	points to the index descriptor.
oldrow	points to the row structure that contains the before-update values.
oldridDesc	points to the row-ID descriptor for the row before the update.
newrow	points to the row structure that contains the updated values.
newridDesc	points to the row-ID descriptor for the updated row.

Usage

The **am_update** function modifies the contents of an existing rowindex entry.

The access method stores the row identifier and fragment identifier for the updated table row in *newridDesc*. To alter the contents of a component in the key, **am_update:**

- Deletes the old key
- Adjusts the key data format in *newrow* to conform to the source data
- Calls the appropriate support functions to make room for the new entry
- Stores the new entry

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,” on page 6-1.

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 3-32.

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 6-8

- "Using FastPath" on page 3-19 and information about the DataBlade API Fastpath facility in the *IBM Informix DataBlade API Programmer's Guide*
- "Inserting, Deleting, and Updating Data" on page 2-5

Chapter 5. Descriptor Function Reference

In This Chapter

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

- "Descriptors" on page 5-2 describes the predefined data structures through which the database server and access method pass information.
- "Include Files" on page 5-8 lists the header files with descriptor and function declarations that the access method must include.
- "Accessor Functions" on page 5-8 lists every function that the IBM Informix database server provides specifically for use with the VII.

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 functions, refer to Chapter 4, "Purpose-Function Reference," on page 4-1.

Descriptors

The application programming interface (API) that is provided with the VII 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 VII provides the following descriptors and accessor functions.

Descriptor	Describes	Accessor-Function Prefix	Reference
key descriptor (MI_AM_KEY_DESC)	Index keys, strategy functions, and support functions	mi_key_	"Key Descriptor" on page 5-3
qualification descriptor (MI_AM_QUAL_DESC)	WHERE clause criteria	mi_qual_	"Qualification Descriptor" on page 5-4
row descriptor (MI_ROW)	Order and data types of projected columns	Various DataBlade API functions	*IBM Informix DataBlade API Programmer's Guide*
row-id descriptor (MI_AM_ROWID_DESC)	Indexed table row location	mi_id_	"Row-ID Descriptor" on page 5-5
scan descriptor (MI_AM_SCAN_DESC)	SELECT clause projection	mi_scan_	"Scan Descriptor" on page 5-6
statistics descriptor (MI_AM_ISTATS_DESC)	Distribution of values	mi_istats_	"Statistics Descriptor" on page 5-6
table descriptor (MI_AM_TABLE_DESC)	Index location and attributes	mi_tab_	"Table Descriptor" on page 5-7

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 5-8.

Important: Because the internal structure of any VII descriptor might change, they are declared 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.

Key Descriptor

The key descriptor, or MI_AM_KEY_DESC structure, identifies the keys and operator class for an index. The following functions extract information from the key descriptor.

Accessor Function	Return Value
mi_key_funcid()	The routine identifier of the UDR that determines the value of a specified key in a functional index
mi_key_nkeys()	The number of columns in an index key
mi_key_opclass(), mi_key_opclass_name()	The identifier or name of the operator class for a specified column of the index key
mi_key_opclass_strat()	The name of one strategy function Typically, an access method calls the **mi_qual_funcid()** function to obtain the routine identifier and does not use **mi_key_opclass_strat()**.
mi_key_opclass_nsupt()	The number of support functions
mi_key_opclass_supt()	The name of one support function For an example of how to use the function names to execute the function, see "Obtaining the Routine Identifier" on page 3-19.

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 VII **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 strategy function occupies within an index entry
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 strategy 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
mi_qual_funcid()	The routine identifier of a strategy function
mi_qual_funcname()	The name of a strategy function
mi_qual_handlenull()	MI_TRUE if the strategy function accepts null arguments
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

Accessor Function	Return Value
mi_qual_stratnum()	The ordinal number of the operator-class strategy function

The following accessor functions set values in the descriptor.

Accessor Function	Value Set
mi_qual_setoutput()	A host-variable value
mi_qual_setreopt()	An indicator to force reoptimization between rescans

Row Descriptor

A row descriptor, or MI_ROW_DESC structure, typically describes the columns that the CREATE INDEX statement establishes for an index. 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 *IBM Informix DataBlade API Programmer's Guide*.

Row-ID Descriptor

A particular row identifier can appear in multiple fragments. For example, row 1 in fragment A describes a different customer than row 1 in fragment B. The unique fragment identifier enables the database server or access method to locate the correct row 1.

A secondary access method sets these values in a row-ID descriptor, or MI_AM_ROWID_DESC structure, during an index scan. The following functions set data in the row-ID descriptor.

Accessor Function	Value Set
mi_id_setrowid()	The row identifier
mi_id_setfragid()	The fragment identifier

The database server fills the row-ID descriptor when it calls:
- **am_insert** or **am_delete** to add or delete a table row
- **am_insert** to build a new index
- **am_insert** and **am_delete** in response to an ALTER FRAGMENT command

The following accessor functions extract information from the descriptor.

Accessor Function	Return Value
mi_id_rowid()	The row identifier
mi_id_fragid()	The fragment identifier

The following system catalog information describes a fragment identifier:

- The **partnum** attribute in the **systables** system catalog table
- The **partn** attribute in the **sysfragments** system catalog table

For detailed information about system catalog tables, refer to the *IBM Informix Guide to SQL: Reference*.

Scan Descriptor

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

- 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 index
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 index 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_ISTATS_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_istats_set2lval()	A pointer to the second largest key value in the index
mi_istats_set2sval()	A pointer to the second smallest key value in the index
mi_istats_setclust()	The degree of clustering A low number indicates fewer clusters and a high degree of clustering.
mi_istats_setnleaves()	The number of leaves in the index
mi_istats_setnlevels()	The number of levels in the index
mi_istats_setnunique()	The number of unique keys in the index

Table Descriptor

The table descriptor, or MI_AM_TABLE_DESC structure, provides information about the index, particularly the data definition from the CREATE INDEX 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 INDEX statement
mi_tab_check_is_recheck()	MI_TRUE if the database server invokes **am_check** to recheck and possibly repair an index
mi_tab_createdate()	The date that the index was created
mi_tab_isindex()	MI_TRUE for a secondary access method
mi_tab_isolevel()	The isolation level
mi_tab_keydesc()	A pointer to the key descriptor
mi_tab_mode()	The input/output mode (read-only, read and write, write-only, and log transactions)
mi_tab_name()	The index name
mi_tab_nextrow()	One entry from shared memory to insert in a new index
mi_tab_niorows()	The number of rows that **mi_tab_setniorows()** sets
mi_tab_nparam_exist()	The number of indexes that are defined for the same combination of table key columns
mi_tab_numfrags()	The number of fragments in the index or 1 for a nonfragmented index
mi_tab_owner()	The index owner
mi_tab_param_exist()	Configuration parameters and values for one of multiple indexes that pertain to the same table and composite key
mi_tab_partnum()	The unique partition number, or fragment identifier, of this index or fragment
mi_tab_rowdesc()	A pointer to a row descriptor that describes the columns in the composite index key

Accessor Function	Return Value
mi_tab_spaceloc()	The extspace location of the index fragment
mi_tab_spacename()	The storage space name for the fragment from the CREATE INDEX statement IN clause
mi_tab_spacetype()	The type of space used for the index: 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_unique()	MI_TRUE if this index should enforce unique keys
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

The following accessor functions set values in the table descriptor.

