

# **Virtual-Index Interface**

## **Programmer's Manual**

Version 9.2  
September 1999  
Part No. 000-6218

Published by Informix® Press

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

© 1999 Informix Corporation. All rights reserved. The following are trademarks of Informix Corporation or its affiliates, one or more of which may be registered in the United States or other jurisdictions:

Answers OnLine™; C-ISAM®; Client SDK™; DataBlade®; Data Director™; Decision Frontier™; Dynamic Scalable Architecture™; Dynamic Server™; Dynamic Server™, Developer Edition™; Dynamic Server™ with Advanced Decision Support Option™; Dynamic Server™ with Extended Parallel Option™; Dynamic Server™ with MetaCube®; Dynamic Server™ with Universal Data Option™; Dynamic Server™ with Web Integration Option™; Dynamic Server™, Workgroup Edition™; Dynamic Virtual Machine™; Enterprise Decision Server™; Formation™; Formation Architect™; Formation Flow Engine™; Gold Mine Data Access®; IIF.2000™; i.Reach™; i.Sell™; Illustra®; Informix®; Informix® 4GL; Informix® InquireSM; Informix® Internet Foundation.2000™; InformixLink®; Informix® Red Brick® Decision Server™; Informix Session Proxy™; Informix® Vista™; InfoShelf™; Interforum™; I-Spy™; Mediazation™; MetaCube®; NewEra™; ON-Bar™; OnLine Dynamic Server™; OnLine/Secure Dynamic Server™; OpenCase®, Orca™; PaVER™; Red Brick® and Design; Red Brick® Data Mine™; Red Brick® Mine Builder™; Red Brick® Decisionscape™; Red Brick® Ready™; Red Brick Systems®; Regency Support®, Rely on Red BrickSM; RISQL®, Solution DesignSM; STARindex™; STARjoin™; SuperView®; TARGETindex™; TARGETjoin™; The Data Warehouse Company®, The one with the smartest data wins.™; The world is being digitized. We're indexing it.SM; Universal Data Warehouse Blueprint™; Universal Database Components™; Universal Web Connect™; ViewPoint®, Visionary™; Web Integration Suite™. The Informix logo is registered with the United States Patent and Trademark Office. The DataBlade logo is registered with the United States Patent and Trademark Office.

Documentation Team: Diana Chase, Kathy Eckardt, Abby Knott

#### GOVERNMENT LICENSE RIGHTS

Software and documentation acquired by or for the US Government are provided with rights as follows:

(1) if for civilian agency use, with rights as restricted by vendor's standard license, as prescribed in FAR 12.212; (2) if for Dept. of Defense use, with rights as restricted by vendor's standard license, unless superseded by a negotiated vendor license, as prescribed in DFARS 227.7202. Any whole or partial reproduction of software or documentation marked with this legend must reproduce this legend.

# Table of Contents

## Introduction

In This Introduction . . . . .	3
About This Manual . . . . .	3
Types of Users . . . . .	3
Software Dependencies . . . . .	3
Assumptions About Your Locale. . . . .	4
Demonstration Databases . . . . .	4
New Features . . . . .	5
Extensibility Enhancements . . . . .	5
Performance Improvements . . . . .	5
Documentation Conventions . . . . .	6
Typographical Conventions . . . . .	6
Icon Conventions . . . . .	7
Syntax Conventions . . . . .	9
Command-Line Conventions . . . . .	12
Sample-Code Conventions. . . . .	13
Additional Documentation . . . . .	14
On-Line Manuals . . . . .	15
Printed Manuals . . . . .	15
Error Message Documentation . . . . .	15
Documentation Notes, Release Notes, Machine Notes . . . . .	16
Related Reading . . . . .	17
Compliance with Industry Standards . . . . .	18
Informix Welcomes Your Comments . . . . .	18

<b>Chapter 1</b>	<b>What Is a Virtual-Index Access Method?</b>	
	In This Chapter . . . . .	1-3
	What Is an Access Method? . . . . .	1-3
	Why Do You Provide Additional Access Methods? . . . . .	1-4
	Access to Storage Spaces . . . . .	1-5
	Seamless Use of SQL . . . . .	1-5
	What Components Define an Access Method? . . . . .	1-7
	What Does Informix Provide? . . . . .	1-7
	What Do You Provide? . . . . .	1-14
	How Does an Access Method Work? . . . . .	1-18
	How Does the Database Server Locate Purpose Functions? . . . .	1-18
	Which Purpose Functions Does the Database Server Use? . . . .	1-20
	What Other Functions Does a Purpose Function Call? . . . . .	1-21
	What Might a More Sophisticated Access Method Do? . . . . .	1-23
 <b>Chapter 2</b>	 <b>Developing an Access Method</b>	
	In This Chapter . . . . .	2-3
	Choosing Features . . . . .	2-4
	Starting and Ending Processing . . . . .	2-7
	Creating and Dropping Database Objects . . . . .	2-8
	Optimizing Queries . . . . .	2-8
	Inserting, Deleting, and Updating Data . . . . .	2-10
	Registering Purpose Functions . . . . .	2-10
	Registering the Access Method . . . . .	2-12
	Specifying an Operator Class . . . . .	2-14
	Writing or Choosing Strategy and Support Functions . . . . .	2-15
	Registering Strategy and Support Functions . . . . .	2-16
	Registering the Operator Class . . . . .	2-17
	Adding a Default Operator Class to the Access Method . . . . .	2-18
	Testing the Access Method . . . . .	2-18
	Creating and Specifying Storage Spaces . . . . .	2-19
	Inserting, Querying, and Updating Data . . . . .	2-23
	Checking Data Integrity . . . . .	2-24
	Dropping an Access Method . . . . .	2-24

## Chapter 3

## Design Decisions

In This Chapter . . . . .	3-3
Storing Data in Shared Memory . . . . .	3-4
Functions That Allocate and Free Memory . . . . .	3-4
Memory-Duration Options . . . . .	3-5
Persistent User Data . . . . .	3-6
Accessing Database and System Catalog Tables . . . . .	3-7
Handling the Unexpected . . . . .	3-8
Using Callback Functions . . . . .	3-8
Using Error Messages . . . . .	3-10
Supporting Data Definition Statements . . . . .	3-12
Interpreting the Table Descriptor . . . . .	3-12
Managing Storage Spaces . . . . .	3-12
Providing Configuration Keywords . . . . .	3-18
Building New Indexes Efficiently . . . . .	3-20
Enabling Alternative Indexes . . . . .	3-21
Supporting Multiple-Column Index Keys . . . . .	3-24
Using FastPath . . . . .	3-26
Obtaining the Routine Identifier . . . . .	3-27
Reusing the Function Descriptor . . . . .	3-28
Processing Queries . . . . .	3-28
Interpreting the Scan Descriptor . . . . .	3-29
Interpreting the Qualification Descriptor . . . . .	3-29
Enhancing Performance . . . . .	3-41
Executing in Parallel . . . . .	3-41
Bypassing Table Scans . . . . .	3-43
Buffering Multiple Results . . . . .	3-44
Supporting Data Retrieval, Manipulation, and Return . . . . .	3-46
Enforcing Unique-Index Constraints . . . . .	3-46
Checking Isolation Levels . . . . .	3-47
Converting to and from Row Format . . . . .	3-49
Determining Transaction Success or Failure . . . . .	3-50
Supplying Error Messages and a User Guide . . . . .	3-51
Avoiding Database Server Exceptions . . . . .	3-52
Avoiding Optimizer Restrictions . . . . .	3-54
Notifying the User About Access-Method Constraints . . . . .	3-55
Documenting Nonstandard Features . . . . .	3-56

## Chapter 4

### Purpose-Function Reference

In This Chapter . . . . .	4-3
Purpose-Function Flow . . . . .	4-3
CREATE Statement Interface. . . . .	4-4
DROP Statement Interface . . . . .	4-4
SELECT...WHERE Statement Interface . . . . .	4-5
INSERT, DELETE, and UPDATE Statement Interface . . . . .	4-6
ALTER FRAGMENT Statement Interface . . . . .	4-8
oncheck Utility Interface . . . . .	4-12
Purpose-Function Syntax . . . . .	4-13
am_beginscan . . . . .	4-14
am_check . . . . .	4-16
am_close. . . . .	4-20
am_create . . . . .	4-21
am_delete . . . . .	4-23
am_drop. . . . .	4-25
am_endscan . . . . .	4-26
am_getnext. . . . .	4-27
am_insert . . . . .	4-29
am_open . . . . .	4-31
am_rescan . . . . .	4-33
am_sancost . . . . .	4-34
am_stats. . . . .	4-38
am_update . . . . .	4-40

## Chapter 5

### Descriptor Function Reference

In This Chapter . . . . .	5-5
Descriptors . . . . .	5-6
Key Descriptor . . . . .	5-8
Qualification Descriptor . . . . .	5-9
Row Descriptor . . . . .	5-11
Row-ID Descriptor . . . . .	5-12
Scan Descriptor . . . . .	5-13
Statistics Descriptor . . . . .	5-15
Table Descriptor . . . . .	5-16
Include Files . . . . .	5-18

Accessor Functions . . . . .	5-19
mi_id_fragid() . . . . .	5-20
mi_id_rowid() . . . . .	5-21
mi_id_setfragid() . . . . .	5-22
mi_id_setrowid() . . . . .	5-23
mi_istats_setclust() . . . . .	5-24
mi_istats_set2lval() . . . . .	5-25
mi_istats_set2sval() . . . . .	5-26
mi_istats_setnlevels() . . . . .	5-27
mi_istats_setnleaves() . . . . .	5-28
mi_istats_setnunique() . . . . .	5-29
mi_key_funcid() . . . . .	5-30
mi_key_nkeys() . . . . .	5-32
mi_key_opclass() . . . . .	5-33
mi_key_opclass_nstrat() . . . . .	5-35
mi_key_opclass_nsupt() . . . . .	5-37
mi_key_opclass_strat() . . . . .	5-39
mi_key_opclass_supt() . . . . .	5-41
mi_qual_boolop() . . . . .	5-43
mi_qual_column() . . . . .	5-45
mi_qual_commuteargs() . . . . .	5-47
mi_qual_constant() . . . . .	5-48
mi_qual_constant_nohostvar() . . . . .	5-50
mi_qual_constisnull() . . . . .	5-52
mi_qual_constisnull_nohostvar() . . . . .	5-53
mi_qual_const_depends_hostvar() . . . . .	5-55
mi_qual_const_depends_outer() . . . . .	5-57
mi_qual_funcid() . . . . .	5-58
mi_qual_funcname() . . . . .	5-60
mi_qual_handlenull() . . . . .	5-61
mi_qual_issimple() . . . . .	5-62
mi_qual_needoutput() . . . . .	5-63
mi_qual_negate() . . . . .	5-64
mi_qual_nquals() . . . . .	5-65
mi_qual_qual() . . . . .	5-66
mi_qual_setoutput() . . . . .	5-67
mi_qual_setreopt() . . . . .	5-68
mi_qual_stratnum() . . . . .	5-69
mi_scan_forupdate() . . . . .	5-70
mi_scan_isolevel() . . . . .	5-71
mi_scan_locktype() . . . . .	5-73
mi_scan_nprojs() . . . . .	5-74
mi_scan_newquals() . . . . .	5-75

mi_scan_projs()	5-76
mi_scan_quals()	5-77
mi_scan_setuserdata()	5-78
mi_scan_table()	5-80
mi_scan_userdata()	5-81
mi_tab_amparam()	5-82
mi_tab_check_is_recheck()	5-84
mi_tab_check_msg()	5-86
mi_tab_check_set_ask()	5-89
mi_tab_createdate()	5-91
mi_tab_isindex()	5-92
mi_tab_isolevel()	5-93
mi_tab_keydesc()	5-94
mi_tab_mode()	5-95
mi_tab_name()	5-97
mi_tab_nextrow()	5-98
mi_tab_niorows()	5-100
mi_tab_nparam_exist()	5-101
mi_tab_numfrags()	5-102
mi_tab_owner()	5-103
mi_tab_param_exist()	5-104
mi_tab_partnum()	5-105
mi_tab_rowdesc()	5-106
mi_tab_setnextrow()	5-107
mi_tab_setniorows()	5-109
mi_tab_setuserdata()	5-111
mi_tab_spaceloc()	5-113
mi_tab_spacename()	5-114
mi_tab_spacetype()	5-116
mi_tab_unique()	5-117
mi_tab_update_stat_mode()	5-118
mi_tab_userdata()	5-119

## Chapter 6 SQL Statements for Access Methods

In This Chapter	6-3
ALTER ACCESS_METHOD	6-4
CREATE ACCESS_METHOD	6-6
DROP ACCESS_METHOD	6-8
Purpose Options	6-10

## Index



# Introduction

In This Introduction . . . . .	3
About This Manual. . . . .	3
Types of Users . . . . .	3
Software Dependencies . . . . .	3
Assumptions About Your Locale. . . . .	4
Demonstration Databases . . . . .	4
New Features. . . . .	5
Extensibility Enhancements . . . . .	5
Performance Improvements . . . . .	5
Documentation Conventions . . . . .	6
Typographical Conventions . . . . .	6
Icon Conventions . . . . .	7
Comment Icons . . . . .	7
Feature, Product, and Platform Icons . . . . .	8
Compliance Icons . . . . .	8
Syntax Conventions . . . . .	9
Elements That Can Appear on the Path . . . . .	9
How to Read a Syntax Diagram. . . . .	11
Command-Line Conventions . . . . .	12
How to Read a Command Line . . . . .	13
Sample-Code Conventions. . . . .	13
SQL Code Conventions . . . . .	13
Access-Method Code Conventions. . . . .	14

Additional Documentation . . . . .	14
On-Line Manuals . . . . .	15
Printed Manuals . . . . .	15
Error Message Documentation . . . . .	15
Documentation Notes, Release Notes, Machine Notes . . . . .	16
Related Reading . . . . .	17
Compliance with Industry Standards. . . . .	18
Informix Welcomes Your Comments . . . . .	18

## In This Introduction

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

---

## About This Manual

This manual explains how to use the Virtual-Index Interface (VII), typically in a DataBlade module, to create a secondary access method that extends the built-in indexing schemes of Informix Dynamic Server 2000.

## Types of Users

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

- Partners and third-party programmers who have index requirements that the B-tree and R-tree indexes do not accommodate
- Informix engineers who support Informix customers, partners, and third-party developers

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

## Software Dependencies

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

## Assumptions About Your Locale

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

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

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

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

## Demonstration Databases

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

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

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

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

---

## New Features

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

- Extensibility enhancements
- Performance improvements

### Extensibility Enhancements

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

- Ability to execute UDRs with the DataBlade API FastPath feature, including the new **mi\_funcdesc\_by\_typeid()** function
- Ability to provide virtual-index information to the **oncheck** utility
- Ability to repair indexes during **oncheck** execution
- Ability to determine the level of distribution statistics desired: high, low, or medium

### Performance Improvements

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

- Parallel processing of access-method functions
- Ability to scan multiple rows into memory
- Ability to fetch multiple index entries from memory while building an index

# Documentation Conventions

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

The following conventions are discussed:

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

## Typographical Conventions

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

Convention	Meaning
KEYWORD	All primary elements in a programming language statement (keywords) appear in uppercase letters in a serif font.
<i>italics</i> <b>italics</b> <i>italics</i>	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.
<b>boldface</b> <b>boldface</b>	Names of program entities (such as classes, events, and tables), environment variables, file and pathnames, and interface elements (such as icons, menu items, and buttons) appear in boldface.
monospace <i>monospace</i>	Information that the product displays and information that you enter appear in a monospace typeface.

(1 of 2)

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




(2 of 2)

## Icon Conventions

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




### Comment Icons

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

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

**Feature, Product, and Platform Icons**


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

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

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

**Compliance Icons**

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

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

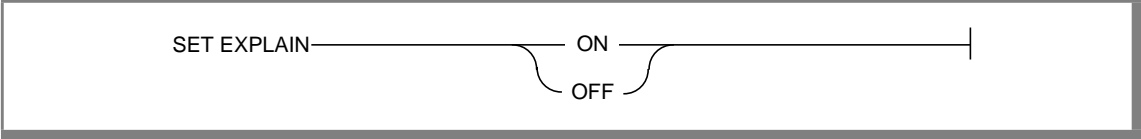
This icon can apply to an entire section or to one or more paragraphs within a section. If an icon appears next to a section heading, the information that applies to the indicated feature, product, or platform ends at the next heading at the same or higher level. A ♦ symbol indicates the end of feature-,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



## Syntax Conventions

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

**Figure 1**  
*Example of a Simple Syntax Diagram*



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



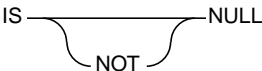
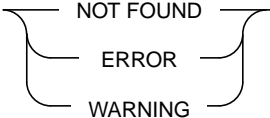
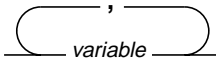
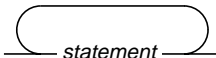
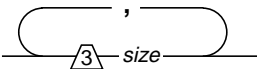

Syntax elements in a path represent terms, keywords, symbols, and segments that can appear in your statement. The path always approaches elements from the left and continues to the right, except in the case of separators in loops. For separators in loops, the path approaches counterclockwise. Unless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

### *Elements That Can Appear on the Path*

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

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

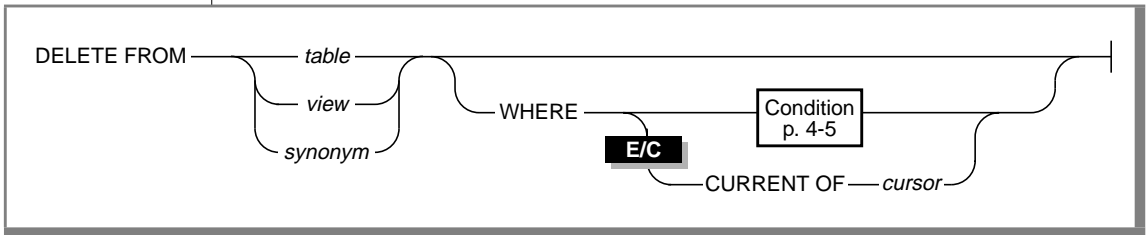
(1 of 2)

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

## How to Read a Syntax Diagram

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

**Figure 2**  
Example of a Syntax Diagram



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

Figure 2 illustrates the following steps:

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

## Command-Line Conventions

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

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

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

## How to Read a Command Line

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

```
setenv INFORMIXC pathname
```

The elements in the diagram are case sensitive.

The previous example illustrates the following steps:

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

## Sample-Code Conventions

Examples of the following code occur throughout this manual:

- SQL examples
- Access-method-module examples



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

### SQL Code Conventions

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

```
CONNECT TO stores_demo
...

DELETE FROM customer
      WHERE customer_num = 121
...

COMMIT WORK
DISCONNECT CURRENT
```



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

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

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

### **Access-Method Code Conventions**

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

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

---

## **Additional Documentation**

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

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

## On-Line Manuals

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

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

`www.informix.com/answers`

## Printed Manuals

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

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

## Error Message Documentation

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

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

Utility	Description
<b>finderr</b>	Displays error messages on line
<b>rofferr</b>	Formats error messages for printing

◆

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

UNIX

WIN NT

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

## Documentation Notes, Release Notes, Machine Notes

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

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

On-Line File	Purpose
<b>VIIDOC_9.2</b>	The documentation notes file for your version of this manual describes topics that are not covered in the manual or that were modified since publication.
<b>SERVERS_9.2</b>	The release notes file describes feature differences from earlier versions of Informix products and how these differences might affect current products. This file also contains information about any known problems and their workarounds.
<b>IDS_9.2</b>	The machine notes file describes any special actions that you must take to configure and use Informix products on your computer. Machine notes are named for the product described.



UNIX



## WIN NT

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

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

Machine notes do not apply to Windows environments. ♦

## Related Reading

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

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

---

## Compliance with Industry Standards

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

---

## Informix Welcomes Your Comments

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

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

Send electronic mail to us at the following address:

`doc@informix.com`

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

We appreciate your suggestions.

# What Is a Virtual-Index Access Method?

In This Chapter . . . . .	1-3
What Is an Access Method? . . . . .	1-3
Why Do You Provide Additional Access Methods? . . . . .	1-4
Access to Storage Spaces . . . . .	1-5
Seamless Use of SQL . . . . .	1-5
What Components Define an Access Method? . . . . .	1-7
What Does Informix Provide? . . . . .	1-7
Virtual-Index Interface . . . . .	1-7
DataBlade API. . . . .	1-12
SQL Extensions . . . . .	1-12
API Libraries . . . . .	1-13
What Do You Provide? . . . . .	1-14
Purpose Functions . . . . .	1-14
User-Defined Routines and Header Files. . . . .	1-16
Operator Class. . . . .	1-16
User Messages and Documentation . . . . .	1-17
How Does an Access Method Work? . . . . .	1-18
How Does the Database Server Locate Purpose Functions? . . . .	1-18
Which Purpose Functions Does the Database Server Use? . . . .	1-20
What Other Functions Does a Purpose Function Call? . . . . .	1-21
What Might a More Sophisticated Access Method Do? . . . . .	1-23





## In This Chapter

This chapter includes the following sections:

- “What Is an Access Method?”
- “Why Do You Provide Additional Access Methods?” on page 1-4
- “What Components Define an Access Method?” on page 1-7
- “How Does an Access Method Work?” on page 1-18

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

---

## What Is an Access Method?

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

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

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

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

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

- The built-in primary access method scans, retrieves, and alters rows in Informix relational tables.  
By default, tables that you create with the CREATE TABLE statement use the built-in primary access method.
- The built-in secondary access method is a generic B-tree index.  
By default, indexes that you create with the CREATE INDEX statement use this built-in secondary access method. For more information about the built-in B-tree index, refer to the [Informix Guide to SQL: Syntax](#).



