IBM Informix User-Defined Routines and Data Types

Developer's Guide

Version 9.4
March 2003
Part No. CT1T0NA

Note:
Before using this information and the product it supports, read the information in the appendix entitled "Notices."

Table of Contents

Introduction

In This Introduction

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

About This Manual

This manual describes how to define new data types and enable user-defined routines (UDRs) to extend IBM Informix Dynamic Server. It describes the tasks you must perform to extend operations on data types, to create new casts, to extend operator classes for secondary-access methods, to write opaque data types, and to create and register routines.

Types of Users

This manual is written for the following users:

- Database-application programmers
- DataBlade module developers

This manual assumes that you have the following background:

- A working knowledge of your computer, your operating system, and the utilities that your operating system provides
- Experience working with relational databases or exposure to database concepts
- Experience with computer programming

If you have limited experience with relational databases, SQL, or your operating system, refer to the *Getting Started Guide* for a list of supplementary titles.

Software Dependencies

This manual assumes that you are using IBM Informix Dynamic Server, Version 9.4, as your database server.

Assumptions About Your Locale

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

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 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 *IBM Informix GLS User's Guide*.

Demonstration Databases

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

- The **stores_demo** database illustrates a relational schema with information about a fictitious wholesale sporting-goods distributor. Many examples in IBM Informix manuals are based on the **stores_demo** database.
- The **superstores_demo** database illustrates an object-relational schema. The **superstores_demo** database includes 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 *IBM Informix DB-Access User's Guide*. For descriptions of the databases and their contents, see the *IBM Informix Guide to SQL: Reference*.

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

New Features

The following table provides information about the new features for IBM Informix Dynamic Server, Version 9.4, that this manual covers. To go to the desired page, click a blue hyperlink. For a description of all new features, see the *Getting Started Guide*.

Extensibility Enhancements

Version 9.4 includes the following improvements in the area of extensibility.

New Features	Reference
Using an iterator function in the FROM clause of a SELECT statement	"Using an Iterator Function" on page 4-11
Naming the return parameters of a UDR	"Naming Return Parameters" on page 4-10
Using multiple OUT parameters and statement local variables	"Using OUT Parameters and Statement-Local Variables (SLVs)" on page 4-8
Setting the collation sequence at runtime (multinationalization)	"Handling Locale-Sensitive Data" on page 10-29
HDR support for extended types	"Writing the Routine" on page 4-22 "Using a UDR With HDR" on page 4-34

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
- Sample-code conventions

Typographical Conventions

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

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

(1 of 2)

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

(2 of 2)

Tip: *When you are instructed to “enter” characters or to “execute” a command, immediately press* RETURN *after the entry. When you are instructed to “type” the text or to “press” other keys, no* RETURN *is required.*

Icon Conventions

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

Comment Icons

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

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

Feature, Product, and Platform Icons

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

Icon	Description
C	Identifies information that relates to C routines
Ext	Identifies information that relates to routines, that is, UDRs written either in C or Java language
GLS	Identifies information that relates to the IBM Informix Global Language Support (GLS) feature
Java	Identifies information that relates to Java routines
SPL	Identifies information that relates to the Stored Procedure Language
UNIX	Identifies information that is specific to UNIX platforms
WIN	Identifies information that is specific to the Windows 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
ANSI	Identifies information that is specific to an ANSI-compliant database

This icons can apply to a row in a table, one or more paragraphs, or an entire section. A ♦ symbol indicates the end of the compliance information.

Sample-Code Conventions

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

```
CONNECT TO stores_demo
...

DELETE FROM customer
   WHERE customer_num = 121
...

COMMIT WORK
DISCONNECT CURRENT
```

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

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

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

Additional Documentation

IBM Informix Dynamic Server documentation is provided in a variety of formats:

- **Online manuals.** You can obtain online manuals at the IBM Informix Online Documentation site at http://www.ibm.com/software/data/informix/pubs/library/. This site enables you to print chapters or entire books.
- **Online help.** This facility provides context-sensitive help, an error message reference, language syntax, and more.
- **Documentation notes and release notes.** Documentation notes, which contain additions and corrections to the manuals, and release notes are located in the directory where the product is installed.

  Please examine these files because they contain vital information about application and performance issues. The following table describes these files.

UNIX

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

Online File	Purpose
ids_creating_udts__docnotes_9.40.html	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.
ids_release_notes_9.40.html	The release notes file describes feature differences from earlier versions of IBM Informix products and how these differences might affect current products. This file also contains information about any known problems and their workarounds.
ids_machine_notes_9.40.txt	The machine notes file describes any special actions that you must take to configure and use IBM Informix products on your computer. Machine notes are named for the product described.

♦

Windows

The following items appear in the **Informix** folder. To display this folder, choose **Start→Programs→Informix → Documentation Notes or Release Notes** from the task bar.

Program Group Item	**Description**
Documentation Notes	This item includes additions or corrections to manuals with information about features that might not be covered in the manuals or that have been modified since publication.
Release Notes	This item describes feature differences from earlier versions of IBM 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 platforms. ♦

- IBM Informix software products provide ASCII files that contain all of the error messages and their corrective actions. For a detailed description of these error messages, refer to *IBM Informix Error Messages* in the IBM Informix Online Documentation site at http://www.ibm.com/software/data/informix/pubs/library/.

UNIX

To read the error messages on UNIX, you can use the **finderr** command to display the error messages online. ♦

WIN

To read error messages and corrective actions on Windows, use the **Informix Error Message** utility. To display this utility, choose **Start→Programs→Informix** from the task bar. ♦

Related Reading

For a list of publications that provide an introduction to database servers and operating-system platforms, refer to your *Getting Started Guide* manual.

Compliance with Industry Standards

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

IBM Welcomes Your Comments

To help us with future versions of our manuals, let us know about any corrections or clarifications that you would find useful. Include the following information:

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

Send electronic mail to us at the following address:

docinf@us.ibm.com

This address is reserved for reporting errors and omissions in our documentation. For immediate help with a technical problem, contact Technical Support at tsmail@us.ibm.com.

Chapter

Extending the Database Server

1

In This Chapter

This manual discusses extending IBM Informix Dynamic Server by using *user-defined routines* (UDRs) and *user-defined data types* (UDTs). You can use UDRs and never use a UDT. Conversely, you can use UDTs and never use UDRs. However, many of the ways that you extend data types require that you write routines to support those extensions.

This chapter summarizes the organization of the chapters in this book and describes which portion of the book you will need to use, depending on your goals.

Creating User-Defined Routines

Extending the database server frequently requires that you create UDRs to support the extensions. A *routine* is a collection of program statements that perform a particular task. A UDR is a routine that you create that can be invoked in an SQL statement, by the database server, or from another UDR.

The next three chapters in this manual discuss the basic aspects of the creation and use of UDRs:

- Chapter 2, "Using a User-Defined Routine"
- Chapter 3, "Running a User-Defined Routine"
- Chapter 4, "Developing a User-Defined Routine"

The Informix database server supports UDRs in the following languages:

- Stored Procedure Language (SPL)
- The C programming language
- The Java programming language

Extending Built-In Data Types

Built-in data types are provided by the database server. The database server already has functions for retrieving, storing, manipulating, and sorting built-in data types.

You can extend built-in data types in the following ways:

- Creating complex data types based on built-in data types
- Creating UDTs (distinct and opaque data types)
- Extending the operations that are allowed for both built-in data types and extended data types

Chapter 5, "Extending Data Types," describes the data type system that the database uses and documents how to extend the database server by building UDTs that are based on built-in data types. The *IBM Informix Database Design and Implementation Guide* also discusses UDTs that are based on built-in data types.

Extending Operators

When you build a UDT, either by extending a built-in data type or by creating an opaque data type, you must provide for the operations that the data type uses. An *operation* is a task that the database server performs on one or more values.

You can write special-purpose routines that extend the built-in operations of the database. The manual discusses the following specific types of operators in detail:

- Arithmetic and relational operators

  The database server provides operator symbols (+, -, =, > and so on) and built-in functions such as **cos()** and **abs()**. You can extend these operators for extended data types.

  Chapter 6, "Extending Operators and Built-In Functions," discusses general aspects of extending an operation and describes how to extend operator symbols and built-in functions.

- Casts

  The database server provides casts for the built-in data types. When you use UDTs, you usually need to provide casts.

  Chapter 7, "Creating User-Defined Casts," describes how to create casts. The *IBM Informix Database Design and Implementation Guide* discusses how to use casts.

- Aggregates

  An aggregate produces one value that summarizes some aspect of a selected column; for example, the average or the count. You can extend aggregates in two ways:

  - Create a new aggregate, such as an aggregate that provides the sum of the square of each value in the column.
  - Extend an existing aggregate, such as AVG or COUNT, to include data types that you have defined.

  Creating a user-defined aggregate and extending an existing aggregate for extended data types require different techniques. For information about both techniques, refer to Chapter 8, "Creating User-Defined Aggregates."

Building Opaque Data Types

An *opaque data type* is an atomic, or fundamental, data type that you define for the database. The database server has no information about the opaque data type until you provide routines that describe it. As you build an opaque data type, you need to consider the following topics:

- How the information in the opaque data type is organized
- How to store and retrieve the data type
- What the standard operations mean with respect to the opaque data type:
  - What does it mean to add two pieces of data? Is it even possible to add the data?
  - When is one data item larger than another?
  - Can you relate this data to built-in data types?

- What unique operations this data has:
  - Does this data type allow you to find a picture?
  - Can you say that one data item is inside another?

Chapter 9, "Creating an Opaque Data Type," describes the basic steps for creating an opaque data type. Chapter 10, "Writing Support Functions," describes the support functions that an opaque data type uses.

Creating an opaque type and all of the routines that are required to support it is a major task. Theoretically, you could sit down and write all of the required routines. However, it is recommended that you use the IBM Informix DataBlade Developer's Kit (DBDK) because DBDK enforces standards that facilitate migration between different versions of the database server.

A *DataBlade module* is a group of database objects and supporting code that manages user-defined data or adds new features. A DataBlade module can include extended data types, routines, casts, aggregates, access methods, SQL code, client code, and installation programs. DataBlade modules that support various special-purpose opaque data types are provided. To find out what DataBlade modules are available, contact your sales representative.

Extending Operator Classes

An operator class is a set of functions that is associated with building an index.

Chapter 11, "Extending an Operator Class," describes how to create a user-defined operator class and how to extend an existing operator class.

Routine Management

Chapter 12, "Managing a User-Defined Routine," covers the following topics:

- Assigning Execute privilege to a UDR
- Reloading a UDR
- Altering a UDR
- Dropping a UDR

Chapter 13, "Improving UDR Performance," discusses ways that you can optimize the performance of your UDR.

Chapter

Using a User-Defined Routine

2

In This Chapter

This chapter introduces user-defined routines (UDRs) and covers the following topics:

- User-Defined Routines
- Tasks That You Can Perform with User-Defined Routines

User-Defined Routines

A UDR can either return values or not, as follows:

- A *user-defined function* returns one or more values and therefore can be used in SQL expressions.

  Use the CREATE FUNCTION statement to register the UDR in the system catalog tables.
- A *user-defined procedure* is a routine that does not return any values. You cannot use a procedure in SQL expressions because it does not return a value.

  Use the CREATE PROCEDURE statement to register the UDR in the system catalog tables.

SPL Routines

Stored Procedure Language (SPL) is part of the database server. Many of the examples in this book are shown in SPL because it is simple to use and requires no support outside the database server.

SPL provides flow-control extensions to SQL. An *SPL routine* is a UDR that is written in SPL and SQL. The body of an SPL routine contains SQL statements and flow-control statements for looping and branching. For information on the syntax of SPL statements, see the *IBM Informix Guide to SQL: Syntax*. For an explanation of how to use SPL statements, refer to the *IBM Informix Guide to SQL: Tutorial*.

The database server parses and optimizes an SPL routine and stores it in the system catalog tables in executable format. If possible, use SPL routines for SQL-intensive tasks.

For more information, see "Creating an SPL Routine" on page 4-26.

External-Language Routines

An *external-language routine* is a UDR that is written in an external language. The body of an external-language routine contains statements for operations such as flow control and looping, as well as special Informix library calls to access the database server. Therefore, you must use the appropriate compilation tool to parse and compile an external-language routine into an executable format.

The database server supports UDRs written in C and in Java.

- Routines in C

  To write routines in C, you need a C compiler. For information about how to write UDRs in C, refer to the *IBM Informix DataBlade API Programmer's Guide* and the *IBM Informix DataBlade API Function Reference*.

- Routines in Java

  To write Java routines, you must have IBM Informix Dynamic Server with J/Foundation. You also need the Java Development Kit (JDK) to compile your Java routines.

  For information about how to write Java UDRs, refer to the *J/Foundation Developer's Guide*.

***Important:** It is recommended that you use the DBDK to develop UDRs in external languages because the DBDK enforces standards that facilitate migration between different versions of the database server.*

Information About User-Defined Routines

The database server stores information about UDRs in the following system catalog tables:

- The **sysprocedures** system catalog table contains information about the UDR, such as its name, owner, and whether it is a user-defined function or user-defined procedure.
- The **sysprocbody** system catalog table contains the actual code of SPL routines.
- The **sysprocauth** system catalog table contains information on which users of the database server can execute a particular UDR.

The CREATE FUNCTION and CREATE PROCEDURE statements do not provide the actual code that makes up the external routine. Instead, they store information about the external routine (including the name of its executable file) in the **sysprocedures** system catalog table. Therefore, unlike SPL routines, the code for the body of an external routine does *not* reside in the system catalog of the database.

The database server stores information on external languages that it supports for UDRs in the following system catalog tables:

- The **sysroutinelangs** system catalog table contains information about the external languages.
- The **syslangauth** system catalog table contains information on which users of the database server can use a particular external language.

For more information, see "Creating an External-Language Routine" on page 4-28.

Tasks That You Can Perform with User-Defined Routines

You can write UDRs to accomplish the following kinds of tasks:

- Extend support for built-in or UDTs
- Provide the end user with new functionality, called an *end-user routine*

The following sections summarize the tasks that a UDR can perform. For information on how to create a UDR, see Chapter 4, "Developing a User-Defined Routine."

Extending Data Type Support

Dynamic Server provides support for the following kinds of UDRs.

UDR Task	SPL Routines	C Routines	Java Routines	For More Information
Cast function	Yes	Yes	Yes	Chapter 7
Cost function	No	Yes	No	Chapter 13
End-user routine	Yes	Yes	Yes	page 2-17
Iterator function	No	Yes	Yes	Chapter 4
Negator function	Yes	Yes	Yes	Chapter 13
Opaque data type support function	No	Yes	Yes	Chapter 9
Operator function	Yes	Yes	Yes	Chapter 6

(1 of 2)

UDR Task	SPL Routines	C Routines	Java Routines	For More Information
Operator-class function	No	Yes	No	Chapter 11
Parallelizable UDR	No	Yes	Yes	Chapter 13
Statistics function	No	Yes	Yes	Chapter 13
Selectivity function	No	Yes	No	Chapter 13
User-defined aggregate	Yes	Yes	Yes (with some limitations)	Chapter 8

(2 of 2)

Tip: *When you want to perform an iteration in SPL, use the WITH RESUME keywords.*

To extend the support for one of these kinds of functions, you can write your own version of the appropriate function and register it with the database.

Supporting User-Defined Data Types

When you create UDTs, you also provide the following routines:

- Support functions that the database server invokes implicitly to operate on the data types
- Cast functions that the database server can invoke implicitly or that users can specify explicitly in SQL statements to convert data from one data type to another
- Optional operator-class functions that extend an index method, such as B-tree or R-tree, to manage the new type
- Optional additional routines that other support functions or the end user can call

Cast Functions

A *cast* performs a conversion between two data types. The database server allows you to write your own cast functions to perform casts. The following sections summarize how you can extend a cast function for built-in and UDTs. For more information on how to extend casts, refer to Chapter 7, "Creating User-Defined Casts."

Tip: *If a DataBlade module defines a data type, it might also provide cast functions between this data type and other data types in the database. For more information on functions that a specific DataBlade module provides, refer to the user guide for that DataBlade module.*

Casting Between Built-In Data Types

The database server provides *built-in casts* that perform automatic conversions between certain built-in data types. For more information on these built-in casts, refer to the *IBM Informix Guide to SQL: Reference*.

You cannot create user-defined casts to allow conversions between two built-in data types for which a built-in cast does not currently exist. For more information on when you might want to write new cast functions, refer to "Creating a User-Defined Cast" on page 7-5.

Casting Between Other Data Types

You can create *user-defined casts* to perform conversions between most data types, including opaque types, distinct types, row types, and built-in types. You can write cast functions in SPL or in external languages. For example, you can define casts for any of the following UDTs:

- Opaque data types

  You can create casts to convert a UDT to other data types in the database. Developers of opaque data types must also provide functions that serve as cast functions between the internal and external representations of the opaque type. For more information, see Chapter 9, "Creating an Opaque Data Type."

- Distinct data types

  The database server cannot directly compare a distinct type to its source type. However, the database server automatically registers explicit casts from the distinct type to the source type and conversely. Although a distinct type inherits the casts and functions of its source type, the casts and functions that you define on a distinct type are not available to its source type.

- Named row types

  You can create casts to convert a named row data type to another type. For information about how to cast between named row types and unnamed row types, see the *IBM Informix Guide to SQL: Tutorial*.

For more information on how to create and register casts on extended data types, refer to Chapter 7, "Creating User-Defined Casts."

End-User Routines

An *end-user routine* is an SQL-invoked function that the SQL user can include in an SQL statement. Such routines provide special functionality that application users often need. An end-user routine might be as simple as "increase the price of every item from XYZ Corporation by 5 percent" or something far more complicated.

This section summarizes how you can extend an end-user routine that operates on the following data types:

- Built-in data types

  The database server provides many functions that end users can use in SQL statements on built-in data types. These functions are called *built-in functions* to distinguish them from SQL-invoked functions that you define.

  You cannot extend an existing built-in function on a built-in data type that it supports. However, you can perform the following extensions:

  - Define your own end-user routines to provide new or similar functionality.
  - Define a UDR that has the same name as a built-in function but operates on a different built-in data type.