Accessor Function	Value Set
mi_tab_check_set_ask()	An indicator that **am_check** detects a problem in an index
mi_tab_setniorows()	The number of rows that shared memory can store from a scan for a new index
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 VII.
- If your access method alters the default memory duration, include the **memdur.h** and **minmdur.h** files.

Global Language Support

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

End of Global Language Support

Accessor Functions

The VII 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 5-2.

This chapter lists detailed information about specific VII 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 Prefix	Descriptor	Accessor-Function Prefix
Key	**mi_key_*()**	Row ID	**mi_id_*()**
Qualification	**mi_qual_*()**	Scan	**mi_scan_*()**
	mi_eval_am_qual()	Statistics	**mi_istats_*()**
	mi_init_am_qual()	Table	**mi_tab_*()**

mi_id_fragid()

The **mi_id_fragid()** function retrieves the fragment identifier from the row-ID descriptor.

Syntax

```
mi_integer mi_id_fragid(MI_AM_ROWID_DESC *rowidDesc)
```

rowidDesc	points to the row-ID descriptor.

Usage

The **am_insert** purpose function calls **mi_id_fragid()** to obtain a value and add it to the index entry with the key.

Return Values

The integer identifies the fragment that contains the row this key indexes.

Related Topics

See the description of functions **mi_id_setfragid(), mi_id_rowid(),** and **mi_id_setrowid().**

mi_id_rowid()

The **mi_id_rowid()** function retrieves the row identifier from the row-ID descriptor.

Syntax

```
mi_integer mi_id_rowid(MI_AM_ROWID_DESC *rowidDesc)
```

rowidDesc	points to the row-ID descriptor.

Usage

The **am_insert** purpose function calls **mi_id_rowid()** to obtain a value and add it to the index entry with the key.

Return Values

The integer identifies the row that this key indexes. For example, the row identifier might offset a fragment identifier to complete the location of the row.

Related Topics

See the description of accessor functions **mi_id_setrowid(), mi_id_fragid(),** and **mi_id_setfragid().**

mi_id_setfragid()

The **mi_id_setfragid()** function sets the fragment identifier for the row.

Syntax

```
void mi_id_setfragid(MI_AM_ROWID_DESC *rowidDesc,
   mi_integer fragid)
```

rowidDesc	points to the row-ID descriptor.
fragid	provides the fragment identifier.

Usage

The **am_getnext** purpose function calls **mi_id_setfragid()** to provide the fragment location for the indexed primary data.

Return Values

None

Related Topics

See the description of functions **mi_id_fragid(), mi_id_rowid(),** and **mi_id_setrowid().**

mi_id_setrowid()

The **mi_id_setrowid()** function sets the row identifier for the row.

Syntax

```
void mi_id_setrowid(MI_AM_ROWID_DESC *rowidDesc,
   mi_integer rowid)
```

rowidDesc	points to the row-ID descriptor.
rowid	provides the row identifier.

Usage

The **am_getnext** purpose function calls **mi_id_setrowid()** so that the database server has the physical location of the indexed primary data.

Return Values

None

Related Topics

See the description of functions **mi_id_setrowid()** and **mi_id_rowid().**

mi_istats_setclust()

The **mi_istats_setclust()** function stores the degree of clustering for an index in the statistics descriptor.

Syntax

```
void mi_istats_setclust(MI_AM_ISTATS_DESC *istatsDesc,
   mi_integer clustering)
```

istatsDesc	points to the statistics descriptor.
clustering	specifies the degree of clustering, from number of pages to number of rows.

Usage

Call this function from **am_stats**. The database server places the value that this function sets in the **clust** column of the **sysindices** system catalog table.

Clustering specifies the degree to which the rows are in the same order as the index. For example, if the index references a table that resides in page-size areas, such as in a dbspace or sbspace, you can estimate clustering as follows:

- The lowest possible *clustering* value equals the number of pages that data occupies, or one cluster per page.
- The highest possible value (and least amount of clustering) equals the number of rows, or one cluster per entry.

Return Values

None

mi_istats_set2lval()

The **mi_istats_set2lval()** function stores the second-largest index-key value in the statistics descriptor.

Syntax

```
void mi_istats_set2lval(MI_AM_ISTATS_DESC *istatsDesc,
   void *2lval)
```

istatsDesc	points to the statistics descriptor.
2lval	points to the second-largest key value in the index.

Usage

To determine the maximum value for an index key while it evaluates a query plan, the optimizer looks at the **colmax** value for the key column in the **syscolumns** system catalog table. The **colmax** column holds a 4-byte integer that represents the second-largest key value in the index. The optimizer assesses the second-largest key value to avoid the distortion that an excessive value can cause to the data distribution.

The **am_stats** purpose function can provide the second-largest value for each key. After storing the value in memory, pass it by reference with the **mi_istats_set2lval()** function. The database server places the first four bytes that begin at address *2lval* as an integer value in the **colmax** column.

Return Values

None

Related Topics

See the description of function **mi_istats_set2sval().**

mi_istats_set2sval()

The **mi_istats_set2sval()** function stores the second-smallest index-key value in the statistics descriptor.

Syntax

```
void mi_istats_set2sval(MI_AM_ISTATS_DESC *istatsDesc,
   void *2sval)
```

IstatsDesc	points to the statistics descriptor.
2sval	points to the second-smallest key value in the index.

Usage

To determine the minimum value for an index key while it evaluates a query plan, the optimizer looks at the **colmin** value for the key column in the **syscolumns** system catalog table. The **colmin** column holds a 4-byte integer that represents the second-smallest key value in the index. The optimizer assesses the second-smallest key value to avoid the distortion that an abnormally low value can cause to the data distribution.

The **am_stats** purpose function can provide the second-largest value for each key. After storing the value in memory, pass it by reference with the **mi_istats_set2sval()** function. The database server places the first four bytes that begin at address *2sval* as an integer value in the **colmin** column.

Return Values

None

Related Topics

See the description of function **mi_istats_set2lval().**

mi_istats_setnlevels()

The **mi_istats_setnlevels()** function stores the number of index levels in the statistics descriptor.

Syntax

```
void mi_istats_setnlevels(MI_AM_ISTATS_DESC *istatsDesc,
   mi_integer nlevels)
```

istatsDesc	points to the statistics descriptor.
nlevels	provides the number of levels in the index.

Usage

Call this function from **am_stats**. The database server places the value that this function sets in the **levels** column of the **sysindices** system catalog table.

Return Values

None

mi_istats_setnleaves()

The **mi_istats_setnleaves()** function stores the number of index leaf nodes in the statistics descriptor.

Syntax

```
void mi_istats_setnleaves(MI_AM_ISTATS_DESC *istatsDesc,
   mi_integer nleaves)
```

istatsDesc	points to the statistics descriptor.
nleaves	provides the number of leaf nodes in the index.

Usage

Call this function from **am_stats**. The database server places the value that this function sets in the **leaves** entry of the **sysindices** system catalog table.

Return Values

None

mi_istats_setnunique()

The **mi_istats_setnunique()** function stores the number of unique index keys in the statistics descriptor.

Syntax

```
void mi_istats_setnunique(MI_AM_ISTATS_DESC *istatsDesc,
   mi_integer nunique)
```

istatsDesc	points to the statistics descriptor.
nunique	indicates the number of unique keys in the index.

Usage

Call this function from **am_stats**. The database server places the value that this function sets in the **nunique** entry of the **sysindices** system catalog table.