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

---

## Why Do You Provide Additional Access Methods?

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

An access method can make any data appear to the end user as rows from an internal relational table 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 manual 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 sbospace  
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 Informix database server does not manage. For example, an extspace might refer to a path and filename that the operating system manages or another database that a different database manager controls.  
The database server does not provide transaction, backup, or recovery services for data that resides in an extspace.

For more information about how to choose the storage spaces that the user-defined access method will support, refer to [“Managing Storage Spaces” on page 3-12](#).

## Seamless Use of SQL

With the aid of a user-defined secondary access method, an SQL statement can use an index that:

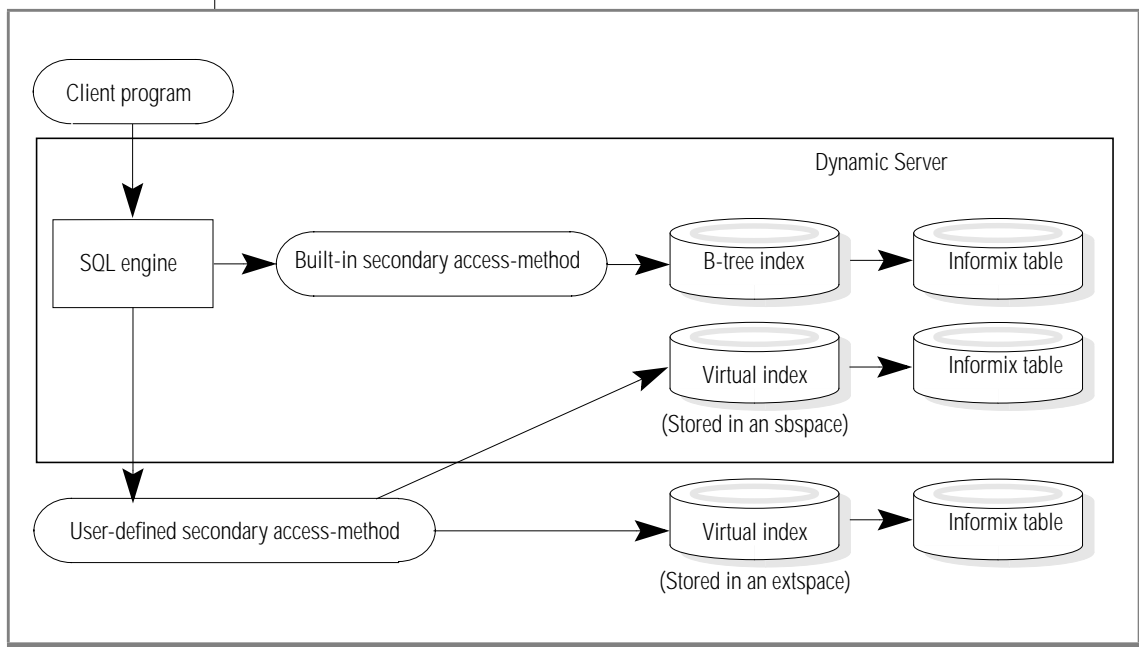
- employs user-defined operators to support user-defined types.
- accesses specific data inside a smart large object.
- belongs to an external database server or data manager.
- accesses nonrelational data.

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 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 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*





---

## What Components Define an Access Method?

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

## What Does Informix Provide?

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

- Virtual-Index Interface
- DataBlade API
- SQL extensions specific to the access method
- Additional 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 the 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 the value that VII provides to indicate success or failure.

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

Descriptors

*Descriptors* are predefined opaque data types that the database server creates to exchange information with a Datablade module or an access method. The 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 5-6](#).

Descriptor Name and Structure	Database Server Entries in the Descriptor
table descriptor MI_AM_TABLE_DESC	<p>The database server puts CREATE INDEX specifications in the table descriptor, including the following items:</p> <ul style="list-style-type: none"><li>■ Identification by index name, owner, storage space, and current fragment</li><li>■ Structural details, such as the number of fragments in the whole index, column names, and data types</li><li>■ Optional user-supplied parameters</li><li>■ Constraints such as read/write mode and unique keys</li></ul>
scan descriptor MI_AM_SCAN_DESC	<p>The database server puts SELECT statement specifications in the scan descriptor, including the following items:</p> <ul style="list-style-type: none"><li>■ Index-key columns</li><li>■ Lock type and isolation level</li><li>■ Pointers to the table descriptor and the qualification descriptor</li></ul>

(1 of 2)

Descriptor Name and Structure	Database Server Entries in the Descriptor
qualification descriptor MI_AM_QUAL_DESC	<p>In the qualification descriptor, the database server describes the functions and Boolean operators that a WHERE clause specifies.</p> <p>A qualification <i>function</i> tests the value in a column against a constant or value that an application supplies. The following examples test the value in the <code>price</code> column against the constant value 80:</p> <pre>WHERE lessthan(price,80) WHERE price &lt; 80</pre> <p>The qualification descriptor for a function identifies the following items:</p> <ul style="list-style-type: none"><li>■ Function name</li><li>■ Arguments that the WHERE clause passes to the function</li><li>■ Negation (NOT) operator, if any</li></ul> <p>A complex qualification combines the results of two previous qualifications with an AND or OR operation, as the following example shows:</p> <pre>WHERE price &lt; 80 AND cost &gt; 60</pre> <p>A complex qualification descriptor contains each Boolean AND or OR operator from the WHERE clause.</p> <p>For examples, refer to <a href="#">“Interpreting the Qualification Descriptor” on page 3-29</a>.</p>

(2 of 2)

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

Descriptor Name and Structure	Access Method Entries in the Descriptor
table descriptor MI_AM_TABLE_DESC	<p>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.</p> <p>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.</p>
scan descriptor MI_AM_SCAN_DESC	<p>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.</p> <p>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.</p>
qualification descriptor MI_AM_QUAL_DESC	<p>As it processes each qualification against a single index entry, the access method can set the following items in the qualification descriptor:</p> <ul style="list-style-type: none"><li>■ A host-variable value for a function with an OUT argument</li><li>■ The MI_VALUE_TRUE or MI_VALUE_FALSE to indicate the result that each function or Boolean operator returns</li><li>■ An indicator that forces the database server to reoptimize between scans for a join or subquery</li></ul>

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 5-19.

### *DataBlade API*

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

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

### *SQL Extensions*

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

### *Registering the Access Method in a Database*

The CREATE 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.”](#)

### *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 3-12](#) and [“Supporting Data Retrieval, Manipulation, and Return” on page 3-46](#).

## **API Libraries**

### **GLS**

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

For information about the complete set of APIs for Dynamic Server, refer to the [Getting Started](#) manual.

## What Do You Provide?

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

- Purpose functions
- Additional UDRs that the purpose functions call
- 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 manual uses a column name from the **sysams** system catalog table as the generic purpose-function name. For example, this manual refers to the UDR that builds a new 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**.*

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

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



**Figure 1-2**  
Purpose Functions

Generic Name	Description	Invoking Statement or Command
am_create	Creates a new virtual 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, <b>am_open</b> 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 <i>key</i>
am_delete	Deletes an existing entry from a virtual index	DELETE, ALTER FRAGMENT, UPDATE <i>key</i>
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

(1 of 2)

Generic Name	Description	Invoking Statement or Command
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	<b>oncheck</b> utility

(2 of 2)

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

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

### ***User-Defined Routines and Header Files***

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

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

### ***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 [Extending Informix Dynamic Server 2000](#).

## ***User Messages and Documentation***

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

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

For samples of user documentation that you must provide, refer to [“Supplying Error Messages and a User Guide” on page 3-51](#).

## How Does an Access Method Work?

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

## How Does the Database Server Locate Purpose Functions?

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

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

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

### UNIX

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



### WIN NT

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



The **get\_next\_record()** declaration has three generic pointer arguments to conform with the prototype of the **am\_getnext** purpose function. For a detailed explanation of the arguments and return value, refer to the description of **am\_getnext** on [page 4-27](#).

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

**Figure 1-3**  
*Partial sysprocedures Entry*

Column Name	Value
procname	get_next_record
owner	informix
procid	163
numargs	3
externalname	\$INFORMIXDIR/extend/am_lib.bld(get_next_record) (for UNIX)
langid	1 (Identifies C in the <b>syslanguages</b> 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 [Figure 1-4](#).

**Figure 1-4**  
*Partial sysams Entry*

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

## Which Purpose Functions Does the Database Server Use?

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 4-3](#).

The example in [Figure 1-4 on page 1-20](#) specifies only the **am\_getnext** purpose for the **super\_access** access method. A SELECT statement on a virtual index that uses **super\_access** initiates the following database server actions:

1. Get the function name for **am\_getnext** that the **super\_access** entry in **sysams** specifies; in this case **get\_next\_record()**.
2. Get the external file name of the executable from the **get\_next\_record()** entry in **sysprocedures** specifies.

The CREATE FUNCTION statement on [page 1-18](#) assigns the executable file as follows.

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

3. Allocate memory for the descriptors that the database server passes by reference through **get\_next\_record()** to the access method.
4. Execute the **am\_getnext** purpose function, **get\_next\_record()**.

## What Other Functions Does a Purpose Function Call?

A query might proceed as follows for the **super\_access** access method, which has only an **am\_getnext** purpose function:

1. The access method **am\_getnext** purpose function, **get\_next\_record()**, uses DataBlade API functions to the initiate callback functions for error handling.
2. The database server prepares a table descriptor to identify the 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.



## What Might a More Sophisticated Access Method Do?

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

The incomplete **super\_access** access method example does not create a virtual 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.

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

GLS



# Developing an Access Method

In This Chapter . . . . .	2-3
Choosing Features . . . . .	2-4
Starting and Ending Processing . . . . .	2-7
Creating and Dropping Database Objects . . . . .	2-8
Optimizing Queries . . . . .	2-8
Providing Optimizer Information . . . . .	2-9
Splitting a Scan . . . . .	2-9
Inserting, Deleting, and Updating Data . . . . .	2-10
Registering Purpose Functions . . . . .	2-10
Registering the Access Method . . . . .	2-12
Specifying an Operator Class . . . . .	2-14
Writing or Choosing Strategy and Support Functions . . . . .	2-15
Registering Strategy and Support Functions . . . . .	2-16
Making a Function Nonvariant . . . . .	2-16
Granting Privileges . . . . .	2-17
Registering the Operator Class . . . . .	2-17
Adding a Default Operator Class to the Access Method . . . . .	2-18
Testing the Access Method . . . . .	2-18
Creating and Specifying Storage Spaces . . . . .	2-19
Using Internal Storage . . . . .	2-19
Using External Storage . . . . .	2-20
Using Fragments . . . . .	2-22
Avoiding Storage-Space Errors . . . . .	2-23
Inserting, Querying, and Updating Data . . . . .	2-23
Checking Data Integrity . . . . .	2-24
Dropping an Access Method . . . . .	2-24



## 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.”](#)

## 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. To process most SQL statements, the database server attempts to invoke a sequence of task-specific purpose functions. You choose the tasks and SQL statements that the access method supports and then write the appropriate purpose functions for those tasks. For more information about the specific purpose functions that the database server executes for specific statements, refer to “[Purpose-Function Flow](#)” on page 4-3.

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

**Figure 2-1**  
*Statements and Their Purpose Functions*

Invoking Statement or Command	Purpose-Function Prototype
All	am_open(MI_AM_TABLE_DESC *)
If you do not supply <b>am_open</b> and <b>am_close</b> , open and close the data source in <b>am_getnext</b> .	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 *)
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 *)

(1 of 2)

Invoking Statement or Command	Purpose-Function Prototype
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 *)
<b>oncheck</b> utility	am_check( MI_AM_TABLE_DESC *, mi_integer *)

(2 of 2)



**Important:** Do not use the purpose label (**am\_open**, **am\_create**, **am\_getnext**) as the actual name of a user-defined purpose function. Do not use the prefix **vii\_**, as in **vii\_getnext**. Assign unique names, such as **image\_open**, **docfile\_open**, and **getnext\_record**. To prevent potential name-space collision, follow the instructions for registering and using an object prefix in the “DataBlade Developers Kit User’s Guide.”

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

As you write the purpose functions, adhere to the syntax provided for each in “[Purpose-Function Syntax](#)” on page 4-13.



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

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



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

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

## Starting and Ending Processing

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

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

## Creating and Dropping Database Objects

In response to a CREATE 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\_scost** 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 3-44](#) describes.
- Register purpose functions as parallelizable, as [“Executing in Parallel” on page 3-41](#) 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 possibly some data from **am\_getnext** to perform any subsequent scans for a join or subquery.

- **am\_endscan**

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

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

## Inserting, Deleting, and Updating Data

The database server calls an **am\_insert** purpose function in response to SQL statements such as INSERT or ALTER FRAGMENT. You might also provide **am\_delete** and **am\_update** purpose functions to handle the SQL statements DELETE and UPDATE, respectively.



**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 NT.

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

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

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

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

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

The **PARALLELIZABLE** routine modifier indicates that you have designed the function to execute safely in parallel. Parallel execution can dramatically speed the throughput of data. By itself, the routine modifier does not guarantee parallel processing. For more information about parallel execution of functions that belong to an access method, refer to [“Executing in Parallel” on page 3-41](#).



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

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



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

## Registering the Access Method

The CREATE FUNCTION statement identifies a function as part of a database, but not necessarily as part of an access method. To register the access method, issue the CREATE 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 2-2](#) 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 2-2**  
*Registering an  
Access Method*

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

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

**Figure 2-3**  
sysams Entry

The statement in Figure 2-2 does not name a purpose function for **am\_stats**, **am\_scancost**, or **am\_check**, or set the **am\_keyscan** or **am\_unique** flag, as the 0 values in Figure 2-3 indicate. The database server sets a 0 value for **am\_parallel** because none of the CREATE FUNCTION statements for the purpose functions included the PARALLELIZATION routine modifier.



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

For syntax and a list of available purpose settings, refer to [Chapter 6, “SQL Statements for Access Methods.”](#)

---

## 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
```

A different function compares values of data type **chord** than that which compares 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. Support functions might 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 might call the same support function to perform multiple tasks. For example, an access method might call a **between()** support function to both 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.*

If you do write strategy or support functions, to ensure that the database server recognizes them as *nonvariant*, complete the following requirements:

- Test that the function always returns identical results for identical arguments.
- In the CREATE FUNCTION statement, specify the NOT VARIANT routine modifier.

## 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 returns consistent results.
- 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 *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 to *f*.

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 potentially 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 *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") ::l1d_job,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 [Informix Guide to SQL: Syntax](#). For more information about environment variables, refer to the [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 3-26](#) and the description of accessor functions `mi_key_opclass_nsupt()` and `mi_key_opclass_supt()` in [Chapter 5, “Descriptor Function Reference.”](#)

## 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 6-4](#). For more information about operator classes, as well as strategy and support functions, refer to [Extending Informix Dynamic Server 2000](#).

---

## Testing the Access Method

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

1. Create one or more storage spaces.
2. Use the access method to create 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 system administrator who is responsible for the configuration of the database server performs steps 1 and 4. A database administrator performs step 2. Anyone with the appropriate SQL privileges to access or update the 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 sbpace holds smart large objects for the database server. This space is physically included in the database server configuration. Informix recommends 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 sbpace**

1. Create an sbpace with the **onspaces** utility.
2. Optionally, set the default sbpace for the database server.
3. Create a virtual index with the CREATE INDEX statement.

### ***Creating an Sbpace***

An sbpace 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.

### UNIX

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

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



### WIN NT

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



### *Specifying the Logical Sbspace Name*

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

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

If you do not intend to specify an sbpace explicitly in the CREATE INDEX statement, specify a default sbpace. To find out how to create a default dbspace, see [“Creating a Default Sbspace” on page 3-14](#).

The following example also creates a virtual index in the sbpace that **SBSpaceName** specifies:

```
CREATE INDEX ix1 ON tab1(coll)  
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:** Informix discourages the use of external storage for secondary access methods 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 extspace_name -l "location_specifier"
```

#### UNIX

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

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

The following example specifies a tape device:

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

◆

#### WIN NT/95

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

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

◆

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

```
CREATE INDEX ix1 ON tab1(coll)
  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 3-4 on page 3-15](#).

### *Using Fragments*

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

The following example shows the creation of 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 SBSPACE exists, and the access method does not create an extspace.
- The **am\_sptype** purpose value is set to 'S', and no default SBSPACE exists.
- The **am\_sptype** purpose value is set to 'X', and the access method does not create an extspace.

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

- The **am\_sptype** purpose value is set to 'D'.
- The IN clause with the CREATE 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 3-17](#). 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. Informix recommends that you document the isolation levels that the access method supports, as “[mi\\_scan\\_isolevel\(\)](#)” on [page 5-71](#) describes. For information about setting isolation levels, refer to the [Informix Guide to SQL: Syntax](#) and the [Informix Guide to SQL: Tutorial](#).

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

For information about SQL statements and keywords that your access method can support, refer to the [Informix Guide to SQL: Syntax](#). For information about the VII functions that determine which statements and keywords the user specifies, refer to [Chapter 5, “Descriptor Function Reference.”](#)

## Checking Data Integrity

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

For more information about how to implement the **oncheck** processing, refer to the description of **am\_check** on [page 4-16](#). For more information about how to specify options on the command line for **oncheck**, refer to the [Administrator's Reference](#).

---

## Dropping an Access Method

To drop an access method, execute the DROP ACCESS\_METHOD statement, as the following example shows:

```
DROP ACCESS_METHOD my_virtual RESTRICT
```

For more information, refer to “[DROP ACCESS\\_METHOD](#)” on [page 6-8](#).

# Design Decisions

In This Chapter . . . . .	3-3
Storing Data in Shared Memory . . . . .	3-4
Functions That Allocate and Free Memory . . . . .	3-4
Memory-Duration Options. . . . .	3-5
Persistent User Data . . . . .	3-6
Accessing Database and System Catalog Tables . . . . .	3-7
Handling the Unexpected . . . . .	3-8
Using Callback Functions . . . . .	3-8
Using Error Messages . . . . .	3-10
Supporting Data Definition Statements . . . . .	3-12
Interpreting the Table Descriptor. . . . .	3-12
Managing Storage Spaces . . . . .	3-12
Choosing DataBlade API Functions . . . . .	3-13
Setting the am_sptype Value . . . . .	3-13
Creating a Default Storage Space . . . . .	3-14
Ensuring Data Integrity . . . . .	3-15
Checking Storage-Space Type . . . . .	3-17
Supporting Fragmentation . . . . .	3-17
Providing Configuration Keywords. . . . .	3-18
Building New Indexes Efficiently . . . . .	3-20
Enabling Alternative Indexes . . . . .	3-21
Supporting Multiple-Column Index Keys . . . . .	3-24
Using FastPath . . . . .	3-26
Obtaining the Routine Identifier . . . . .	3-27
Reusing the Function Descriptor . . . . .	3-28

Processing Queries . . . . .	3-28
Interpreting the Scan Descriptor . . . . .	3-29
Interpreting the Qualification Descriptor . . . . .	3-29
Simple Functions . . . . .	3-30
Runtime Values as Arguments . . . . .	3-31
Negation . . . . .	3-33
Complex Boolean Expressions . . . . .	3-33
Multiple-Index Restrictions . . . . .	3-34
Qualifying Data . . . . .	3-35
Supporting Query-Plan Evaluation. . . . .	3-39
Enhancing Performance . . . . .	3-41
Executing in Parallel . . . . .	3-41
Bypassing Table Scans . . . . .	3-43
Buffering Multiple Results . . . . .	3-44
Supporting Data Retrieval, Manipulation, and Return . . . . .	3-46
Enforcing Unique-Index Constraints . . . . .	3-46
Checking Isolation Levels . . . . .	3-47
Converting to and from Row Format . . . . .	3-49
Determining Transaction Success or Failure . . . . .	3-50
Supplying Error Messages and a User Guide . . . . .	3-51
Avoiding Database Server Exceptions . . . . .	3-52
Statements That the Access Method Does Not Support . . . . .	3-52
Keywords That the Access Method Does Not Support . . . . .	3-52
Storage Spaces and Fragmentation . . . . .	3-53
Features That the Interface Does Not Support . . . . .	3-53
Avoiding Optimizer Restrictions . . . . .	3-54
Notifying the User About Access-Method Constraints . . . . .	3-55
Data Integrity Limitations . . . . .	3-55
WHERE Clause Limitations . . . . .	3-56
Documenting Nonstandard Features . . . . .	3-56

## In This Chapter

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

- [“Storing Data in Shared Memory,”](#) which follows this section
- [“Accessing Database and System Catalog Tables”](#) on page 3-7
- [“Handling the Unexpected”](#) on page 3-8
- [“Using FastPath”](#) on page 3-26

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

- [“Supporting Data Definition Statements”](#) on page 3-12
- [“Processing Queries”](#) on page 3-28
- [“Enhancing Performance”](#) on page 3-41
- [“Supporting Data Retrieval, Manipulation, and Return”](#) on page 3-46

The chapter ends with guidelines for helping end users and application developers to use the access method in [“Supplying Error Messages and a User Guide”](#) on page 3-51.