  For more information about built-in functions, see Chapter 6, "Extending Operators and Built-In Functions."

- Extended data types

  You can write an end-user routine on any data type that is registered in the database.

For more information about end-user routines, see "Creating an End-User Routine" on page 2-17.

Aggregate Functions

An *aggregate function* is an SQL-invoked function that takes values that depend on all the rows that the query selects and returns information about these rows. The database server supports aggregate functions that you write, called *user-defined aggregates*. You can write user-defined aggregates in SPL or in external languages.

You can extend an aggregate function for built-in and UDTs, as follows:

- The database server provides built-in aggregate functions, such as COUNT, SUM, or AVG, that operate on built-in data types.

  You cannot create a user-defined aggregate that has the same name as a built-in aggregate and that handles a built-in data type. However, you can define a new aggregate that operates on a built-in data type.
- When you create a UDT, you can write user-defined aggregates to provide aggregates that handle this data type. The database server provides two ways to extend aggregates:
  - Extend a built-in aggregate to handle the data type.

    You overload the support functions for the built-in aggregate.
  - Define a new aggregate.

    You write a user-defined aggregate with a name that is different from any existing aggregate function. You then register a new aggregate in the database.

Tip: *If a DataBlade module defines a data type, it might also provide user-defined aggregate functions on this data type. For more information on functions that a specific DataBlade module provides, refer to the user guide for that DataBlade module.*

For more information about aggregate functions, see Chapter 8, "Creating User-Defined Aggregates." Aggregate functions use the support functions to compute the aggregate result. For information on support functions, see Chapter 10, "Writing Support Functions."

Operator Functions

An *operator function* is an SQL-invoked function that has a corresponding operator symbol (such as '=' or '+'). These operator symbols are used within expressions in an SQL statement.

Operator binding is the implicit invocation of an *operator function* when an *operator symbol* is used in an SQL statement. The database server implicitly maps a built-in operator function name to a built-in operator. For example, you can compare two values for equality in either of the following ways.

Method of Comparison	Operator Used
Operator function	**equal**(value1, value2)
Operator symbol	value1 = value2

The following sections summarize how you can extend an operator on built-in and UDTs. For more information on how to extend operators, refer to Chapter 6, "Extending Operators and Built-In Functions."

Operators on Built-In Data Types

The database server provides operator functions that operate on most built-in data types. For a complete list of operator functions, see Chapter 6, "Extending Operators and Built-In Functions." You *cannot* extend an operator function that operates on a built-in data type.

Operators on User-Defined Data Types

You can extend an existing operator to operate on a UDT. When you define the appropriate operator function, operator binding enables SQL statements to use both the function name and its operator symbol on the UDT. You can write operator functions in SPL or an external language.

For example, suppose you create a data type, called **Scottish**, that represents Scottish names, and you want to order the data type in a different way than the U.S. English collating sequence. You might want the names `McDonald` and `MacDonald` to appear together on a phone list. The default relational operators (for example, =) for character strings do not achieve this ordering.

To treat `Mc` and `Mac` as the same, you can create a **compare()** function that compares two Scottish-name values, treating `Mc` and `Mac` as identical. *Routine overloading* is the ability to use the same name for multiple functions to handle different data types. The database server uses the **compare** (Scottish, Scottish) function when it compares two Scottish-name values. For more information, refer to "Overloading Routines" on page 3-13.

Tip: *The relational operators (such as =) are the operator-class functions of the built-in secondary-access method, the generic B-tree. Therefore, if you redefine the relational operators to handle a UDT, you also enable that type to be used in a B-tree index. For more information, see "Operator-Class Functions" in the following section.*

Operator-Class Functions

An *operator class* is the set of operators that the database server associates with a *secondary-access method* for query optimization and building the index. A secondary-access method (sometimes referred to as an *index-access method*) is a set of database server functions that build, access, and manipulate an index structure such as a B-tree, an R-tree, or an index structure that a DataBlade module provides.

The query optimizer uses an operator class to determine if an index can be considered in the cost analysis of query plans. The query optimizer can consider use of the index for the given query when the following conditions are true:

- An index exists on the particular column or columns in the query.
- For the index that exists, the operation on the column or columns in the query matches one of the operators in the operator class that is associated with the index.

For more information on how to optimize queries with UDRs, refer to "Optimizing a User-Defined Routine" on page 13-3. For more information on how to extend operator classes, refer to "Extending an Existing Operator Class" on page 11-9.

Tip: *If a DataBlade module provides a secondary-access method, it might also provide operator classes with the strategy and support functions. For more information on functions that a specific DataBlade module provides, refer to the user guide for that DataBlade module.*

Operator-Class Functions on Built-In Data Types

The database server provides the default operator class for the built-in secondary-access method, the generic B-tree. These operator-class functions handle the built-in data types. You can write new operator-class functions that operate on built-in data types if you want to do the following:

- Extend the default operator class for the generic B-tree to redefine the ordering scheme that these operators support.

  The **compare()** function implements the ordering scheme for a B-tree index. The strategy functions (**greaterthan()**, **lessthan()**, and so on) let the query optimizer use the index for optimizing SQL statements.

  Because of routine overloading, these functions can have the same name as the functions in the default operator class. For more information, refer to "Overloading Routines" on page 3-13.

- Define a new operator class to provide an entirely new set of operators that operate on the built-in data type.

  You write operator-class functions with names that are different from any existing operating-class functions associated with the secondary-access method. You then register a new operator class that contains these new operators. The query optimizer can choose an index on this data type when the index uses this new operator class and the SQL statement contains one of the operators in this operator class.

Operator Classes on User-Defined Data Types

When you create a opaque data type, you can write operator-class functions to do the following:

- Extend the default operator class for an existing secondary-access method to handle the indexing scheme that these operators support.

  You write operator-class functions with the same names as those in the existing operator class. These functions extend the existing operator class by implementing its indexing scheme on the opaque data type. The query optimizer can choose an index on this data type when the index uses this operator class and the SQL statement contains one of the operators in this operator class.

  Because of routine overloading, these functions can have the same name as the functions in the default operator class. For more information on routine overloading, refer to "Overloading Routines" on page 3-13.

- Define a new operator class to provide an entirely new set of operators that operate on the opaque type.

  You supply the support and strategy functions that the access method requires. These functions define the new operators that the query optimizer can recognize as associated with the secondary-access method. The requirements for the support and strategy functions vary from one access method to another. You must consult the documentation for the access method before defining a new operator class.

Optimization Functions

Optimization functions help the query optimizer determine the most efficient query plan for a particular SQL statement. These optimization functions are as follows.

Optimization Function	Description
Negator function	Specifies the function to use for a NOT condition that involves a Boolean UDR
Cost function	Specifies the cost factor for execution of a particular UDR
Selectivity function	Specifies the percentage of rows for which a Boolean UDR is expected to return `true`
Parallel UDR	A UDR that can be run in parallel and therefore can be run in parallel queries
Statistics function	Creates distribution statistics for a UDT

The database server provides optimization functions for the built-in data types. You can write an optimization function on any UDT that is registered in the database. You cannot extend existing optimization for built-in types through optimization functions.

For more information about optimization functions, see Chapter 13, "Improving UDR Performance."

Opaque Data Type Support Functions

When you define a new opaque data type, you provide support functions that enable the database server to operate on the data type. The database server requires some support functions, and others are optional. The following list shows the standard functions that you define to support opaque data types:

- Text input and output routines
- Binary send and receive routines
- Text import and export routines
- Binary import and export routines

For more information on support functions for opaque data types, refer to Chapter 10, "Writing Support Functions."

Access-Method Purpose Functions

An *access method* is a set of functions that the database server uses to access and manipulate a table or an index. The two types of access methods are as follows:

- Primary-access methods, which create and manipulate tables

  A *primary-access method* is a set of routines that perform all the operations needed to make a table available to a database server, such as create, drop, insert, delete, update, and scan. The database server provides a built-in primary-access method.
- Secondary-access methods, which create and manipulate indexes

  A *secondary-access method* is a set of routines that perform all the operations needed to make an index available to a database server, such as create, drop, insert, delete, update, and scan. The database server provides the B-tree and R-tree secondary-access methods. For information about R-tree indexes, refer to the *IBM Informix R-Tree Index User's Guide*.

DataBlade modules can provide other primary- and secondary-access methods. For more information, refer to the *IBM Informix Virtual-Table Interface Programmer's Guide* and the *IBM Informix Virtual-Index Interface Programmer's Guide*.

Creating an End-User Routine

You can write end-user routines to accomplish the following tasks:

- Encapsulate multiple SQL statements
- Create triggered actions for multiple applications
- Restrict who can read data, change data, or create objects
- Create iterators

Routines also can accomplish tasks that address new technologies, including the following ones:

- Manipulate large objects
- Manage new data domains, such as images, web publishing, and spatial

Encapsulating Multiple SQL Statements

You create a routine to simplify writing programs or to improve performance of SQL-intensive tasks.

Simplifying Programs

A UDR can consolidate frequently performed tasks that require several SQL statements. Both SPL and external languages offer program control statements that extend what SQL can accomplish alone. You can test database values in a UDR and perform the appropriate actions for the values that the routine finds.

By encapsulating several statements in a single routine that the database server can call by name, you reduce program complexity. Different programs that use the same code can execute the same routine, so that you need not include the same code in each program. The code is stored in only one place, eliminating duplicate code.

Simplifying Changes

UDRs are especially helpful in a client/server environment. If a change is made to application code, it must be distributed to every client computer. A UDR resides in the database server, so only database servers need to be changed.

Instead of centralizing database code in client applications, you create UDRs routines to move this code to the database server. This separation allows applications to concentrate on user-interface interaction, which is especially important if multiple types of user interfaces are required.

SPL

Improving Performance Using SPL

Because an SPL routine contains native database language that the database server parses and optimizes as far as possible when you create the routine, rather than at runtime, SPL routines can improve performance for some tasks. SPL routines can also reduce the amount of data transferred between a client application and the database server.

For more information on performance considerations for SPL routines, refer to Chapter 13, "Improving UDR Performance."

Creating Triggered Actions

An SQL *trigger* is a database mechanism that executes an action automatically when a certain event occurs. The event that can trigger an action can be an INSERT, DELETE, or UPDATE statement on a specific table. The table on which the triggered event operates is called the *triggering table*.

An SQL trigger is available to any user who has permission to use it. When the trigger event occurs, the database server executes the trigger action. The actions can be any combination of one or more INSERT, DELETE, UPDATE, EXECUTE PROCEDURE, or EXECUTE FUNCTION statements.

Because a trigger resides in the database and anyone who has the required privilege can use it, a trigger lets you write a set of SQL statements that multiple applications can use. It lets you avoid redundant code when multiple programs need to perform the same database operation. By invoking triggers from the database, a DBA can ensure that data is treated consistently across application tools and programs.

You can use triggers to perform the following actions as well as others that are not found in this list:

- Create an audit trail of activity in the database

  For example, you can track updates to the orders table by updating corroborating information in an audit table.
- Implement a business rule

  For example, you can determine when an order exceeds a customer's credit limit and display a message to that effect.
- Derive additional data that is not available within a table or within the database

  For example, when an update occurs to the quantity column of the items table, you can calculate the corresponding adjustment to the **total_price** column.

For more information on triggers, refer to the *IBM Informix Guide to SQL: Tutorial*.

SPL

Restricting Access to a Table

SPL routines offer the ability to restrict access to a table. For example, if a database administrator grants insert permissions to a user, that user can use ESQL/C, DB-Access, or an application program to insert a row. This situation could create a problem if an administrator wants to enforce any business rules.

Using the extra level of security that SPL routines provide, you can enforce business rules. For example, you might have a business rule that a row must be archived before it is deleted. You can write an SPL routine that accomplishes both tasks and prohibits users from directly accessing the table.

Rather than granting insert privileges, an administrator can force users to execute a routine to perform the insert.

Creating Iterators

An *iterator function* returns an active set of items. Each iteration of the function returns one item of the active set. To execute an iterator function, you must associate the function with a database cursor.

The database server does not provide any built-in iterator functions. However, you can write iterator functions and register them with the ITERATOR routine modifier. For more information, see "Using an Iterator Function" on page 4-11.

Invoking a User-Defined Routine

A UDR can be invoked either explicitly or implicitly. For more information, see Chapter 3, "Running a User-Defined Routine."

Explicit Invocation

You can use the EXECUTE PROCEDURE and EXECUTE FUNCTION statements to execute a UDR from:

- A UDR
- DB-Access
- A client application (such as an ESQL/C application)

In addition, you can use a user-defined function in an SQL expression in the SELECT clause or WHERE clause. You *cannot* use a procedure in an SQL expression because a procedure does not return a value.

Implicit Invocation

The database server can invoke a UDR implicitly for following reasons.

Implicit Call of UDR	UDR Called
Built-in operator binding	Operator function
Implicit casting	Implicit cast function
Opaque-type processing	Opaque-type support functions and statistics functions
Query processing	Optimization functions and operator-class functions

Chapter

Running a User-Defined Routine

3

In This Chapter

This chapter discusses the following topics:

Invoking a UDR in an SQL Statement

You can invoke a UDR from within an SQL statement in the following ways:

- You can directly invoke a UDR with the EXECUTE FUNCTION or the EXECUTE PROCEDURE statement.
- You can invoke a user-defined function within an expression.

Invoking a UDR with an EXECUTE Statement

For details about the syntax of the EXECUTE FUNCTION and EXECUTE PROCEDURE statements, see the *IBM Informix Guide to SQL: Syntax*. For more information about creating UDRs, refer to Chapter 4, "Developing a User-Defined Routine."

Invoking a Function

Suppose **result** is a variable of type INTEGER. The following example shows how to register and invoke a C user-defined function called **nFact()** that returns N-factorial (n!):

```
CREATE FUNCTION nFact(arg1 n)
   RETURNING INTEGER;
   SPECIFIC nFactorial
   WITH (HANDLESNULLS, NOT VARIANT)
   EXTERNAL NAME '/usr/lib/udtype2.so(nFactorial)'
   LANGUAGE C;
EXECUTE FUNCTION nFact (arg1);
```

Using a SELECT Statement in a Function Argument

As another example, suppose you create the following type hierarchy and functions:

```
CREATE ROW TYPE emp_t
    (name VARCHAR(30), emp_num INT, salary DECIMAL(10,2));
CREATE ROW TYPE trainee_t (mentor VARCHAR(30)) UNDER emp_t;
CREATE TABLE trainee OF TYPE trainee_t;
INSERT INTO  trainee VALUES ('sam', 1234, 44.90, 'joe');

CREATE FUNCTION func1 (arg1 trainee_t) RETURNING row;
DEFINE newrow trainee_t;
LET newrow = ROW( 'sam', 1234, 44.90, 'juliette');
RETURN newrow;
END FUNCTION;
```

The following EXECUTE FUNCTION statement invokes the **func1()** function, which has an argument that is a query that returns a row type:

```
EXECUTE FUNCTION
   func1 ((SELECT * from trainee where emp_num = 1234)) ...
```

Important: *When you use a query for the argument of a user-defined function invoked with the EXECUTE FUNCTION statement, you must enclose the query in an additional set of parentheses.*

Invoking a Procedure

The following EXECUTE PROCEDURE statement invokes the **log_compare()** function:

```
EXECUTE PROCEDURE log_compare (arg1, arg2)
```

Invoking a User-Defined Function in an Expression

You can invoke a user-defined function in an expression in the select list of a SELECT statement, or in the WHERE clause of an INSERT, SELECT, UPDATE, or DELETE statement.

For example, with the factorial function described in "Invoking a Function" on page 3-4, you might write the following SELECT statement:

```
SELECT * FROM tab_1 WHERE nFact(col1) > col3
```

Invoking a Function That Is Bound to an Operator

Functions that are bound to specific operators get invoked automatically without explicit invocation. Suppose an **equal()** function exists that takes two arguments of **type1** and returns a Boolean. If the equal operator (=) is used for comparisons between two columns, **col_1** and **col_2**, that are of **type1**, the **equal()** function is invoked automatically. For example, the following query implicitly invokes the appropriate **equal()** function to evaluate the WHERE clause:

```
SELECT * FROM tab_1
WHERE col_1 = col_2
```

The preceding query evaluates as though it had been specified as follows:

```
SELECT * FROM tab_1
WHERE equal (col_1, col_2)
```

Invoking a UDR in an SPL Routine

You use the CALL statement only to invoke a UDR from within an SPL program. You can use CALL to invoke both user-defined functions and user-defined procedures, as follows:

- When you invoke a user-defined function with the CALL statement, you must include a RETURNING clause and the name of the value or values that the function returns.

  The following statement invokes the **equal()** function:

  ```
  CALL equal (arg1, arg2) RETURNING result
  ```

  You cannot use the CALL statement to invoke a user-defined function that contains an OUT parameter.

- A RETURNING clause is *never* present when you invoke a user-defined procedure with the CALL statement because a procedure does not return a value.

  The following CALL statement invokes the **log_compare()** procedure:

  ```
  CALL log_compare (arg1, arg2)
  ```

Executing a User-Defined Routine

When you invoke a UDR, the database server must execute it. To execute a UDR in one of these SQL statements, the database server takes the following steps:

1. Calls the query parser
2. Calls the query optimizer
3. Executes the UDR

Parsing the SQL Statement

The query parser breaks the SQL statement into its syntactic parts. If the statement contains a UDR, the query parser performs the following steps on the SQL statement:

- Parses the routine call to obtain the routine signature
- Performs any necessary routine resolution on the UDR calls to determine which UDR to execute

For a description of routine resolution, refer to "Understanding Routine Resolution" on page 3-11.

Optimizing the SQL Statement

Once the query parser has separated the SQL statement into its syntactic parts, the query optimizer can create a *query plan* that efficiently organizes the execution of the SQL statement. The query optimizer formulates a query plan to fetch the data rows that are required to process a query.

For more information, see "Optimizing a User-Defined Routine" on page 13-3.

Executing the Routine

For SPL routines, the routine manager executes the SPL p-code that the database server has compiled and stored in the **sysprocbody** system catalog table.