Return Values

None

mi_key_funcid()

The **mi_key_funcid()** function retrieves the identifier of the function that computes the key values in a functional index.

Syntax

```
mi_integer mi_key_funcid(MI_AM_KEY_DESC *keyDesc,
   mi_integer keyNum)
```

keyDesc	points to the key descriptor.
keyNum	specifies the column number of the index-based key or 0 for a single-key index. For the first (or only) key, pass 0 as *keyNum.* Increment *keyNum* by one for each subsequent key in a composite index.

Usage

A UDR returns the values that make up a functional index. For example, the following statement creates an index from the values that the **box()** function returns:

```
CREATE INDEX box_func_idx ON zones (box(x1,y1,x2,y2)) USING map_am;
```

Use the DataBlade API FastPath facility to obtain values for function-based index keys.

To execute a function on a key column:

1. Call **mi_key_funcid()** to extract the routine identifier from the qualification descriptor.
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, which executes the function in a virtual processor.

Return Values

A positive integer identifies the function that creates the values in the *keyNum* position of a composite-key index.

A return value of `0` indicates that the specified *keyNum* contains column values and does not belong to a functional index.

A negative value indicates that the CREATE INDEX statement specifies an unknown function to create the key.

Related Topics

See the discussions of:

- Fastpath functions in the *IBM Informix DataBlade API Programmer's Guide*, including functions **mi_func_desc_by_typeid()** and **mi_routine_exec()**.
- CREATE INDEX in the *IBM Informix Guide to SQL: Syntax*, particularly functional index information.

mi_key_nkeys()

The **mi_key_nkeys()** function returns the number of columns in the index key.

Syntax

```
mi_integer mi_key_nkeys(MI_AM_KEY_DESC *keyDesc)
```

keyDesc	points to the key descriptor.

Return Values

The integer indicates the number of keys in the index.

mi_key_opclass(), mi_key_opclass_name()

Identify the **mi_key_opclass()** and **mi_key_opclass_name()** functions by identifier number or name, the operator class that provides the support, and strategy functions for a specified column in a key.

Syntax

```
mi_integer
mi_key_opclass(MI_AM_KEY_DESC *keyDesc, mi_integer keyNum)
mi_string *
mi_key_opclass_name(
   MI_AM_KEY_DESC *keyDesc, mi_integer keyNum)
```

keyDesc	points to the key descriptor.
keyNum	specifies the column number of a key in a composite-key index or 0 for a single-key index.

Usage

An operator class consists of the strategy and support functions with which the access method manages a particular data type. To determine which operator class to use for a particular key, identify the key as an argument to **mi_key_opclass()** or **mi_key_opclass_name().** To obtain the operator class identifier number, call **mi_key_opclass()**. To obtain the operator class name, call **mi_key_opclass_name()**.

Identifying the Key: The integer argument *keyNum* identifies the column number in the index entry. A one-column index contains only *keyNum* `0`. A two-column key contains *keyNum* `0` and `1`. To determine the number of columns in a key, call **mi_key_nkeys()**.

Identifying the Operator Class: The access method can execute **mi_key_opclass()** or **mi_key_opclss_name()** for each column in a multiple-column key because the columns do not necessarily all use the same operator class. A CREATE INDEX statement can assign different operator classes to individual columns in a multiple-column key. The following example defines an index with multiple operator classes:

```
CREATE OPCLASS str_ops FOR video_am
   STRATEGIES (lessthan(char, char), lessthanorequal(char, char),
            equal(char, char),
            greaterthanorequal(char, char), greaterthan(char, char))
   SUPPORT(compare)
CREATE OPCLASS int_ops FOR video_am
   STRATEGIES (lessthan(int, int), lessthanorequal(int, int),
         equal(int, int),
         greaterthanorequal(int, int), greaterthan(int,int))
   SUPPORT(compare)

CREATE TABLE videos (title char(50), year int, copies int)
CREATE INDEX vidx ON videos (title str_ops, year int_ops) USING video_am
```

As the access-method creator, you must assign a default operator class for the access method. To assign a default operator class, set the **am_defopclass** purpose value with the ALTER ACCESS_METHOD statement. If the CREATE INDEX statement does not specify the operator class to use, the **mi_key_opclass()** or **mi_key_opclass_name()** function specifies the default operator class.

Return Values

For **mi_key_opclass(),** a positive return value identifies the operator class in the **sysopclass** system catalog table. A return value of `-1` indicates that the function passed an invalid *keyNum* value.

For **mi_key_opclass_name()**, a non-NULL pointer identifies the name of the operator class. A return value of null indicates that the function passed an invalid *keyNum* value.

Related Topics

See the description of:

- The **am_defopclass** purpose value in "Setting Purpose Functions, Flags, and Values" on page 6-8
- Accessor function **mi_key_nkeys()**

mi_key_opclass_nstrat()

The **mi_key_opclass_nstrat()** function retrieves the number of strategy functions in the operator class associated with the key.

Syntax

```
mi_integer mi_key_opclass_nstrat(MI_AM_KEY_DESC *keyDesc,
    mi_integer keyNum)
```

keyDesc	points to the key descriptor.
keyNum	specifies the column number of a key in a composite-key index or 0 for a single-key index. For the first (or only) key, pass 0 as *keyNum.* Increment *keyNum* by 1 for each subsequent key in a composite index.

Usage

The access method can use either the function name or routine identifier to execute a strategy function. Use **mi_key_opclass_nstrat()** if the access method needs strategy-function names. The **mi_key_opclass_nstrat()** returns the number of function names to retrieve for a single key-column with the **mi_key_opclass_strat()** function.

For a multiple-column key, **mi_key_opclass_nstrat()** might return different values for each column. The integer argument *keyNum* specifies a column by sequential position in the index key. A one-column index contains only *keyNum* `0`. A two-column composite key contains *keyNum* `0` and `1`. To determine the maximum *keyNum* value, call **mi_key_nkeys(**). If **mi_key_nkeys()** returns a value of `1` or greater, the index contains multiple key columns.

Return Values

A positive integer indicates the number of strategy functions that the key descriptor contains for the specified column in the key.

A value of `-1` indicates that *keyNum* specifies an invalid column number for the key.

Related Topics

See the descriptions of:

- Functions **mi_key_opclass_strat(), mi_key_nkeys(,** and **mi_key_opclass(), mi_key_opclass_name()**
- "Supporting Multiple-Column Index Keys" on page 3-17

mi_key_opclass_nsupt()

The **mi_key_opclass_nsupt()** function retrieves the number of support functions in the operator class associated with the key.

Syntax

```
mi_integer mi_key_opclass_nsupt(MI_AM_KEY_DESC *keyDesc,
   mi_integer keyNum)
```

keyDesc	points to the key descriptor.
keyNum	specifies the column number of a key in a composite-key index or 0 for a single-key index. For the first (or only) key, pass 0 as *keyNum.* Increment *keyNum* by 1 for each subsequent key in a composite index.

Usage

The **mi_key_opclass_nsupt()** function returns the number of operator class support functions for a column in the index. It can be used to obtain the function names with the **mi_key_opclass_supt()** function.

For a multiple-column key, **mi_key_opclass_nsupt()** might return different values for each column. The integer argument *keyNum* specifies a column by sequential position the index key. A one-column index contains only *keyNum* `0`. A two-column composite key contains *keyNum* `0` and `1`. To determine the maximum *keyNum* value, call **mi_key_nkeys(**). If **mi_key_nkeys()** returns a value of `1` or greater, the index contains multiple key columns.