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

---

## Storing Data in Shared Memory

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

- Which function to call
- What duration to assign

## Functions That Allocate and Free Memory

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

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

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

For most purposes, UDRs, including access methods, can allocate shared memory with the public DataBlade API memory-management functions, **mi\_alloc()**, **mi\_dalloc()**, or **mi\_zalloc()**. UDRs share access to memory that a public function allocates with the pointer that the allocation function returns. For an example that allocates memory and stores a pointer, refer to [“Persistent User Data” on page 3-6](#). The public **mi\_free()** function frees the memory that a public function allocates.

The memory that you allocate with public functions is available only to UDRs that execute during a single-thread 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 deference the pointer to access the memory.

For detailed information about the following items, refer to the [DataBlade API Programmer's Manual](#):

- Functions that allocate public memory
- Duration keywords

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

## Persistent User Data

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

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

typedef struct my_row
{
    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*



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

Figure 3-2  
Storing and Retrieving User-Data Pointers

Descriptor	User-Data Duration	Stores Pointer to User Data	Retrieves Pointer to User Data
Table descriptor	PER STATEMENT	<a href="#">mi_tab_setuserdata()</a>	<a href="#">mi_tab_userdata()</a>
Scan descriptor	PER COMMAND	<a href="#">mi_scan_setuserdata()</a>	<a href="#">mi_scan_userdata()</a>

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

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

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

## Accessing Database and System Catalog Tables

Although the 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_exec_prepared_statement()`. SQL provides data directly from the system catalog tables and enables the access method to create tables to hold user data on the database server.

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

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

For more information on querying database tables, consult the [DataBlade API Programmer's Manual](#).



**Warning:** A parallelizable UDR must not call **`mi_exec()`**, **`mi_prepare()`**, **`mi_exec_prepared_statement()`**, or a UDR that calls these functions. A database server exception results if a parallelizable UDR calls any UDR that prepares or executes SQL. For more information about parallelizable access-method functions, refer to [“Executing in Parallel” on page 3-41](#).

## 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: <ul style="list-style-type: none"> <li>■ Warnings</li> <li>■ Runtime errors</li> </ul>
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 functions*. A callback function automatically executes when the database server indicates that the event of a particular type has occurred.

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

```

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

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

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

```

**Figure 3-3**  
Registering a  
Callback Function

The example in [Figure 3-3](#) accomplishes the following actions:

- Registers the **error\_callback()** function as a callback function to handle the MI\_Exception event
- Stores the callback handle that **mi\_register\_callback()** returns in **error\_cback** field of the **my\_data** memory

For more information about detecting if a transaction commits or rolls back, refer to [“Checking Isolation Levels” on page 3-47](#).

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

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

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

For a complete discussion of callback processing and the DataBlade API **mi\_register\_callback()** function, refer to the [DataBlade API Programmer's Manual](#). For code samples, follow the link to the DataBlade Corner on the Informix Developer Network Web page, [www.informix.com/idn](http://www.informix.com/idn).

## Using Error Messages

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

The following example shows how an access method might avoid an unexpected exit due to a user error that the database server cannot detect. The CREATE 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:

```
switch (mi_tab_amparam(tableDesc)
    case 'searchmode')
    ...
    case 'wildcard'
    ...
    default:
        mi_db_error_raise( connection, MI_EXCEPTION,
            "Invalid keyword in the USING clause.");
```

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

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

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

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

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

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

For more information, refer to the following sections:

- [“Checking Isolation Levels” on page 3-47](#)
- [“Notifying the User About Access-Method Constraints” on page 3-55](#)
- [“Accessor Functions” on page 5-19](#)

---

## 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 5-16](#).

### 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” on page 3-13](#)
- [“Creating a Default Storage Space” on page 3-14](#)
- [“Ensuring Data Integrity” on page 3-15](#)
- [“Checking Storage-Space Type” on page 3-17](#)
- [“Supporting Fragmentation” on page 3-17](#)

## Choosing DataBlade API Functions

The type of storage space determines whether you use **mi\_file\_\*()** functions or **mi\_lo\_\*()** functions to open, close, read from, and write to data.

To have the access method store data in an sbspace, use the smart-large-object interface of the DataBlade API. The names of most functions of the smart-large-object interface begin with the **mi\_lo\_** prefix. For example, you open a smart large object in an sbspace with **mi\_lo\_open()** or one of the smart-large-object creation functions: **mi\_lo\_copy()**, **mi\_lo\_create()**, **mi\_lo\_expand()**, or **mi\_lo\_from\_file()**.

If the access method stores data on devices that the operating system manages, use the DataBlade API file-access functions. Most file-access functions begin with the **mi\_file\_** prefix. For example, the **am\_open** purpose function might open a disk file with **mi\_file\_open()**.



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

For more information about smart-large-object functions and file-access functions, refer to the [DataBlade API Programmer's Manual](#).

## 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,”](#) 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, follow the link to the DataBlade Corner on the Informix Developer Network Web page, [www.informix.com/idn](http://www.informix.com/idn).



**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 the SBSPACENAME parameter of the ONCONFIG file.
3. Initialize the database server with the **oninit** utility.

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

1. From the command line, create the *sbspace* with logging turned on:  

```
onspaces -c -S vspace -p path -o offset -s size -Df "LOGGING=ON"
```
2. Edit the ONCONFIG file to insert the following line:  

```
SBSPACENAME vspace # Default sbspace name
```
3. Take the database server offline and then bring it online again to initialize memory with the updated configuration.  

```
onmode -ky  
oninit
```

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



*Creating a Default Extspace*

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

- Raise an error
- Specify an external storage space

The example in [Figure 3-4](#) specifies a directory path as the default extspace on a UNIX system.

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

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

**Figure 3-4**  
*Creating a Default  
Extspace*

*Ensuring Data Integrity*

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

- Locks
- Logging
- Backup and recovery
- Transaction management

### Activating Automatic Controls in Sbspaces

The following advantages apply to data that resides in sbspaces:

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

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

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

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



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

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



**Tip:** To save log space, call the DataBlade API **mi\_lo\_specset\_flags()** function at the start of the **am\_create** purpose function to temporarily turn off transaction logging. After the access method builds the new index, turn logging on with the same function. For more information about the **mi\_lo\_specset\_flags()** function and the settings that turn logging off and on, refer to the “[DataBlade API Programmer’s Manual](#).”

For more information about metadata logging and transaction logging, refer to the [Administrator’s Guide](#).

### *Adding Controls for Extspaces*

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

- Locks
- Logging and recovery
- Transaction commit and rollback management (described in [“Checking Isolation Levels” on page 3-47](#))

### *Checking Storage-Space Type*

The database server issues an error if the CREATE INDEX statement specifies 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 [mi\\_tab\\_spacetype\(\)](#) on [page 5-116](#).

For more information about errors that occur from inappropriate storage-space type, refer to [“Avoiding Storage-Space Errors” on page 2-23](#). For more information about documenting potential errors and intercepting error events, refer to [“Supplying Error Messages and a User Guide” on page 3-51](#).

### *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 2-22](#). For a detailed discussion about the benefits of and approaches to fragmentation, refer to the [Informix Guide to Database Design and Implementation](#).

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 the functions that [“Row-ID Descriptor” on page 5-12](#) 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 4-8.



***Tip:** For an ALTER FRAGMENT statement, the database server creates a scan descriptor but not a qualification descriptor. The **mi\_scan\_qual()** 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 the description of **mi\_scan\_qual()** on page 5-77.*

For information about the FRAGMENT BY clause, refer to the [Informix Guide to SQL: Syntax](#).

## Providing Configuration Keywords

You can provide configuration keywords that the access method interrogates to tailor its behavior. The user specifies one or more parameter choices in the USING clause of the CREATE 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 sbpace  
USING sbpace_access_method ("setbuffer=10")
```

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

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

**Figure 3-5**  
Checking a  
Configuration  
Parameter Value



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

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

```
CREATE 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 3-6** shows how the **am\_insert** purpose function writes multiple new index entries.

**Figure 3-6**  
*Processing Multiple 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_integer  nrows;
    mi_integer  rowid;
    mi_integer  fragid;

    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.) */
}
```

In [Figure 3-6](#), 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. Loop 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.”](#)

## 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 indexes **word** and **pattern** 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\_ainparam()** 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\_ainparam()**, so the access method alerts the user that it cannot create index **fuzzy**.



Figure 3-7 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 == other_param) == NULL))
        || ((index_param == other_param) == '\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 3-7**  
*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.”](#)

---

## 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.  
An operator class must exist for the data type of each key column.
- The access method determines the number of required strategy and support functions.
- The operator class provides the names of the required strategy and support functions for the data types that the access method handles.  
To bind an operator class to a single data type, specify explicit UDR signatures in the CREATE OPCLASS statement. To create a single an operator class for multiple data types, specify support function names without the parameters or return value in the CREATE OPCLASS statement.

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-column key.  
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 3-8**  
*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-column key.  
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 3-8 on page 3-25](#).

---

## 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. Obtain a routine identifier for the desired UDR.  
To find out how to obtain the routine identifier, refer to [“Obtaining the Routine Identifier”](#) on this page.
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.

For complete information about FastPath functions and the function descriptor (MI\_FUNC\_DESC), see the [DataBlade API Programmer's Manual](#).



**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()`**, as [Figure 3-15 on page 3-38](#) shows. 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 the 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, follow the link to the DataBlade Corner of the Informix Developer Network Web page, [www.informix.com/idn](http://www.informix.com/idn).



## 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 3-4](#).

---

## 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_qual()` function returns. A `NULL`-valued pointer indicates that the database server did not construct a qualification descriptor.



**Important:** If `mi_scan_qual()` returns a `NULL`-valued pointer, the access method must format and return all possible index keys.

For more information about the information that the scan descriptor provides, refer to [“Scan Descriptor” on page 5-13](#) and the scan-descriptor accessor functions that begin on [page 5-70](#).

## 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
<a href="#">mi_qual_nquals()</a>	Determines the number of simple functions and Boolean operators in a complex qualification
<a href="#">mi_qual_qual()</a>	Pointer to one qualification in a complex qualification descriptor or to the only qualification
<a href="#">mi_qual_issimple()</a> <a href="#">mi_qual_boolop()</a>	Determine which of the following qualifications the descriptor describes: <ul style="list-style-type: none"><li>■ A simple function</li><li>■ A complex AND or OR expression</li></ul>
<a href="#">mi_qual_funcid()</a> or <a href="#">mi_qual_funcname()</a>	Identify a simple function by function identifier or function name
<a href="#">mi_qual_column()</a>	Identifies the column argument of a function
<a href="#">mi_qual_constant()</a>	Extracts the value from the constant argument of a function
<a href="#">mi_qual_negate()</a>	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 5-9](#).

### Simple Functions

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

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

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



**Figure 3-9**  
Generic Function Prototypes

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

### ***Runtime Values as Arguments***

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

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

### ***Statement-Local Variables***

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

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

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

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

The 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 description of the [mi\\_qual\\_needoutput\(\)](#) function on [page 5-63](#) and the [mi\\_qual\\_setoutput\(\)](#) function on [page 5-67](#).

For more information about output parameters, the OUT keyword, and SLVs, refer to [Extending Informix Dynamic Server 2000](#).

### *Host Variables*

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

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

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

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

Because the value of a host variable applies to every 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.”

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

Topic	Manual
Setting values for host variables in client applications	<a href="#"><i>Informix ESQL/C Programmer's Manual</i></a>
Using DataBlade API functions from client applications	<a href="#"><i>DataBlade API Programmer's Manual</i></a>
Using host variables in SQL statements	<a href="#"><i>Informix Guide to SQL: Syntax</i></a>

## ***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 3-10** 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)
```

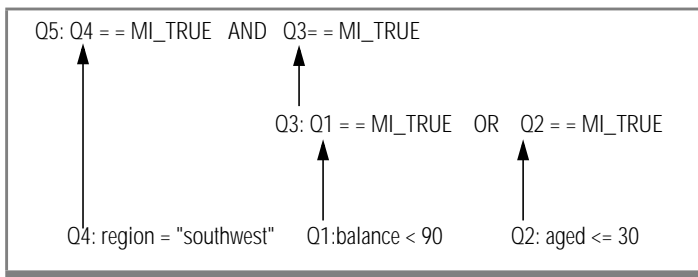
**Figure 3-10**  
Complex WHERE Clause

**Figure 3-11** and **Figure 3-12** represent the structure of the qualification descriptor that corresponds to the WHERE clause in **Figure 3-10**.

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

**Figure 3-11**  
Function Nesting

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



**Figure 3-12**  
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 5-9.

## Multiple-Index Restrictions

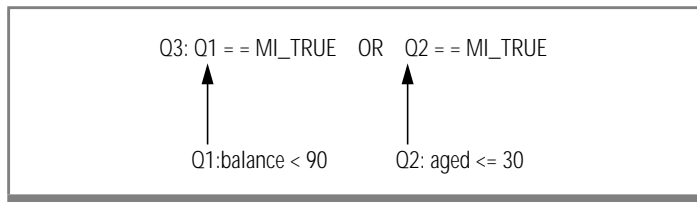
All the keys that a complex qualification descriptor tests belong to the same index. For example, **Figure 3-12** represents a qualification descriptor that could test three keys from the **receivables** index in the following example:

```
CREATE INDEX receivables ON customer(region, balance, aged) USING ledger_am
SELECT company, address, customer_id FROM customer
      WHERE region = "southwest" AND (balance < 90 OR aged <= 30)
```

If a qualification involves multiple indexes, the optimizer scans one index and ignores the others. To process the following query, the optimizer evaluates the query plan for both indexes and selects the more efficient index:

```
CREATE INDEX receivables ON customer(balance, aged) USING ledger_am
CREATE INDEX regionx ON customer(region) USING ledger_am
SELECT company, address, customer_id FROM customer
WHERE region = "southwest" AND (balance < 90 OR aged <= 30)
```

In this example, the **receivables** index contains **balance** and **aged**. If the optimizer invokes the access method to scan the **receivables** index, the qualification descriptor does not include the test for **region**.



**Figure 3-13**  
Qualification-  
Descriptor for a  
Two-Key Index

The access method returns row identifiers for any row that contains a qualifying value in either the **balance** or **aged** column. The optimizer retrieves those rows and then scans the reduced set for the value `southwest` in the **region** column.

### 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 3-26](#).



**Tip:** You can obtain the function descriptor in the **am\_beginscan** purpose function, store the function descriptor in the **PER\_COMMAND** user data, and call **mi\_scan\_setuserdata()** to store a pointer to the user data. In the **am\_getnext** purpose function, call **mi\_scan\_userdata()** to retrieve the pointer, access the function descriptor, and execute the function with **mi\_routine\_exec()**. For examples, follow the link to the DataBlade Corner on the Informix Developer Network Web page, [www.informix.com/idn](http://www.informix.com/idn).

### Using the Function Name

To extract the function name from the qualification descriptor, the access method calls the **mi\_qual\_funcname()** accessor function.

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

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

### Processing Complex Qualifications

In [Figure 3-14](#), 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.

**Figure 3-14**  
*Sample am\_getnext  
 Purpose Function*

```
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.*/
```

In [Figure 3-15](#), the `eval_qual()` function loops recursively through the qualification descriptor. After it executes all simple functions, it can apply any Boolean operators. Eventually, `eval_qual()` returns `MI_FALSE` OR `MI_TRUE` to indicate whether the current index key satisfies all the conditions in the qualification descriptor.

**Figure 3-15**  
*Evaluating  
Qualifications*

```
static mi_boolean  eval_qual(sd, qd, my_data)
MI_AM_SCAN_DESC  *sd;
MI_AM_QUAL_DESC  *qd;
my_data_t        *my_data;
{
    mi_integer      i;
    mi_boolean      return_val;
    MI_FUNC_DESC * fd;
    MI_CONNECTION *my_data->conn

    if (mi_qual_issimple(qd))
    { /* Qualification is strategy function.*/
        *Use FastPath to execution by the database server.
        *Use routineidentifier to get function descriptor. */
        fd = mi_func_desc_by_typeid(conn, mi_qual_funcid(qd));

        /* Get constant with mi_qual_constant(qd) */
        /* Get column_value with mi_qual_column(qd) */
        /* Get and adjust data types, as needed. */

        /* FastPath execution of the strategy function. */
        /* Return the mi_boolean result that the function returns. */
        return (mi_routine_exec(conn, fd, error, column_value, constant);

    } /* END: if (mi_qual_issimple(qd)) */

    else

    { /* Complex qualification (has AND or OR) */
        for (i = 0; i < mi_qual_nquals(qd); i++)
        {
            switch (mi_qual_boolop(qd))
            case MI_BOOLOP_OR
                if (eval_qual(mi_qual_qual(qd, i), my_data)== MI_TRUE)
                    return MI_TRUE; /* If one is true, then OR is true. */
            case MI_BOOLOP_AND
                if (eval_qual(mi_qual_qual(qd, i), my_data)== MI_FALSE)
                    return MI_FALSE; /*If one is false then AND is false.*/
        } /* END: for each mi_qual_nquals(qd) */
    } /* END: Complex qualification (has AND or OR) */
} /*End eval_qual() */
```



## Supporting Query-Plan Evaluation

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

### Calculating Statement-Specific Costs

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

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

For more information about query plans, refer to the [Performance Guide](#).

If the query involves a user-defined access method, the database server executes the **am\_scancost** purpose function to request cost information from the access method. For a description of the factors that **am\_scancost** calculates, refer to [page 4-34](#).

To avoid error messages, the access method can use the **am\_scancost** purpose function to notify the optimizer when it does not support all the requirements specified in a query. If necessary, **am\_scancost** can return a negative cost so that the optimizer excludes this access method from the query plan. For an example, refer to [Figure 4-13 on page 4-36](#).

### *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.”](#)

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 Informix database, refer to [“Accessing Database and System Catalog Tables” on page 3-7.](#)

---

## 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 [DataBlade API Programmer's Manual](#):

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

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

**To make an access method parallelizable**

1. Create a *basic set* of parallelizable purpose functions.  
The basic set, which enables a SELECT statement to execute in parallel, includes the following purpose functions: **am\_open**, **am\_close**, **am\_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)	<b>am_insert</b>
SELECT INTO TEMP	<b>am_insert</b>
DELETE	<b>am_delete</b>
UPDATE	<b>am_update</b>



**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 6-10](#).

## 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 data types (UDTs).

For more information about **am\_keyscan**, refer to [“Purpose Options” on page 6-10](#).

## 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:

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 3-16 shows the preceding steps. For more information about these functions, refer to [Chapter 5, “Descriptor Function Reference.”](#)

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

mi_integer sample_getnext(MI_AM_SCAN_DESC *sd, MI_ROW **retrow,
                          MI_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 3-16**  
*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 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 contains UDTs.

For information about **am\_keyscan**, refer to [“Bypassing Table Scans” on page 3-42.](#)

---

## Supporting Data Retrieval, Manipulation, and Return

The following topics 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 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 6-12](#).



## Checking Isolation Levels

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

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

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

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

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

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

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

To determine which of the following isolation levels the user or application specifies, the access method can call either the **mi\_tab\_isolevel()** or **mi\_scan\_isolevel()** function.

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

For more information about how applications use isolation levels, consult the *Informix Guide to SQL: Reference*, *Informix Guide to SQL: Syntax*, and *Informix Guide to SQL: Tutorial*. For information about determining isolation level, refer to **mi\_scan\_isolevel()** or **mi\_tab\_isolevel()** in Chapter 5, “Descriptor Function Reference.”

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

- The virtual 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 3-16. To find out how to provide for logging with ONCONFIG parameters, refer to your *Administrator’s Guide*.

The access method must provide the code to enforce isolation levels 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 3-19 on page 3-55](#).

## 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 [DataBlade API Programmer's Manual](#).
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 [DataBlade API Programmer's Manual](#).

## Determining Transaction Success or Failure

The access method can register an end-of-transaction callback function to handle the MI\_EVENT\_END\_XACT event, which the database server raises at the end of a transaction. In that callback function, test the return value of the DataBlade API **mi\_transition\_type()** function to determine the state of the transaction, as follows.

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



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

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

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

For detailed information about the following items, refer to the [DataBlade API Programmer's Manual](#):

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

For an example of a secondary access method that provides a state-transition callback function, follow the link to the DataBlade Corner on the Informix Developer Network Web page, [www.informix.com/idn](http://www.informix.com/idn).