For routines written in external languages, the *routine manager* executes the UDR in the appropriate language. The routine manager is the specific part of the database server that manages the execution of UDRs.

SPL

Executing an SPL Routine

Unlike a routine in C or Java, whose executable code resides in an external file, the executable code for an SPL routine is stored directly in the **sysprocbody** system catalog table of the database. When you create an SPL routine, the database server parses the SPL routine, compiles it, and stores the executable code in the **sysprocbody** system catalog table. When a statement invokes an SPL routine, the database server executes the SPL routine from the internally-stored compiled code.

When you execute an SPL routine with the EXECUTE FUNCTION, EXECUTE PROCEDURE, or CALL statement, the database server performs the following tasks:

- Retrieves the p-code, execution plan, and dependency list from the system catalog and converts them to binary format
- Parses and evaluates the arguments passed by the EXECUTE FUNCTION, EXECUTE PROCEDURE, or CALL statement
- Checks the dependency list for each SQL statement that will be executed

  If an item in the dependency list indicates that reoptimization is needed, optimization occurs at this point. If an item needed in the execution of the SQL statement is missing (for example, a column or table has been dropped), an error occurs at this time.
- Executes the p-code instructions

An SPL routine with the WITH RESUME clause of the RETURN statement causes multiple executions of the same SPL routine in the same routine sequence. However, an SPL routine does not have access to the user state of its routine sequence.

Executing an External Language Routine

The routine manager performs the following steps to handle execution of external-language routines:

- Loads the external-language executable code
- Creates a routine sequence
- Manages the actual execution of the UDR

Loading an Executable Code into Memory

To execute a UDR written in an external language, the executable code must reside in database server memory. On the first invocation of a UDR, the routine manager loads into memory the file that contains the UDR. The database server locates that file from the **externalname** column in the **sysprocedures** system catalog table.

C

Use the **onstat** command-line utility with the **-g dll** option to view the dynamically loaded libraries in which your UDRs reside. For information about the **onstat** command, refer to the *Administrator's Reference*. ♦

You must install shared libraries and **.jar** files on all database servers that need to run the UDRs, including database servers involved in Enterprise Replication (ER) and High-Availability Data Replication (HDR). The shared object files and **.jar** files need to be installed under the same absolute path name.

After the routine manager has loaded an external-language routine into memory, this file remains in memory until it is explicitly unloaded or the database server is shut down. For more information, see "Dropping a User-Defined Routine" on page 12-11.

Creating the Routine Sequence

The *routine sequence* contains dynamic information that is necessary to execute an instance of the routine in the context of an SQL or SPL statement. The routine manager receives information about the UDR from the query parser. With this information, the routine manager creates a routine sequence for the associated UDR. Each instance of a UDR, implicit or explicit, in an SQL or SPL statement creates at least one independent routine sequence. Sometimes, a routine sequence consists of the single call to the UDR, as follows:

```
EXECUTE PROCEDURE update_log(log_name)
```

However, often a UDR can be invoked on more than a row. For example, in the following SELECT statement, the **running_avg()** function is called for *each* matching row of the query:

```
SELECT name, running_avg(price)
FROM stock_history
WHERE running_avg(price) > 5.00
```

In the preceding query, the WHERE clause causes the database server to invoke two functions: the **running_avg()** UDR and, implicitly, the built-in **greaterthan()** function. The database server calls the **running_avg()** function for each row that it processes and executes the function in its own separate routine sequence, independent from the routine sequence for **running_avg()** in the SELECT clause.

For a fragmented **stock_history** table, the routine instance in the WHERE clause might have more than one routine sequence if **running_avg()** was created with the PARALLELIZABLE option. For example, if the **stock_history** table has four fragments, the database server uses five routine sequences to execute **running_avg()** in the WHERE clause:

- One routine sequence for the primary thread
- Four routine sequences, one for each fragment in the table, for the secondary PDQ threads

Each individual call to a UDR within a routine sequence is called a *routine invocation*.

The routine manager creates a *routine-state space* to hold UDR information that the routine sequence needs. The database server obtains this information from the query parser and passes it to the routine manager. The routine-state space holds the following information about a UDR:

- Argument information:
  - The number of arguments passed to the UDR
  - The data types of each argument
- Return-value information (user-defined functions only):
  - The number of return values passed from the UDR
  - The data type of each return value

Important: *This argument information in the routine-state space does not include the actual argument values. It contains information only about the argument data types.*

The routine-state space also includes private *user-state information* for use by later invocations of the routine in the same routine sequence. The UDR can use this information to optimize the subsequent invocations. The user-state information is stored in the routine-state space.

C

For a C UDR, the routine manager creates an MI_FPARAM structure to hold information about routine arguments and return values. The MI_FPARAM structure that the routine manager creates to hold information about routine arguments and return values can also contain a pointer to user-state information. For more information, see the chapter on how to execute UDRs in the *IBM Informix DataBlade API Programmer's Guide*. ♦

Java

For a Java UDR, the **UDREnv** interface provides most of the information that MI_FPARAM provides for a C UDR. This interface has public methods for returning the SQL data types of the return values, for iterator use, and for the user-state pointer. The interface also provides facilities for logging and tracing. For more information, refer to the *J/Foundation Developer's Guide*. ♦

Managing Routine Execution

After the routine sequence exists, the routine manager can execute the UDR, as follows:

1. It pushes arguments onto the stack for use by the routine.
2. It invokes the routine.
3. It handles the return of any UDR result.

All invocations of the same UDR within the same routine sequence have access to the same routine-state space.

Understanding Routine Resolution

You can assign the same name to different UDRs, as long as the routine signature is unique. It is the *routine signature* that uniquely identifies a UDR, not the routine name alone. A routine that has many versions is called an *overloaded routine*. When you invoke an overloaded routine, the database server must uniquely identify which routine to execute. This process of identifying the UDR to execute is called *routine resolution*.

This section provides the following information about routine resolution:

- What is the routine signature?
- What is an overloaded routine?

- How to you create overloaded routines?
- What is the routine-resolution process?

You need to understand the routine-resolution process to:

- Obtain the data results that you expect from a UDR.
- Avoid unintentional side effects if the wrong UDR executes.
- Understand when you need to write an overloaded routine.

The Routine Signature

The *routine signature* uniquely identifies the routine. The query parser uses the routine signature when you invoke a UDR. The routine signature includes the following information:

- The type of routine: procedure or function
- The routine name
- The number of parameters
- The data types of the parameters
- The order of the parameters
- ANSI The owner name ♦

Important: *The signature of a routine does not include return types. Consequently, you cannot create two user-defined functions that have the same signature but different return types.*

Using ANSI and Non-ANSI Routine Signatures

In a database that is *not* ANSI compliant, the routine signature must be unique within the entire database, irrespective of the owner. If you explicitly qualify the routine name with an owner name, the signature includes the owner name as part of the routine name.

ANSI

In an ANSI-compliant database, the routine signature must be unique within the name space of the user. The routine name always begins with the owner, in the following format:

```
owner.routine_name
```

♦

When you register the routine signature in a database with the CREATE FUNCTION or CREATE PROCEDURE statement, the database server stores the routine signature in the **sysprocedures** system catalog table. For more information, see "Registering a User-Defined Routine" on page 4-23.

Using the Routine Signature to Perform DBA Tasks

The database server uses the routine signature when you use SQL statements to perform DBA tasks (DROP, GRANT, REVOKE, and UPDATE STATISTICS) on routines. The signature identifies the routine on which to perform the DBA task. For example, the DROP statement that Figure 3-1 shows uses a routine signature.

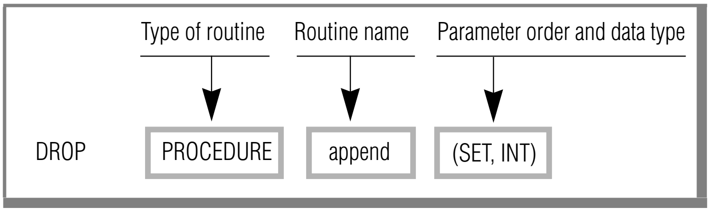


Figure 3-1
Example of Routine Signature

Overloading Routines

Routine overloading refers to the ability to assign one name to multiple routines and specify parameters of different data types on which the routines can operate. Because the database server supports routine overloading, you can register more than one UDR with the same name.

Creating Overloaded Routines

The database server can support routine overloading because it supports *polymorphism*: the ability to have many entities with the same name and to choose the entity most relevant to a particular usage.

You can have more than one routine with the same name but different parameter lists, as in the following situations:

- You create a routine with the same name as a built-in function, such as **equal()**, to process a new UDT.
- You create *type hierarchies*, in which subtypes inherit data representation and functions from supertypes.

- You create *distinct types*, which are data types that have the same internal storage representation as an existing data type, but have different names. Distinct types cannot be compared to the source type without casting. Distinct types inherit UDRs from their source types.

For example, you might create each of the following user-defined functions to calculate the area of different data types (each data type represents a different geometric shape):

```
CREATE FUNCTION area(arg1 circle) RETURNING DECIMAL...
CREATE FUNCTION area(arg1 rectangle) RETURNING DECIMAL....
CREATE FUNCTION area(arg1 polygon) RETURNING DECIMAL....
```

These three CREATE FUNCTION statements create an overloaded routine called **area()**. Each CREATE FUNCTION statement registers an **area()** function for a particular argument type. You can overload a routine so that you have a customized **area()** routine for every data type that you want to evaluate.

The advantage of routine overloading is that you do not need to invent a different name for a routine that performs the same task for different arguments. When a routine has been overloaded, the database server can choose which routine to execute based on the arguments of the routine when it is invoked.

Assigning a Specific Routine Name

Due to routine overloading, the database server might not be able to uniquely identify a routine by its name alone. When you register an overloaded UDR, you can assign a *specific name* to a particular signature of a routine. The specific name serves as a shorthand identifier that refers to a particular overloaded version of a routine.

A specific name can be up to 128 characters long and is unique in the database. Two routines in the same database cannot have the same specific name, even if they have different owners. To assign a unique name to an overloaded routine with a particular data type, use the SPECIFIC keyword when you create the routine. You specify the specific name, in addition to the routine name, in the CREATE PROCEDURE or CREATE FUNCTION statement.

You can use the specific name instead of the full routine signature in the following SQL statements:

- ALTER FUNCTION, ALTER PROCEDURE, ALTER ROUTINE
- DROP FUNCTION, DROP PROCEDURE, DROP ROUTINE
- GRANT
- REVOKE
- UPDATE STATISTICS

For example, suppose you assign the specific name **eq_udtype1** to the UDR that the following statement creates:

```
CREATE FUNCTION equal (arg1 udtype1, arg2 udtype1)
   RETURNING BOOLEAN
   SPECIFIC eq_udtype1
   EXTERNAL NAME
     '/usr/lib/udtype1/lib/libbtype1.so(udtype1_equal)'
   LANGUAGE C
```

You can then refer to the UDR with either the routine signature or the specific name. The following two GRANT statements are equivalent:

```
GRANT EXECUTE ON equal(udtype1, udtype1) to mary
GRANT EXECUTE ON SPECIFIC eq_udtype1 to mary
```

Specifying Overloaded Routines During Invocation

When you invoke an overloaded routine, you *must* specify an argument list for the routine. If you invoke an overloaded routine by the routine name only, the routine-resolution process fails because the database server cannot uniquely identify the routine without the arguments.

For example, the following SQL statement shows how you can invoke the overloaded **equal()** function on a new data type, **udtype1**:

```
CREATE TABLE atest (col1 udtype1, col2 udtype1, ...)
...
SELECT * FROM employee WHERE equal(col1, col2)
```

Because the **equal()** function is an operator function bound to the equal (=) symbol, you can also invoke the **equal()** function with an argument on either side of the operator symbol, as follows:

```
SELECT * FROM employee WHERE col1 = col2
```

SPL

In SPL, the following statements show ways that you can invoke the **equal()** function:

```
EXECUTE FUNCTION equal(col1, col2) INTO result

CALL equal(col1, col2) RETURNING result
LET result = equal(col1, col2)
```

♦

For more information about overloaded operator functions, refer to Chapter 6, "Extending Operators and Built-In Functions."

Overloading Built-In SQL Functions

The database server provides built-in SQL functions that provide some basic mathematical operations. You can overload most of these built-in SQL functions. For example, you might want to create a **sin()** function on a UDT that represents complex numbers. For a complete list of built-in SQL functions that you can overload, see "Built-In Functions" on page 6-7.

The Routine-Resolution Process

Routine resolution refers to the process that the database server uses when you invoke a routine. The database server also invokes routine resolution when another routine invokes a UDR. If the routine is overloaded, the query parser resolves the UDR from the system catalog tables, based on its routine signature. The parser performs any routine resolution necessary to determine which UDR to execute.

The Routine Signature

When a user or another routine invokes a routine, the database server searches for a routine signature that matches the routine name and arguments. If no exact match exists, the database server searches for a substitute routine, as follows:

1. When several arguments are passed to a routine, the database server searches the **sysprocedures** system catalog table for a routine whose signature is an exact match for the invoked routine:
   a. The database server checks for a candidate routine that has the same data type as the leftmost argument.

      For more information, see "Candidate List of Routines" on page 3-17.
   b. If no exact match exists for the first argument, the database server searches the candidate list of routines using a precedence order of data types.

      For more information, see "Precedence List of Data Types" on page 3-18.
2. The database server continues matching the arguments from left to right. If the database contains a routine with a matching signature, the database server executes this routine.

Important: *If one of the arguments for the routine is null, more than one routine might match the routine signature. If that situation occurs, the database server generates an error. For more information, see "Null Arguments in Overloaded Routines" on page 3-26.*

Candidate List of Routines

The database server finds a list of candidate routines from the **sysprocedures** system catalog table that have the following characteristics:

- The same routine name
- The same routine type (function or procedure)
- The same number of arguments
- The Execute privilege on the routine in the current session
- ANSI: Belong to the current user or user **informix** ♦

If the candidate list does not contain a UDR with the same data type as an argument specified in the routine invocation, the database server checks for the existence of cast routines that can implicitly convert the argument to a data type of the parameter of the candidate routines.

For example, suppose you create the following two casts and two routines:

```
CREATE IMPLICIT CAST (type1 AS type2)
CREATE IMPLICIT CAST (type2 AS type1)
CREATE FUNCTION g(type1, type1) ...
CREATE FUNCTION g(type2, type2) ...
```

Suppose you invoke function **g** with the following statement:

```
EXECUTE FUNCTION g(a_type1, a_type2)
```

The database server considers both functions as candidates. The routine-resolution process selects the function **g(type1, type1)** because the leftmost argument is evaluated first. The database server executes the second cast, **cast(type2 AS type1)**, to convert the second argument before the function **g(type1, type1)** executes.

For more information about casting, refer to Chapter 7, "Creating User-Defined Casts."

***Tip:** Consider the order in which the database casts data and resolves routines as part of your decision to overload a routine.*

Precedence List of Data Types

To determine which routine in the candidate list might be appropriate to an argument type, the database server builds a precedence list of data types for the argument. The routine-resolution process builds a precedence list, which is a partially ordered list of data types to match. It creates the precedence list as follows (from highest to lowest):

1. The database server checks for a routine whose data type matches the argument passed to a routine.
2. If the argument passed to the routine is a *named row type* that is a subtype in a type hierarchy, the database server checks up the type-hierarchy tree for a routine to execute.

   For more information, refer to "Routine Resolution with User-Defined Data Types" on page 3-22.

3. If the argument passed to the routine is a *distinct typ*e, the database server checks the source data type for a routine to execute.

   If the source type is itself a distinct type, the database server checks the source type of that distinct type. For more information, refer to "Routine Resolution with Distinct Data Types" on page 3-24.

4. If the argument passed to the routine is a *built-in data type*, the database server checks the candidate list for a data type in the built-in data type precedence list for the passed argument.

   For more information, refer to "Precedence List for Built-In Data Types" on page 3-19.

   If a match exists in this built-in data type precedence list, the database server searches for an implicit cast function.

5. The database server *adds implicit casts of the data types* in steps 1 through 4 to the precedence list, in the order that the data types were added.

6. If the argument passed to the routine is a *collection type*, the database server adds the generic type of the collection to the precedence list for the passed argument.

7. The database server adds data types for which there are implicit casts between any data type currently on the precedence list (except the built-in data types) and some other data type.

If no qualifying routine exists, the database server returns the following error message:

```
-674: Routine routine-name not found.
```

If the routine-resolution process locates more than one qualifying routine, the database server returns this error message:

```
-9700: Routine routine-name cannot be resolved.
```

Precedence List for Built-In Data Types

If a routine invocation contains a data type that is not included in the candidate list of routines, the database server tries to find a candidate routine that has a parameter contained in the precedence list for the data type. Figure 3-2 lists the precedence for the built-in data types when an argument in the routine invocation does not match the parameter in the candidate list.

Figure 3-2
Precedence of Built-In Data Types

Data Type	Precedence List
CHAR	VARCHAR, LVARCHAR
VARCHAR	None
NCHAR	NVARCHAR
NVARCHAR	None
SMALLINT	INT, INT8, DECIMAL, SMALLFLOAT, FLOAT
INT	INT8, DECIMAL, SMALLFLOAT, FLOAT, SMALLINT
INT8	DECIMAL, SMALLFLOAT, FLOAT, INT, SMALLINT
SERIAL	INT, INT8, DECIMAL, SMALLFLOAT, FLOAT, SMALLINT
SERIAL8	INT8, DECIMAL, SMALLFLOAT, FLOAT, INT, SMALLINT
DECIMAL	SMALLFLOAT, FLOAT, INT8, INT, SMALLINT
SMALLFLOAT	FLOAT, DECIMAL, INT8, INT, SMALLINT
FLOAT	SMALLFLOAT, DECIMAL, INT8, INT, SMALLINT
MONEY	DECIMAL, SMALLFLOAT, FLOAT, INT8, INT, SMALLINT
DATE	None
DATETIME	None
INTERVAL	None
BYTE	None
TEXT	None

The following example shows overloaded **test** functions and a query that invokes the **test** function. This query invokes the function with a DECIMAL argument, **test(2.0)**. Because a **test** function for a DECIMAL argument does not exist, the routine-resolution process checks for the existence of a **test** function for each data type that the precedence list in Figure 3-2 shows.

```
CREATE FUNCTION test(arg1 INT) RETURNING INT...
CREATE FUNCTION test(arg1 MONEY) RETURNING MONEY....

CREATE TABLE mytab (a real, ...
SELECT * FROM mytab WHERE a=test(2.0);
```

Figure 3-3 shows the order in which the database server performs a search for the overloaded function, **test()**. The database server searches for a qualifying **test()** function that takes a single argument of type INTEGER.