Return Values

A positive integer indicates the number of support functions that the key descriptor contains for the specified key column.

A value of `-1` indicates that *keyNum* specifies an invalid column number for the key.

Related Topics

See the descriptions of:

- Functions **mi_key_opclass_supt(), mi_key_nkeys(),** and **mi_key_opclass(), mi_key_opclass_name()**
- "Supporting Multiple-Column Index Keys" on page 3-17

mi_key_opclass_strat()

The **mi_key_opclass_strat()** function retrieves the name of an operator-class strategy function.

Syntax

```
mi_string* mi_key_opclass_strat(MI_AM_KEY_DESC *keyDesc,
   mi_integer keyNum,
   mi_integer strategyNum)
```

keyDesc	points to the key descriptor.
keyNum	specifies the column number of a key in a composite-key index or 0 for a single-key index.
strategyNum	identifies the strategy function.

Usage

Each call to **mi_key_opclass_strat()** returns the name of one strategy function for one key column.

The *strategyNum* value for the first support function is 0. To determine the number of strategy functions that **mi_key_opclass_strat()** can return for a particular key column, call **mi_key_opclass_nstrat()**. To determine the maximum *keyNum* value, first call **mi_key_nkeys()**.

The **mi_key_opclass_strat()** returns strategy function names in the order that the CREATE OPCLASS statement names them.

To obtain the name of a strategy function in a WHERE clause, the access method can call the **mi_qual_funcname()** access function instead of **mi_key_opclass_strat()**.

Return Values

The string contains the strategy function name.

A NULL-valued pointer indicates that the function arguments contain an invalid value for either *keyNum* or *strategyNum*.

Related Topics

See the descriptions of:

- Functions **mi_key_opclass_nstrat(), mi_key_nkeys(), mi_key_opclass(), mi_key_opclass_name(),** and **mi_qual_funcname()**
- "Supporting Multiple-Column Index Keys" on page 3-17

mi_key_opclass_supt()

The **mi_key_opclass_supt()** function returns the name of an operator-class support function.

Syntax

```
mi_string* mi_key_opclass_supt(MI_AM_KEY_DESC *keyDesc,
   mi_integer keyNum,
   mi_integer supportNum)
```

keyDesc	points to the key descriptor.
keyNum	specifies the column number of a key in a composite-key index or 0 for a single-key index. For the first (or only) key, pass 0 as *keyNum.* Increment *keyNum* by 1 for each subsequent key in a composite index.
supportNum	identifies this support function.

Usage

Each call to **mi_key_opclass_supt()** returns the name of one support function for one key column.

The *supportNum* value for the first support function is `0`. To determine the number of support functions that **mi_key_opclass_supt()** can return for a particular key column, call **mi_key_opclass_nsupt()**. To determine the maximum *keyNum* value, first call **mi_key_nkeys()**. For an example of how to use these functions together, refer to Figure 3-7 on page 3-18.

The **mi_key_opclass_supt()** returns support function names in the order that the CREATE OPCLASS statement names them.

The access method can optionally use the support function name to get the function descriptor that the DataBlade API FastPath facility uses to execute the support function. For more information, refer to "Using FastPath" on page 3-19, particularly "Obtaining the Routine Identifier" on page 3-19.

Return Values

The string contains the support-function name.

A NULL-valued pointer indicates an invalid value for either the *keyNum* or *strategyNum* argument.

Related Topics

See the descriptions of:

- Functions **mi_key_opclass_nsupt(), mi_key_nkeys(),** and **mi_key_opclass(), mi_key_opclass_name()**
- "Supporting Multiple-Column Index Keys" on page 3-17

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.

Warning: The database server has no means to detect if a secondary access method does not set values for complex expressions.

If the access method has no code to evaluate AND or OR, the **am_scancost** purpose function can take the following precautions:

1. Call **mi_qual_boolop()**.
2. If **mi_qual_boolop()** indicates the presence of an AND or OR operator, return a negative value from **am_scancost** to ensure that the optimizer does not use the access method to process the query.

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 Topics

See the descriptions of:

- Function **mi_qual_issimple()**
- "Qualifying Data" on page 3-24

mi_qual_column()

The **mi_qual_column()** function identifies the key-column argument to a strategy 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 *IBM Informix DataBlade API Programmer's Guide*

mi_qual_commuteargs()

The **mi_qual_commuteargs()** function determines if the constant precedes the column in a strategy-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 strategy-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 strategy function, call **mi_qual_constant()** from the **am_beginscan** or **am_getnext** purpose function.

Strategy functions evaluate the contents of a column against some criteria, such as a supplied constant value.

If a strategy 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 strategy 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 the `?` 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 Table 3-2 on page 3-21
- MI_DATUM in the *IBM Informix DataBlade API Programmer's Guide*

mi_qual_constant_nohostvar()

The **mi_qual_constant_nohostvar**() function returns an explicit constant value, if any, from the strategy-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 Values

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 a NULL for each host variable.

If the function arguments do not explicitly specify a constant value, this function returns a NULL value.

Related Topics

See the descriptions of:

- Accessor functions **mi_qual_constisnull_nohostvar()** and **mi_qual_constant()**
- "Runtime Values as Arguments" on page 3-22
- MI_DATUM in the *IBM Informix DataBlade API Programmer's Guide*
- Host variables in the *IBM Informix DataBlade API Programmer's Guide*, *IBM Informix User-Defined Routines and Data Types Developer's Guide*, and the *IBM Informix ESQL/C Programmer's Manual*

mi_qual_constisnull()

The **mi_qual_constisnull()** function determines whether the arguments to a strategy 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 whether a strategy-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 3-22
- Host variables in the *IBM Informix DataBlade API Programmer’s Guide*, *IBM Informix User-Defined Routines and Data Types Developer’s Guide*, and the *IBM 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 strategy 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 3-22
- Host variables in the *IBM Informix DataBlade API Programmer's Guide*, *IBM Informix User-Defined Routines and Data Types Developer's Guide*, and *IBM 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 strategy 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 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 strategy 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 3-24
- "Using FastPath" on page 3-19

In the *IBM Informix DataBlade API Programmer's Guide*, 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 strategy 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 strategy 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 3-24.

Return Values

The return string contains the name of a simple function in the qualification.

mi_qual_handlenull()

The **mi_qual_handlenull()** function determines if the strategy 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 whether a qualification is a function. A function has one of the formats that Table 3-2 on page 3-21 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 strategy-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 3-25.

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 Topics

See the description of:

- Accessor function **mi_qual_boolop()**
- "Simple Functions" on page 3-21

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 3-22.

Return Values

MI_TRUE indicates that the strategy function involves an SLV argument.
MI_FALSE indicates that the strategy function does not specify an SLV argument.

Related Topics

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 strategy function should be negated. MI_FALSE indicates that the strategy function should not be negated.

Related Topics

See the description of "Negation" on page 3-23.

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 Topics

See the description of “Complex Boolean Expressions” on page 3-23.

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 non-zero 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 3-25.

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 3-22.

Return Values

None

Related Topics

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 index.

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_stratnum()

The **mi_qual_stratnum()** function locates a strategy function that a WHERE clause specifies in the list of strategy functions for the corresponding operator class.

Syntax

```
mi_integer  mi_qual_stratnum(MI_AM_QUAL_DESC *qualDesc)
```

qualDesc	points to the qualification descriptor.