---

## Supplying Error Messages and a User Guide

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

- The SQL statement syntax in the [Informix Guide to SQL: Syntax](#)
- The [Informix Guide to SQL: Tutorial](#) and the [Informix Guide to Database Design and Implementation](#)

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

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

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

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

## Avoiding Database Server Exceptions

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

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

Specify access-method support for the following items in the **sysams** system catalog table with a CREATE SECONDARY ACCESS\_METHOD or ALTER ACCESS\_METHOD statement:

- Statements
- Keywords
- Storage space type

### *Statements That the Access Method Does Not Support*

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

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

### *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 2-19](#).

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 2-22](#).

### ***Features That the Interface Does Not Support***

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

## Avoiding Optimizer Restrictions

The optimizer does not use an index if a WHERE clause qualification specifies keys from multiple indexes. The database server does not issue an error, but it also does not use the index, which potentially increases the time that the user waits for the query to complete.

The following examples show how to avoid qualifications that specify multiple indexes. Use them as a basis for the examples with which you can alert the user to rephrase the queries that separate the indexes.

The WHERE clause in each of the examples examines the values in two columns, **title** and **year**, to qualify a row of data. A **titlex** index references the **title** column and a **yearx** index references **year** column.

Figure 3-17 shows how to rephrase an OR expression that involves two indexes as UNION.

**Figure 3-17**  
*Replacing OR with UNION*

Results in Sequential Scan	Results in Index Scan
SELECT * FROM videos WHERE title = 'Hamlet' OR year > 1980;	SELECT * FROM videos WHERE title = 'Hamlet' UNION SELECT * FROM videos WHERE year > 1980;

Figure 3-18 shows how to use aliases in an AND query that combines two indexes.

**Figure 3-18**  
*Combining Aliases with AND*

Results in Sequential Scan	Results in Index Scan
SELECT * FROM videos WHERE title = 'Hamlet' AND year > 1980	SELECT v1.* FROM videos v1, videos v2 WHERE v1.rowid=v2.rowid AND v1.title = 'Hamlet' AND v2.year > 1980;



## 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 3-5 on page 3-19](#).

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

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

**Figure 3-19**  
*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.  
For an example that demonstrates the use of UNION, refer to [Figure 3-17 on page 3-54](#).
- In the **am\_scost** 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 4-13 on page 4-36](#).
- 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 3-21](#).
- Output from the **oncheck** utility  
For more information about the options that the **oncheck** provides, refer to the *Administrator’s Reference*. For more information about providing **oncheck** functionality, refer to the description of the **am\_check** purpose function on [page 4-16](#).

# Purpose-Function Reference

In This Chapter . . . . .	4-3
Purpose-Function Flow . . . . .	4-3
CREATE Statement Interface . . . . .	4-4
DROP Statement Interface . . . . .	4-4
SELECT...WHERE Statement Interface . . . . .	4-5
INSERT, DELETE, and UPDATE Statement Interface . . . . .	4-6
ALTER FRAGMENT Statement Interface . . . . .	4-8
oncheck Utility Interface . . . . .	4-12
Purpose-Function Syntax. . . . .	4-13
am_beginscan . . . . .	4-14
am_check . . . . .	4-16
am_close . . . . .	4-20
am_create . . . . .	4-21
am_delete . . . . .	4-23
am_drop . . . . .	4-25
am_endscan . . . . .	4-26
am_getnext . . . . .	4-27
am_insert . . . . .	4-29
am_open . . . . .	4-31
am_rescan . . . . .	4-33
am_scancost . . . . .	4-34
am_stats . . . . .	4-38
am_update . . . . .	4-40



## 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 4-13](#) specifies the predefined function-call syntax and suggests usage for each purpose function.

---

## Purpose-Function Flow

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

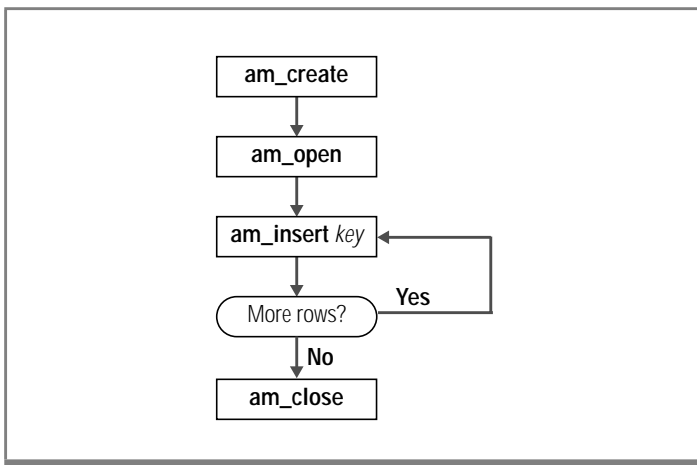
The complexity of the purpose-function flow for each statement determines the order in which the statement appears in this section. The following sections describe the purpose-function interface for SQL statements:

- [“CREATE Statement Interface” on page 4-4](#)
- [“DROP Statement Interface” on page 4-4](#)
- [“SELECT...WHERE Statement Interface” on page 4-5](#)
- [“INSERT, DELETE, and UPDATE Statement Interface” on page 4-6](#)
- [“ALTER FRAGMENT Statement Interface” on page 4-8](#)

Also see [“oncheck Utility Interface” on page 4-12](#).

## CREATE Statement Interface

[Figure 4-1](#) shows the order in which the database server executes purpose functions for a CREATE INDEX statement. If the IN clause specifies multiple storage spaces to fragment the index, the database server repeats the sequence of purpose functions that [Figure 4-1](#) shows for each storage space.

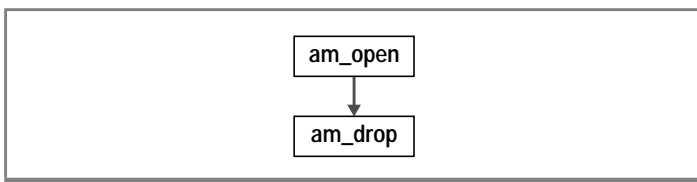


**Figure 4-1**  
Processing a  
CREATE INDEX  
Statement

For more information about implementing the CREATE INDEX statement in the access method, refer to [“Supporting Data Definition Statements” on page 3-12](#).

## DROP Statement Interface

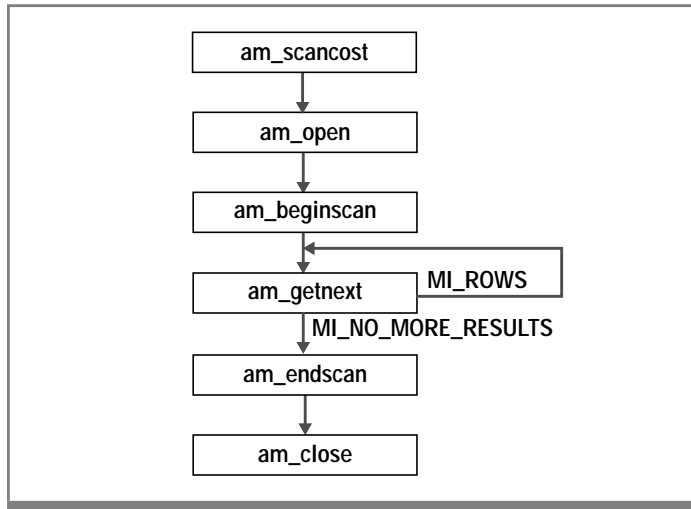
[Figure 4-2](#) shows the processing for each fragment of a DROP INDEX or DROP DATABASE statement.



**Figure 4-2**  
Processing a DROP  
Statement

## SELECT...WHERE Statement Interface

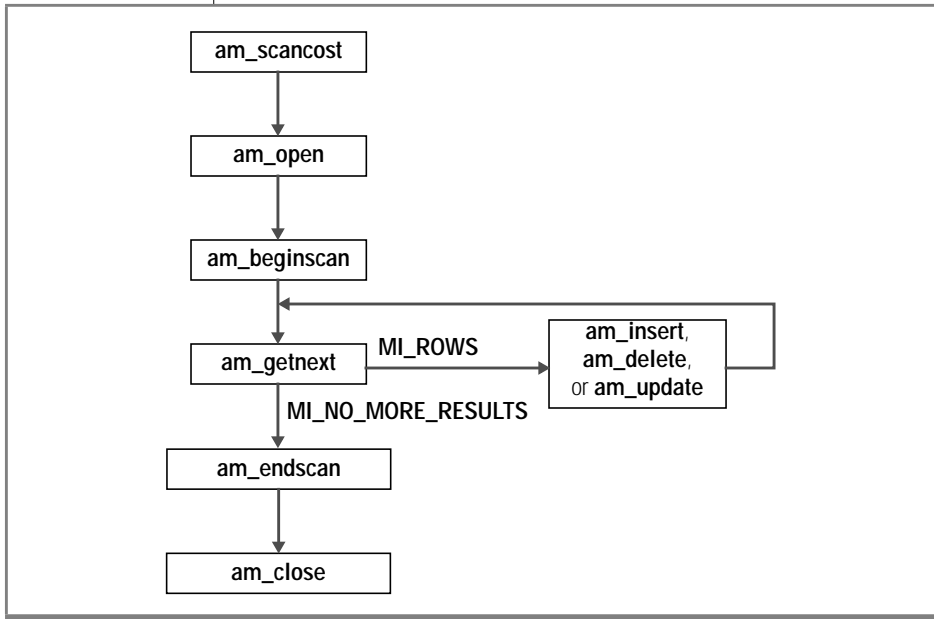
Figure 4-3 shows the order in which the database server executes purpose functions for a SELECT statement with a WHERE clause. For information about how to process the scan and qualifications, refer to “[Processing Queries](#)” on page 3-28.



**Figure 4-3**  
*Processing a  
SELECT Statement  
Scan*

## INSERT, DELETE, and UPDATE Statement Interface

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

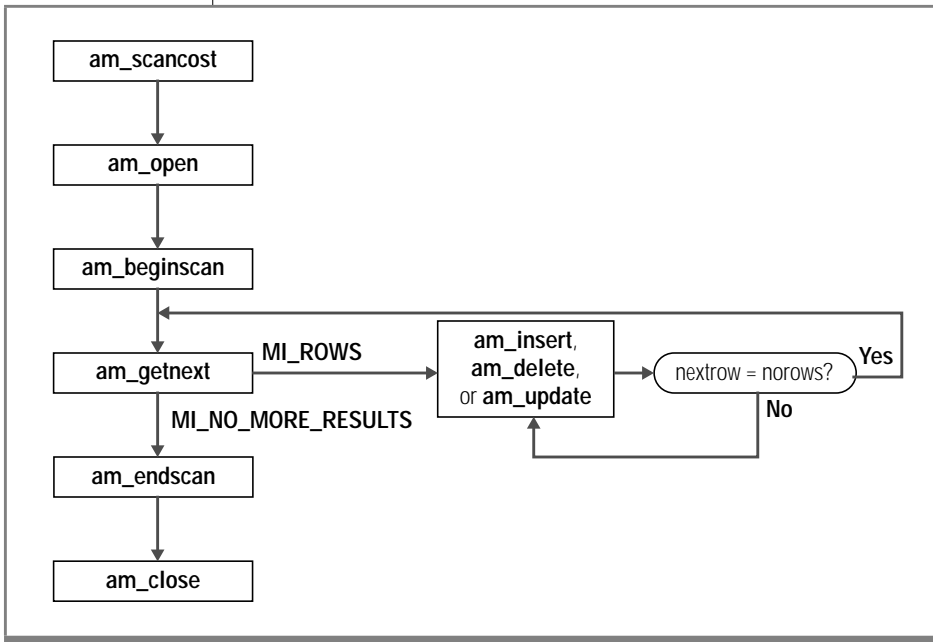


**Figure 4-4**  
*INSERT, DELETE,  
or UPDATE in a  
Subquery*



Figure 4-5 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 4-5**  
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 3-46](#).

## 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 4-6](#) creates and fragments a **jobsx** index.

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

**Figure 4-6**  
*SQL to Create the  
Fragmented Jobsx  
Index*

The statement in [Figure 4-7](#) changes the fragment expression for **jobsx**, which redistributes the index entries.

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

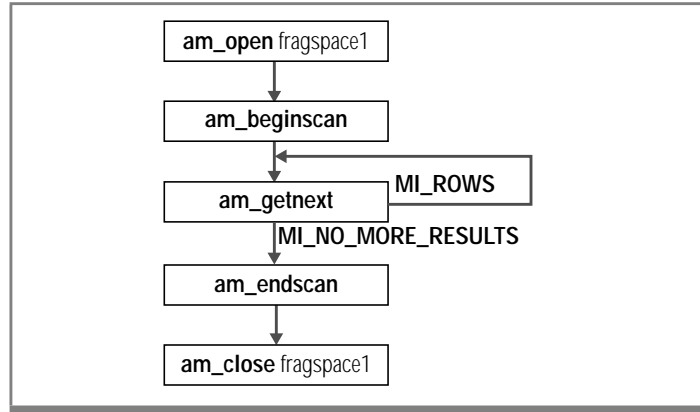
**Figure 4-7**  
*SQL to Alter the  
Jobsx Fragments*

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

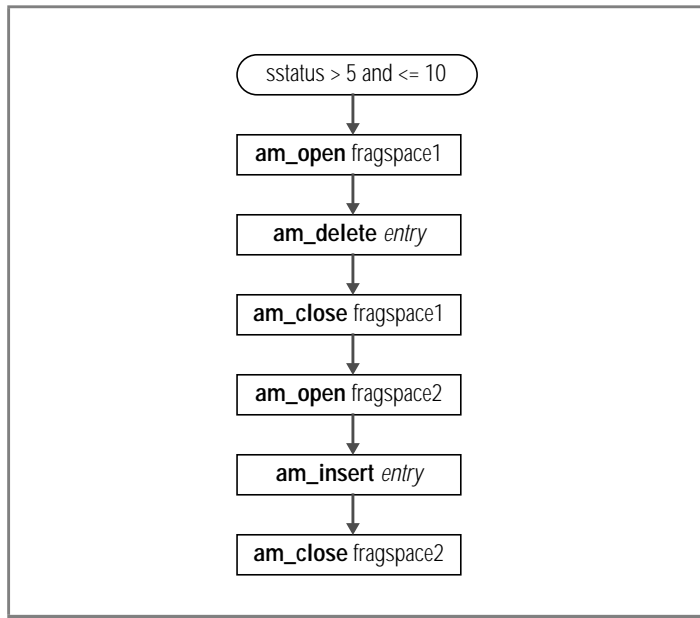
[Figures 4-8 through Figure 4-11](#) show the separate sequences of purpose functions that create the fragments and distribute the data for the SQL ALTER FRAGMENT statement in [Figure 4-7](#). The database server performs steps 1, 2, and 3 to move fragments from **fragspace1** to **fragspace2** and then performs steps 1 through 3 to move fragments from **fragspace2** to **fragspace3**.

Figure 4-8 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.



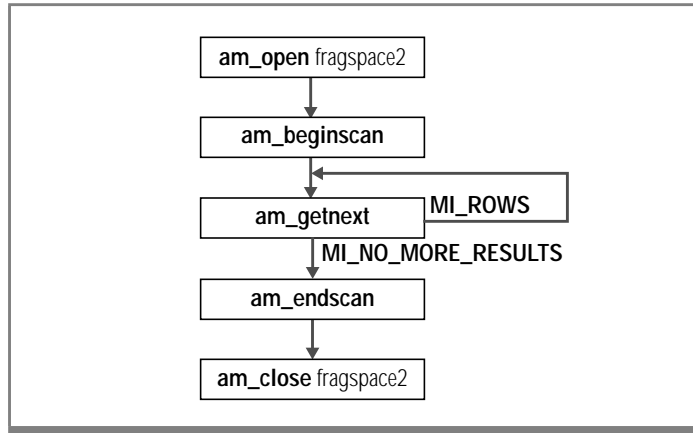
**Figure 4-8**  
Getting All the  
Entries in  
Fragment 1

In Figure 4-9, the database server returns the row identifiers that the access method should delete from **fragspace1** and insert in **fragspace2**.



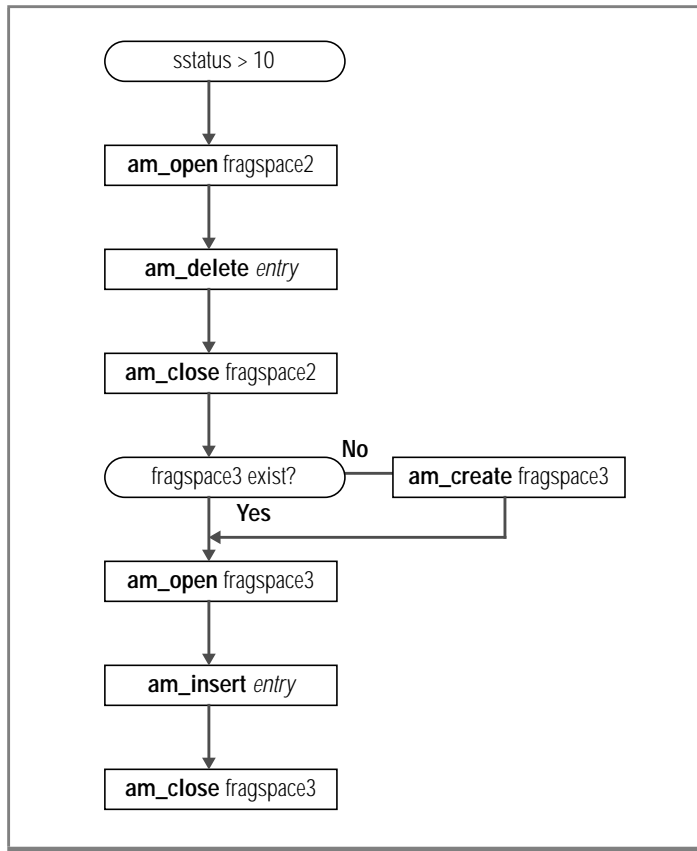
**Figure 4-9**  
Moving Entries  
Between Fragments

**Figure 4-10** again shows the sequential scan in step 1. This scan returns all the rows from **fragment2**.



**Figure 4-10**  
*Getting All the  
Entries in  
Fragment 2*

Figure 4-11 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**.



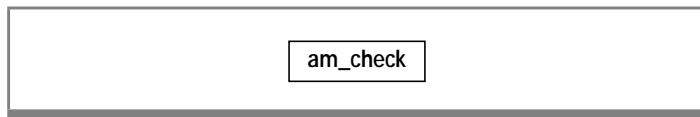
**Figure 4-11**  
Adding and Filling a  
Fragment

For more information about fragments that a VII-based access method manages, refer to [“Supporting Fragmentation” on page 3-17](#).

## oncheck Utility Interface

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

As [Figure 4-12](#) 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 4-12**  
*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 **vindex\_open()**. To associate the unique purpose-function names to the corresponding prototype names, use the CREATE SECONDARY ACCESS\_METHOD statement, as [“CREATE ACCESS\\_METHOD” on page 6-6](#) specifies.

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

Purpose functions are simply entry points from which the access method calls other routines from the access-method library, DataBlade API functions, and the VII functions that [“Accessor Functions” on page 5-19](#) 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:

- Obtain the qualification descriptor from the scan descriptor
- Parse the criteria in the qualification descriptor  
For a more detailed discussion, refer to [“Processing Queries” on page 3-28](#).
- Determine the need for data type conversions to process qualification expressions
- Call 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 3-35](#) and [“Using FastPath” on page 3-26](#).
- Based on the information in the qualification descriptor, initiate a search for data that fulfills the qualification
- Allocate PER\_COMMAND memory to build user data and then store the user data in the scan descriptor for the **am\_getnext** function  
For more information about memory allocation, refer to [“Storing Data in Shared Memory” on page 3-4](#).



You can also choose to defer any processing of qualifications until the **am\_getnext** function.

## Return Values

MI\_OK indicates success. MI\_ERROR indicates failure.

## Related Topics

See the descriptions of:

- purpose functions [am\\_endscan](#), [am\\_getnext](#), and [am\\_rescan](#).
- [“Optimizing Queries” on page 2-8.](#)

---

## **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 *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
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**:

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

**Checking and Displaying Index State**

The access method can call accessor function **mi\_tab\_spacetype()** to determine whether the specified index resides in an sbpace or extspace. If the data resides in an sbpace, the **am\_check** purpose function can duplicate the expected behavior of the **oncheck** utility. For information about the behavior for each **oncheck** option, refer to the *Administrator's Reference*.

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

To provide detailed information about the state of the 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.”

## 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 might:

- deallocate user-data memory that **am\_open** allocated with a PER\_STATEMENT duration.
- call **mi\_file\_close()**, **mi\_lo\_close()**, or one of the DataBlade API functions that copies smart-large-object data to a file.



**Important:** Do not call the DataBlade API **mi\_close()** function to free a database connection handle that you open (in the **am\_open** purpose function) with **mi\_open()**. Because the database connection has a PER\_COMMAND duration, not a duration, the database server frees the handle before it calls the **am\_close** purpose function.