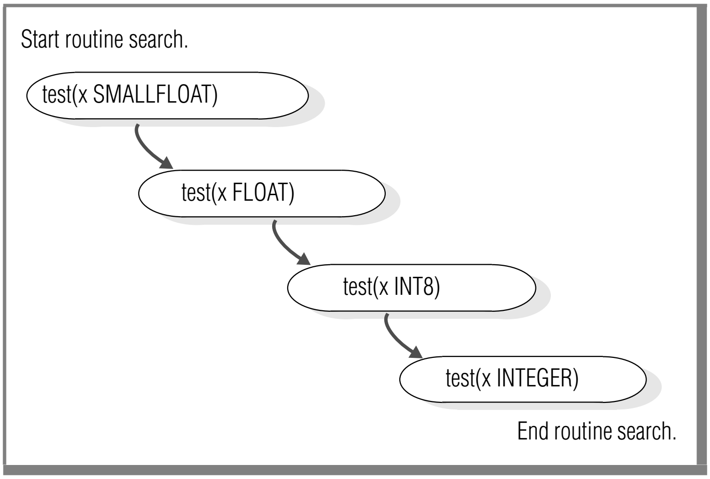


Figure 3-3
Example of Data Type Precedence During Routine Resolution

Routine Resolution with User-Defined Data Types

The following sections discuss routine resolution when one or more of the arguments in the routine signature are UDTs.

Routine Resolution in a Type Hierarchy

A *type hierarchy* is a relationship that you define among named row types in which subtypes inherit representation (data *fields*) and behavior (routines, operators, rules) from a named row above it (*supertype)* and can add additional fields and routines. The subtype is said to *inherit* the attributes and behavior from the supertype.

For information about creating type hierarchies, refer to the discussion of type and table hierarchies in the *IBM Informix Database Design and Implementation Guide*.

When a UDR has named row types in its parameter list, the database server must resolve which type in the type hierarchy to pass to the UDR. When a data type in the argument list does not match the data type of the parameter in the same position of the routine signature, the database server searches for a routine with a parameter in the same position that is the closest supertype of that argument.

Suppose you create the following type hierarchy and register the overloaded function **bonus()** on the root supertype, **emp**, and the **trainee** subtype:

```
CREATE ROW TYPE emp
   (name VARCHAR(30),
    age INT,
    salary DECIMAL(10,2));
CREATE ROW TYPE trainee UNDER emp ...
CREATE ROW TYPE student_emp (gpa FLOAT) UNDER trainee;

CREATE FUNCTION bonus (emp,INT) RETURNS DECIMAL(10,2) ...
CREATE FUNCTION bonus(trainee,FLOAT) RETURNS DECIMAL(10,2).
```

Then you invoke the **bonus()** function with the following statement:

```
EXECUTE FUNCTION bonus(student_emp, INT);
```

To resolve the data type of the UDR parameter when it is a named row type, the database server takes the following steps:

1. The database server processes the leftmost argument first:
   - **a.** It looks for a candidate routine named **bonus** with a row type parameter of **student_emp**.

     No candidate routines exist with this parameter, so the database server continues with the next data type precedence, as described in "Precedence List of Data Types" on page 3-18.
   - **b.** Because **student_emp** is a subtype of **trainee**, the database server looks for a candidate routine with a parameter of type **trainee** in the first position.

     The first parameter of the second function, **bonus(trainee,float)**, matches the first argument in the routine invocation. Therefore, this version of **bonus()** goes on the precedence list.
2. The database server processes the second argument next:
   - **a.** It looks for a candidate routine with a second parameter of data type INTEGER.

     The matching candidate routine from step 1b has a second parameter of data type FLOAT. Therefore, the database server continues with the next data type precedence as "Precedence List of Data Types" on page 3-18 describes.
   - **b.** Because the second parameter is the INTEGER built-in data type, the database server goes to the precedence list that Figure 3-2 on page 3-20 shows.

     The database server searches the candidate list of routines for a second parameter that matches one of the data types in the precedence list for the INTEGER data type.
   - **c.** Because a built-in cast exists from the INTEGER data type to the FLOAT data type, the database server casts the INTEGER argument to FLOAT before the execution of the **bonus()** function.
3. Because of the left-to-right rule for processing the arguments, the database server executes the second function, **bonus(trainee,float)**.

Routine Resolution with Distinct Data Types

A distinct data type has the same internal storage representation as an existing data type, but it has a different name and cannot be compared to the source type without casting. Distinct types inherit functions from their source types. For more information, refer to "Distinct Data Type" on page 5-11.

When a UDR has distinct types in its parameter list, the database server resolves the routine signature, as follows:

- When a routine signature contains a parameter that matches the distinct data type in the same position of the routine invocation, the routine-resolution process selects that routine to execute.
- When a distinct data type in the argument list does not match the data type of the parameter in the same position of the routine signature, the database server searches for a UDR that accepts one of the following data types in the position of that argument:
  - A data type to which the user has defined an implicit cast from the type of the argument specified in the routine invocation

    For more information on casts, refer to "Cast Functions" on page 2-8.
  - The source data type of the distinct type

The following sections describe source data type restrictions and provide procedures for routine resolution with these source types.

Routine Resolution with Two Different Distinct Data Types

The candidate list can contain a routine with a parameter that is the source data type of the invoked routine argument. If the source type is itself a distinct type, the database server checks the source type of that distinct type. However, if the source type is not in the precedence list for that data type, the routine-resolution process eliminates that candidate.

For example, suppose you create the following distinct data types and table:

```
CREATE DISTINCT TYPE pounds AS INT;
CREATE DISTINCT TYPE stones AS INT;
CREATE TABLE test(p pounds, s stones);
```

Figure 3-4 shows a sample query that an SQL user might execute.

```
SELECT * FROM test WHERE p=s;
```

Figure 3-4
Sample Distinct Type Invocation

Although the source data types of the two arguments are the same, this query fails because **p** and **s** are different distinct data types. The **equal()** function cannot compare these two different data types.

Alternate SELECT Statements for Different Distinct Data Types

The database server chooses the built-in **equals** function when you explicitly cast the arguments. If you modify the SELECT statement as follows, the database server can invoke the **equals(int,int)** function, and the comparison succeeds:

```
SELECT * FROM test WHERE p::INT = s::INT;
```

You can also write and register the following additional functions to allow the SQL user to use the SELECT statement that Figure 3-4 shows:

- An overloaded function **equals(pounds,stones)** to handle the two distinct data types:

  ```
  CREATE FUNCTION equals(pounds, stones) ...
  ```

  The advantage of creating an overloaded **equals()** function is that the SQL user does not need to know that these are new data types that require explicitly casting.

- Implicit cast functions from the data type **pounds** to **stones** and from **stones** to INT:

  ```
  CREATE IMPLICIT CAST (pounds AS stones);
  CREATE IMPLICIT CAST (stones AS INT);
  ```

Routine Resolution with Built-In Data Types as Source

If the source type is a built-in data type, the distinct type does not inherit any built-in casts provided for the built-in type, but it does inherit any user-defined casts that are defined on the source type. For example, suppose you create the following distinct data type and table:

```
CREATE DISTINCT TYPE inches AS FLOAT;
CREATE TABLE test(col1 inches);
INSERT INTO test VALUES (2.5::FLOAT::inches);
```

An SQL user might execute the following sample query:

```
SELECT 4.8 + col1 FROM test;
```

Although the source data type of the **col1** argument has a built-in cast function to convert from FLOAT to DECIMAL (the `4.8` is DECIMAL), this query fails because the distinct type **inches** does not inherit the built-in cast.

You must use explicit casts in the SQL query. The following queries succeed:

```
SELECT 4.8 + col1::INT from test;
SELECT 4.8::FLOAT::inches + col1 FROM test;
```

Routine Resolution with Collection Data Types

A *collection data type* is a complex data type whose instances are groups of elements of the same data type that are stored in a SET, MULTISET, or LIST. An element within a collection can be an opaque data type, distinct data type, built-in data type, collection data type, or row type.

Null Arguments in Overloaded Routines

The database server might return an error message when you call a UDR and both of the following conditions are true:

- The argument list of the UDR contains a null value.
- The UDR invoked is an overloaded routine.

Suppose you create the following user-defined functions:

```
CREATE FUNCTION func1(arg1 INT, arg2 INT) RETURNS BOOLEAN...
CREATE FUNCTION func1(arg1 MONEY, arg2 INT)
   RETURNS BOOLEAN...
CREATE FUNCTION func1(arg1 REAL, arg2 INT) RETURNS BOOLEAN...
```

The following statement creates a table, **new_tab**:

```
CREATE TABLE new_tab (col_int INT);
```

The following query is successful because the database server locates only one **func1()** function that matches the function argument in the expression:

```
SELECT *
FROM new_tab
WHERE func1(col_int, NULL) = "t";
```

The null value acts as a wildcard for the second argument and matches the second parameter type for each function **func1()** defined. The only **func1()** function with a leftmost parameter of type INT qualifies as the function to invoke.

If more than one qualifying routine exists, the database server returns an error. The following query returns an error because the database server cannot determine which **func1()** function to invoke. The null value in the first argument matches the first parameter of each function; all three **func1()** functions expect a second argument of type INTEGER.

```
SELECT *
FROM new_tab
WHERE func1(NULL, col_int) = "t";
```

To avoid ambiguity, use null values as arguments carefully.

Chapter

Developing a User-Defined Routine

4

In This Chapter

This chapter describes the design and creation of UDRs. It covers the following topics:

Planning the Routine

When you write a UDR, consider the following:

- Naming your routine
- Defining routine parameters
- Defining a return value (user-defined functions only)
- Adhering to coding standards

The routine name and routine parameters make up the routine signature for the routine. The routine signature uniquely identifies the UDR in the database. For more information, see "The Routine Signature" on page 3-12.

Consider the following questions about routine naming and design:

- Are any of my routines modal? That is, does the behavior of the routine depend on one of its arguments?
- Can I describe what each type and routine does in two sentences?
- Do any of my routines take more than three arguments?
- Have I used polymorphism effectively?

The maximum size of a UDR depends on the language in which it is written in and the platform where it is used. For UDRs written in C, you can create very large shared objects. The limit depends on the compiler and the machine architecture. The size limit for UDRs written in Java is similarly high, depending on the size of the **.jar** files that you can create. For SPL you are limited to the maximum size of an SQL statement at 64 kilobytes.

Naming the Routine

Choose sensible names for your routines. Make the routine name easy to remember and have it succinctly describe what the routine does. The database server supports *polymorphism*, which allows multiple routines to have the same name. This ability to assign one name to multiple routines is called routine overloading. For more information on routine overloading, refer to "Overloading Routines" on page 3-13.

Routine overloading is contrary to programming practice in some high-level languages. For example, a C programmer might be tempted to create functions with the following names that return the larger of their arguments:

```
bigger_int(integer, integer)
bigger_real(real, real)
```

In SQL, these routines are better defined in the following way:

```
bigger(integer, integer)
bigger(real, real)
```

The naming scheme in the second example allows users to ignore the types of the arguments when they call the routine. They simply remember what the routine does and let the database server choose which routine to call based on the argument types. This feature makes the UDR simpler to use.

Defining Routine Parameters

When you invoke a UDR, you can pass it optional *argument* values. Each argument value corresponds to a *parameter* of the routine.

Number of Arguments

Limit the number of arguments in your UDRs and make sure that these arguments do not make the routine *modal*. A modal routine uses a special argument as a sort of flag to determine which of several behaviors it should take. For example, the following statement shows a routine call to compute containment of spatial values:

```
Containment(polygon, polygon, integer);
```

This routine determines whether the first polygon contains the second polygon or whether the second contains the first. The caller supplies an integer argument (for example, 1 or 0) to identify which value to compute. This is modal behavior; the mode of the routine changes depending on the value of the third argument.

In the following example, the routine names clearly explain what computation is performed:

```
Contains(polygon, polygon)
ContainedBy(polygon, polygon)
```

Always construct your routines to be nonmodal, as in the second example.

Declaring Routine Parameters

You define routine parameters in a *parameter list* when you declare the routine. In the parameter list, each parameter provides the name and data type of a value that the routine expects to handle. Routine parameters are optional; you can write a UDR that has no input parameters.

When you invoke the routine, the argument value must have a data type that is compatible with the parameter data type. If the data types are not the same, the database server tries to resolve the differences. For more information, see "The Routine-Resolution Process" on page 3-16.

The way that you declare a UDR depends on the language in which you write that routine.

SPL

The parameters in an SPL routine must be declared with SQL data types, either built-in or user defined. For more information, see "Executing an SPL Routine" on page 3-8. ♦

Ext

For routines written in C or Java, you use the syntax of that language to declare the routine. The routine parameters indicate the argument data types that the routine expects to handle.

You declare the routine parameters with data types that the external language supports. However, when you register the routine with CREATE FUNCTION or CREATE PROCEDURE, you use SQL data types for the parameters. (For more information, see "Registering Parameters and a Return Value" on page 4-32.) Therefore, you must ensure that these external data types are compatible with the SQL data types that the routine registration specifies. ♦

C

For C UDRs, the DataBlade API provides special data types for use with SQL data types. For most of these special data types, you must use the pass by reference mechanism. However, for a few data types, you can use the pass-by-value mechanism. For more information, see the chapter on DataBlade API data types in the *IBM Informix DataBlade API Programmer's Guide* and the *IBM Informix DataBlade API Function Reference*. ♦

Java

Every Java UDR maps to an external Java static method whose class resides in a JAR file that has been installed in a database. The SQL-to-Java data type mapping is done according to the JDBC specification. For more information, refer to the *J/Foundation Developer's Guide* and your Java documentation. ♦

Returning Values

A common use of a UDR is to return values to the calling SQL statement. A UDR that returns a value is called a user-defined *function*.

For information on how to specify the data type of the return value of a user-defined function, see "Registering a User-Defined Routine" on page 4-23.

Returning a Variant or Nonvariant Value

By default, a user-defined function is a variant function. A *variant function* has any of the following characteristics:

- It returns different results when it is invoked with the same arguments.

  For example, a function whose return value is computed based on the current date or time is a variant function.
- It has variant side effects, such as:
  - Modifying some database table, variable state, or external file
  - Failing to locate an external file, or a table or row in a database, and returning an error

You can explicitly specify a variant function with the VARIANT keyword. However, because a function is variant by default, this keyword is not required.

A *nonvariant function* always returns the same value when it receives the same argument, and it has none of the preceding variant side effects. Therefore, nonvariant functions cannot access external files or contain SQL statements, even if the SQL statements only SELECT static data and always return the same results. You specify a nonvariant function with the NOT VARIANT keywords.

You can create a *functional index* only on a nonvariant function. The return result for a functional index cannot contain a smart large object. Functional indexes are indexed on the value returned by the specified function rather than on the value of a column. The value returned by a functional index cannot contain a smart large object.

The database server can execute a nonvariant function during query compile time if all the arguments passed to it are constants. In that case, the result replaces the UDR expression in the query tree. This action by the database server is *constant elimination*. The database server cannot execute an SQL statement during constant elimination, thus a nonvariant function cannot execute even nonvariant SQL.

For information about creating a functional index, refer to the CREATE INDEX statement in the *IBM Informix Guide to SQL: Syntax*.

Using OUT Parameters and Statement-Local Variables (SLVs)

You use OUT parameters to pass values from the called function to the caller. The SPL, C, or Java called function sets the value of this parameter and returns a new value through the parameter. Any or all arguments of a UDR can be an OUT parameter. You cannot use OUT parameters to pass values *to* the called function; OUT parameters are passed as `NULL` to the UDR.

The syntax for creating a UDR with OUT parameters is:

```
CREATE FUNCTION udr ([IN/OUT] arg0 datatype0, ...,
                     [IN/OUT] argN datatypeN)
                    RETURNING returntype;
...
END FUNCTION;
```

By default, a parameter is considered an IN parameter unless you define it as an OUT parameter by specifying the OUT keyword.

For example, the following CREATE FUNCTION statement specifies one IN parameter, *x*, and two OUT parameters, *y* and *z*.

```
CREATE FUNCTION my_func(x INT, OUT y INT, OUT z INT)
RETURNING INT
EXTERNAL NAME '/usr/lib/local_site.so'
LANGUAGE C
```

A statement-local variable (SLV) is an OUT parameter used in the WHERE clause of a SELECT statement. See "Using SLVs" on page 4-9 for more information.

Important: *You cannot execute UDRs with OUT parameters in Data Manipulation Language (DML) SQL statements, except by using an SLV. The statements SELECT, UPDATE, INSERT and DELETE are DML statements.*

Important: *You cannot use the EXECUTE FUNCTION statement to invoke a user-defined function that contains an OUT parameter, unless you are using JDBC.*

Important: *You cannot execute remote UDRs that contain OUT parameters.*

Using SLVs

An SLV transmits OUT parameters from a user-defined function to other parts of an SQL statement. An SLV is local to the SQL statement; that is, it is valid *only* for the life of the SQL statement. It provides a temporary name by which to access an OUT parameter value. Any or all user-defined function arguments can be an SLV.

In the SQL statement that calls the user-defined function, you declare the SLV with the syntax: *SLV_name* # *SLV_type*, where *SLV_name* is the name of the SLV variable and *SLV_type* is its data type, as in:

```
SELECT SLV_name1, SLV_nameN FROM table WHERE
   udr (param1, SLV_name1 # SLV_type1, ...
        SLV_nameN # SLV_typeN, paramN);
```

For example, the following SELECT statement declares SLVs *x* and *z* that are typed as INTEGER in its WHERE clause and then accesses both SLVs in the projection list:

```
SELECT x, z WHERE my_func(x # INT, y, z # INT) < 100
    AND (x = 3) AND (z = 5)
```

For more information on the syntax and use of an SLV, see the description of function expressions within the Expression section in the *IBM Informix Guide to SQL: Syntax*.