Usage

The return value from **mi_qual_stratnum()** provides an offset to retrieve the strategy function name from the key descriptor. To obtain the strategy-function name, the access method can pass the return value from **mi_qual_stratnum()** to the **mi_key_opclass_strat()** function.

Tip: The access method can alternatively use the **mi_qual_funcname()** function to obtain the name of a particular strategy function that the WHERE clause specifies from the qualification descriptor.

Return Values

The return integer indicates the order in which the strategy function name occurs in the key descriptor. The **mi_qual_stratnum()** returns `0` for the first strategy function and 1 for the second strategy function name. For each subsequent strategy function, the return value increments by 1.

Related Topics

See the descriptions of functions **mi_key_opclass_strat()** and **mi_qual_funcname().**

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 Topics

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 3-30.

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 3-30
- Sample isolation-level language for access-method documentation (Figure 3-13 on page 3-34)

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 *IBM Informix Performance Guide*

mi_scan_nprojs()

The **mi_scan_nprojs()** function returns a value that is 1 less than the number of key columns.

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 of times to loop through the related **mi_scan_projs()** function.

Return Values

The integer return value indicates the number of key columns in an index entry.

Related Topics

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 key column.

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()**.

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 *IBM Informix DataBlade API Programmer's Guide*

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 of index keys 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 index and returns all index entries.

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 index entries.

Related Topics

See the description of the accessor functions in “Qualification Descriptor” on page 5-4.

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
*userdata);
```

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:

To retain user data in memory during the scan:

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 3-2

mi_scan_table()

The **mi_scan_table()** function retrieves a pointer to the table descriptor for the index 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 an index entry.

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 5-7
- Accessor functions for the row descriptor in the *IBM Informix DataBlade API Programmer's Guide*

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 index 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 entry in the index.

Return Values

This function returns a pointer to a user-data structure that the access method creates during the scan.

Related Topics

See the description of:

- Function **mi_scan_setuserdata()**
- "Storing Data in Shared Memory" on page 3-2

mi_tab_amparam()

The **mi_tab_amparam()** function retrieves any user-defined configuration values for the index.

Syntax

```
mi_string* mi_tab_amparam(MI_AM_TABLE_DESC *tableDesc);
```

tableDesc	points to the index 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.

To support multiple indexes on the same key column or composite of columns, use the configuration keywords as the example in "Enabling Alternative Indexes" on page 3-15 demonstrates.

To ensure that a CREATE INDEX statement does not duplicate the definition of another index, use the functions **mi_tab_param_exist()** and **mi_tab_nparam_exist()** as Figure 3-6 on page 3-17 shows.

Return Values

The pointer accesses a string that contains user-specified keywords and values. A NULL-valued pointer indicates that the CREATE INDEX statement specified no configuration keywords.

Related Topics

See the descriptions of:

- Functions **mi_tab_param_exist()** and **mi_tab_nparam_exist()**
- "Enabling Alternative Indexes" on page 3-15
- "Providing Configuration Keywords" on page 3-13
- The USING clause of the CREATE INDEX statement in the *IBM 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 index 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 indexes. 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 an 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 *IBM Informix DataBlade API Programmer's Guide*.

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**
- Accessor functions **mi_tab_check_is_recheck()** and **mi_tab_check_set_ask()**
- DataBlade API function **mi_db_error_raise()** in the *IBM Informix DataBlade API Programmer's Guide*, particularly the information about raising custom messages
- **oncheck** in the *IBM Informix Administrator's Reference*

mi_tab_check_is_recheck()

The **mi_tab_check_is_recheck()** function indicates whether the current execution of the **am_check** purpose function should repair a specific problem that the previous execution detected.

Syntax

```
mi_boolean mi_tab_check_is_recheck(MI_AM_TABLE_DESC *tableDesc)
```

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

Usage

Call this function in **am_check** purpose function to determine if the following sequence of events occurred:

1. A user issued an **oncheck** request but did not include **-y** or **-n** in the option arguments.
2. In response to an **oncheck** request, the database server invoked **the am_check** purpose function.
3. During the first execution of **am_check**, the purpose function detected a problem with the index, called **mi_tab_check_set_ask()** to alert the database server, and exited.
4. The database server prompted the user to indicate if the access method should repair the index.
5. The user answered `y` or `yes` to the prompt, and the database server executed **am_check** again for the same index with **-y** appended to the original options.

In addition to **mi_tab_check_is_recheck()**, the access method should do the following to support index repair during **oncheck**:

- Store a description of the problem in PER_STATEMENT memory and call **mi_tab_setuserdata()** to place a pointer to the PER_statement memory in the table descriptor.
- Contain the logic required to repair the index.
- If **mi_tab_check_is_recheck()** returns MI_True, execute the logic that repairs the index.

Return Values

MI_TRUE indicates that this execution of **am_check** is a recheck and should attempt to repair the index. MI_FALSE indicates that this is the first execution of **am_check** for a new **oncheck** request.

Related Topics

See the descriptions of:

- Purpose function **am_check**
- Accessor functions **mi_tab_check_msg()** and **mi_tab_check_set_ask()**

mi_tab_check_set_ask()

The **mi_tab_check_set_ask()** function sets a flag in the table descriptor to indicate that **am_check** detects a repairable problem in the index.

Syntax

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

tableDesc	points to the table descriptor of the index that the current **oncheck** command specifies.
option	contains an encoded version of the current command-line option string for the **oncheck** utility.

Usage

Call this function from the **am_check** purpose function to alert the database server of the following conditions:

- The access method detects a structural problem or data-integrity problem in an index.
- The access method contains appropriate logic to repair the problem.
- The user does not specify **-y** or **-n** with an **oncheck** command.

A user includes a **-y** option to indicate that the **oncheck** utility should repair any index problems that it detects. To indicate that **oncheck** should report problems but not repair them, the user includes a **-n** option with **oncheck**.

The **am_check** purpose function can check for the **-y** option with the **MI_CHECK_YES_TO_ALL()** macro and for **-n** with **MI_CHECK_NO_TO_ALL().** If both **MI_CHECK_YES_TO_ALL()** and **MI_CHECK_NO_TO_ALL()** return MI_FALSE, the user did not specify a preference to repair or not repair problems. Because it does not know how to proceed, **am_check** can call accessor function **mi_tab_check_set_ask()**, which causes the database server to ask if the user wants the index repaired.

Return Values

MI_OK validates the index structure as error free.

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

Related Topics

See the descriptions of:

- Purpose function **am_check**
- Accessor functions **mi_tab_check_msg()** and **mi_tab_check_is_recheck()**

mi_tab_createdate()

The **mi_tab_createdate()** function returns the date that the index was created.

Syntax

```
mi_date * mi_tab_createdate(MI_AM_TABLE_DESC *tableDesc);
```

tableDesc	points to the index descriptor.

Return Values

The date indicates when the CREATE INDEX statement was issued.

mi_tab_isindex()

The **mi_tab_isindex()** function indicates whether the table descriptor describes an index.

Syntax

```
mi_boolean mi_tab_isindex(MI_AM_TABLE_DESC *tableDesc)
```

tableDesc	points to the index descriptor.

Usage

If the access method shares source files with a primary access method, use this function to verify that the table descriptor pertains to the secondary access method.

Return Values

MI_TRUE verifies that the table descriptor actually describes an index. MI_FALSE indicates that it describes a 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 3-30
- Sample isolation-level language for access-method documentation (Figure 3-13 on page 3-34)

mi_tab_keydesc()

The **mi_tab_keydesc()** function returns a pointer to the key descriptor.

Syntax