### Return Values

MI\_OK indicates success. MI\_ERROR indicates failure.

### Related Topics

See the description of:

- purpose function **am\_open**.
- DataBlade API functions, such as **mi\_file\_close()** or **mi\_lo\_close()**, in the [DataBlade API Programmer's Manual](#).
- “Starting and Ending Processing” on page 2-7.

---

## am\_create

The database server calls **am\_create** to process a CREATE 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 **sysindices**. For example, a user might issue the CREATE INDEX statement 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 CREATE INDEX specifies.
- validates CREATE INDEX statements that specify identical keys, as described in [“Enabling Alternative Indexes” on page 3-21](#).
- calls the appropriate DataBlade API functions to create a smart large object or interact with the operating system for file creation, as described in [“Managing Storage Spaces” on page 3-12](#).
- 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 3-26](#).



**Important:** By default, transaction logging is disabled in sbspaces. To find out how to turn logging on, refer to [“Ensuring Data Integrity” on page 3-15](#).

## Return Values

MI\_OK indicates success. MI\_ERROR indicates failure.

## Related Topics

In this manual, see the description of:

- purpose function [am\\_drop](#).
- [“Creating and Dropping Database Objects” on page 2-8](#).

In the [DataBlade API Programmer's Manual](#), see the descriptions of:

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



## am\_delete

The database server calls **am\_delete** for:

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

## Syntax

```
mi_integer am_delete(MI_AM_TABLE_DESC *tableDesc,
                    MI_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 index 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 might need to identify and execute support functions to adjust the index structure after the delete. For more information, refer to [“Using FastPath” on page 3-26](#).



**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 3-51](#).

## Return Values

MI\_OK indicates success. MI\_ERROR indicates failure.

## Related Topics

See the descriptions of:

- purpose functions [am\\_insert](#) and [am\\_update](#).
- [“Inserting, Deleting, and Updating Data”](#) on page 2-10.

---

## am\_drop

The database server calls **am\_drop** for a DROP 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 Topic

See the descriptions of:

- purpose function [am\\_create](#).
- [“Creating and Dropping Database Objects” on page 2-8.](#)

---

## am\_endscan

The database server calls **am\_endscan** when **am\_getnext** finds no more rows.

### Syntax

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

*scanDesc*        points to the scan descriptor.

### Usage

The **am\_endscan** purpose function might:

- deallocate the PER\_COMMAND user-data memory that the **am\_beginscan** purpose function allocates and stores in the scan descriptor.  
For more information on the PER\_COMMAND memory and memory deallocation, refer to [“Storing Data in Shared Memory” on page 3-4](#).
- check for transaction commit or rollback.  
For more information about transaction processing, see [“Determining Transaction Success or Failure” on page 3-50](#).

### 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 2-8](#).



## 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 index entry, an **am\_getnext** purpose function might:

- read index data into user data.
- execute strategy functions in the qualification descriptor.
- 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 3-43](#) describes.

To find out how to create a row, refer to [“Converting to and from Row Format” on page 3-49](#).

The **am\_getnext** purpose function can loop to fill a shared-memory buffer with multiple index entries. For more information about buffering, see [“Buffering Multiple Results” on page 3-44](#) and the example of an **am\_getnext** loop in [“Buffering Multiple Results” on page 3-44](#).

The database server calls the **am\_getnext** purpose function until that function returns `MI_NO_MORE_RESULTS`. Then the database server calls the **am\_endscan** purpose function, if any, that the access method supplies.

If the access method does not provide an **am\_rescan** purpose function, **am\_getnext** stores interim data for subsequent scans in memory that persists between executions of the access method. For more information on memory duration, refer to [“Storing Data in Shared Memory” on page 3-4](#).

## Return Values

`MI_ROWS` indicates the return of a 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\\_qual\(\)](#), [mi\\_tab\\_niorows\(\)](#), and [mi\\_tab\\_setnextrow\(\)](#) in Chapter 5, “Descriptor Function Reference.”
- the **am\_keyscan** purpose flag in [“Purpose Options” on page 6-10](#).
- DataBlade API function [mi\\_row\\_create\(\)](#) in the [DataBlade API Programmer’s Manual](#).
- [“Executing Qualification Functions” on page 3-35](#) and [“Using FastPath” on page 3-26](#).
- [“Optimizing Queries” on page 2-8](#).

## 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)
```

<i>tableDesc</i>	points to the index descriptor.
<i>row</i>	points to a row structure in shared memory that contains the values for the access method to insert.
<i>ridDesc</i>	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 5-98](#).

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.

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 3-26](#). Call **mi\_tab\_userdata()** to retrieve the pointer to PER\_STATEMENT user data. Call **mi\_routine\_exec()** to execute the support function.

***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 3-51](#).*

## Return Values

MI\_OK indicates success. MI\_ERROR indicates failure.

## Related Topics

See the descriptions of:

- purpose functions [am\\_delete](#) and [am\\_update](#).
- [“Using FastPath” on page 3-26](#) and information about the DataBlade API FastPath facility in the [DataBlade API Programmer's Manual](#).
- [“Inserting, Deleting, and Updating Data” on page 2-10](#).



## 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** might:

- determine the reason, or mode, for the open, as described in [“mi\\_tab\\_mode\(\)” on page 5-95](#).
- allocate PER\_STATEMENT memory for a user-data structure as described in [“Persistent User Data” on page 3-6](#).
- open a database connection with the DataBlade API **mi\_open()** function.

To enable subsequent purpose functions to use the database, **am\_open** can copy the connection handle that **mi\_open()** returns into the user-data structure.

- register callback functions to handle exceptions, as described in [“Handling the Unexpected” on page 3-8](#).
- call the appropriate DataBlade API functions to obtain a file handle for an extspace or an LO handle for a smart large object.
- call **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 3-20](#).

## Return Value

MI\_OK indicates success. MI\_ERROR indicates failure.

## Related Topics

See the descriptions of:

- purpose function [am\\_close](#).
- memory allocation, callback functions, and the functions to open files or smart large objects in the [DataBlade API Programmer's Manual](#).
- [mi\\_tab\\_mode\(\)](#) and [mi\\_tab\\_setniorows\(\)](#) in Chapter 5, “Descriptor Function Reference.”
- “Starting and Ending Processing” on page 2-7.

## am\_rescan

The database server typically calls **am\_rescan** to process a join or subquery that requires multiple scans on the same 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.”
- “Optimizing Queries” on page 2-8.

---

## am\_scost

The query optimizer calls **am\_scost** during a SELECT statement, before it calls **am\_open**.

### Syntax

```
mi_real * am_scost(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 index key must satisfy to qualify for retrieval.

### Usage

The **am\_scost** purpose function estimates the cost to fetch and qualify data for the current query. The optimizer relies on the **am\_scost** 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\_scost** 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 disk accesses per row:

1. Fetch the entry from the index
2. Fetch the row 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\_scost**, set the **am\_costfactor** purpose value. The database server multiplies the cost that **am\_scost** returns by the value of **am\_costfactor**, which defaults to 1 if you do not set it. To find out how to set purpose values, refer to [Chapter 6, “SQL Statements for Access Methods.”](#)

### ***Forcing Reoptimization***

The optimizer might need a new scan cost for subsequent scans of the same index; for example, due to a join. To execute **am\_scost** before each rescan, call the **mi\_qual\_setreopt()** function. For a list of VII accessor functions that **am\_scost** can call to help evaluate cost and the need to reoptimize, refer to [“Related Topics” on page 4-37.](#)

### ***Returning a Negative Cost***

If the query specifies a feature that the access method does not support, return a value from **am\_scost** that forces the optimizer to pursue another path. In [Figure 4-13](#), an access method that does not process Boolean operators checks the qualification descriptor for Boolean operators and returns a negative value if it finds one.

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

***Figure 4-13***  
*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 that has no code to evaluate AND or OR, call accessor function **mi\_qual\_boollop()** or **mi\_qual\_issimple()** to determine if the qualification descriptor contains a Boolean operator.*

## Return Value

A pointer to an **mi\_real** data type that contains the cost value

## Related Topics

See the descriptions of:

- purpose function [am\\_stats](#).
- purpose flag **am\_scancost** in “Setting Purpose Functions, Flags, and Values” on page 6-12.
- accessor functions [mi\\_qual\\_const\\_depends\\_hostvar\(\)](#), [mi\\_qual\\_constisnull\\_nohostvar\(\)](#), [mi\\_qual\\_constant\\_nohostvar\(\)](#), [mi\\_qual\\_boollop\(\)](#), [mi\\_qual\\_issimple\(\)](#), and [mi\\_qual\\_setreopt\(\)](#) in Chapter 5, “Descriptor Function Reference.”

## am\_stats

The database server calls **am\_stats** to process an UPDATE STATISTICS statement.

### Syntax

```
mi_integer am_stats(MI_AM_TABLE_DESC *tableDesc,
MI_AM_ISTATS_DESC *istatsDesc)
```

*tableDesc*        points to the index descriptor.

*istatsDesc*       points to the statistics descriptor.

### Usage

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

Adjust the sampling rate in your version of the **am\_stats** purpose function according to the distribution-level keyword that the user specifies in the UPDATE STATISTICS statement. To determine which keyword, LOW, MEDIUM, or HIGH, an UPDATE STATISTICS statement specifies, call the **mi\_tab\_update\_stat\_mode()** function. For detailed information about the sampling rates that each keyword implies, refer to the description of UPDATE STATISTICS in the [Informix Guide to SQL: Syntax](#).

The **am\_stats** purpose function calls the various VII accessor functions that set values in statistics descriptor 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 sbospace.



## 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\\_istats\\_\\*](#) in [Chapter 5, “Descriptor Function Reference.”](#)
- the “Statistics Descriptor” on page 5-15.
- “Updating Statistics” on page 3-40.

## 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 index 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.

**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 3-51](#).



## Return Values

MI\_OK indicates success. MI\_ERROR indicates failure.

## Related Topics

See the descriptions of:

- purpose functions [am\\_delete](#) and [am\\_insert](#).
- [“Using FastPath” on page 3-26](#) and information about the DataBlade API FastPath facility in the *DataBlade API Programmer’s Manual*.
- [“Inserting, Deleting, and Updating Data” on page 2-10](#).



# Descriptor Function Reference

In This Chapter . . . . .	5-5
Descriptors . . . . .	5-6
Key Descriptor . . . . .	5-8
Qualification Descriptor . . . . .	5-9
Row Descriptor . . . . .	5-11
Row-ID Descriptor . . . . .	5-12
Scan Descriptor . . . . .	5-13
Statistics Descriptor . . . . .	5-15
Table Descriptor . . . . .	5-16
Include Files . . . . .	5-18
Accessor Functions . . . . .	5-19
mi_id_fragid() . . . . .	5-20
mi_id_rowid() . . . . .	5-21
mi_id_setfragid() . . . . .	5-22
mi_id_setrowid() . . . . .	5-23
mi_istats_setclust() . . . . .	5-24
mi_istats_set2lval() . . . . .	5-25
mi_istats_set2sval() . . . . .	5-26
mi_istats_setnlevels() . . . . .	5-27
mi_istats_setnleaves() . . . . .	5-28
mi_istats_setnunique() . . . . .	5-29
mi_key_funcid() . . . . .	5-30
mi_key_nkeys() . . . . .	5-32
mi_key_opclass() . . . . .	5-33
mi_key_opclass_nstrat() . . . . .	5-35
mi_key_opclass_nsupt() . . . . .	5-37
mi_key_opclass_strat() . . . . .	5-39

<code>mi_key_opclass_supt()</code> . . . . .	5-41
<code>mi_qual_boolop()</code> . . . . .	5-43
<code>mi_qual_column()</code> . . . . .	5-45
<code>mi_qual_commuteargs()</code> . . . . .	5-47
<code>mi_qual_constant()</code> . . . . .	5-48
<code>mi_qual_constant_nohostvar()</code> . . . . .	5-50
<code>mi_qual_constisnull()</code> . . . . .	5-52
<code>mi_qual_constisnull_nohostvar()</code> . . . . .	5-53
<code>mi_qual_const_depends_hostvar()</code> . . . . .	5-55
<code>mi_qual_const_depends_outer()</code> . . . . .	5-57
<code>mi_qual_funcid()</code> . . . . .	5-58
<code>mi_qual_funcname()</code> . . . . .	5-60
<code>mi_qual_handlenull()</code> . . . . .	5-61
<code>mi_qual_issimple()</code> . . . . .	5-62
<code>mi_qual_needoutput()</code> . . . . .	5-63
<code>mi_qual_negate()</code> . . . . .	5-64
<code>mi_qual_nquals()</code> . . . . .	5-65
<code>mi_qual_qual()</code> . . . . .	5-66
<code>mi_qual_setoutput()</code> . . . . .	5-67
<code>mi_qual_setreopt()</code> . . . . .	5-68
<code>mi_qual_stratnum()</code> . . . . .	5-69
<code>mi_scan_forupdate()</code> . . . . .	5-70
<code>mi_scan_isolevel()</code> . . . . .	5-71
<code>mi_scan_locktype()</code> . . . . .	5-73
<code>mi_scan_nprojs()</code> . . . . .	5-74
<code>mi_scan_newquals()</code> . . . . .	5-75
<code>mi_scan_projs()</code> . . . . .	5-76
<code>mi_scan_quals()</code> . . . . .	5-77
<code>mi_scan_setuserdata()</code> . . . . .	5-78
<code>mi_scan_table()</code> . . . . .	5-80
<code>mi_scan_userdata()</code> . . . . .	5-81
<code>mi_tab_amparam()</code> . . . . .	5-82
<code>mi_tab_check_is_recheck()</code> . . . . .	5-84
<code>mi_tab_check_msg()</code> . . . . .	5-86
<code>mi_tab_check_set_ask()</code> . . . . .	5-89
<code>mi_tab_createdate()</code> . . . . .	5-91

mi_tab_isindex()	5-92
mi_tab_isolevel()	5-93
mi_tab_keydesc()	5-94
mi_tab_mode()	5-95
mi_tab_name()	5-97
mi_tab_nextrow()	5-98
mi_tab_niorows()	5-100
mi_tab_nparam_exist()	5-101
mi_tab_numfrags()	5-102
mi_tab_owner()	5-103
mi_tab_param_exist()	5-104
mi_tab_partnum()	5-105
mi_tab_rowdesc()	5-106
mi_tab_setnextrow()	5-107
mi_tab_setniorows()	5-109
mi_tab_setuserdata()	5-111
mi_tab_spaceloc()	5-113
mi_tab_spacename()	5-114
mi_tab_spacetype()	5-116
mi_tab_unique()	5-117
mi_tab_update_stat_mode()	5-118
mi_tab_userdata()	5-119





## In This Chapter

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

- [“Descriptors” on page 5-6](#) describes the predefined data structures through which the database server and access method pass information.
- [“Include Files” on page 5-18](#) lists the header files with descriptor and function declarations that the access method must include.
- [“Accessor Functions” on page 5-19](#) lists every function that Informix 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 function, refer to [Chapter 4, “Purpose-Function Reference.”](#)

## Descriptors

The application programming interface (API) that Informix provides 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 Virtual Index Interface (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_	<a href="#">“Key Descriptor” on page 5-8</a>
Qualification descriptor (MI_AM_QUAL_DESC)	WHERE clause criteria	mi_qual_	<a href="#">“Qualification Descriptor” on page 5-9</a>
Row descriptor (MI_ROW)	Order and data types of projected columns	Various DataBlade API functions	<a href="#">DataBlade API Programmer’s Manual</a>
Row-id descriptor (MI_AM_ROWID_DESC)	Indexed table row location	mi_id_	<a href="#">“Row-ID Descriptor” on page 5-12</a>
Scan descriptor (MI_AM_SCAN_DESC)	SELECT clause projection	mi_scan_	<a href="#">“Scan Descriptor” on page 5-13</a>
Statistics descriptor (MI_AM_ISTATS_DESC)	Distribution of values	mi_istats_	<a href="#">“Statistics Descriptor” on page 5-15</a>
Table descriptor (MI_AM_TABLE_DESC)	Index location and attributes	mi_tab_	<a href="#">“Table Descriptor” on page 5-16</a>



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 5-19](#).

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

## 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
<a href="#"><code>mi_key_funcid()</code></a>	The routine identifier of the UDR that determines the value of a specified key in a functional index
<a href="#"><code>mi_key_nkeys()</code></a>	The number of columns in an index key
<a href="#"><code>mi_key_opclass()</code></a>	The identifier of the operator class for a specified column of the index key
<a href="#"><code>mi_key_opclass_nstrat()</code></a>	The number of operator-class strategy functions
<a href="#"><code>mi_key_opclass_strat()</code></a>	The name of one strategy function Typically, an access method calls the <a href="#"><code>mi_qual_funcid()</code></a> function to obtain the routine identifier and does not use <a href="#"><code>mi_key_opclass_strat()</code></a> .
<a href="#"><code>mi_key_opclass_nsupt()</code></a>	The number of support functions
<a href="#"><code>mi_key_opclass_supt()</code></a>	The name of one support function For an example of how to use the function names to execute the function, see <a href="#">“Obtaining the Routine Identifier” on page 3-27</a> .

## 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 3-28](#).

Use the VII [mi\\_scan\\_qual\(\)](#) 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
<a href="#">mi_qual_boolop()</a>	The operator type (AND or OR) of a qualification that is a complex expression
<a href="#">mi_qual_column()</a>	The position that the column argument to a strategy function occupies within an index entry
<a href="#">mi_qual_commuteargs()</a>	MI_TRUE if the argument list begins with a constant rather than a column value
<a href="#">mi_qual_const_depends_hostvar()</a>	MI_TRUE if a constant argument to a qualification function acquires a value at runtime from a host variable
<a href="#">mi_qual_const_depends_outer()</a>	MI_TRUE if the value of a particular constant argument can change each rescan
<a href="#">mi_qual_constant()</a>	The runtime value of the constant argument to a strategy function
<a href="#">mi_qual_constant_nohostvar()</a>	The value specified in the WHERE clause for the constant argument to a qualification function
<a href="#">mi_qual_constisnull()</a>	MI_TRUE if the value of a constant argument to a qualification function is NULL

(1 of 2)

Accessor Function	Return Value
<a href="#">mi_qual_constisnull_nohostvar()</a>	MI_TRUE if the WHERE clause specifies a NULL value as the constant argument to a qualification function
<a href="#">mi_qual_funcid()</a>	The routine identifier of a strategy function
<a href="#">mi_qual_funcname()</a>	The name of a strategy function
<a href="#">mi_qual_handlenull()</a>	MI_TRUE if the strategy function accepts NULL as an argument
<a href="#">mi_qual_issimple()</a>	MI_TRUE if the qualification contains one function rather than a complex expression
<a href="#">mi_qual_needoutput()</a>	MI_TRUE if the qualification function supplies an output parameter value Obtain and set a pointer to the output-parameter value with <a href="#">mi_qual_setoutput()</a> .
<a href="#">mi_qual_negate()</a>	MI_TRUE if the qualification includes the operator NOT
<a href="#">mi_qual_nquals()</a>	The number of nested qualifications in a complex expression, or 0 for a simple qualification that contains no Boolean operators
<a href="#">mi_qual_qual()</a>	Pointer to one qualification in a complex qualification descriptor or to the only qualification
<a href="#">mi_qual_stratnum()</a>	The ordinal number of the operator-class strategy function

(2 of 2)

The following accessor functions set values in the descriptor.

Accessor Function	Value Set
<a href="#">mi_qual_setoutput()</a>	A host-variable value
<a href="#">mi_qual_setreopt()</a>	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 [\*DataBlade API Programmer's Manual\*](#).

# 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
<a href="#">mi_id_setrowid()</a>	The row identifier
<a href="#">mi_id_setfragid()</a>	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
<a href="#">mi_id_rowid()</a>	The row identifier
	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 [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
<a href="#"><b>mi_scan_forupdate()</b></a>	MI_TRUE if a SELECT statement includes a FOR UPDATE clause
<a href="#"><b>mi_scan_isolevel()</b></a>	The isolation level for the index
<a href="#"><b>mi_scan_locktype()</b></a>	The lock type for the scan
<a href="#"><b>mi_scan_newquals()</b></a>	MI_TRUE if the qualification descriptor changes after the first scan for a join or subquery
<a href="#"><b>mi_scan_nprojs()</b></a>	The number of columns in the projected row that the access method returns to the query
<a href="#"><b>mi_scan_projs()</b></a>	A pointer to an array that identifies which columns from the row descriptor make up the projected row that the query returns
<a href="#"><b>mi_scan_quals()</b></a>	A pointer to the qualification descriptor or a NULL-valued pointer if the database server does not create a qualification descriptor
<a href="#"><b>mi_scan_table()</b></a>	A pointer to the table descriptor for the index that the access method scans
<a href="#"><b>mi_scan_userdata()</b></a>	A pointer to the user-data area of memory

The following accessor function sets data in the qualification descriptor.

Accessor Function	Value Set
<a href="#">mi_scan_setuserdata()</a>	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
<a href="#"><code>mi_istats_set2lval()</code></a>	A pointer to the second largest key value in the index
<a href="#"><code>mi_istats_set2sval()</code></a>	A pointer to the second smallest key value in the index
<a href="#"><code>mi_istats_setclust()</code></a>	The degree of clustering A low number indicates fewer clusters and a high degree of clustering.
<a href="#"><code>mi_istats_setnleaves()</code></a>	The number of leaves in the index
<a href="#"><code>mi_istats_setnlevels()</code></a>	The number of levels in the index
<a href="#"><code>mi_istats_setnunique()</code></a>	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
<a href="#"><code>mi_tab_amparam()</code></a>	Parameter values from the <code>USING</code> clause of the <code>CREATE INDEX</code> statement
<a href="#"><code>mi_tab_check_is_recheck()</code></a>	<code>MI_TRUE</code> if the database server invokes <code>am_check</code> to recheck and possibly repair an index
<a href="#"><code>mi_tab_createdate()</code></a>	The date that the index was created
<a href="#"><code>mi_tab_isindex()</code></a>	<code>MI_TRUE</code> for a secondary access method
<a href="#"><code>mi_tab_isolevel()</code></a>	The isolation level
<a href="#"><code>mi_tab_keydesc()</code></a>	A pointer to the key descriptor
<a href="#"><code>mi_tab_mode()</code></a>	The input/output mode (read-only, read and write, write-only, and log transactions)
<a href="#"><code>mi_tab_name()</code></a>	The index name
<a href="#"><code>mi_tab_nextrow()</code></a>	One entry from shared memory to insert in a new index
<a href="#"><code>mi_tab_niorows()</code></a>	The number of rows that <a href="#"><code>mi_tab_setniorows()</code></a> sets
<a href="#"><code>mi_tab_nparam_exist()</code></a>	The number of indexes that are defined for the same combination of table key columns
<a href="#"><code>mi_tab_numfrags()</code></a>	The number of fragments in the index or 1 for a single-fragment index
<a href="#"><code>mi_tab_owner()</code></a>	The index owner

(1 of 2)

Accessor Function	Return Value
<a href="#">mi_tab_param_exist()</a>	Configuration parameters and values for one of multiple indexes that pertain to the same table and composite key
<a href="#">mi_tab_partnum()</a>	The unique partition number, or fragment identifier, of this index or fragment
<a href="#">mi_tab_rowdesc()</a>	A pointer to a row descriptor that describes the columns in the index key
<a href="#">mi_tab_spaceloc()</a>	The extspace location of the index fragment
<a href="#">mi_tab_spacename()</a>	The storage space name for the fragment from the CREATE INDEX statement IN clause
<a href="#">mi_tab_spacetype()</a>	The type of space used for the index: X for an extspace or S for an sbpace Any other value means that neither an IN clause nor the <b>sysams</b> system catalog table specifies the type of storage space.
<a href="#">mi_tab_unique()</a>	MI_TRUE if this index should enforce unique keys
<a href="#">mi_tab_update_stat_mode()</a>	The level of statistics that an UPDATE STATISTICS statement generates: LOW, MEDIUM, or HIGH
<a href="#">mi_tab_userdata()</a>	A pointer to the user-data area of memory

(2 of 2)

The following accessor functions set values in the table descriptor.

Accessor Function	Value Set
<a href="#">mi_tab_check_set_ask()</a>	An indicator that <b>am_check</b> detects a problem in an index
<a href="#">mi_tab_setniorows()</a>	The number of rows that shared memory can store from a scan for a new index
<a href="#">mi_tab_setnextrow()</a>	One row of the number that <a href="#">mi_tab_setniorows()</a> allows
<a href="#">mi_tab_setuserdata()</a>	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.
- To call GLS routines for internationalization, include **ifxgls.h**. ♦

# 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 5-6](#).

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 Name or Prefix
Key	mi_key_*()
Qualification	mi_qual_*()
Row ID	mi_id_*()
Scan	mi_scan_*()
Statistics	mi_istats_*()
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 *ridDesc)
```