SPL Procedures With No Return Values

SPL procedures with no return values are only accessible through the JDBC **CallableStatement** interface. SPL procedures with no return values can use OUT parameters. The syntax for creating such a procedure is:

```
CREATE PROCEDURE spl_udr ([IN/OUT] arg0 datatype0, ...,
                          [IN/OUT] argN datatypeN);
...

END PROCEDURE;
```

For example, the following SQL statement creates an SPL procedure with two OUT parameters and one IN parameter:

```
CREATE PROCEDURE myspl (OUT arg1 int, arg2 int, OUT arg3 int);
LET arg1 = arg2;
LET arg3 = arg2 * 2;
END PROCEDURE;
```

SPL procedures that do not return values cannot be used in the WHERE clause of a SELECT statement and therefore cannot generate SLVs.

Naming Return Parameters

You can define names for each return parameter of an SPL UDR. Specify the names in the RETURNS/RETURNING clause of the CREATE PROCEDURE/FUNCTION statement.

The syntax for the CREATE PROCEDURE/FUNCTION statement is:

```
RETURNS/RETURNING data_type AS return_param_name [{, data_type AS
return_param_name}]
```

The *return_param_name* parameter defines the name of the return parameter and follows the same rules as for table column names. Either *all* return parameters should have names or *none* should have names. The names of the return parameters for a function or procedure should be unique. Return parameter names cannot be referenced within the body of the procedure. There is no relation between the names of the return parameters and any variables within the function or procedure itself, as shown in the following example:

```
CREATE PROCEDURE NamedRetProc()
RETURNING int AS p_customer_num, char(20) AS p_fname, char(20) AS p_lname;

DEFINE v_id int;
DEFINE v_fname char(15);
DEFINE v_lname char(15);

FOREACH curA FOR SELECT customer_num, fname, lmname
    INTO v_id, v_fname, v_lname FROM customer
RETURN v_id,v_fname, v_lname WITH RESUME;
END FOREACH;

ENDPROCEDURE;
```

The **NamedRetProc()** procedure returns data with the return parameter names shown above the returned values, as below, instead of the name *expression* that appears if you do not name return parameters:

```
p_customer_num p_fname p_lname
       101     Ludwig   Pauli
       102     Carole   Sadler
```

Avoid naming return parameters if you intend to export the database to a pre-9.4 version of IBM Informix Dynamic Server that does not support this syntax. When you export a database containing stored procedures that have names for return parameters, the schema creation scripts also have these names. If you try to import the database using a pre-9.4 version of IBM Informix Dynamic Server, errors will be returned. If you decide to go ahead and import the stored procedures without the names for return parameters, you can manually edit the schema creation scripts to be able to import.

Tip: *When you call a stored procedure in the projection list of a SELECT statement, return parameter names are not displayed. Instead, the output string "expression" appears. If you want to display the return parameter name, use the AS keyword, as in: SELECT some_func(a,b) AS name1,... .*

Using an Iterator Function

By default, a user-defined function returns one value; that is, it calculates its return value and returns only once to its calling SQL statement. User-defined functions that return their result in a single return to the calling SQL statement are called *noncursor functions* because they do *not* require a database cursor to be executed. For information on how to invoke noncursor functions, see "Invoking a UDR in an SQL Statement" on page 3-3.

However, you can write a user-defined function that returns to its calling SQL statement several times, each time returning a value. Such a user-defined function is called an *iterator function*. An iterator function is a *cursor function* because it must be associated with a cursor when it is executed. The cursor holds the values that the cursor function repeatedly returns to the SQL statement. The calling program can then access the cursor to obtain each returned value, one at a time. The contents of the cursor are called an *active set*. Each time the iterator function returns a value to the calling SQL statement, it adds one item to the active set.

Important: *You cannot use OUT parameters in iterator functions.*

Creating an Iterator Function

You can write iterator functions in SPL, C, or Java. Each language uses different statements, functions, and methods to manage iterator tasks:

- An SPL iterator function uses the FOREACH keyword in conjunction with the RETURN WITH RESUME statement.
- A C-language iterator function uses DataBlade API functions, such as **mi_fp_setisdone()** and **mi_fp_request()**, to handle each return item of the active set. MI_FPARAM maintains the iterator state that **mi_fp_setisdone()** and **mi_fp_request()** access.
- A Java iterator function uses the **UDREnv** interface, which provides all necessary methods and constants.

Registering an Iterator Function

By default, a function written in an external language is *not* an iterator. To define an iterator function written in C or Java, you must register the function with the ITERATOR routine modifier. The following sample CREATE FUNCTION statement shows how to register the function **TopK()** as an iterator function in C:

```
CREATE FUNCTION TopK(INTEGER, INTEGER)
   RETURNS INTEGER
   WITH (ITERATOR, NOT VARIANT)
   EXTERNAL NAME
      '/usr/lib/extend/misc/topkterms.so(topk_integers)'
   LANGUAGE C
```

Tip: *An SPL iterator function does not need to be registered using the ITERATOR modifier.*

Invoking an Iterator Function

You can invoke an iterator function using one of the following methods:

- Directly with the EXECUTE FUNCTION statement:
  - From DB-Access
  - In a prepared cursor in an external routine
  - In an external routine
  - In an SPL FOREACH loop

- With an EXECUTE FUNCTION statement as part of an INSERT statement:
  - From DB-Access
  - In a prepared cursor in ESQL/C or an external routine
  - In a DataBlade API database server routine
  - In an SPL FOREACH loop
- In the FROM clause of a SELECT statement

  Instead of a table, the result set of the iterator function is the source from which the query selects data. The return values from the iterator function are mapped to a virtual table. Using an iterator function in a FROM clause is described in detail, next.

  Existing iterator UDRs from pre-9.4 releases can be used in the FROM clause of a SELECT statement.

Using an Iterator Function in the FROM Clause of a SELECT Statement

In addition to tables, an iterator function can be specified as a source for a SELECT statement. This means you can query the return result set of an iterator UDR using a table interface. Therefore, you can manipulate the iterator result set in a number of ways, such as by using the WHERE clause to filter the result set; by joining the UDR result set with other table scans; by running GROUP BY, aggregation, and ORDER BY operations, and so on.

Syntax and Usage

The syntax for using an iterator function in the FROM clause is:

```
FROM TABLE (FUNCTION iterator_func_name ([argument_list]))
[[AS] virtual_table_name] [(virtual_column_list)]
```

The *virtual_table_name* parameter is unqualified (do not include the owner or database name) and specifies the name of the virtual table that holds the result set from the iterator function.

Important: *The virtual table can only be referenced within the context of this SELECT query. After the SELECT statement completes, the virtual table no longer exists.*

The *virtual_column_list* parameter is a comma-separated list of unqualified column names for the virtual table. The number of columns must match the number of values returned by the iterator (SPL functions can return more than one value).

If you want to reference virtual table columns in other parts of the SELECT statement, for example, in the projection list, WHERE clause, or HAVING clause, you must specify the virtual table name and virtual column names in the FROM clause. You do not have to specify the virtual table name or column names in the FROM clause if you use wildcard characters in the projection list of the SELECT clause:

```
SELECT * FROM ...
```

As an example, the following statement retrieves the result set from the function called **fibseries()**. This result set is held in the virtual table called **vtab**.

```
SELECT col FROM TABLE (FUNCTION fibseries(10)) vtab(col);
```

If a SELECT statement specifying an iterator in the FROM clause returns unexpected results, execute the iterator function separately to verify the function is behaving correctly. For example, run your function in DB-Access with a command like this:

```
execute function iterator_udr(args)
```

The SQL Explain output section for a virtual table derived from an iterator UDR is marked *ITERATOR UDR SCAN*.

Ensure that you call **mi_fp_setisdone()** in a C UDR or **UDREnv.setSetIterationIsDone(true)** in a JAVA UDR when the iterator UDR is finished. The server checks this flag internally to determine when to stop calling the iterator UDR.

Allocating Memory

For iterator functions written in C, the default memory duration for return values set by the server should be sufficient.

The MI_FPARAM data structure should be allocated a duration that lasts for all iterations, usually a PER_COMMAND duration.

Running Parallel Queries

If you are running queries in parallel using the IBM Informix Dynamic Server parallel database query (PDQ) feature and the iterator UDR in the FROM clause is *not* parallelizable, query parallelism is turned off for the SELECT query. However, if the iterator UDR in the FROM clause is parallelizable and no other factors disable the query parallelism, the query can run in parallel. When PDQ is on, functional tables are treated as single non-fragmented tables.

In the following example, the GROUP BY and aggregation operations can be run by multiple PDQ threads and the **fibseries()** function can be run by a secondary thread.

```
SELECT col1,col2, COUNT(*) FROM TABLE (FUNCTION fibseries(10))
tab1(col1),tab2
GROUP BY col1,col2;
```

Refer to your *IBM Informix Dynamic Server Performance Guide* for information about running queries in parallel.

Restrictions

The following restrictions apply to using iterator functions in the FROM clause:

- Iterator functions cannot refer to other columns in the FROM clause. For example, the following query is invalid because the **fibseries** iterator function specifies the column **t.x** as an argument:

  ```
  SELECT t.x, vtab.col
  FROM t, TABLE (FUNCTION fibseries(t.x)) vtab(col);
  ```

  However, iterator functions can refer to other columns when used in an outer query, as in:

  ```
  SELECT t.x FROM t
  WHERE t.y IN
   (SELECT col FROM TABLE (FUNCTION fibseries(t.y)) vtab(col));
  ```

- Iterator functions cannot generate OUT parameters and statement-local variables.

- You cannot use iterator functions as the target in INSERT, UPDATE, or DELETE statements.
- UDRs used in the FROM clause must be iterator functions.

Example SPL Iterator Function

To create an SPL iterator function to be used in the FROM clause, your function must use the RETURN WITH RESUME construct, as shown in the following example.

Because an SPL UDR can return more than one value, you can specify multiple column names in the virtual column list in the FROM clause. You can reference any of these virtual column names in the target list of the SELECT query.

```
create function find_top_earners()
    returning integer,decimal,lvarchar
define ret_empid integer;
define ret_salary decimal;
define ret_empname lvarchar;
    foreach select emp_id,salary into
        ret_empid,ret_salary from salary
        if (ret_salary > 100000.00)
          select emp_name into ret_empname from employee
          where emp_id = ret_empid;
        return ret_empid,ret_salary,ret_empname with
          resume;
        end if;
    end foreach;
end function;
```

The following query uses the above iterator UDR, **find_top_earners()**, to retrieve the top earners sorted by employee name.

```
select vemp_name,vemp_id,vemp_sal from
    table (function find_top_accounts())
    vtab1(vemp_name,vemp_id,vemp_sal)
    order by vemp_name;
```

Example C Iterator Function

To write an iterator C function, you use DataBlade API functions, such as **mi_fp_request()**, **mi_fp_setfuncstate()**, **mi_fp_setisdone()**, and so on, with the MI_FPARAM data structure.

A C UDR can return only one value; therefore, there can be only one column in the virtual column list in the FROM clause. However, a C UDR can return a row type, which can capture multiple return values as a unit.

The following example demonstrates how to write a C iterator function and use it in the FROM clause; relevant DataBlade API and iterator states are highlighted.

The function **fibseries()** is an iterator function that returns the Fibonacci series up to the value passed to it as an argument.

```
create function fibseries(int x)
returns int with (handlesnulls,iterator, parallelizable)
external name "$USERFUNCDIR/fib.so"
language c;

/* A Function to return a set of integer. This function takes
stop val as a parameter and returns a fibonaucci series up to
stop val.

* Three states of fparam :
*
* SET_INIT: Allocate the the function state structure defined.
This State Structure is allocated in PER_COMMAND duration to
hold the memory till the end of the command.
Make the fparam structure point to the State Structure.
Set the first two numbers of the series i.e 0 and 1; And
set the stop val field of State Structure to the stop val passed
to the function.

* SET_RETONE: Computes the next number in the series. Compares
it with the stop val to check if the exit criteria is met.
num1 = num2;num2 = next number in the series.

* SET_END: Frees the user Allocated Func State structure.
*/

#include <milib.h>
typedef struct fibState1 {
   mi_integer fib_prec1;
   mi_integer fib_prec2;
   mi_integer fib_ncomputed;
   mi_integer fib_endval;
}fibState;
mi_integer
fibseries(endval,fparam)
mi_integer endval;
MI_FPARAM  *fparam;
{
     fibState   *fibstate;
     mi_integer next;
     switch(mi_fp_request(fparam)) {
        case SET_INIT :
             fibstate = (fibState *) mi_dalloc
(sizeof(fibState),PER_COMMAND);
```

```
                mi_fp_setfuncstate(fparam,(void *)fibstate);
                if (mi_fp_argisnull(fparam,0) || endval < 0) {
                   mi_fp_setreturnisnull(fparam,0,1);
                   break;
                }
                if (endval < 1) {
                   fibstate->fib_prec1 = 0;
                   fibstate->fib_prec2 = 1;
                   fibstate->fib_ncomputed = 1;
                   fibstate->fib_endval = endval;
                }
                else {
                   fibstate->fib_prec1 = 0;
                   fibstate->fib_prec2 = 1;
                   fibstate->fib_ncomputed = 0;
                   fibstate->fib_endval = endval;
                }
                break;
           case SET_RETONE :
                fibstate = mi_fp_funcstate(fparam);
                if (fibstate->fib_ncomputed < 2) {
                   return((fibstate->fib_ncomputed++ == 0) ? 0 : 1);
                }
                next = fibstate->fib_prec1 + fibstate->fib_prec2;
                if (next > fibstate->fib_endval) {
                   mi_fp_setisdone(fparam,1);
                   return 0;
                }
                if (next == 0) {
                   fibstate->fib_prec1 = 0;
                   fibstate->fib_prec1 = 1;
                }
                else {
                   fibstate->fib_prec1 = fibstate->fib_prec2;
                   fibstate->fib_prec2 = next;

                }
                return (next);
           case SET_END :
                fibstate = mi_fp_funcstate(fparam);
                mi_free(fibstate);
                break;
        }
     }
```

This function can be used in the FROM clause of a SELECT query:

```
select vcol1 from table (function fibseries(100)) vtab1(vcol1);
```

Example Java Iterator Function

The **UDREnv** interface provides all necessary methods and constants. A Java UDR can return only one value; therefore, there can be only one column in the virtual column list in the FROM clause.

The following example demonstrates how to write a Java iterator function and use it in FROM clause; relevant DataBlade API and iterator states are highlighted.

The iterator UDR **jenv_iter()** takes an integer parameter and returns a row of CHAR(40) columns. The parameter passed in determines the number of rows it returns.

```
public interface UDREnv
{
...

// for maintaining state across UDR invocations
void setUDRState(Object state);
Object getUDRState();
// for set/iterator processing
public static final int UDR_SET_INIT = 1;
public static final int UDR_SET_RETONE = 2;
public static final int UDR_SET_END = 3;
int getSetIterationState();
void setSetIterationIsDone(boolean value);

...
}
import java.lang.*;
import java.sql.*;
import com.informix.udr.*;
import informix.jvp.*;
public class Env
{
    public int count;
    //
    // test UDR meta
    //

    public static String envTest1(int i, String xchar, String
xvchar, String xlvarchar)
    throws SQLException
    {
UDREnv env = UDRManager.getUDREnv();
String res = env.getName() + "#" +
env.getReturnTypeName() + "#";
String param[] = env.getParamTypeName();
for (int j = 0; j < param.length; ++ j)
    res += param[j] + "#";

res += i + xchar + xvchar + xlvarchar;
return res;
    }
    public static String envTest2(int i, String s[])
    throws SQLException
    {
UDREnv env = UDRManager.getUDREnv();
UDRLog log = env.getLog();
String res = env.getName() + "#" +
env.getReturnTypeName() + "#";
String param[] = env.getParamTypeName();
for (int j = 0; j < param.length; ++ j)
    res += param[j] + "#";
res += i;
log.log(res);
s[0] = res;
return res;
    }
    //
    //test env state, iterator, log, traceable, and
properties
    //
    public static String envIter(int num)
throws SQLException
    {
UDREnv env = UDRManager.getUDREnv();
UDRLog log = env.getLog();
UDRTraceable tr = env.getTraceable();
JVPProperties pr = env.getProperties();
int iter = env.getSetIterationState();
Env state = (Env)env.getUDRState();
        if (iter == UDREnv.UDR_SET_INIT)
    {
    state = new Env();
    state.count = num;
    env.setUDRState(state);
    log.log("SET INIT" + state.count + " " +
state.toString());

    tr.tracePrint("UDR.ENVITER", 0, "SET INIT");
       env.setSetIterationIsDone(false);
    pr.setProperty("ENVITERPROP", "AFTER INIT");
    return "INIT";
    }
        else if (iter == UDREnv.UDR_SET_END)
    {
    log.log("SET DONE");
    tr.tracePrint("UDR.ENVITER", 0, "SET DONE");
       env.setSetIterationIsDone(true);
    return "DONE";
    }
        else if (iter == UDREnv.UDR_SET_RETONE)
    {
    log.log("SET RETONE" + state.count + " " +
state.toString());
    tr.tracePrint("UDR.ENVITER", 0, "SET RETONE");
    String prv = pr.getProperty("ENVITERPROP");

    if (state.count <= 0)
env.setSetIterationIsDone(true);
    else
env.setSetIterationIsDone(false);
    -- state.count;
    pr.setProperty("ENVITERPROP", "AFTER RETONE" +
(state.count + 1));
    return new String("ELEMENT " + (state.count + 1) );
//+ prv);
    }
        else
    throw new SQLException("Bad iter code");
    }
}
```

The following statement creates the Java iterator UDR, **jenv_iter()**.

```
create function jenv_iter(int)
    returning char(40)
    with (class = "jvp", iterator)
    external name `Env.envIter(int)'
    language java;
```

Adhering to Coding Standards

The SQL/PSM standard is available for UDR development. In addition, a collection of standards is available for DataBlade module development from the IBM Informix Developer Zone at www.ibm.com/software/data/developer/informix. The most important rules govern the naming of data types and routines. DataBlade modules share these name spaces, so you must follow the naming guidelines to guarantee that no problems occur when you register multiple DataBlade modules in a single database.