```
MI_AM_KEY_DESC* mi_tab_keydesc(MI_AM_TABLE_DESC *tableDesc)
```

tableDesc	points to the index descriptor.

Usage

The **mi_tab_keydesc()** function describes the individual key columns in an index entry. After the access method obtains the pointer, it can pass it to the accessor functions that extract information from the key descriptor.

Return Values

The pointer enables the access method to locate the active key descriptor.

Related Topics

See the description of accessor functions in "Key Descriptor" on page 5-3.

mi_tab_mode()

The **mi_tab_mode()** function retrieves the I/O mode of the index from the table descriptor.

Syntax

```
mi_unsigned_integer
mi_tab_tab_mode(MI_AM_TABLE_DESC *tableDesc)
```

tableDesc	points to the index 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 Table 5-1 for interpretation.

   Each macro returns either MI_TRUE or MI_FALSE.

Table 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**.

The **am_open** purpose function can use the **MI_OUTPUT()** macro to verify that a CREATE INDEX statement is in progress. If **MI_OUTPUT()** returns MI_TRUE, the access method can call **mi_tab_setniorows()** to set the number of index entries for **am_insert** to process.

Return Values

The integer indicates whether an input or output request is active.

To interpret the returned integer, use the macros that Table 5-1 on page 5-68 describes.

Related Topics

See the descriptions of

- "Buffering Multiple Results" on page 3-28
- Purpose functions **am_beginscan** and **am_getnext**

- “Building New Indexes Efficiently” on page 3-14
- Purpose functions **am_open** and **am_insert**
- Setting logging preferences in Figure 3-4 on page 3-14

mi_tab_name()

The **mi_tab_name()** function retrieves the index 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 index descriptor.

Return Values

The string specifies the name of the index to access.

mi_tab_nextrow()

The **mi_tab_nextrow()** function fetches the next index entry from several that the database server stores in shared memory.

Syntax

```
mi_integer
mi_tab_nextrow(MI_AM_TABLE_DESC  *tableDesc,
   MI_ROW **row,
   mi_integer *rowid,
   mi_integer *fragid)
```

tableDesc	points to the index descriptor.
row	points to the address of a row structure. The row structure contains the index entry that the access method reformats, if necessary, and inserts into the virtual index.
rowid	points to the row identifier of the associated table row.
fragid	points to the fragment identifier of the associated table row.

Usage

Use this function from the **am_insert** purpose function if **am_insert** can insert more than one new index entry. The values in *row*, *rowid,* and *fragid* replace the new row and row-ID descriptor that the database server passes to **am_insert** if shared memory holds only one new index entry.

The **mi_tab_nextrow()** function works together with the following related accessor functions:

- The **mi_tab_setniorows()** function sets a number of rows to pass to **am_insert.**
- The **mi_tab_niorows()** function gets the number of rows to expect.

For an example of how these three functions work together, refer to Figure 3-5 on page 3-15.

Return Values

The return value increments for each call to **am_insert**. The first call to **mi_tab_nextrow()** returns `0`, the second returns `1`, and so forth. A negative return value indicates an error.

Related Topics

See the descriptions of:

- Purpose function **am_insert**
- Accessor functions **mi_tab_setniorows()** and **mi_tab_niorows()**
- "Building New Indexes Efficiently" on page 3-14

mi_tab_niorows()

The **mi_tab_niorows()** function retrieves the number of rows that the database server expects to process in **am_getnext** or **am_insert**.

Syntax

```
mi_integer
mi_tab_niorows(MI_AM_TABLE_DESC *tableDesc)
```

tableDesc	points to the index 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.

Call this function from **am_insert** and then use the return value to determine how many times to loop through shared memory to get the next row.

Return Values

The integer specifies the actual number of rows that the database server has placed in shared memory for **am_insert** to insert in a new index or 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_nextrow(), mi_tab_setniorows(),** and **mi_tab_setnextrow().**

mi_tab_nparam_exist()

The **mi_tab_nparam_exist()** function returns the number of virtual indexes that contain identical key columns.

Syntax

```
mi_integer mi_tab_nparam_exist(MI_AM_TABLE_DESC *tableDesc)
```

tableDesc	points to the index descriptor.

Usage

Call this function to determine how many alternative configuration-parameter entries the table descriptor contains. The return value is the array position of the last parameter entry in the table descriptor. Thus, this function returns `0` for the first and only parameter entry. If two parameter entries exist, this function returns `1`, and so forth. Use the return value from this function to extract parameter entries from the array with the **mi_tab_param_exist()** function.

Return Values

The integer indicates the number of configuration-parameter specifications, and therefore indexes, on identical columns. A value of `0` indicates one index on a group of columns. A value of *n* indicates the existence of `n+1` indexes.

Related Topics

See the descriptions of:

- Functions **mi_tab_param_exist()** and **mi_tab_amparam()**
- "Enabling Alternative Indexes" on page 3-15

mi_tab_numfrags()

The **mi_tab_numfrags()** function retrieves the number of fragments in the index.

Syntax

```
mi_integer mi_tab_numfrags(MI_AM_TABLE_DESC *tableDesc)
```

tableDesc	points to the index 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 index 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 Topics

See the description of the Owner Name segment in the *IBM Informix Guide to SQL: Syntax*.

mi_tab_param_exist()

The **mi_tab_param_exist()** function retrieves the index-configuration parameters that are available for one of multiple indexes that consist of the same key columns.

Syntax

```
mi_string * mi_tab_param_exists(MI_AM_TABLE_DESC *tableDescr,
   mi_integer n)
```

tableDesc	points to the index descriptor.
n	specifies a particular index from among multiple indexes on equivalent columns. The first CREATE INDEX statement for those columns creates index 0. To select that index, set *n* to 0. To select the second index created on the same columns, set *n* to 1.

Usage

To support multiple search schemes for the same set of columns, the VII enables the user to identify each search scheme with a set of keyword parameters. The user specifies these parameters in the CREATE INDEX statements for these indexes. The access method uses the related functions together to determine if CREATE INDEX statements specify new or duplicate keyword values.

For an example, refer to "Enabling Alternative Indexes" on page 3-15.

Return Values

The string lists keywords and their values from the **amparam** column of the **sysindexes** system catalog table for index *n*.

Related Topics

See the descriptions of functions **mi_tab_nparam_exist()** and **mi_tab_amparam().**

mi_tab_partnum()

The **mi_tab_partnum()** function retrieves the fragment identifier for the index.

Syntax

```
mi_integer mi_tab_partnum(MI_AM_TABLE_DESC *tableDesc)
```

tableDesc	points to the index descriptor.

Usage

If a CREATE INDEX 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.

For a fragmented index, 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 index that the table descriptor identifies.

Syntax

```
MI_ROW_DESC* mi_tab_rowdesc(MI_AM_TABLE_DESC *tableDesc)
```

tableDesc	points to the index 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 that make up the index.

The order of the columns in the row descriptor corresponds to the order of the columns in the CREATE INDEX 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 *IBM Informix DataBlade API Programmer's Guide* 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 index descriptor.
row	points to the address of a row structure that contains fetched data under the following conditions: • The query projects only index-key columns. • The **am_keyscan** purpose flag is set. Otherwise, *row* might not exist.
rowid	points to the row identifier of the table row that contains the key values.
fragid	points to the fragment identifier of the associated table row.

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*, *rowid*, *fragid* replace arguments that the database server passes to **am_getnext** if shared memory accommodates only one fetched index entry.

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 3-29.

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 3-28

mi_tab_setniorows()

The **mi_tab_setniorows()** function indicates:

- 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 index descriptor.
nrows	specifies the maximum number of rows that **am_getnext** or **am_insert** 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.

A secondary access method can set up a multiple-row area in shared memory for use in one or both of the following purpose functions:

- The database server can place multiple entries in shared memory that the **am_insert** purpose function retrieves and writes to disk.
- The **am_getnext** purpose function can fetch multiple rows into shared memory in response to a query.

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(), mi_tab_nextrow(),** 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 *userdata)
```

tableDesc	points to the index 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.