*ridDesc*            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 Topic**

See the description of function [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 *ridDesc)
```

*ridDesc*            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 Topic**

See the description of accessor functions [mi\\_id\\_setrowid\(\)](#), 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 *ridDesc,  
                    mi_integer fragid)
```

*ridDesc*            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 Topic**

See the description of function, [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 *ridDesc,  
                   mi_integer rowid)
```

*ridDesc*            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 Topic**

See the description of function [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 page-size areas, such as in a dbspace or sbpace, 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 4 bytes that begin at address *2lval* as an integer value in the **colmax** column.

### Return Values

None

### Related Topic

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 4 bytes that begin at address *2sval* as an integer value in the **colmin** column.

### Return Values

None

### Related Topic

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 [DataBlade API Programmer's Manual](#), including functions **mi\_func\_desc\_by\_typeid()** and **mi\_routine\_exec()**.
- CREATE INDEX in the [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()

The **mi\_key\_opclass()** function identifies 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)
```

*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()**.

#### *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()** 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()** function specifies the default operator class.

## Return Values

A positive 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.

## Related Topics

See the description of:

- the **am\_defopclass** purpose value in [“Setting Purpose Functions, Flags, and Values” on page 6-12](#).
- 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-column key 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 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-column keys.

## 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 Topic

See the descriptions of:

- functions [mi\\_key\\_opclass\\_strat\(\)](#), [mi\\_key\\_nkeys\(\)](#), and [mi\\_key\\_opclass\(\)](#).
- “Supporting Multiple-Column Index Keys” on page 3-24.



## 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()** returns the number of function names to retrieve for a single-column key 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-column keys.

## 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 Topic

See the descriptions of:

- functions [mi\\_key\\_opclass\\_supt\(\)](#), [mi\\_key\\_nkeys\(\)](#), and [mi\\_key\\_opclass\(\)](#).
- “Supporting Multiple-Column Index Keys” on page 3-24.

## 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\(\)](#), and [mi\\_qual\\_funcname\(\)](#).
- [“Supporting Multiple-Column Index Keys”](#) on page 3-24.

## 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 3-8 on page 3-25](#).

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 3-26](#), particularly [“Obtaining the Routine Identifier” on page 3-27](#).

## 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\(\)](#).
- “Supporting Multiple-Column Index Keys” on page 3-24.

## mi\_qual\_boollop()

The **mi\_qual\_boollop()** function retrieves the Boolean operator that combines two qualifications in a complex expression.

### Syntax

```
MI_AM_BOOLLOP mi_qual_boollop(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\_boollop()** function. For an example, refer to [Figure 3-15 on page 3-38](#).



**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\_boollop()**.
2. If **mi\_qual\_boollop()** 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 Topic

See the descriptions of:

- function [mi\\_qual\\_issimple\(\)](#).
- [“Qualifying Data” on page 3-35.](#)



---

## **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 [DataBlade API Programmer's Manual](#).

---

## 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 `?` in the following function:

```
WHERE equal(name,?)
```



**Important:** *Because the value that an application binds to host variables can change between scans, the results of **mi\_qual\_constant()** might change between calls to **am\_getnext**.*

To determine if a function involves a host variable argument, execute **mi\_qual\_const\_depends\_hostvar()** in the **am\_scancost** purpose function. If **mi\_qual\_const\_depends\_hostvar()** returns **MI\_TRUE**, call **mi\_qual\_constant()** from **am\_getnext** to retrieve the most recent value for the host variable and do not save the value from **mi\_qual\_constant()** in user data for subsequent scans.

## Return Values

The **MI\_DATUM** structure contains the value of the constant argument.

## Related Topics

See the descriptions of:

- functions [mi\\_qual\\_column\(\)](#), [mi\\_qual\\_constisnull\(\)](#), and [mi\\_qual\\_const\\_depends\\_hostvar\(\)](#).
- generic functions in [Figure 3-9 on page 3-31](#).
- **MI\_DATUM** in the [DataBlade API Programmer's Manual](#).

---

## mi\_qual\_constant\_nohostvar()

The **mi\_qual\_constant\_nohostvar()** function returns an explicit constant value, if any, from the 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 Value

If the argument list of a function includes a specified constant value, **mi\_qual\_constant\_nohostvar()** returns that value in an ML\_DATUM structure.

If the specified constant contains multiple values, this function returns all provided values and substitutes NULL for each host variable.

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

## Related Topics

See the descriptions of:

- accessor functions **mi\_qual\_constisnull\_nohostvar()** and **mi\_qual\_constant()**.
- “Runtime Values as Arguments” on page 3-31.
- ML\_DATUM in the *DataBlade API Programmer’s Manual*.
- host variables in the *DataBlade API Programmer’s Manual*, *Extending Informix Dynamic Server 2000*, and the *Informix ESQ/C Programmer’s Manual*.

## mi\_qual\_constisnull()

The **mi\_qual\_constisnull()** function determines if 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
<b>function</b> (column, 10)	The arguments specify the explicit non-NULL constant value 10.	MI_FALSE
<b>function</b> (column, NULL)	The arguments specify an explicit NULL value.	MI_TRUE

The form **function**(column,?) should not occur because the qualification descriptor that the database server passes to the **am\_beginscan** or **am\_getnext** purpose function contains values for any host-variable argument.

Do not call this function from the **am\_scancost** purpose function. Use **mi\_qual\_constisnull\_nohostvar()** instead.

### Return Values

MI\_TRUE indicates that the arguments include an explicit NULL-valued constant.



## mi\_qual\_constisnull\_nohostvar()

The **mi\_qual\_constisnull\_nohostvar()** function determines if a 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\_sancost** 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
<b>function</b> (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 <b>mi_qual_constisnull_nohostvar()</b> .	MI_FALSE
<b>function</b> (column, ROW(NULL,?,20))	The first field in the row constant specifies an explicit NULL value.	MI_TRUE
<b>function</b> (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

ML\_TRUE indicates one of the following conditions in the argument list:

- An explicit NULL-valued constant
- No explicit values

ML\_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 3-31.
- host variables in the *DataBlade API Programmer’s Manual*, *Extending Informix Dynamic Server 2000*, and the *Informix ESQL/C Programmer’s Manual*.

---

## **mi\_qual\_const\_depends\_hostvar()**

The **mi\_qual\_const\_depends\_hostvar()** function indicates whether the value of a host variable influences the evaluation of a qualification.

### **Syntax**

```
mi_boolean  
mi_qual_const_depends_hostvar(MI_AM_QUAL_DESC *qualDesc)
```

*qualDesc*        points to the qualification descriptor.

### **Usage**

Call **mi\_qual\_const\_depends\_hostvar()** in the **am\_sancost** purpose function to determine whether a strategy function contains a host variable but no explicit constant value.

Because the database server executes **am\_sancost** 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\_sancost** 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 3-31.
- host variables in the *DataBlade API Programmer’s Manual*, *Extending Informix Dynamic Server 2000*, and the *Informix ESQL/C Programmer’s Manual*.

---

## **mi\_qual\_const\_depends\_outer()**

The **mi\_qual\_const\_depends\_outer()** function indicates that an outer join provides the constant in a qualification.

### **Syntax**

```
mi_boolean  
mi_qual_const_depends_outer(MI_AM_QUAL_DESC *qualDesc)
```

*qualDesc*        points to the qualification descriptor.

### **Usage**

If this **mi\_qual\_const\_depends\_outer()** evaluates to MI\_TRUE, the join or subquery can produce a different constant value for each rescan.

Call **mi\_qual\_const\_depends\_outer()** in **am\_rescan**. If your access method has no **am\_rescan** purpose function, call **mi\_qual\_const\_depends\_outer()** in **am\_beginscan**.

### **Return Values**

MI\_TRUE indicates that the constant depends on an outer join. MI\_FALSE indicates that the constant remains the same on a rescan.

### **Related Topics**

See the description of accessor function [mi\\_qual\\_constant\(\)](#).

---

## mi\_qual\_funcid()

The **mi\_qual\_funcid()** function returns the routine identifier of a 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 the DataBlade API FastPath facility, the access method needs a valid routine identifier. The **mi\_qual\_funcid()** function provides a routine identifier, if available, for the 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 3-36.
- “Using FastPath” on page 3-26.

In the [DataBlade API Programmer's Manual](#), see the descriptions of:

- the function descriptor (MI\_FUNC\_DESC data structure) and its accessor functions.
- FastPath function execution, including DataBlade API functions [mi\\_func\\_desc\\_by\\_typeid\(\)](#) and [mi\\_routine\\_exec\(\)](#).

---

## **mi\_qual\_funcname()**

The `mi_qual_funcname()` function returns the name of a 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 3-36](#).

### **Return Value**

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



---

## **mi\_qual\_handlenull()**

The **mi\_qual\_handlenull()** function determines if the 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 if a qualification is a function. A function has one of the formats that [Figure 3-9 on page 3-31](#) 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 3-36](#).

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\_boollop()** accessor function.

### Return Values

MI\_TRUE indicates that the qualification is a function. MI\_FALSE indicates that the qualification is not a function.

### Related Topic

See the description of:

- accessor function [mi\\_qual\\_boollop\(\)](#).
- [“Simple Functions” on page 3-30](#).

## 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 3-31](#).

### 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 Topic

See the description of accessor function [mi\\_qual\\_setoutput\(\)](#).

---

## **mi\_qual\_negate()**

The **mi\_qual\_negate()** function indicates whether the NOT Boolean operator applies to the results of the specified qualification. The NOT operator can negate the return value of a function or a Boolean expression.

### **Syntax**

```
mi_boolean mi_qual_negate(MI_AM_QUAL_DESC *qualDesc);
```

*qualDesc*        points to the qualification descriptor.

### **Return Values**

MI\_TRUE indicates that the strategy function should be negated. MI\_FALSE indicates that the strategy function should not be negated.

### **Related Topic**

See the description of [“Negation”](#) on page 3-33.

---

## **mi\_qual\_nquals()**

The **mi\_qual\_nquals()** function retrieves the number of qualifications in an AND or OR qualification expression.

### **Syntax**

```
mi_integer mi_qual_nquals(MI_AM_QUAL_DESC *qualDesc);
```

*qualDesc*        points to the qualification descriptor.

### **Return Values**

The return integer indicates the number of qualifications in an AND or OR qualification expression. A return value of 0 indicates that the qualification contains one simple function and no Boolean operators.

### **Related Topic**

See the description of [“Complex Boolean Expressions”](#) on page 3-33.

---

## 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\_qual()**.

If **mi\_qual\_nquals()** returns a nonzero value, the qualification descriptor combines nested qualifications in a complex expression. The access method can loop through **mi\_qual\_qual()** to process each qualification from those that AND or OR combine. For an example, refer to [“Processing Complex Qualifications” on page 3-36](#).

### 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);
```

<i>qualDesc</i>	points to the qualification descriptor.
<i>n</i>	is always set to 0 to indicate the first and only argument that needs a value.
<i>value</i>	passes the output value in a MI_DATUM data structure.
<i>null_flag</i>	is MI_TRUE if <i>value</i> 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 3-31](#).

### Return Values

None

### Related Topic

See the description of accessor function [mi\\_qual\\_needoutput\(\)](#).

---

## **mi\_qual\_setreopt()**

The **mi\_qual\_setreopt()** function sets an indicator in the qualification descriptor to force reoptimization.

### **Syntax**

```
void    mi_qual_setreopt(MI_AM_QUAL_DESC *qualDesc)
```

*qualDesc*        points to the qualification descriptor.

### **Usage**

The **am\_scancost** purpose function can call the **mi\_qual\_setreopt()** to indicate that the optimizer should reevaluate the query path between scans. For example, if either the **mi\_qual\_const\_depends\_hostvar()** or **mi\_qual\_const\_depends\_outer()** function returns **MI\_TRUE**, the access method can call **mi\_qual\_setreopt()** to alert the optimizer that the constant-argument value in a qualification descriptor might change between scans on the same 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 Topic**

See the description of accessor functions [mi\\_scan\\_locktype\(\)](#) and [mi\\_tab\\_mode\(\)](#).

---

## **mi\_scan\_isolevel()**

The **mi\_scan\_isolevel()** function retrieves the isolation level that the database server expects for the table that **am\_getnext** scans.

### **Syntax**

```
MI_ISOLATION_LEVEL mi_scan_isolevel(MI_AM_SCAN_DESC  
*scanDesc);
```

*scanDesc*        points to the scan descriptor.

### **Usage**

If the access method supports isolation levels, it can call **mi\_scan\_isolevel()** from **am\_beginscan** to determine the appropriate isolation level. For a detailed description of isolation levels, see [“Checking Isolation Levels” on page 3-47](#).

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\_NOTTRANSACTION 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 3-47](#).
- sample isolation-level language for access-method documentation in [Figure 3-19 on page 3-55](#).

---

## mi\_scan\_locktype()

The **mi\_scan\_locktype()** function retrieves the lock type that the database server expects for the table that **am\_getnext** scans.

### Syntax

```
MI_LOCK_TYPE mi_scan_locktype(MI_AM_SCAN_DESC *scanDesc);
```

*scanDesc*      points to the scan descriptor.

### Usage

If the access method supports locking, use the return value from this function to determine whether you need to lock an object during **am\_getnext**.

### Return Values

MI\_LCK\_S indicates a shared lock on the table.

MI\_LCK\_X indicates an exclusive lock on the table.

MI\_LCK\_IS\_S indicates an intent-shared lock on the table and shared lock on the row.

MI\_LCK\_IX\_X indicates intent-exclusive lock on the table and exclusive lock on the row.

MI\_LCK\_SIX\_X indicates an intent-shared exclusive lock on the table and an exclusive lock on the row.

### Related Topics

See the descriptions of:

- functions [mi\\_scan\\_isolevel\(\)](#) and [mi\\_scan\\_forupdate\(\)](#).
- locks in the [Performance Guide](#).

---

## **mi\_scan\_nprojs()**

The **mi\_scan\_nprojs()** function returns a value that is 1 less than the number of 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 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 Topic**

See the description of accessor function [mi\\_scan\\_projs\(\)](#).

---

## **mi\_scan\_newquals()**

The **mi\_scan\_newquals()** function indicates whether the qualification descriptor includes changes between multiple scans for the same query statement.

### **Syntax**

```
mi_boolean mi_scan_newquals(MI_AM_SCAN_DESC *scanDesc);
```

*scanDesc*      points to the scan descriptor.

### **Usage**

This function pertains to multiple-scan queries, such as a join or subquery. If the access method provides a function for the **am\_rescan** purpose, that rescan function calls **mi\_scan\_newquals()**.

If this function returns **MI\_TRUE**, retrieve information from the qualification descriptor and obtain function descriptors. If it returns **MI\_FALSE**, retrieve state information that the previous scan stored in user data.

### **Return Values**

**MI\_TRUE** indicates that the qualifications have changed since the start of the scan (**am\_beginscan**). **MI\_FALSE** indicates that the qualifications have not changed.

---

## **mi\_scan\_projs()**

The **mi\_scan\_projs()** function identifies each 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 [DataBlade API Programmer's Manual](#).



## mi\_scan\_qual()

The **mi\_scan\_qual()** 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_qual(MI_AM_SCAN_DESC *scanDesc);
```

*scanDesc*      points to the scan descriptor.

### Usage

The **am\_getnext** purpose function calls **mi\_scan\_qual()** 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 5-9](#).

---

## **mi\_scan\_setuserdata()**

The **mi\_scan\_setuserdata()** function stores a pointer to user data in the scan descriptor.

### **Syntax**

```
void mi_scan_setuserdata(MI_AM_SCAN_DESC *scanDesc, void  
*user_data);
```

*scanDesc*        points to the scan descriptor.

*user\_data*       points to the user data.

### **Usage**

The access method can create a user-data structure in shared memory to store reusable information, such as function descriptors for qualifications and to maintain a row pointer for each execution of the **am\_getnext** purpose function. To retain user data in memory during the scan (starting when **am\_beginscan** is called and ending when **am\_endscan** is called), follow these steps:

1. In the **am\_beginscan** purpose function, call the appropriate DataBlade API function to allocate memory for the user-data structure.  
Allocate the user-data memory with a duration of PER\_COMMAND.
2. In **am\_getnext**, populate the user-data structure with scan-state information.
3. Before **am\_getnext** exits, call **mi\_scan\_setuserdata()** to store a pointer to the user-data structure in the scan descriptor.
4. In the **am\_endscan** purpose function, call the appropriate DataBlade API function to deallocate the user-data memory.

## Return Values

None

## Related Topics

See the descriptions of:

- function [mi\\_scan\\_userdata\(\)](#).
- DataBlade API functions for memory allocation and duration in [“Storing Data in Shared Memory”](#) on page 3-4.

---

## **mi\_scan\_table()**

The **mi\_scan\_table()** function retrieves a pointer to the table descriptor for the 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 5-16](#).
- accessor functions for the row descriptor in the [DataBlade API Programmer's Manual](#).

---

## **mi\_scan\_userdata()**

The **mi\_scan\_userdata()** function retrieves the pointer from the scan descriptor that points to a user data structure.

### **Syntax**

```
void* mi_scan_userdata(MI_AM_SCAN_DESC *scanDesc);
```

*scanDesc*      points to the scan descriptor.

### **Usage**

If the access method allocates user-data memory to hold scan-state information, it places a pointer to that user data in the scan descriptor. Use the **mi\_scan\_userdata()** function to retrieve the pointer for access to the user data.

For example, the **am\_getnext** might maintain a row pointer to keep track of its progress through the 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 Topic**

See the description of:

- function [mi\\_scan\\_setuserdata\(\)](#).
- [“Storing Data in Shared Memory”](#) on page 3-4.

---

## **mi\_tab\_amparam()**

The **mi\_tab\_amparam()** function retrieves any user-defined configuration values for the 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 3-21](#) 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 3-7 on page 3-23](#) 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 3-21.
- “[Providing Configuration Keywords](#)” on page 3-18.
- the USING clause of the CREATE INDEX statement in the [Informix Guide to SQL: Syntax](#).

---

## 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 invokes the **am\_check** purpose function.
3. During the first execution of **am\_check**, the purpose function detects a problem with the index, calls **mi\_tab\_check\_set\_ask()** to alert the database server, and exits.
4. The database server prompts the user to indicate if the access method should repair the index.
5. The user answers **y** or **yes** to the prompt, and the database server executes **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** execution:

- 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\_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])
```

<i>tableDesc</i>	points to the descriptor for the index that the <b>oncheck</b> command line specifies.
<i>msg_type</i>	<p>indicates where <b>oncheck</b> should look for the message.</p> <p>If <i>msg_type</i> is MI_SQL, an error occurred. The <b>syserrors</b> system catalog table contains the message.</p> <p>If <i>msg_type</i> is MI_MESSAGE, the pointer in the <i>msg</i> argument contains the address of an information-only message string.</p>
<i>msg</i>	<p>points to a message string of up to 400 bytes if <i>msg_type</i> is MI_MESSAGE.</p> <p>If <i>msg_type</i> is MI_SQL, <i>msg</i> points to a 5-character <b>SQLSTATE</b> value. The value identifies an error or warning in the <b>syserrors</b> system catalog table.</p>
<i>marker_n</i>	specifies a marker name in the <b>syserrors</b> 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 [DataBlade API Programmer's Manual](#).