Tip: *It is recommended that you use the DBDK, Version 4.0 or later, to manage DataBlade development. It is especially important to use the SQL registration scripts that the DBDK generates so that BladeManager can correctly process DataBlade upgrades.*

In addition, the standards for 64-bit clean implementation, safe function-calling practices, thread-safe development, and platform portability are important. Adherence to these standards ensures that UDR modules are portable across platforms.

Ask yourself the following questions when you code your UDR:

- Do I obey all naming standards?
- Is my design 64-bit clean and portable across platforms?
- Is my design thread-safe?

Writing the Routine

The source for an external routine resides in a separate text file. For information about C UDRs, refer to the *IBM Informix DataBlade API Programmer's Guide* and the *IBM Informix DataBlade API Function Reference*. For information about Java UDRs, refer to the *J/Foundation Developer's Guide*.

Tip: *It is recommended that you use the DBDK to help write UDRs. DBDK enforces standards that facilitate migration between different versions of the database server.*

Because external-language routines are external to the database, you must compile the UDR source code and store it where the database server can access it when the routine is invoked. To prepare UDR source code:

- Compile the C-language UDR and store the executable version in a shared-object file.

  For information about how to create shared-object files, refer to the *IBM Informix DataBlade API Programmer's Guide*.
- Compile the Java-language UDR and store the executable version in a **.jar** file.

  For information about how to prepare **.jar** files, refer to your Java documentation.

You must install shared object files and **.jar** files on all database servers that need to run the UDRs, including database servers involved in Enterprise Replication (ER) and High-Availability Data Replication (HDR). The shared object files and **.jar** files need to be installed under the same absolute path name.

Registering a User-Defined Routine

The database server recognizes the following SQL statements for the registration of UDRs in the database:

- The CREATE FUNCTION statement registers UDRs that return a value.
- The CREATE PROCEDURE statement registers UDRs that do not return a value.

You must register UDRs in all databases in which they will be used, unless the database is on the secondary database server of an HDR pair.

To register a user-defined routine

1. Ensure that you have the correct privileges to register a UDR.
2. Use a CREATE FUNCTION or CREATE PROCEDURE statement to register the UDR:
   - For SPL routines, the statement lists the routine code and then compiles and registers the routine.
   - For external-language routines, the statement specifies the location of the routine code (with an EXTERNAL NAME clause) and registers the routine.

The following example shows the syntax of a CREATE FUNCTION statement:

```
CREATE FUNCTION func_name(parameter_list) RETURNS ret_type
   WITH (NOT VARIANT)
   EXTERNAL NAME 'pathname'
   LANGUAGE C
```

This SQL statement provides the following information to the database:

- The name, *func_name*, and owner of the support function
- An optional specific name for the support function (not shown)
- The data types of the parameters, *parameter_list*, and return value, *ret_type*, of the support function
- The location, *pathname*, of the source code for the support function
- The language of the support function: LANGUAGE C.
- The routine modifier NOT VARIANT that indicates that the function does not return different results with different arguments.

E/C

You cannot use the CREATE FUNCTION directly in an ESQL/C program. To register an opaque-type support function from within an ESQL/C application, you must put the CREATE FUNCTION statement in an operating-system file. Then use the CREATE FUNCTION FROM statement to identify the location of this file. The CREATE FUNCTION FROM statement sends the contents of the operating-system file to the database server for execution. ♦

Setting Privileges for a Routine

A user must have the following privileges to issue a CREATE FUNCTION or CREATE PROCEDURE statement that registers a UDR in the database:

- Database-level privilege
- Language-level privilege

After you register the UDR, you can assign routine-level privileges. For information about how to assign privileges, refer to the GRANT statement in the *IBM Informix Guide to SQL: Syntax*.

Database-Level Privilege

Database-level privileges control the ability to extend the database by registering or dropping a UDR. The following users qualify to register a new routine in the database:

- Any user with the DBA privilege can register a routine with or without the DBA keyword in the CREATE FUNCTION or CREATE PROCEDURE statement.
- A non-DBA user needs the Resource privilege to register a routine.

  The creator has owner privileges on the routine. A user who does not have the DBA privilege cannot use the DBA keyword in the CREATE FUNCTION or CREATE PROCEDURE statement to register the routine.

Tip: *For an explanation of the DBA keyword, see "Executing a UDR as DBA" on page 12-6.*

A DBA must grant the Resource privilege required for any non-DBA user to create a routine. The DBA can revoke the Resource privilege, which prevents that user from creating additional routines.

A DBA or the routine owner can cancel the registration with the DROP ROUTINE, DROP FUNCTION, or DROP PROCEDURE statement. A DBA or routine owner can register a modification to the routine with the ALTER ROUTINE, ALTER FUNCTION, or ALTER PROCEDURE statement.

Language-Level Privilege

The language-level Usage privilege controls the ability to write a UDR in a particular UDR language. This privilege needs to be granted by user **informix** or by another user who has been granted the DBA privilege with WITH GRANT OPTION.

UDR languages have the following GRANT and REVOKE requirements for the Usage privilege:

- The DBA can grant or revoke the Usage privilege to *any* language that the database server supports.
- Another user can grant the Usage privilege if the DBA applied the WITH GRANT keywords in the GRANT EXECUTE ON statement.

The following GRANT statement grants Usage privilege on Java UDRs to the user named **dorian**:

```
GRANT USAGE ON LANGUAGE JAVA TO dorian
```

By default, the database server:

- Does not grant Usage privilege on external languages to PUBLIC
- Does grants Usage privilege on SPL to PUBLIC

For more information, see the description of privileges in the *IBM Informix Database Design and Implementation Guide* and the description of the GRANT statement in the *IBM Informix Guide to SQL: Syntax*

Routine-Level Privilege

When you register a UDR, you automatically receive the Execute privilege on that routine. The Execute privilege allows you to invoke the UDR. For information about allowing other users to execute your routine, see "Assigning the Execute Privilege to a Routine" on page 12-3.

SPL

Creating an SPL Routine

For an SPL routine, the CREATE FUNCTION or CREATE PROCEDURE statement performs the following tasks:

- Parses and optimizes all SQL statements, if possible

  The database server puts the SQL statements in an *execution plan.* An execution plan is a structure that enables the database server to store and execute the SQL statements efficiently.

  The database server optimizes each SQL statement within the SPL routine and includes the selected query plan in the execution plan. For more information on SPL routine optimization, refer to "Optimizing an SPL Routine" on page 13-4.
- Builds a dependency list

  A dependency list contains items that the database server checks to decide if an SPL routine needs to be reoptimized at execution time. For example, the database server checks for the existence of all tables, indexes, and columns involved in the query.
- Parses SPL statements and convert them to p-code

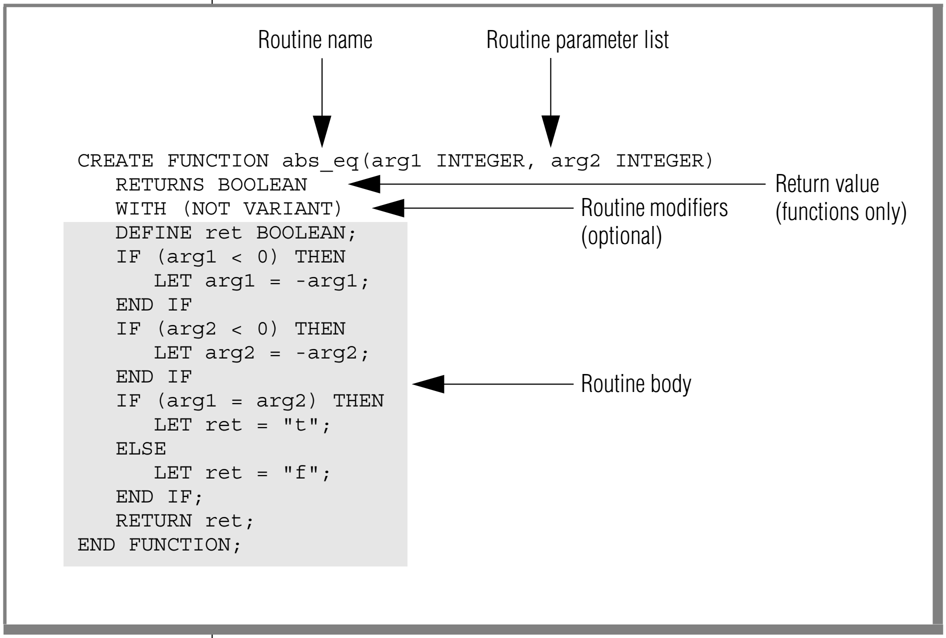
  The term *p-code* refers to pseudocode that an interpreter can execute quickly.
- Converts the p-code, execution plan, and dependency list to ASCII format

  The database server stores these ASCII formats as character columns in the system catalog tables, **sysprocbody** and **sysprocplan**.
- Stores information about the procedure, such as routine name parameters and modifiers, in the **sysprocedures** system catalog table
- Stores permissions for the procedure in the **sysprocauth** system catalog table

For information on how to optimize an SPL routine, see Chapter 13, "Improving UDR Performance."

For a summary of the UDR information in the system catalog tables, refer to "Reviewing Information about User-Defined Routines" on page 4-33.

Figure 4-1 shows the parts of a CREATE FUNCTION statement that registers a user-defined function called **abs_eq()**.



```
CREATE FUNCTION abs_eq(arg1 INTEGER, arg2 INTEGER)
   RETURNS BOOLEAN
   WITH (NOT VARIANT)
   DEFINE ret BOOLEAN;
   IF (arg1 < 0) THEN
      LET arg1 = -arg1;
   END IF
   IF (arg2 < 0) THEN
      LET arg2 = -arg2;
   END IF
   IF (arg1 = arg2) THEN
      LET ret = "t";
   ELSE
      LET ret = "f";
   END IF;
   RETURN ret;
END FUNCTION;
```

Figure 4-1
Registering an SPL Function

When you create an SPL function, you can specify optional routine modifiers that affect how the database server executes the function. Procedures in SPL do not allow routine modifiers. Use the WITH clause of the CREATE FUNCTION statement to list function modifiers. SPL functions allow the following routine modifiers:

- INTERNAL
- NEGATOR
- NOT VARIANT
- VARIANT

In Figure 4-1, the NOT VARIANT modifier indicates that the **abs_eq()** SPL function is written so that it always returns the same value when passed the same arguments.

For more information about the CREATE FUNCTION and CREATE PROCEDURE statements and about the syntax of SPL, refer to the *IBM Informix Guide to SQL: Syntax*. For information about creating using SPL routines, refer to the *IBM Informix Guide to SQL: Tutorial*.

Ext

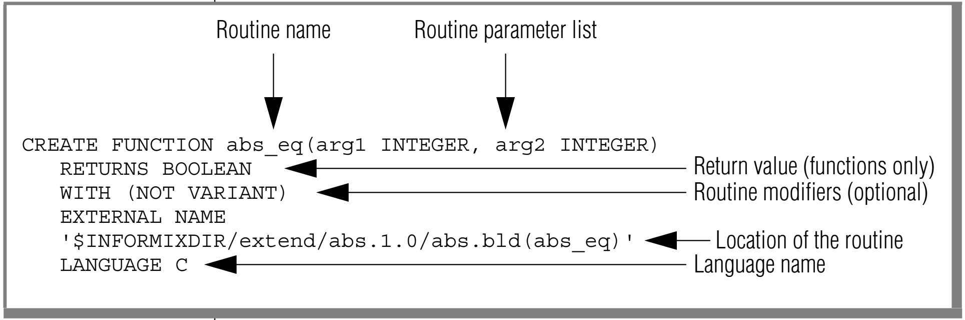
Creating an External-Language Routine

You can write a routine in an external language that the database server supports. After you create a routine, you register the routine with a CREATE FUNCTION or CREATE PROCEDURE statement.

The CREATE FUNCTION and CREATE PROCEDURE statements specify the location of the external routine, as follows:

- For C UDRs, the location is the full pathname of the shared-object module, qualified with the name of the C function that implements the function or procedure.
- For Java UDRs, location is the name of the **.jar** file, followed by the name of the Java class and the name of the method within that class, including its arguments.

For example, Figure 4-2 shows a CREATE FUNCTION statement that registers a user-defined function called **abs_eq()** that is written in C. The corresponding C function is in a shared-object file called **abs.bld**.



```
CREATE FUNCTION abs_eq(arg1 INTEGER, arg2 INTEGER)
   RETURNS BOOLEAN
   WITH (NOT VARIANT)
   EXTERNAL NAME
   '$INFORMIXDIR/extend/abs.1.0/abs.bld(abs_eq)'
   LANGUAGE C
```

Figure 4-2
Registering an External-Language Function

Registering a Routine Written in C

To register a C routine, write the body of the routine, compile it, and create a shared-object file, and then use the CREATE FUNCTION or CREATE PROCEDURE statement to register the function. The RETURNING clause of CREATE FUNCTION specifies the return data type of the function.

For example, the following CREATE FUNCTION statement registers a C function called **equal()** that takes two arguments, **arg1** and **arg2**, of data type **udtype1** and returns a single value of the data type BOOLEAN:

```
CREATE FUNCTION equal (arg1 udtype1, arg2 udtype1)
RETURNING BOOLEAN
EXTERNAL NAME '/usr/lib/udtype1/lib/libbtype1.so(udtype1_equal)'
LANGUAGE C
END FUNCTION;
```

Tip: *In the preceding example, the END FUNCTION keywords are optional. C user-defined-routines can use either RETURNS or RETURNING.*

For more information, see the CREATE FUNCTION and CREATE PROCEDURE statements in the *IBM Informix Guide to SQL: Syntax*. For information about how to create a shared-object file, refer to the *IBM Informix DataBlade API Programmer's Guide*.

Registering a Routine Written in Java

To register a Java routine, write the body of the routine, compile it, create a **.jar** file, and register the **.jar** file with **install_jar()**. Then use the CREATE FUNCTION or CREATE PROCEDURE statement to register the function. For example:

```
CREATE PROCEDURE showusers()
   WITH (class='jvp')
   EXTERNAL NAME 'thisjar:admin.showusers()'
   LANGUAGE java;
```

A UDR written in Java runs on a JVP by default. Therefore, the CLASS routine modifier in the preceding example is optional. However, it is recommended that, to improve readability of your SQL statements, you include the CLASS routine modifier when you register a UDR.

For more information, see the CREATE FUNCTION and CREATE PROCEDURE statements in the *IBM Informix Guide to SQL: Syntax*. For information about how to create a Java routine, refer to the *J/Foundation Developer's Guide*.

Registering an External Routine with Modifiers

When you create a routine in an external language, you can specify optional modifiers that tell the database server about attributes of the UDR. Use the WITH clause of the CREATE FUNCTION and CREATE PROCEDURE statements to list routine modifiers. Following the WITH keyword, the modifiers that you want to specify are enclosed within parentheses and separated by commas.

For more information about using routine modifiers, refer to the *IBM Informix DataBlade API Programmer's Guide*.

C

Modifiers in a C UDR

The following table shows the routine modifiers that are valid for C routines.

		Valid for	
Routine Modifier	**Description**	**External Function**	**External Procedure**
CLASS	Specifies a virtual-processor class in which to run the UDR	Yes	Yes
COSTFUNC	Specifies the name of the cost function for this UDR	Yes	Yes
HANDLESNULLS	Specifies that the UDR can handle null arguments	Yes	Yes
INTERNAL	Specifies that the UDR is an internal routine; that is, that the routine is not available for use in an SQL or SPL statement	Yes	Yes
ITERATOR	Specifies that the UDR is an iterator function	Yes	No
NEGATOR	Specifies that the UDR is a negator function	Yes	No
NOT VARIANT	Specifies that all invocations of the UDR with the same arguments return the same value	Yes	No

(1 of 2)

		Valid for	
Routine Modifier	**Description**	**External Function**	**External Procedure**
PARALLELIZABLE	Routine can be executed in parallel	Yes	Yes
PERCALL_COST	Specifies the cost of execution for the UDR	Yes	Yes
SELCONST	Specifies the selectivity of the UDR	Yes	No
SELFUNC	Specifies the name of the selectivity function for this UDR	Yes	No
STACK	Specifies the stack size for the UDR	Yes	Yes
VARIANT	Specifies that all invocations of the UDR with the same arguments do *not* necessarily return the same value	Yes	No

(2 of 2)

The following example shows how to use the WITH clause to specify a set of modifiers when you create an external-language function:

```
CREATE FUNCTION lessthan (arg1 basetype2, arg2 basetype2)
RETURNING BOOLEAN
WITH (HANDLESNULLS, NOT VARIANT)
EXTERNAL NAME
'/usr/lib/basetype2/lib/libbtype2.so(basetype2_lessthan)'
LANGUAGE C
```

In this example, the HANDLESNULLS modifier indicates that the **basetype2_lessthan()** function (in the shared library **/usr/lib/basetype2/lib/libbtype2.so)** is coded to recognize SQL null. If HANDLESNULL is not set, the routine manager does not execute the UDR if any arguments of the routine are null; it simply returns null.

Java

Modifiers in a Java UDR

The following table shows the routine modifiers that are valid for Java routines.

Routine Modifier	Type of UDR
CLASS	Access to JVP
HANDLESNULLS	UDR that handles SQL null values as arguments
ITERATOR	Iterator function
NEGATOR	Negator function
NOT VARIANT	All invocations of the UDR with the same arguments return the same value
PARALLELIZABLE	Parallelizable UDR
VARIANT	All invocations of the UDR with the same arguments do *not* necessarily return the same value

Registering Parameters and a Return Value

The CREATE FUNCTION and CREATE PROCEDURE statements specify any parameters and return value for a C UDR. These statements use SQL data types for parameters and the return value. For example, suppose a C UDR has the following C declaration:

```
mi_double_precision *func1(parm1, parm2)
   mi_integer parm1;
   mi_double_precision *parm2;
```

The following CREATE FUNCTION statement registers the **func1()** user-defined function:

```
CREATE FUNCTION func1(INTEGER, FLOAT)
RETURNS FLOAT
```

Use the opaque SQL data type, POINTER, to specify a data type for an external-language routine whose parameter or return type has no equivalent SQL data type. The CREATE FUNCTION or CREATE PROCEDURE statement uses the POINTER data type when the data structure that the routine receives or returns is a private data type, not one that is available to users.