   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 3-3 and "Persistent User Data" on page 3-3 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 3-2

mi_tab_spaceloc()

The **mi_tab_spaceloc()** function retrieves the location of the extspace in which the index resides.

Syntax

```
mi_string* mi_tab_spaceloc(MI_AM_TABLE_DESC *tableDesc)
```

tableDesc	points to the index descriptor.

Usage

A user, usually a database server administrator, can assign a short name to an extspace with the **onspaces** utility. When a user creates an index, the CREATE INDEX 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 INDEX idx_remote on TABLE remote...
   IN dc39
   USING access_method
```

Return Values

A string identifies the extspace.

If the index 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 index resides.

Syntax

```
mi_string* mi_tab_spacename(MI_AM_TABLE_DESC *tableDesc)
```

tableDesc	points to the index 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 an index, the CREATE INDEX 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 INDEX idx_remote on TABLE remote...
   IN dc39
   USING access_method
```

The statement that creates the index 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 INDEX idx_remote on TABLE remote...
   IN '/tmp'
   USING access_method
```

If the IN clause specifies multiple storage spaces, each makes up a fragment of the index 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 a new smart large object or virtual index. The database server assigns the default sbspace to a virtual index under the following circumstances:

- A CREATE INDEX 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 an index due to a previous SQL statement.

For more information, refer to "Creating a Default Storage Space" on page 3-10.

Return Values

A string identifies the sbspace or extspace that the CREATE INDEX statement associates with the index. A NULL-valued pointer indicates that the index 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 index resides.

Syntax

```
mi_char1 mi_tab_spacetype(MI_AM_TABLE_DESC *tableDesc)
```

tableDesc	points to the index descriptor.

Return Values

The letter `S` indicates that the index resides in an sbspace. The letter `X` indicates that the index resides in an extspace. The letter `D` indicates that the index resides in a dbspace and is reserved for IBM Informix use only.

Important: A user-defined access method cannot create indexes in dbspaces.

mi_tab_unique()

The **mi_tab_unique()** function determines if a CREATE INDEX statement specifies that the index contains only unique keys.

Syntax

```
mi_boolean mi_tab_unique(MI_AM_TABLE_DESC *tableDesc)
```

tableDesc	points to the index descriptor.

Usage

The access method can call this function from the **am_create** or **am_insert** purpose function. As the access method builds an index, it checks for unique key values if the **mi_tab_unique()** function returns MI_TRUE.

Return Values

MI_TRUE indicates that the secondary access method must enforce unique keys for this index. MI_FALSE indicates that the secondary access method should not enforce unique keys for this index.

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 index 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 4-26**
- UPDATE STATISTICS in the *IBM Informix Guide to SQL: Syntax* and the *IBM Informix 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 index 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 index 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 Topics

See the descriptions of:
- Function **mi_tab_setuserdata()**
- "Storing Data in Shared Memory" on page 3-2

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 SECONDARY ACCESS_METHOD
- DROP ACCESS_METHOD

For information about how to interpret the syntax diagrams in this chapter, refer to "Syntax Diagrams" on page x.

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

```
►►──ALTER──ACCESS_METHOD──access-method name───────────────────────────────►


    ┌─,───────────────────────────────┐
    ▼                                 │
►────┬───────────────────────────────┬┴──────────────────────────────────►◄
     │                     (1)       │
     ├─┬─ADD────┬─┤ Purpose Option ├─┤
     │ └─MODIFY─┘                    │
     └─DROP──purpose name────────────┘
```

Notes:

1 See "Purpose Options" on page 6-7

Element	Purpose	Restrictions	Syntax
access-method name	The access method to alter	A previous CREATE SECONDARY ACCESS_METHOD statement must register the access method in the database.	Database Object Name segment; see *IBM 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.	Table 6-1 on page 6-8

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 6-7.

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_READWRITEAM_UNIQUE,
DROP AM_CHECK,
MODIFY AM_SPTYPE = ' SX' );
```

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_readwriteam_unique** 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 *IBM Informix Database Design and Implementation Guide* or the GRANT statement in the *IBM Informix Guide to SQL: Syntax*

CREATE ACCESS_METHOD (+)

Use the CREATE SECONDARY ACCESS_METHOD statement to register a new secondary access method. When you register an access method, the database server places an entry in the **sysams** system catalog table.

Syntax

```
►►──CREATE──SECONDARY──ACCESS_METHOD──access-method name─────────────────────►


    ┌─,──────────────────────────┐
    ▼                        (1) │
►─(───┤ Purpose Option ├─────────┴──)──────────────────────────────────────►◄
```

Notes:

1 See “Purpose Options” on page 6-7

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 *IBM Informix Guide to SQL: Syntax*.

Usage

The CREATE SECONDARY 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 6-7.

You must have the DBA or Resource privilege to create an access method. For information about privileges, refer to the *IBM Informix Database Design and Implementation Guide* or the GRANT statement in the *IBM Informix Guide to SQL: Syntax*.

Sample Statements

The following statement creates a secondary access method named **T-tree** that resides in an sbspace. The **am_getnext** purpose function is assigned to a function name that already exists. The **T_tree** access method supports unique keys and clustering.

```
CREATE SECONDARY ACCESS_METHOD T_tree(
AM_GETNEXT = ttree_getnext,
AM_UNIQUE,
AM_CLUSTER,
AM_SPTYPE = ' S ' );
```

Figure 6-2. Sample CREATE SECONDARY 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 *IBM Informix Database Design and Implementation Guide* or the GRANT statement in the *IBM 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 *IBM Informix Guide to SQL: Syntax*.

Usage

The RESTRICT keyword is required. You cannot drop an access method if indexes 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 *IBM Informix Guide to SQL: Syntax*
- Privileges in the *IBM Informix Database Design and Implementation Guide* or the GRANT statement in the *IBM 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 SECONDARY ACCESS_METHOD and ALTER ACCESS_METHOD statements specify purpose attributes with the following syntax.

Syntax

Purpose Option:

```
├──┬─purpose function──=──function name──────────┬──────────────────────────────────────────┤
   ├─purpose value──=──┬─string value──┬─────────┤
   │                   └─numeric value─┘         │
   └─purpose flag────────────────────────────────┘
```

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 Table 6-1 on page 6-8.
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 Table 6-1 on page 6-8.
purpose flag	A keyword that indicates which feature a flag enables	The interface specifies flag names.	Flag purpose category; see Table 6-1 on page 6-8.
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 *IBM Informix Guide to SQL: Syntax*.
string value	An indicator that is expressed as one or more characters	None	Quoted String segment; see *IBM Informix Guide to SQL: Syntax*.
numeric value	A value that can be used in computations	None	A numeric literal

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 Table 1-1 on page 1-8 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 SECONDARY 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 6-4 sets the **am_getnext** purpose-function name to the UDR name **ttree_getnexttextfile_getnext**. This example creates a new access method.