**Important:** Do not use msg\_type values **MI\_FATAL** or **MI\_EXCEPTION** with **mi\_tab\_check\_msg()**. These message types are reserved for the DataBlade API function **mi\_db\_error\_raise()**.

## Return Values

None

## Related Topics

See the descriptions of:

- purpose function [am\\_check](#) on [page 4-16](#).
- accessor functions [mi\\_tab\\_check\\_is\\_recheck\(\)](#) and [mi\\_tab\\_check\\_set\\_ask\(\)](#).
- DataBlade API function [mi\\_db\\_error\\_raise\(\)](#) in the [DataBlade API Programmer's Manual](#), particularly the information about raising custom messages.
- **oncheck** in the *Administrator's Reference*.

## mi\_tab\_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 did 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 the **-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.

*mi\_tab\_check\_set\_ask()*

## 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 Value**

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\_NOTTRANSACTION 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 3-47](#).
- sample isolation-level language for access-method documentation in [Figure 3-19 on page 3-55](#).

---

## **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 Value**

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 5-8.

# 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_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 [Figure 5-1](#) for interpretation.

Each macro returns either MI\_TRUE or MI\_FALSE.

**Figure 5-1**  
Macro Modes

Macro	Mode Verified
MI_INPUT()	Open for input only, usually in the case of a SELECT statement
MI_OUTPUT()	Open for output only, usually in the case of an INSERT statement
MI_INOUT()	Open for input and output, usually in the case of an UPDATE statement
MI_NOLOG()	No logging required

In the following example, the access method calls **mi\_tab\_mode()** to verify that a query is read-only. If **MI\_INOUT()** returns **MI\_FALSE**, the access method requests a multiple-row buffer because the access method can return several rows without interruption by an update:

```
if (MI_INOUT(tableDesc) == MI_FALSE)
    mi_tab_setniorows(tableDesc, 10);
```

If **MI\_INOUT()** returns **MI\_TRUE**, the access method can process only one row identifier with each call to **am\_getnext**.

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 [Figure 5-1 on page 5-95](#) describes.

## Related Topics

See the descriptions of

- [“Buffering Multiple Results” on page 3-44.](#)
- purpose functions **am\_beginscan** and **am\_getnext**.
- [“Building New Indexes Efficiently” on page 3-20.](#)
- purpose functions **am\_open** and **am\_insert**.
- setting logging preferences in [Figure 3-5 on page 3-19.](#)

---

## **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 name specifies 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 3-6 on page 3-20](#).

## 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 3-20.

---

## **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 1 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\_ainparam()**.
- [“Enabling Alternative Indexes” on page 3-21.](#)

---

## **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 Topic**

See the description of the Owner Name segment in the [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 3-21](#).

### Return Values

The string lists keywords and their values from the **amparam** column of the **sysindices** 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 from 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 [DataBlade API Programmer's Manual](#) for the descriptions of:

- DataBlade API row-descriptor accessor functions  
**mi\_column\_bound()**, **mi\_column\_count()**, **mi\_column\_id()**,  
**mi\_column\_name()**, **mi\_column\_nullable()**, **mi\_column\_scale()**,  
**mi\_column\_type\_id()**, and **mi\_column\_typedesc()**.
- the row descriptor (MI\_ROW\_DESC data structure).

## mi\_tab\_setnextrow()

The **am\_getnext** purpose function calls **mi\_tab\_setnextrow()** to store the next entry that qualifies for selection.

### Syntax

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

*tableDesc* points to the 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*, and *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 3-18 on page 3-54](#).

## 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 3-44](#).



## mi\_tab\_setniorows()

The **mi\_tab\_setniorows()** function indicates:

- that the access method can handle more than one row per call.
- the number of rows for which the database server should allocate memory.

## Syntax

```
mi_integer mi_tab_setniorows(MI_AM_TABLE_DESC *tableDesc,
                             mi_integer nrows)
```

*tableDesc*      points to the 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 *user_data)
```

*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.
4. Pass the pointer as the *user\_data* argument.

Typically, an access method performs the preceding procedure in the **am\_open** purpose function and deallocates the user-data memory in the **am\_close** purpose function. To have the table descriptor retain the pointer to the user data as long as the table remains open, specify a memory duration of PER\_STATEMENT, as [“Memory-Duration Options” on page 3-5](#) and [“Persistent User Data” on page 3-6](#) describe.

To retrieve the pointer from the table descriptor to access the table-state user data, call the **mi\_tab\_userdata()** function in any purpose function between **am\_open** and **am\_close**.

## Return Values

None

## Related Topics

See the descriptions of:

- function [mi\\_tab\\_userdata\(\)](#).
- purpose functions [am\\_open](#) and [am\\_close](#).
- DataBlade API functions for memory allocation and duration in [“Storing Data in Shared Memory”](#) on page 3-4.

## mi\_tab\_spaceloc()

The **mi\_tab\_spaceloc()** function retrieves the location of the extspace in which the 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 system 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 sbpace, 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 new smart large objects or virtual indexes. 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 3-14](#).

### **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 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 4-38](#).
- UPDATE STATISTICS in the [Informix Guide to SQL: Syntax](#) and the [Performance Guide](#).

---

## **mi\_tab\_userdata()**

The **mi\_tab\_userdata()** function retrieves, from the table descriptor, a pointer to a user-data structure that the access method maintains in shared memory.

### **Syntax**

```
void* mi_tab_userdata(MI_AM_TABLE_DESC *tableDesc)
```

*tableDesc*      points to the 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 Topic**

See the descriptions of:

- function [mi\\_tab\\_setuserdata\(\)](#).
- [“Storing Data in Shared Memory”](#) on page 3-4.



---

# SQL Statements for Access Methods

In This Chapter . . . . .	6-3
ALTER ACCESS_METHOD . . . . .	6-4
CREATE ACCESS_METHOD . . . . .	6-6
DROP ACCESS_METHOD. . . . .	6-8
Purpose Options . . . . .	6-10



## 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 Conventions](#)” on page 9 of the [Introduction](#).

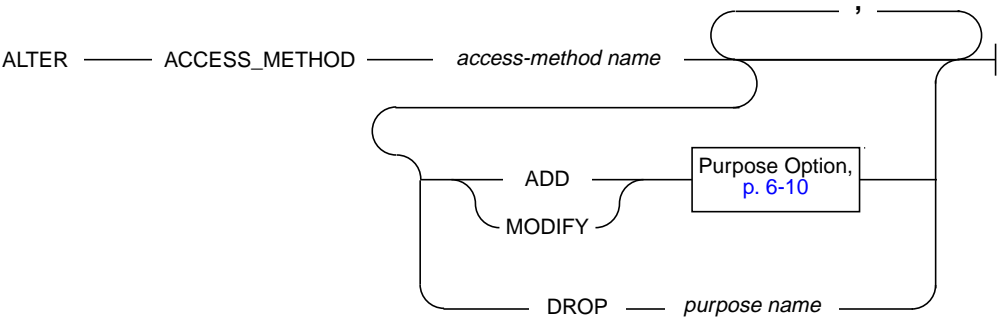
This chapter also provides the valid purpose-function, purpose-flag, and purpose-value settings.

+

## ALTER ACCESS\_METHOD

The ALTER ACCESS\_METHOD statement changes the attributes of a user-defined access method in the **sysams** system catalog table.

### Syntax



Element	Purpose	Restrictions	Syntax
<i>access-method name</i>	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 <a href="#">Informix Guide to SQL: Syntax</a> .
<i>purpose name</i>	A keyword that indicates which purpose function, purpose value, or purpose flag to drop	A previous statement must associate the purpose name with this access method.	"Purpose-Name Keyword" on <a href="#">page 6-13</a> .

### 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 6-10](#).

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_UNIQUE,
DROP AM_CHECK,
MODIFY AM_SPTYPE = ' S' );
```

**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\_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 [Informix Guide to Database Design and Implementation](#) or the GRANT statement in the [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 — ( — 

Purpose Option,  
p. 6-12

 — ) —
```

Element	Purpose	Restrictions	Syntax
<i>access-method name</i>	The access method to add	The access method must have a unique name in the <b>sysams</b> system catalog table.	Database Object Name segment; see <a href="#">Informix Guide to SQL: Syntax</a> .

### 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 6-10](#).

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

### Sample Statements

The following statement creates a 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 [Informix Guide to Database Design and Implementation](#) or the GRANT statement in the [Informix Guide to SQL: Syntax](#).

+

# DROP ACCESS\_METHOD

Use the DROP ACCESS\_METHOD statement to remove a previously defined access method from the database.

## Syntax