Reviewing Information about User-Defined Routines

The following table shows where the database server stores information from CREATE FUNCTION and CREATE PROCEDURE statements in the **sysprocedures** system catalog table.

UDR Information	CREATE Statement Syntax	Column of sysprocedures
Routine type: function or procedure	FUNCTION or PROCEDURE keyword	**isproc**
Owner name (optional)	Precedes the routine name: *owner.routine_name* Defaults to the creator of the routine	**owner**
Routine name	After FUNCTION or PROCEDURE keyword	**procname**
Specific name (optional)	SPECIFIC keyword	**specificname**
Routine parameters	Parameter list	**numargs, paramstyle**, **paramtypes**
Routine modifiers	WITH clause	**variant, handlesnulls, iterator**, **percallcost**, **negator**, **selfunc**, **internal**, **class**, **stack**, **parallelizable, costfunc**, **selconst, modifiers**
Location of the routine (if it is external)	EXTERNAL NAME	**externalname**
Routine language	LANGUAGE	**langid**

The database server assigns a unique identifying number to each UDR and stores this number in the **procid** column of **sysprocedures** table.

For SPL routines, the database server also stores routine information in the **sysprocbody** and **sysprocplan** system catalog tables. The **sysprocbody** table stores both the text and the compiled version (which is not legible) of the SPL routine. The **sysprocplan** table stores a compiled version of the execution plan, which is not legible.

Using a UDR With HDR

If you are using High-Availability Data Replication (HDR), there are some rules you must follow when running UDRs:

- Install the UDR object file on both servers of an HDR pair under the same absolute path name.
- Name the UDR object file identically on both servers of an HDR pair.
- Register the UDR only on the primary server.
- Do not use the UDR to create any persistent external files or persistent memory objects.

Chapter

5 Extending Data Types

In This Chapter

You can extend Dynamic Server by extending existing data types or by creating *user-defined data types* (UDTs). This chapter reviews basic information about the data types. It covers the following topics:

- Understanding the Data Type System
- Understanding Data Types
- Extending the Data Type System

When you create a new data type or extend an existing data type, you use the UDRs that were introduced in Chapter 2, "Using a User-Defined Routine."

Understanding the Data Type System

The data type system that the database server uses is an *extensible* data type system. That is, the data type system is flexible enough to let you:

- Use the data types that the data type system defines and supports.
- Define your own data types.
- Extend the data type system to support additional behavior for data types.

The data type system handles the interaction with the data types. A *data type* is a descriptor that is assigned to a variable or column to indicate the type of data that the variable or column can hold. The database server uses a data type to determine the following information:

- The *data types* that the database server can use

  The data type determines the layout or *internal structure* that the database server can use to store the data type values on disk.
- The *operations* (such as multiplication, string concatenation, casting, or aggregation) that the database server can apply to values of a particular data type

  An operation must be defined on a particular data type. Otherwise, the database server does not allow the operation to be performed.
- The *access methods* that the database server can use for values in columns of this data type:
  - The *primary-access metho*d handles storage and retrieval of a particular data type in a table. If the primary-access method does not handle a particular data type, the database server cannot access values of that type.
  - The *secondary-access method* handles storage and retrieval of a particular data type in an index. If the secondary-access method does not handle a particular data type, you cannot build an index on that data type.
- The *casts* that the database server can use to perform data conversion between values of two different data types

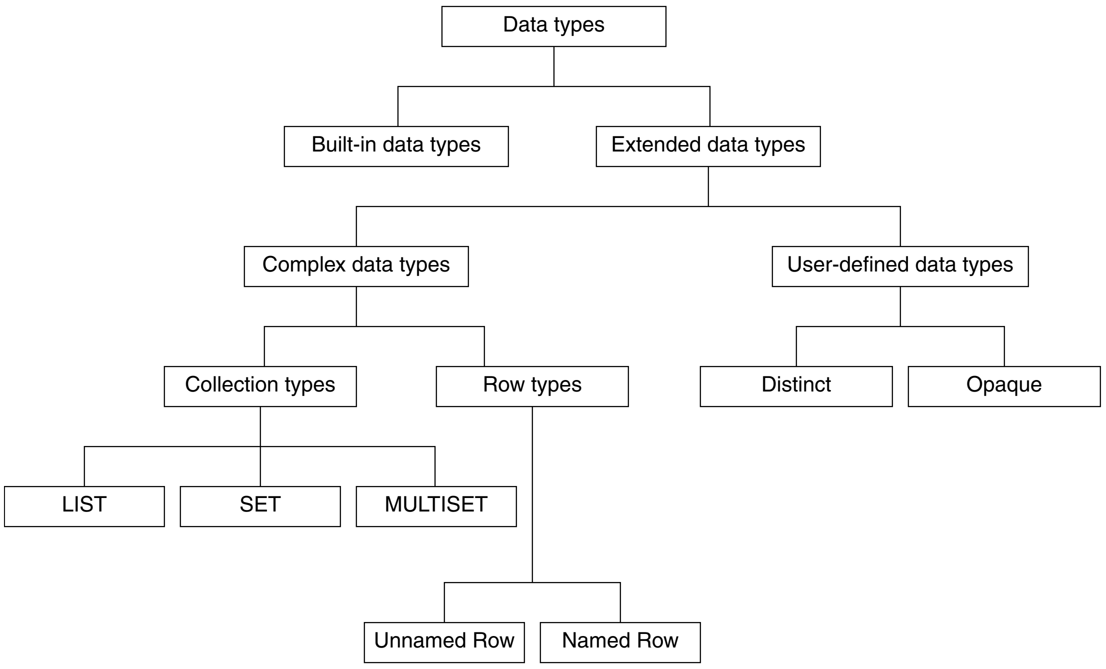
  The database server uses casts to perform data conversion between values of two different data types.

The data type system knows how to provide this behavior for its built-in data types. When you create a UDT, you must provide this information for your data type.

Understanding Data Types

This section gives a brief summary of the data types that the database server supports. Figure 5-1 summarizes the data types.

Figure 5-1
Data Types That the Database Server Supports



For a more detailed description of data types, see the *IBM Informix Database Design and Implementation Guide*.

Built-In Data Types

A built-in data type is a fundamental data type that the database server defines. A fundamental data type is atomic; that is, it cannot be broken into smaller pieces. Built-in data types serve as building blocks for other data types. Figure 5-2 summarizes the built-in data types that the database server provides.

Figure 5-2
Built-In Data Types

Data Type	**Explanation**
BLOB	Stores binary data in smart large objects in a format that supports random access
BOOLEAN	Stores the Boolean values for true and false
BYTE	Stores binary data in chunks that are not random access
CHAR(*n*)	Stores single-byte or multibyte sequences of characters, including letters, numbers, and symbols of fixed length Collation is code-set dependent.
CHARACTER(*n*)	Is a synonym for CHAR
CHARACTER VARYING(*m*,*r*)	Is an ANSI-compliant version of the VARCHAR data type
CLOB	Stores text in smart large objects in a format that supports random access
DATE	Stores a calendar date
DATETIME	Stores a calendar date combined with the time of day
DEC	Is a synonym for DECIMAL
DECIMAL	Stores numbers with definable scale and precision
DOUBLE PRECISION	Behaves the same way as FLOAT

(1 of 3)

Data Type	Explanation
FLOAT(*n*)	Stores double-precision floating-point numbers that correspond to the **double** data type in C (on most platforms)
INT	Is a synonym for INTEGER
INT8	Stores an 8-byte integer value These whole numbers can be in the range -(2^{63}-1) to 2^{63}-1.
INTEGER	Stores whole numbers from -(2^{31}-1) to 2^{31}-1
INTERVAL	Stores a span of time
LVARCHAR	Stores single-byte or multibyte strings of letters, numbers, and symbols of varying length to a maximum of 32 kilobytes It is also the external storage format for opaque data types. Collation is code-set dependent.
MONEY(*p,s*)	Stores a currency amount
NCHAR(*n*)	Stores single-byte and multibyte sequences of characters, including letters, numbers, and symbols Collation is locale dependent. For more information, see the *IBM Informix GLS User's Guide*.
NUMERIC(*p,s*)	Is a synonym for DECIMAL
NVARCHAR(*m,r*)	Stores single-byte and multibyte sequences of characters, including letters, numbers, and symbols of varying length to a maximum of 255 bytes Collation is locale dependent. For more information, see the *IBM Informix GLS User's Guide*.
REAL	Is a synonym for SMALLFLOAT
SERIAL	Stores sequential integers; has the same range of values as INTEGER

(2 of 3)

Data Type	Explanation
SERIAL8	Stores large sequential integers; has the same range of values as INT8
SMALLFLOAT	Stores single-precision floating-point numbers that correspond to the **float** data type in C (on most platforms)
SMALLINT	Stores whole numbers from -(2^{15}-1) to 2^{15}-1
TEXT	Stores text data in chunks that are not random access
VARCHAR(*m*,*r*)	Stores single-byte or multibyte strings of letters, numbers, and symbols of varying length to a maximum of 255 bytes Collation is code-set dependent.

(3 of 3)

Extended Data Types

The extensible data type system allows you to:

- Define new data types, called *extended data types*, to extend the data type system
- Define the behavior of extended data types:
  - The *operations* that are supported on the extended data types
  - New *operator class* that supports the extended data type and provides new functionality for a secondary-access method
  - Additional *casts* to provide data conversions between the extended data types and other data types
  - Functions that collect *statistics* for the optimizer

You can define the following extended data types:

- Complex data types
  - Collection types
  - Row types
- UDTs
  - Opaque data types
  - Distinct data types

The database server stores information about extended data types in the **sysxtdtypes** and **sysxtdtypeauth** system catalog tables. For information about these tables, refer to the *IBM Informix Guide to SQL: Reference*.

Complex Data Types

A *complex data type* is built from a combination of other data types. An SQL statement can access individual components within the complex type. The two kinds of complex types are as follows:

- *Collection types* have instances that are groups of elements of the same data type, which can be any built-in or complex data type.

  The requirements for elements with ordered position and uniqueness among the elements determine whether the collection is a SET, LIST, or MULTISET.
- *Row types* have instances that are groups of related data *fields*, of any data type, that form a template for a record.

  The assignment of a name to the row type determines whether the row type is a named row type or an unnamed row type.

Figure 5-3 summarizes the complex data types that the database server supports.

Figure 5-3
Complex Data Types of the Database Server

Data Type	**Explanation**
LIST(*e*)	Stores a collection of values that have an implicit position (first, second, and so on) and allows duplicate values All elements have the same element type, *e*.
MULTISET(*e*)	Stores a collection of values that have no implicit position and allows duplicate values All elements have the same element type, *e*.
Named row type	A row type created with the CREATE ROW TYPE statement This row type has a defined name and *inheritance* properties and can be used to construct a *typed table*. A named row type is not equivalent to another named row type, even if its field definitions are the same.
ROW (Unnamed row type)	A row type created with the SQL keyword ROW This row type has no defined name and no inheritance properties. Two unnamed row types are equivalent if they have the same number of fields and if corresponding fields have the same data type, even if the fields have different names.
SET(*e*)	Stores a collection of values that have no implicit position and does not allow duplicate values All elements have the same element type, *e*.

User-Defined Data Types

Figure 5-4 summarizes the UDTs that the database server supports.

Figure 5-4
User-Defined Data Types

Data Type	Explanation
Distinct	Has the same internal representation as the source data type on which it is based but has different casts and functions defined over it than those on the source type
Opaque	Fundamental data type that the user defines A fundamental data type is atomic; that is, it cannot be broken into smaller pieces, and it can serve as the building block for other data types.

ANSI

In an ANSI-compliant database, columns defined using user-defined types should be in the **owner.object** format.

Distinct Data Type

A distinct type has the same internal structure as an existing data type. However, it has a distinct name and therefore distinct functions that make it different from its source type. When you define a distinct type, you provide the following information:

- The source data type, which defines the *internal structure* of the distinct data type

  The functions of the source data type determine how the database server interacts with this internal structure.
- The *operations* that are valid on the distinct data type

  You define operator functions, built-in functions, or end-user routines that handle the distinct type. For information about building operator functions, see Chapter 6, "Extending Operators and Built-In Functions."
- Extensions of the *operator class* of a secondary-access method so that its strategy and support functions handle the distinct data type

  For information about support functions, see Chapter 10, "Writing Support Functions."

- Cast functions to provide the data conversions to and from the distinct type

  The database server automatically creates explicit casts between the distinct type and its source type. Because these two data types have the same internal format, this cast does not require a cast function. You can write cast functions to support data conversion between the distinct type and other data types in the database or to support implicit casts between the distinct type and its source data type. For information about writing casts, see Chapter 7, "Creating User-Defined Casts."

You create a distinct data type with the CREATE DISTINCT TYPE statement. After you create the distinct type, you can use it anywhere that other data types are valid. For more information, refer to the description of this statement in the *IBM Informix Guide to SQL: Syntax*.

Opaque Data Type

Unlike other data types (built in, complex, and distinct), the internal structure of the opaque data type is not known to the database server. Therefore, when you define an opaque type, you must provide the following information:

- The *internal structure* of the opaque data type, which provides the format of the data

  You define the *support functions* of the opaque type to tell the database server how to interact with this internal structure.
- The *operations* that are valid on the opaque data type

  You define operator functions, built-in functions, or end-user routines that handle the opaque type.
- Extensions of the *operator class* of a secondary-access method so that its strategy and support functions handle the opaque data type
- Cast functions to provide the data conversions to and from the opaque type

  The support functions of the opaque type also serve as cast functions.

You register an opaque data type with the CREATE OPAQUE TYPE statement. For information about this statement, refer to the *IBM Informix Guide to SQL: Syntax*. For more information, see Chapter 9, "Creating an Opaque Data Type," and Chapter 10, "Writing Support Functions."

IBM Informix DataBlade Modules

In addition to the extended data types that you explicitly define, you can use the pre-packaged extended data types that are provided. For example, an IBM Informix DataBlade module might contain the routines required to support a spherical coordinate system. For more information on IBM Informix DataBlade modules, consult your sales representative or refer to the user guides for the DataBlade modules.

Extending the Data Type System

You can extend the data type system by writing routines that provide the following additional behavior for existing built-in or extended data types:

- Define *operators* to provide additional operations on data types.
- Define *operator classes* to provide new functionality for a secondary-access method (an index) on a data type.
- Define *casts* to provide conversions between data types.
- Define functions that provide information for the optimizer.

You must register each new function in the database with the CREATE FUNCTION statement.

Operations

A data type tells the database server which *operations* it can perform on the data type values. The database server provides the following types of operations on data types:

- An *operator function* implements a particular operator symbol.

  The **plus()** and **times()** functions are examples of operator functions for the + and * operators, respectively.
- A *built-in function* is a predefined function that the database server provides for use in SQL statements.

  The **cos()** and **hex()** functions are examples of built-in functions.
- An *aggregate function* returns a single value for a set of retrieved rows.

  The SUM and AVG functions are examples of aggregate functions.
- An *end-user routine* is a UDR that end users can use in SQL statements to perform some useful action.

The database server provides operator functions, built-in functions, and aggregate functions that handle the data types that it provides. For a description of these operations and how to extend them, see Chapter 6, "Extending Operators and Built-In Functions."

Casts

The database server looks for a cast in the **syscasts** system catalog table to determine which function to use to convert the data type value to a different type. A *cast* performs the necessary operations for conversion from the data type to another data type. When two data types have different internal formats, the database server calls a *cast function* to convert one data type to another. For example, when you add an integer value to a decimal value, the database server performs a cast to change the integer into a decimal so that it can perform the addition.

The database server provides casts between the built-in data types. You might want to create additional casts to provide data conversion between an existing data type and an extended data type that you create. If the two data types have different internal formats, you must define a cast function to perform the data conversion. You must register the cast function with the CREATE FUNCTION statement and create the cast with the CREATE CAST statement before it can be used. For more information on casts, see Chapter 7, "Creating User-Defined Casts."

Operator Classes

An operator class tells the database server which data type (or types) it can index using a secondary-access method. The operator class must follow the requirements of the access method. The secondary-access method builds and accesses an index. An operator class associates a group of operators with a secondary-access method. When you extend an operator class, you provide additional functions that can be used as filters in queries and for which the database server can use an index.

The database server provides a default operator class for the built-in secondary-access method, a generic B-tree. This default operator class uses the relational operators (<, >, =, and so on) to order values in the generic B-tree. These relational operators are defined for the built-in data types.

Providing Additional Operator Classes

To provide additional sequences in which the B-tree can order values in the index, you might want to create an additional operator class for the generic B-tree.

Extending Operator Classes

The default operator class provides only for built-in data types. You might want to extend an operator class to support an extended data type for the following reasons:

- To enable the default operator class to handle values of the extended data type in a generic B-tree
- To provide a new sequence for the values of the extended data type to be stored in a generic B-tree
- To extend an operator class of some other secondary-access method so that it handles the extended data type

To extend or implement an operator class, you must define *strategy and support functions* that handle each extended data type you want to index. For more information, see Chapter 11, "Extending an Operator Class."

You must register each new operator class in the database with the CREATE OPCLASS statement. For information about this statement, refer to the *IBM Informix Guide to SQL: Syntax*.

Optimizer Information

The UPDATE STATISTICS statement collects information for built-in data types. The optimizer uses the information to determine the cost associated with a query.

To collect statistics on opaque and distinct UDTs, you must provide the functions that collect the information. For more information on these functions, see Chapter 13, "Improving UDR Performance."

Chapter

Extending Operators and Built-In Functions

6

In This Chapter

This chapter discusses the operators and built-in functions that you can extend for use with UDTs. An *operation* is a task that the database server performs on one or more values.