   The example in Figure 6-1 on page 6-3 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

Table 6-1 describes the possible settings for the **sysams** columns that contain purpose-function names, purpose flags, and purpose values. The items in Table 6-1 appear in the same order as the corresponding **sysams** columns.

Table 6-1. 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 index in a dbspace.	Value	`A`
am_defopclass	The name of the default operator class for this access method. Because the access method must exist before you can define an operator class for it, you set this purpose with the ALTER ACCESS_METHOD statement.	Value	None

Table 6-1. Purpose Functions, Purpose Flags, and Purpose Values (continued)

Purpose-Name Keyword	Explanation	Purpose category	Default Setting
am_keyscan	A flag that, if set indicates that **am_getnext** returns rows of index keys If query selects only the columns in the index key, the database server uses the row of index keys that the secondary access method puts in shared memory, without reading the table.	Flag	Not set
am_unique	A flag that you set if the secondary access method checks for unique keys	Flag	Not set
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
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 index to the database	Function	None
am_drop	The name of a user-defined function that drops a virtual index	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

Table 6-1. Purpose Functions, Purpose Flags, and Purpose Values (continued)

Purpose-Name Keyword	Explanation	Purpose category	Default Setting
am_insert	The name of a user-defined function that inserts an index entry	Function	None
am_delete	The name of a user-defined function that deletes an index entry	Function	None
am_update	The name of a user-defined function that changes the values in a rowkey	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
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 performs an integrity check on an index	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_getnext	The name of the required user-defined function that scans for the next item that satisfies the query	Function	None

The following rules apply to the purpose-option specifications in the CREATE SECONDARY ACCESS_METHOD and ALTER ACCESS_METHOD statements:

- To specify multiple purpose options in one statement, separate them with commas.
- The CREATE SECONDARY 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.
- You can specify the **am_defopclass** value only with the ALTER ACCESS_METHOD statement.

  You must first register an access method with the CREATE SECONDARY ACCESS_METHOD statement before you can assign a default operator class.

References

In this publication, see the following topics:

- "Managing Storage Spaces" on page 3-9
- "Executing in Parallel" on page 3-27
- "Registering Purpose Functions" on page 2-5 and "Registering the Access Method" on page 2-6

- “Specifying an Operator Class” on page 2-8
- “Enforcing Unique-Index Constraints” on page 3-30
- “Calculating Statement-Specific Costs” on page 3-25
- “Bypassing Table Scans” on page 3-28
- Chapter 4, “Purpose-Function Reference,” on page 4-1

In the *IBM 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.
- CREATE OPERATOR CLASS statement.

Chapter 7. Using XA-Compliant External Data Sources

The Dynamic Server Transaction Manager recognizes XA-compliant external data sources, which can participate in two-phase commit transactions. You can invoke support routines for each XA-compliant, external data source that participates in a distributed transaction at a particular transactional event, such as prepare, commit, or rollback. This interaction conforms to X/Open XA interface standards.

Creating a Virtual-Index Interface for XA Data Sources

You can create a virtual-index interface to provide data access mechanisms for external data from XA data sources. The interaction between Dynamic Server and external data sources is through a set of purpose routines, such as **xa_open( )**, **xa_start( )**, **xa_prepare( )**, **xa_rollback( )**, **xa_commit( )**, **xa_recover( )**, **xa_complete( )**, **xa_forget( )**, **xa_close( )**, and **xa_end( )** For more information these purpose functions, see the *IBM Informix DataBlade API Programmer's Guide*.

You can create and drop XA-compliant data source types and instances of XA-compliant data sources. After you create an external XA-compliant data source, transactions can register and unregister the data source using the **mi_xa_register_xadatasource( )** or **ax_reg( )** and **mi_xa_unregister_xadatasource( )** or **ax_unreg( )** functions. For information on creating and dropping XA-compliant data source types and instances of XA-compliant data sources and information on the functions that transactions use to register and unregister the data source, see the *IBM Informix DataBlade API Programmer's Guide* and the *IBM Informix DataBlade API Function Reference* .

The MQ DataBlade module is an example of a set of user-defined routines that provide data access mechanisms for external data from XA data sources and provides XA-support functions to provide transactional support for the interaction between Dynamic Server and IBM Websphere MQ. For more information, see the *IBM Informix Database Extensions User's Guide*.

Appendix. Accessibility

IBM strives to provide products with usable access for everyone, regardless of age or ability.

Accessibility features for IBM Informix Dynamic Server

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use information technology products successfully.

Accessibility Features

The following list includes the major accessibility features in IBM Informix Dynamic Server. These features support:

- Keyboard-only operation.
- Interfaces that are commonly used by screen readers.
- The attachment of alternative input and output devices.

Tip: The IBM Informix Dynamic Server Information Center and its related publications are accessibility-enabled for the IBM Home Page Reader. You can operate all features using the keyboard instead of the mouse.

Keyboard Navigation

This product uses standard Microsoft® Windows® navigation keys.

Related Accessibility Information

IBM is committed to making our documentation accessible to persons with disabilities. Our publications are available in HTML format so that they can be accessed with assistive technology such as screen reader software. The syntax diagrams in our publications are available in dotted decimal format. For more information about the dotted decimal format, go to "Dotted Decimal Syntax Diagrams."

You can view the publications for IBM Informix Dynamic Server in Adobe Portable Document Format (PDF) using the Adobe Acrobat Reader.

IBM and Accessibility

See the *IBM Accessibility Center* at http://www.ibm.com/able for more information about the commitment that IBM has to accessibility.

Dotted Decimal Syntax Diagrams

The syntax diagrams in our publications are available in dotted decimal format, which is an accessible format that is available only if you are using a screen reader.

In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), the elements can appear on the same line, because they can be considered as a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that your screen reader is set to read punctuation. All syntax elements that have the same dotted decimal number (for example, all syntax elements that have the number 3.1) are mutually exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, the word or symbol is preceded by the backslash (\) character. The * symbol can be used next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is read as 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol that provides information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, this identifies a reference that is defined elsewhere. The string following the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you should refer to a separate syntax fragment OP1.

The following words and symbols are used next to the dotted decimal numbers:

? Specifies an optional syntax element. A dotted decimal number followed by the ? symbol indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element (for example, 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that syntax elements NOTIFY and UPDATE are optional; that is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.

! Specifies a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicates that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the same dotted decimal number can specify a ! symbol. For example, if you hear the lines

2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In this example, if you include the FILE keyword but do not specify an option, default option KEEP is applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1! (KEEP), and 2.1.1 (DELETE), the default option KEEP only applies to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

*	Specifies a syntax element that can be repeated zero or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line 5.1* data-area, you know that you can include more than one data area or you can include none. If you hear the lines 3*, 3 HOST, and 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

Notes:

1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you could write HOST STATE, but you could not write HOST HOST.
3. The * symbol is equivalent to a loop-back line in a railroad syntax diagram.

+	Specifies a syntax element that must be included one or more times. A dotted decimal number followed by the + symbol indicates that this syntax element must be included one or more times. For example, if you hear the line 6.1+ data-area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. As for the * symbol, you can only repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loop-back line in a railroad syntax diagram.

Notices

Trademarks

Index

Printed in USA

SC23-9439-00

Spine information:

IBM Informix **Version 11.50** **IBM Informix Virtual-Index Interface Programmer's Guide**