```
DROP ACCESS_METHOD access-method name RESTRICT
```

Element	Purpose	Restrictions	Syntax
<i>access-method name</i>	The access method to drop	The access method must be registered in the <b>sysams</b> system catalog table with a previous CREATE ACCESS_METHOD statement.	Database Object Name segment; see <a href="#">Informix Guide to SQL: Syntax</a> .

## 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

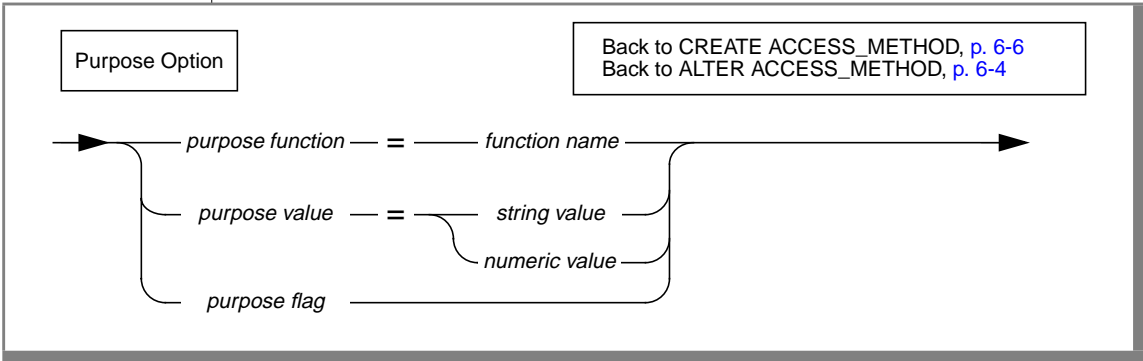
See the descriptions of:

- [CREATE ACCESS\\_METHOD](#) and [ALTER ACCESS\\_METHOD](#) statements in this chapter.
- keyword RESTRICT in the *Informix Guide to SQL: Syntax*.
- privileges in the *Informix Guide to Database Design and Implementation* or the GRANT statement in the *Informix Guide to SQL: Syntax*.

## Purpose Options

The database server recognizes a registered access method as a set of attributes, including the access-method name and options called *purposes*. The CREATE SECONDARY ACCESS\_METHOD and ALTER ACCESS\_METHOD statements specify purpose attributes with the following syntax.

### Syntax



Element	Purpose	Restrictions	Syntax
<i>purpose function</i>	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 <a href="#">Figure 6-3 on page 6-13</a> .
<i>purpose value</i>	A keyword that identifies configuration information	The interface specifies the predefined configuration keywords to which you can assign values.	Value purpose category; see <a href="#">Figure 6-3 on page 6-13</a> .
<i>purpose flag</i>	A keyword that indicates which feature a flag enables	The interface specifies flag names.	Flag purpose category; see <a href="#">Figure 6-3 on page 6-13</a> .

(1 of 2)

Element	Purpose	Restrictions	Syntax
<i>function name</i>	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 <a href="#">Informix Guide to SQL: Syntax</a> .
<i>string value</i>	An indicator that is expressed as one or more characters	None.	Quoted String segment; see <a href="#">Informix Guide to SQL: Syntax</a> .
<i>numeric value</i>	A value that can be used in computations	None.	A numeric literal.

(2 of 2)

## Usage

Each purpose-name keyword corresponds to a column name in the **sysams** system catalog table. The database server uses the following types of purpose attributes:

- Purpose functions  
A purpose-function attribute maps the name of a user-defined function to one of the prototype purpose functions that [Figure 1-2 on page 1-15](#) describes.
- Purpose flags  
Each flag indicates whether an access method supports a particular SQL statement or keyword.
- Purpose values  
These string, character, or numeric values provide configuration information that a flag cannot supply.

You specify purpose options when you create an access method with the CREATE 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 6-7](#) sets the **am\_getnext** purpose-function name to the UDR name **tree\_getnext**. This example creates a new access method.

The example in [Figure 6-1 on page 6-5](#) adds a purpose function to an existing access method.

To enable a purpose flag, specify the purpose name without a corresponding value.

To clear a purpose-option setting in the **sysams** system catalog table, use the DROP clause of the ALTER ACCESS\_METHOD statement.

## Setting Purpose Functions, Flags, and Values

[Figure 6-3](#) describes the possible settings for the **sysams** columns that contain purpose-function names, purpose flags, and purpose values. The items in [Figure 6-3](#) appear in the same order as the corresponding **sysams** columns.



**Figure 6-3**  
Purpose Functions, Purpose Flags, and Purpose Values

Purpose-Name Keyword	Explanation	Purpose category	Default Setting
<b>am_sptype</b>	<p>A character that specifies what type of storage space the access method supports</p> <p>For a user-defined access method, <b>am_sptype</b> can have any of the following settings:</p> <ul style="list-style-type: none"> <li>■ 'X' indicates that the access method accesses only extspaces</li> <li>■ 'S' indicates that the access method accesses only sbspaces</li> <li>■ 'A' indicates that the access method can provide data from extspaces and sbspaces</li> </ul> <p>You can specify <b>am_sptype</b> only for a new access method. You cannot change or add an <b>am_sptype</b> value with ALTER ACCESS_METHOD.</p> <p>Do not set <b>am_sptype</b> to 'D' or attempt to store a virtual index in a dbspace.</p>	Value	'A'
<b>am_defopclass</b>	<p>The name of the default operator class for this access method</p> <p>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.</p>	Value	None
<b>am_keyscan</b>	<p>A flag that, if set indicates that <b>am_getnext</b> returns rows of index keys</p> <p>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.</p>	Flag	Not set
<b>am_unique</b>	A flag that you set if the secondary access method checks for unique keys	Flag	Not set
<b>am_cluster</b>	A flag that you set if the access method supports clustering of tables	Flag	Not set

(1 of 3)

Purpose-Name Keyword	Explanation	Purpose category	Default Setting
<b>am_parallel</b>	<p>A flag that the database server sets to indicate which purpose functions can execute in parallel</p> <p>If set, the hexadecimal <b>am_parallel</b> flag contains one or more of the following bit settings:</p> <ul style="list-style-type: none"> <li>■ The 1 bit is set for parallelizable scan.</li> <li>■ The 2 bit is set for parallelizable delete.</li> <li>■ The 4 bit is set for parallelizable update.</li> <li>■ The 8 bit is set for parallelizable insert.</li> </ul>	Flag	Not set
<b>am_costfactor</b>	<p>A value by which the database server multiplies the cost that the <b>am_scancost</b> purpose function returns</p> <p>An <b>am_costfactor</b> value from 0.1 to 0.9 reduces the cost to a fraction of the value that <b>am_scancost</b> calculates. An <b>am_costfactor</b> value of 1.1 or greater increases the <b>am_scancost</b> value.</p>	Value	1.0
<b>am_create</b>	The name of a user-defined function that adds a virtual index to the database	Function	None
<b>am_drop</b>	The name of a user-defined function that drops a virtual index	Function	None
<b>am_open</b>	The name of a user-defined function that makes a fragment, extspace, or sbspace available	Function	None
<b>am_close</b>	The name of a user-defined function that reverses the initialization that <b>am_open</b> performs	Function	None
<b>am_insert</b>	The name of a user-defined function that inserts an index entry	Function	None
<b>am_delete</b>	The name of a user-defined function that deletes an index entry	Function	None
<b>am_update</b>	The name of a user-defined function that changes the values in a key	Function	None
<b>am_stats</b>	The name of a user-defined function that builds statistics based on the distribution of values in storage spaces	Function	None

(2 of 3)

Purpose-Name Keyword	Explanation	Purpose category	Default Setting
<b>am_scancost</b>	The name of a user-defined function that calculates the cost of qualifying and retrieving data	Function	None
<b>am_check</b>	The name of a user-defined function that performs an integrity check on an index	Function	None
<b>am_beginscan</b>	The name of a user-defined function that sets up a scan	Function	None
<b>am_endscan</b>	The name of a user-defined function that reverses the setup that AM_BEGINSCAN initializes	Function	None
<b>am_rescan</b>	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
<b>am_getnext</b>	The name of the required user-defined function that scans for the next item that satisfies the query	Function	None

(3 of 3)

The following rules apply to the purpose-option specifications in the CREATE 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 manual, see the following topics:

- “Managing Storage Spaces” on page 3-12.
- “Executing in Parallel” on page 3-41.
- “Registering Purpose Functions” on page 2-10 and “Registering the Access Method” on page 2-12.
- “Specifying an Operator Class” on page 2-14.
- “Enforcing Unique-Index Constraints” on page 3-46.
- “Calculating Statement-Specific Costs” on page 3-39.
- “Bypassing Table Scans” on page 3-43.
- Chapter 4, “Purpose-Function Reference.”

In the *Informix Guide to SQL: Syntax*, see the descriptions of:

- Database Object Name segment (for a routine name), Quoted String segment, and Literal Number segment.
- CREATE FUNCTION statement
- CREATE OPERATOR CLASS statement.

# Index

## A

### Access method

- attributes 6-10
- choosing features 2-4
- configuring 6-10
- default operator class,
  - assigning 2-18, 6-13
- defined 6-10
- developing, steps in 2-3
- documenting 3-51
- dropping 2-24
- privileges needed
  - to alter 6-4
  - to drop 6-8
  - to register 6-6
- purpose functions. *See* Purpose functions.
- purpose options 6-10
- registering 2-12, 6-6
- sysams system catalog table
  - settings 6-11
- testing and using 2-18

### ALTER ACCESS\_METHOD

- statement
  - default operator class syntax 2-18
  - privileges needed 6-4
  - syntax 6-4

### ALTER FRAGMENT statement

- access-method support for 3-12
- am\_delete purpose function 4-23
- am\_insert purpose function 4-29
- purpose-function flow 4-8

### am\_beginscan purpose function

- allocating memory 3-5
- buffer setup 3-44, 5-109
- syntax 4-14
- usage 2-9

### am\_check purpose function

- creating output 5-86
- macros 4-17
- syntax 4-16

### am\_close purpose function,

- syntax 4-20

### am\_cluster purpose flag

- description 6-13

### am\_costfactor purpose value

- setting 6-14
- usage 4-36

### am\_create purpose function

- syntax 4-21
- usage 2-8
- with fragments 4-4

### am\_defopclass purpose value

- description 6-13
- example 2-18

### am\_delete purpose function

- design decisions 3-46
- parallel execution 3-42
- syntax 4-23
- usage 2-10

### am\_drop purpose function

- syntax 4-25
- usage 2-8

### am\_endscan purpose function

- syntax 4-26
- usage 2-10

**am\_getnext** purpose function  
     design decisions 3-46  
     mi\_tab\_setnext() function 5-107  
     number of rows to fetch 5-100  
     parallel execution 3-42  
     returning keys as rows 3-43  
     syntax 4-27  
     unique keys only 3-46  
     usage 2-9  
**am\_insert** purpose function  
     design decisions 3-46  
     multiple-entry buffering 3-21  
     parallel execution of 3-42  
     syntax 4-29  
     unique keys only 3-46  
     usage 2-10  
     zeros as arguments 4-29  
**am\_keyscan** purpose flag  
     description 6-13  
     effects 3-43  
**am\_open** purpose function  
     allocating memory 3-5  
     buffer setup 3-44, 5-109  
     buffered index-build  
         example 3-21  
     syntax 4-31  
     usage 2-7  
**am\_parallel** purpose flag,  
     description 6-14  
**am\_rescan** purpose function  
     detecting qualification  
         changes 5-75  
     syntax 4-33  
     usage 2-10  
**am\_scancost** purpose function  
     factors to calculate 4-35  
     functions to call 5-50, 5-68  
     syntax 4-34  
     usage 2-9, 3-39  
**am\_sptype** purpose value  
     description 6-13  
     error related to 2-23  
**am\_stats** purpose function  
     syntax 4-38  
     usage 2-9, 3-40  
**am\_unique** purpose flag  
     description 6-13  
     usage 3-46

**am\_update** purpose function  
     design decisions 3-46  
     parallel execution of 3-42  
     syntax 4-40  
     usage 2-10  
 ANSI compliance level Intro-18  
 API, defined 1-7  
 Application programming  
     interface. *See* API.

## B

Backup and restore in  
     sbspaces 3-16  
 Boldface type Intro-6  
 Buffering multiple results  
     filling buffer with  
         mi\_tab\_setnextrow()  
         function 5-107  
     specifying number to return 3-44

## C

Callback function  
     defined 3-8  
     for end-of-transaction 3-50  
     for unsupported features 3-51  
     registering 3-8  
 Callback handle 3-9  
 Clustering  
     degree of 5-24  
     specifying support for 6-13  
 Code set, ISO 8859-1 Intro-4  
 Code, sample, conventions  
     for Intro-13  
 Command-line conventions  
     elements of Intro-12  
     example Intro-13  
     how to read Intro-13  
 Comment icons Intro-7  
 Compliance  
     icons Intro-8  
     with industry standards Intro-18  
 Configuration parameters  
     documenting 3-56  
     retrieving 5-82  
     usage 3-18  
 Contact information Intro-18

Conventions,  
     documentation Intro-6  
 Converting data type 4-14  
 CREATE FUNCTION statement  
     NOT VARIANT routine modifier  
         requirement 2-16  
     PARALLELIZABLE routine  
         modifier in 2-11  
     privileges needed 2-11  
     registering purpose  
         functions 2-10  
     registering strategy and support  
         functions 2-16  
 CREATE INDEX statement  
     access-method support for 3-12  
     buffer setup for 3-21  
     example 2-21  
     fragmentation example 2-22  
     multiple-entry buffer,  
         example 3-21  
     purpose functions for 4-21, 4-29  
     purpose-function flow 4-4  
 CREATE OPCLASS statement 2-17  
 CREATE SECONDARY  
     ACCESS\_METHOD statement  
         syntax 6-6  
         usage 2-12  
 Customization 3-18

## D

Data definition statements 3-12  
 Data distribution 4-35  
 Data type conversion 4-14  
 DataBlade API functions  
     for callback 3-8  
     for end-of-transaction 3-50  
     for error messages 3-10  
     for FastPath UDR execution 3-26  
 DB-Access utility Intro-4  
 Default locale Intro-4  
 DELETE statement  
     am\_delete purpose function 4-23  
     parallel execution of 3-42  
     purpose-function flow 4-6  
 Demonstration databases Intro-4  
 Dependencies, software Intro-3

## Descriptor

*See individual descriptor names.*

Development process 2-3

Disk file, extspace for 2-21

DISTINCT keyword,  
enforcing 3-46

Documentation notes Intro-16

Documentation, types of  
documentation notes Intro-16  
error message files Intro-15  
machine notes Intro-16  
on-line manuals Intro-15  
printed manuals Intro-15  
related reading Intro-17  
release notes Intro-16

DROP ACCESS\_METHOD  
statement  
privileges needed 6-8  
syntax 6-8  
usage 2-24

DROP DATABASE or INDEX  
statement  
purpose function for 4-25  
purpose-function flow 4-4

## E

Environment variables Intro-6

en\_us.8859-1 locale Intro-4

Error message files Intro-15

Error messages  
creating 3-10  
from oncheck utility 5-86

Event handling 3-8

Extension to SQL, symbol  
for Intro-8

Extspace  
adding to system catalog  
tables 4-21

creating 2-20  
defined 2-20  
determining location 5-17  
determining name 5-114  
fragments 2-22

Extspace-only access method,  
specifying 3-13

## F

FastPath, defined 3-26

Feature icons Intro-8

Features of this product,  
new Intro-5

Find Error utility Intro-15

finderr utility Intro-15

Fragment  
defined 3-17  
partnum (fragment  
identifier) 5-17, 5-105

Fragmentation  
testing for 3-10  
usage 2-22

Fragments, number of 5-102

Function descriptor, getting and  
using 3-26

Functional index 5-8

Functional index key 5-30

## G

Global Language Support  
(GLS) Intro-4

## I

Icons

compliance Intro-8

feature Intro-8

Important Intro-7

platform Intro-8

product Intro-8

Tip Intro-7

Warning Intro-7

ifxgls.h 5-18

Important paragraphs, icon  
for Intro-7

IN clause

determining space type 5-17

errors from 2-23

specifying storage space 2-21

Include files 5-18

## Index

checking for duplicate 3-21

keys in 5-32

leaf nodes in 5-28

levels of 5-27

multiple, on identical keys  
example 3-21

number of 5-101

operator class for 5-33

repairing 4-18

resolving function for key 5-30

unique keys  
checking requirement for 5-117  
number of 5-29

specifying support for 6-13

various data types in 2-14

Index-key range 5-25, 5-26

Industry standards, compliance  
with Intro-18

INFORMIXDIR/bin  
directory Intro-4

INSERT statement

am\_insert purpose function 4-29

parallel execution of 3-42

purpose-function flow 4-6

Internationalization 5-18

ISO 8859-1 code set Intro-4

Isolation level

definitions of each 3-47

determining 3-11, 5-13, 5-16

documenting 3-55

retrieving 5-71, 5-93

## J

Join, purpose function for 4-33

## K

Key descriptor

description 5-8

retrieving pointer to 5-94

## L

Locale Intro-4  
     default Intro-4  
     en\_us.8859-1 Intro-4  
 Locks  
     for extspaces 3-17  
     for sbspaces 3-16  
     retrieving type 3-11, 5-13, 5-73  
 Logging  
     checking for 3-11, 5-16  
     enabling for sbspaces 3-15  
     extspaces 3-17  
     sbspaces 3-16

## M

Machine notes Intro-16  
 memdur.h 5-18  
 Memory allocation  
     for user data 4-31, 5-111  
     functions for 3-4  
 Memory deallocation 4-26  
 Memory duration  
     changing 3-5  
     keywords for specifying 3-5  
 Message file for error  
     messages Intro-15  
 miami.h 5-18  
 mi.h 5-18  
 MI\_AM\_ISTATS\_DESC  
     structure 5-15  
 MI\_AM\_KEY\_DESC structure 5-8  
 MI\_AM\_QUAL\_DESC  
     structure 5-9  
 MI\_AM\_ROWID\_DESC  
     structure 5-12  
 MI\_AM\_SCAN\_DESC  
     structure 5-13  
 MI\_AM\_TABLE\_DESC  
     structure 5-16  
 mi\_dalloc() function 3-5  
 mi\_db\_error\_raise() function 3-10  
 MI\_EVENT\_END\_XACT  
     event 3-50  
 MI\_Exception event  
     callback function 3-9  
 mi\_file\_\* functions 3-13

MI\_FUNC\_DESC structure 3-26  
 mi\_func\_desc\_by\_typeid()  
     function 3-26  
 mi\_id\_fragid() function,  
     syntax 5-20  
 mi\_id\_rowid() function,  
     syntax 5-21  
 mi\_id\_setfragid() function,  
     syntax 5-22  
 mi\_id\_setrowid() function,  
     syntax 5-23  
 mi\_istats\_set2lval() function,  
     syntax 5-25  
 mi\_istats\_set2sval() function,  
     syntax 5-26  
 mi\_istats\_setclust() function,  
     syntax 5-30  
 mi\_istats\_setnleaves() function,  
     syntax 5-28  
 mi\_istats\_setnlevels() function,  
     syntax 5-27  
 mi\_istats\_setnunique() function,  
     syntax 5-29  
 mi\_key\_funcid() function,  
     syntax 5-30  
 mi\_key\_nkeys() function  
     example 5-33  
     syntax 5-32  
     usage 5-35, 5-37  
 mi\_key\_opclass() function,  
     syntax 5-33  
 mi\_key\_opclass\_nstrat() function,  
     syntax 5-35  
 mi\_key\_opclass\_nsupt() function,  
     syntax 5-37  
 mi\_key\_opclass\_strat() function,  
     syntax 5-39  
 mi\_key\_opclass\_supt() function,  
     syntax 5-41  
 mi\_lo\_\* functions 3-13  
 MI\_NO\_MORE\_RESULTS return  
     value 4-28  
 mi\_qual\_column() function,  
     syntax 5-45  
 mi\_qual\_commuteargs() function,  
     syntax 5-47  
 mi\_qual\_constant() function,  
     syntax 5-48

mi\_qual\_constant\_nohostvar()  
     function, syntax 5-50  
 mi\_qual\_constisnull() function,  
     syntax 5-52  
 mi\_qual\_constisnull\_nohostvar()  
     function, syntax 5-53  
 mi\_qual\_const\_depends\_hostvar()  
     function, syntax 5-55  
 mi\_qual\_const\_depends\_outer()  
     function, syntax 5-57  
 mi\_qual\_depends\_hostvar()  
     function, syntax 5-55  
 mi\_qual\_funcid() function,  
     syntax 5-58  
 mi\_qual\_funcname() function  
     example 3-36  
     syntax 5-60  
 mi\_qual\_handlenull() function,  
     syntax 5-61  
 mi\_qual\_issimple() function  
     example 3-38  
     syntax 5-62  
 mi\_qual\_needoutput() function,  
     syntax 5-63  
 mi\_qual\_negate() function,  
     syntax 5-64  
 mi\_qual\_nquals() function  
     syntax 5-65  
     usage 5-66  
 mi\_qual\_qual() function,  
     syntax 5-66  
 mi\_qual\_setoutput() function,  
     syntax 5-67  
 mi\_qual\_setreopt() function,  
     syntax 5-68  
 mi\_qual\_stratnum() function,  
     syntax 5-69  
 mi\_register\_callback() function 3-8  
 mi\_routine\_exec() function 3-26  
 mi\_row\_create() function 3-49  
 MI\_ROW\_DESC structure 5-11  
 mi\_scan\_forupdate() function  
     syntax 5-70  
 mi\_scan\_isolevel() function  
     syntax 5-71  
     usage 3-11  
 mi\_scan\_locktype() function  
     syntax 5-73  
     usage 3-11



mi\_scan\_nprojs() function  
     syntax 5-74  
     usage 5-76  
 mi\_scan\_projs() function  
     syntax 5-76  
 mi\_scan\_qual() function  
     syntax 5-77  
 mi\_scan\_setuserdata() function  
     syntax 5-78  
     usage 3-7  
 mi\_scan\_table() function  
     syntax 5-80  
 mi\_scan\_userdata() function  
     syntax 5-81  
     usage 3-7  
 MI\_SQL exception level 5-87  
 mi\_switch\_mem\_duration()  
     function 3-5  
 mi\_tab\_amparam() function  
     example 3-22  
     syntax 5-82  
 mi\_tab\_check\_is\_recheck() function  
     syntax 5-84  
     usage 4-19  
 mi\_tab\_check\_msg() function,  
     syntax 5-86  
 mi\_tab\_check\_set\_ask() function  
     syntax 5-89  
     usage 4-19  
 mi\_tab\_id() function, syntax 5-92  
 mi\_tab\_isindex() function,  
     syntax 5-92  
 mi\_tab\_isolevel() function  
     syntax 5-93  
     usage 3-11  
 mi\_tab\_keydesc() function  
     example 3-27  
     syntax 5-94  
 mi\_tab\_mode() function  
     syntax 5-94  
     usage 3-11  
 mi\_tab\_name() function  
     syntax 5-97  
 mi\_tab\_nextrow() function,  
     syntax 5-98  
 mi\_tab\_niorows() function  
     syntax 5-100  
     usage 3-21, 3-44

mi\_tab\_nparam\_exist() function  
     example 3-22  
     syntax 5-98, 5-101  
 mi\_tab\_numfrags() function  
     syntax 5-102  
     using to catch SQL error 3-10  
 mi\_tab\_owner() function  
     syntax 5-103  
 mi\_tab\_param\_exist() function  
     example 3-22  
     syntax 5-104  
 mi\_tab\_partnum() function,  
     syntax 5-105  
 mi\_tab\_rowdesc() function,  
     syntax 5-106  
 mi\_tab\_setnextrow() function,  
     syntax 5-107  
 mi\_tab\_setniorows() function  
     syntax 5-109  
     usage 3-21, 3-44  
 mi\_tab\_setuserdata() function  
     syntax 5-111  
     usage 3-7  
 mi\_tab\_spaceloc() function,  
     syntax 5-113  
 mi\_tab\_spacename() function,  
     syntax 5-114  
 mi\_tab\_spacetype() function  
     syntax 5-116  
     usage 3-17  
 mi\_tab\_update\_stat\_mode()  
     function, syntax 5-118  
 mi\_tab\_userdata() function  
     syntax 5-119  
     usage 3-7  
 mi\_transition\_type() function 3-50  
 Multiple indexes, example 5-82  
 Multiple-row read-write  
     example 3-20, 3-45  
     get next row for 5-107  
     number in memory 5-100  
     setup 3-21, 3-44, 5-109

## N

New features of this  
     product Intro-5  
 NOT VARIANT routine modifier,  
     requirement for 2-16

## O

oncheck utility  
     documenting output from 3-56  
     implementing 4-16  
     options 4-17  
     output for 5-86  
     purpose-function flow 4-12  
     repairing an index 4-18  
 ONCONFIG file setting for  
     sbspace 3-14  
 On-line manuals Intro-15  
 onspaces utility  
     creating storage spaces with 2-19,  
         2-21  
     extspace creation 2-21  
     required for sbspace  
         fragments 3-53  
     sbspace creation 2-20  
 Operator class  
     creating functions for 2-14  
     default 2-18, 6-13  
     defined 1-14, 1-17, 2-14  
     for index key 5-33  
     for multiple data types 3-24  
 NOT VARIANT  
     requirement 2-16  
     parallel execution with 3-42  
     privilege needed 2-17  
     strategy function 2-15  
     support function 2-15  
 Optimization 3-39  
 OUT keyword  
     defined 3-31  
     setting 5-63

## P

Parallel execution 2-11  
Parallelizable purpose functions 3-42  
Parallelizable purpose functions, requirements for 3-42  
PARALLELIZABLE routine  
    modifier 2-11, 3-41  
Parallelizable UDR  
    defined 3-41  
    restrictions on 3-8  
Performance considerations  
    building indexes efficiently 3-20  
    creating parallelizable UDRs 2-11  
    optimizing queries 2-8  
    returning keys as rows 3-43  
    returning multiple rows 3-44  
PER\_COMMAND memory 3-5  
PER\_ROUTINE memory 3-5  
PER\_STATEMENT memory 3-5  
Platform icons Intro-8  
Printed manuals Intro-15  
Product icons Intro-8  
Program group  
    Documentation notes Intro-17  
    Release notes Intro-17  
Purpose flags  
    adding and deleting 6-5  
    list of 6-12  
Purpose functions  
    adding, changing, and dropping 6-5  
    characteristics of 1-8  
    choosing and writing 2-5  
    defined 1-14  
    flow diagrams 4-3  
    for SQL statements 4-3  
    naming 4-13  
    parallel execution 3-42  
    parallel-execution indicator 6-14  
    registering 2-10  
    registering as parallelizable 2-11  
    setting names for 6-14  
    SQL errors from 3-52  
    syntax reference 4-13

Purpose values  
    adding, changing, and dropping 6-5  
    valid settings 6-12  
Purpose, defined 6-10

## Q

Qualification  
    Boolean 5-43  
    column number in 5-45  
    constant value in 5-48  
    defined 3-29  
    host variable needed 5-55  
    multiple indexes in 3-35  
    NOT operator in 5-64  
    NULL constant in 5-52, 5-53  
    OUT value needed 5-55, 5-63, 5-67  
    OUT value, setting 5-67  
    outer join in 5-57  
    routine identifier for 5-58  
    simple predicate 5-62  
Qualification descriptor  
    accessor functions 5-9  
    array size 5-65  
    changed for rescan 5-75  
    complex 3-29  
    defined 3-29  
    nested structure 3-29  
    NULL-valued pointer to 5-77  
    retrieving 5-77  
    retrieving pointer to 5-77  
Query  
    complex examples 3-36  
    privilege to execute function in 2-17  
query  
    complex examples 3-36  
Query plan  
    components 4-35  
    cost 4-34  
    defined 3-39

## R

Related reading Intro-17  
Release notes Intro-16  
Reoptimize 5-68  
rofferr utility Intro-15  
Row descriptor  
    description 5-11  
    retrieving 5-106  
Row-ID descriptor 5-12  
Row, creating from source data 3-49

## S

Sample-code conventions Intro-13  
SBSPACENAME parameter 3-14  
Sbspaces  
    creating 2-19  
    creating a default 3-14  
    creating for fragmentation 2-22, 3-53  
    enabling logging 3-15  
    retrieving the name 5-114  
    specifying logging with onspaces utility 3-16  
    using the default 2-20  
Scan  
    cleanup 4-26  
    fetch routine 4-27  
    isolation level for 3-11, 5-13  
    lock type for 3-11, 5-13  
    setup 4-14  
Scan descriptor  
    accessor functions for 5-13  
    NULL-valued pointer in 5-77  
    relationship to SELECT clause 3-29  
    user data 3-5  
SELECT statement  
    defined 3-28  
    INTO TEMP clause 3-42  
    parallel execution 3-42  
    purpose functions for 4-14, 4-26, 4-27, 4-34  
    purpose-function flow 4-5  
Simple predicate, defined 3-30  
Software dependencies Intro-3

SQL code Intro-13

SQL errors

- avoiding 2-23
- causes of 3-52
- missing purpose function 4-23, 4-30, 4-40
- unsupported storage space 3-14

SQL statements

- executing inside access method 3-7
- extensions 1-12
- for data definition 3-12
- for data retrieval and manipulation 3-46
- See also entry for a keyword.*

SQLSTATE status value 5-86

Statistics descriptor, accessor functions for 5-15

Storage-space type

- access-method support for 3-12
- retrieving 5-116

stores\_demo database Intro-4

Strategy functions. *See* Operator class.

Structured Query Language (SQL). *See* SQL statements.

Subquery, purpose function for 4-33

superstores\_demo database Intro-4

Support functions. *See* Operator class.

Syntax conventions

- description of Intro-9
- example diagram Intro-11

Syntax diagrams, elements in Intro-9

sysams system catalog table

- columns in 6-11
- setting values in 6-3

sysindexes system catalog table

- statistics for 4-38

sysindices system catalog table

- adding an index 4-21
- deleting an index 4-25
- setting clust value 5-24
- setting leaves value 5-28
- setting levels value 5-27
- setting nunique 5-29

systables system catalog table

- statistics for 4-38

System catalog tables

- querying 3-7
- See also individual table names.*

System requirements

- database Intro-3
- software Intro-3

## T

Table

- mode, determining 5-95
- owner 5-103

Table descriptor

- accessor functions for 5-16
- defined 3-12
- retrieving a pointer to 5-80

Tape-device extspace 2-21

Testing 2-18

Tip icons Intro-7

Transaction management

- determining commit success 3-50
- for sbspaces 3-16

## U

UDR

- defined 1-8
- executing 3-26

UNIQUE keyword, enforcing 3-46

UNIX operating system

- default locale for Intro-4

UPDATE statement

- am\_delete purpose function 4-23
- am\_insert purpose function 4-29
- am\_update purpose function 4-40
- parallel execution of 3-42
- purpose-function flow 4-6

UPDATE STATISTICS statement

- described 3-40
- purpose function for 4-38

User data

- declaring structure for 3-6
- defined 3-6
- for scan
- retrieving 5-81
- storing 5-78
- for statement
- retrieving 5-119
- storing 5-111
- table-state memory 5-111

User guide 3-51

Users, types of Intro-3

USING clause

- configuration parameters in 3-19, 5-16
- specifying access method 2-21
- specifying alternative index 3-21

## V

Variant function, defined 2-16

## W

Warning icons Intro-7

WHERE clause

- defined 3-28
- qualifications in 3-29, 3-33
- See also* Qualification.

Windows NT

- default locale for Intro-4

## X

X/Open compliance level Intro-18