The database server provides SQL-invoked functions that provide operations within SQL statements:

- Operator symbols (such as +, -, /, and *) and their associated operator functions
- Built-in functions such as **cos()** and **abs()**
- Aggregate functions such as SUM and AVG

These functions handle the built-in data types. For a UDT to use any of these functions, you can write a new function that has the same name but accepts the UDT in its parameter list.

The property called *routine overloading* allows you to create a user-defined function whose name is already defined in the database but whose parameter list is different. All functions with the same name have the same functionality, but they operate on different data types.

For more information on routine overloading and routine resolution, refer to "Understanding Routine Resolution" on page 3-11. For information about aggregate functions, refer to Chapter 8, "Creating User-Defined Aggregates."

Operators and Operator Functions

An *operator function* implements a particular operator symbol. The database server provides special SQL-invoked functions, called *operator functions*, that implement operators. An operator function processes one to three arguments and returns a value. When an SQL statement contains an operator, the database server automatically invokes the associated operator function.

The association between an operator and an operator function is called *operator binding*. You can overload an operator function to provide the operator for a UDT. The SQL user can then use the operator with the UDT as well as with the built-in data types. When an SQL statement contains an operator, the database server automatically invokes the associated operator function.

Arithmetic Operators

Arithmetic operators usually operate on numeric values. The following table lists the operator functions for the arithmetic operators that the database server provides.

Arithmetic Operator	Operator Function
+ (binary)	**plus()**
- (binary)	**minus()**
*	**times()**
+ (unary)	**positive()**
- (unary)	**negate()**
/	**divide()**

You can overload these operators so that you can use them with user-defined types. For an example of overloading the plus() and divide() functions, refer to "Example of a User-Defined Aggregate" on page 8-14.

Text Operators

Text operators operate on character strings. The following table lists the text operators that the database server provides.

Text Operator	Operator Function
LIKE	**like()**
MATCHES	**matches()**
||	**concat()**

For information on syntax and use of the LIKE and MATCHES operators, see the Condition segment in the *IBM Informix Guide to SQL: Syntax*.

Relational Operators

Relational operators operate on expressions of numeric and string values. The following table lists the operator functions that the database server provides.

Relational Operator	Operator Function
=	**equal()**
<> and !=	**notequal()**
>	**greaterthan()**
<	**lessthan()**
>=	**greaterthanorequal()**
<=	**lessthanorequal()**

All relational operator functions must return a Boolean value. For more information on relational operators, see the Relational Operator segment in the *IBM Informix Guide to SQL: Syntax*.

For end users to be able to use values of a new data type with relational operators, you must write new relational-operator functions that can handle the new data type. In these functions, you can:

- Determine what the relational operators mean for that data type.

  For example, you might create the **circle** opaque data type to implement a circle. A circle is a spatial object that does not have a single value to compare. However, you can define relational operators on this data type that can use the value of its area: one **circle** is less than a second **circle** if its area is less than the area of the second.

- Change from lexicographical sequence to some other ordering for a data type.

  For example, suppose you create a data type, **ScottishName**, that holds Scottish names, and you want to order the data type in a different way than the U.S. English collating sequence. You might want the names `McDonald` and `MacDonald` to appear together on a phone list. You can define relational operators for this data type that equate the strings `Mc` and `Mac`. For more information, see "Changing the Sort Order" on page 11-12.

  After you define the relational operators, you can use SQL statements such as the following one:

  ```
  SELECT * FROM employee
     WHERE emp_name = 'McDonald'::ScottishName
  ```

The relational-operator functions are strategy functions for the built-in secondary-access method, a generic B-tree. For information on strategy functions, see "Operator Classes" on page 11-5.

Overloading an Operator Function

When you write a new version of an operator function, follow these rules:

- The name of the operator function must match the name of an arithmetic, text, or relational-operator function. The name is case insensitive; the **plus()** function is the same as the **Plus()** function.
- The operator function must handle the correct number of parameters.
- The operator function must return the correct data type.

Tip: *Although the **compare()** function is not strictly an operator function, when you overload the relational operators, you should prepare a corresponding **compare()** function, because the database server uses **compare()** to process queries that SELECT DISTINCT or have an ORDER BY clause.*

Built-In Functions

The database server provides special SQL-invoked functions, called *built-in functions*, that provide some basic mathematical operations. For detailed information about built-in functions, see the Expression segment in the *IBM Informix Guide to SQL: Syntax*.

Built-In Functions That You Can Overload

You can overload built-in functions that provide basic operations and certain text and time functions, including the following ones.

abs()	trunc()	atan()	extend()
hex()	exp()	atan2()	decode()
mod()	log10()	length()	nvl()
pow()	logn()	char_length()	initcap()
root()	cos(), sin()	character_length()	lower()
round()	tan()	octet_length()	lpad(), rpad()
sqrt()	acos(), asin()	atan2()	upper()

Built-In Functions That You Cannot Overload

The following sections list built-in functions that you *cannot* overload.

Built-In Aggregates

Each aggregate function uses built-in functions to generate the aggregate result. You *cannot* overload a built-in aggregate function. Instead, you overload the necessary support functions. For a list of the aggregate functions and their related operator functions, refer to "Overloading Operators for Built-In Aggregates" on page 8-4.

Status Functions

You *cannot* overload the following functions that describe time, date, the database server, and the user.

cardinality()	day()	month()	user
current	dbinfo()	sitename	weekday()
date()	dbservername	today	year()
datetime()	mdy()	trim()	

Tip: *Technically, CURRENT, DBSERVERNAME, SITENAME, TODAY, and USER, are not built-in functions, but built-in macros. You can register overloaded routines by those names, but you cannot use them in SQL statements.*

Optical Subsystem Functions

The following table lists the built-in functions for the Optical Subsystem that you *cannot* overload.

descr()	volume()	family()

Overloading a Built-In Function

The database server provides functions that handle the built-in data types. You can write a new version of a built-in function that allows the function to operate on your new data type. If you write a new version of a built-in function, follow these rules:

- The function must be one that you can overload, as listed in "Built-In Functions That You Can Overload" on page 6-7. The name is case insensitive; the **abs()** function is the same as the **Abs()** function.
- The function must handle the correct number of parameters, and these parameters must be the correct data type.
- The function must return the correct data type, where appropriate.

Chapter

Creating User-Defined Casts

7

In This Chapter

A *cast* is a mechanism that converts a value from one data type to another. The database server supports two kinds of cast:

- Understanding Casts
- Creating a User-Defined Cast
- Dropping a Cast

This chapter describes how to create casts for UDTs.

Understanding Casts

Casts allow you to make comparisons between values of different data types or substitute a value of one data type for a value of another data type. For example, when you add a floating-point number to an integer, the computer must change (cast) the integer to a floating-point value before it can perform the addition.

Built-In Casts

A *built-in cast* performs an automatic conversion between two built-in data types. The database server provides casts between most of the built-in data types.

For more information on built-in casts, refer to the chapter on data types in the *IBM Informix Database Design and Implementation Guide*.

User-Defined Casts

A *user-defined cast* is a cast that you define to perform conversion from one UDT to another data type, either built-in or user-defined. You can create user-defined casts to perform conversions between most data types, including opaque types, distinct types, row types, and built-in types.

Opaque Data Types

When you create an opaque data type, you define casts to handle conversions between the internal and external representations of the opaque data type. You might also create casts to handle conversions between the opaque data type and other data types in the database.

For information about how to create and register casts for opaque data types, see "Creating Casts for Opaque Data Types" on page 9-14.

Distinct Data Types

When you create a distinct data type, the database server automatically registers explicit casts from the distinct data type to the source data type and from the source data type to the distinct data type. You must create casts on distinct types to handle conversions between the new distinct data type and other data types in the database or use explicit casts in your SQL statements.

For more information and examples that show how you can create and use casts for distinct types, refer to the chapter on casting in the *IBM Informix Database Design and Implementation Guide*.

Named Row Types

In most cases, you can explicitly cast a named row type to another row type value without creating the cast. However, in some cases, you might want to create a cast that allows for comparisons between a named row type and some other data type.

For information about how to cast between named row types and unnamed row types, refer to the chapter on casting in the *IBM Informix Database Design and Implementation Guide*.

Casts That You Cannot Create

You cannot create a user-defined cast that includes any of the following data types as either the source data type or target data type for the cast:

- Collection data types: LIST, MULTISET, or SET
- Unnamed row types
- Smart-large-object data types: CLOB or BLOB
- Simple-large-object data types: TEXT or BYTE

Creating a User-Defined Cast

You create a user-defined cast with the CREATE CAST statement, which registers the cast in the **syscasts** system catalog table. The person who registers a cast with CREATE CAST owns the cast.

For information about the syntax of the CREATE CAST statement, refer to the *IBM Informix Guide to SQL: Syntax*. For a general discussion of using casts, refer to the *IBM Informix Database Design and Implementation Guide*.

The CREATE CAST statement provides the following information about the cast to the database server:

- The kind of user-defined cast to create
- The cast mechanism that the database server is to use to perform the data conversion
- The direction of the cast

  The CREATE CAST statement specifies the source and target data types to determine the direction of the cast. For full data conversion between two data types, you must define one cast in each direction of the conversion.

Choosing the Kind of User-Defined Cast

You specify how a database server treats a cast when you use the CREATE CAST statement. The database server supports two kinds of user-defined casts:

- Implicit cast
- Explicit cast

  The database server invokes an explicit cast to perform conversions between two data types only when you specify the CAST AS keywords or the double colon (::) cast operator.

Implicit Cast

An implicit cast governs what automatic data conversion occurs for an operation that involves two different data types. All casts between built-in data types are implicit.

The database server automatically invokes an implicit cast when it performs the following tasks:

- It passes arguments of one data type to a UDR whose parameters are of another data type.
- It evaluates expressions and needs to operate on two similar data types.

Conversion of one data type to another can involve loss of data. Be careful of creating implicit casts for such conversions. The end user cannot control when the database server invokes an implicit cast and therefore cannot avoid the loss of data that is inherent to such a conversion.

The database server invokes an implicit cast automatically, without a cast operator. However, you also can explicitly invoke an implicit cast with the CAST AS keywords or the :: cast operator.

To create an implicit cast, specify the IMPLICIT keyword of the CREATE CAST statement. The following CREATE CAST statement creates an implicit cast from the **percent** data type to the DECIMAL data type:

```
CREATE IMPLICIT CAST (percent AS DECIMAL)
```

Explicit Cast

An explicit cast governs what data conversion an end user can specify for UDTs (such as opaque data types, distinct data types, and row types). The database server invokes an explicit cast *only* when it encounters one of the following syntax structures:

- The CAST AS keywords

  For example, the following expression uses the CAST AS keywords to invoke an explicit cast between the **percent** and INTEGER data types:

  ```
  WHERE col1 > (CAST percent AS INTEGER)
  ```
- The :: cast operator

  For example, the following expression uses the cast operator to invoke an explicit cast between the **percent** and INTEGER data types:

  ```
  WHERE col1 > (percent::INTEGER)
  ```

The conversion of one data type to another can involve loss of data. If you define such conversions as explicit casts, the end user can control when the loss of data that is inherent to such a conversion is acceptable.

To create an explicit cast, specify the EXPLICIT keyword of the CREATE CAST statement. If you omit the keyword, the default is an explicit cast. Both of the following CREATE CAST statements create explicit casts from the **percent** data type to the INTEGER data type:

```
CREATE EXPLICIT CAST (percent AS INTEGER)
CREATE CAST (percent AS INTEGER)
```

Choosing the Cast Mechanism

The CREATE CAST statement can optionally specify the name of a cast function that implements the cast. The database server does not automatically perform data conversion on extended data types. You must specify a cast function if the two data types have different internal structures.

The database server can implement a cast with one of following mechanisms:

- Perform a straight cast if two data types have internal structures that are the same
- Call a cast function to perform the data conversion

Straight Cast

A *straight cast* tells the database server that two data types have the same internal structure. With such a cast, the database server does not need to manipulate data to convert from the source data type to the target data type. Therefore, you do not need to specify a WITH clause in the CREATE CAST statement.

For example, suppose you need to compare the values of an INTEGER data type and a UDT **my_int** that has the same internal structure as the INTEGER data type. This conversion does not require a cast function because the database server does not need to perform any manipulation on the values of these two data types to compare them. The following CREATE CAST statements create the explicit casts that allow you to convert between values of data type INT and **my_int**:

```
CREATE CAST (INT AS my_int)
CREATE CAST (my_int AS INT)
```

The first cast defines a valid conversion from INT to **my_int**, and the second cast defines a valid conversion from **my_int** to INT.

Built-in casts have no cast function associated with them. Because a distinct data type and its source data type have the same internal structure, distinct types do not require cast functions to be cast to their source data type. The database server automatically creates explicit casts between a distinct data type and its source data type.

Cast Function

You can create special SQL-invoked functions, called cast functions, that implement data conversion between two dissimilar data types. When two data types have different storage structures, you must create a cast function that defines how to convert the data in the source data type to data of the target data type.

To create a cast that has a cast function

1. Write the cast function.

   The cast function takes the source data type as its argument and returns the target data type.
2. Register the cast function with the CREATE FUNCTION statement.
3. Register the cast with the CREATE CAST statement.

   Use the WITH clause of the CREATE CAST statement to specify the cast function. To invoke a cast function, the function must reside in the current database. However, the cast function does not need to exist when you register the cast.

Example of a Cast Function

For example, suppose you want to compare values of two opaque data types, **int_type** and **float_type**. Both types have an external LVARCHAR format that you can use as an intermediate type for converting from one to the other. The CREATE FUNCTION statement in Figure 7-1 creates and registers an SPL function, **int_to_float()**, as an argument. It casts the **int_type** input value to an LVARCHAR, and then casts the LVARCHAR result to **float_type** and returns the **float_type** result.

```
CREATE FUNCTION int_to_float(int_arg int_type)
   RETURNS float_type
   RETURN CAST(CAST(int_arg AS LVARCHAR) AS float_type);
END FUNCTION;
```

Figure 7-1
An SPL Function as a Cast Function from int_type to float_type

The **int_to_float()** function uses a nested cast and the support functions of the **int_type** and **float_type** opaque types to obtain the return value, as follows:

1. The **int_to_float()** function converts the **int_type** argument to LVARCHAR with the inner cast:

   ```
   CAST(int_arg AS LVARCHAR)
   ```

   The output support function of the **int_type** opaque data type serves as the cast function for this inner cast. This output support function must be defined as part of the definition of the **int_type** opaque data type; it converts the internal format of **int_type** to its external (LVARCHAR) format.

2. The **int_to_float()** function converts the LVARCHAR value to **float_type** with the outer cast:

   ```
   CAST((LVARCHAR value from step 1) AS float_type)
   ```

   The input support function of the **float_type** opaque data type serves as the cast function for this outer cast. This input support function must be defined as part of the definition of the **float_type** opaque data type; it converts the external (LVARCHAR) format of **float_type** to its internal format.

For information about input and output support functions, refer to "Locale-Sensitive Input and Output Support Functions" on page 10-30.

After you create this cast function, use the CREATE CAST statement to register the function as a cast. You cannot use the function as a cast until you register it with the CREATE CAST statement. The CREATE CAST statement in Figure 7-2 creates an explicit cast that uses the **int_to_float()** function as its cast function.

```
CREATE EXPLICIT CAST (int_type AS float_type
   WITH int_to_float);
```

Figure 7-2
An Explicit Cast from int_type to a float_type

After you register the function as an explicit cast, the end user can invoke the function with the CAST AS keywords or with the :: cast operator to convert an **int_type** value to a **float_type** value. For the syntax of the CREATE FUNCTION and CREATE CAST statements, refer to the *IBM Informix Guide to SQL: Syntax*.

Defining the Direction of the Cast

A cast tells the database server how to convert from a source data type to a target data type. The CREATE CAST statement provides the name of the source and target data types for the cast. The source data type is the data type that needs to be converted, and the target data type is the data type to which the source data type should be converted. For example, the following CREATE CAST statement creates a cast whose source data type is DECIMAL and whose target data type is a UDT called **percent**:

```
CREATE CAST (DECIMAL AS percent)
```

When you register a user-defined cast, the combination of source data type and target data type must be unique within the database.

To provide data conversion between two data types, you must define a cast for each direction of the conversion. For example, the explicit cast in Figure 7-2 enables the database server to convert from the **int_type** opaque data type to the **float_type** opaque data type. Therefore, the end user can perform the following cast in an INSERT statement to convert an **int_type** value, **it_val**, to a **float_type** column, **ft_col**:

```
INSERT INTO table1 (ft_col) VALUES (it_value::float_type)
```

However, this cast does *not* provide the inverse conversion: from **float_type** to **int_type**. If you try to insert a **float_type** value in an **int_type** column, the database server generates an error. To enable the database server to perform this conversion, you need to define another cast function, one that takes a **float_type** argument and returns an **int_type** value. Figure 7-3 shows the CREATE FUNCTION statement that defines the **float_to_int()** SPL function.

```
CREATE FUNCTION float_to_int(float_arg float_type)
   RETURNS int_type
   RETURN CAST(CAST(float_arg AS LVARCHAR) AS int_type);
END FUNCTION;
```

Figure 7-3
An SPL Function as a Cast Function from float_type to int_type

The **float_to_int()** function also uses a nested cast and the support functions of the **int_type** and **float_type** opaque types to obtain the return value:

1. The **float_to_int()** function converts the **float_type** value to LVARCHAR with the inner cast.

   ```
   CAST(float_arg AS LVARCHAR)
   ```

   The output support function of the **float_type** opaque data type serves as the cast function for this inner cast. This output support function must be defined as part of the definition of the **float_type** opaque data type; it converts the internal format of **float_type** to its external (LVARCHAR) format.

2. The **float_to_int()** function converts the LVARCHAR value to **int_type** with the outer cast.

   ```
   CAST(LVARCHAR value AS int_type)
   ```

   The input support function of the **int_type** opaque data type serves as the cast function for this outer cast. This input support function must be defined as part of the definition of the **int_type** opaque data type; it converts the external (LVARCHAR) format of **int_type** to its internal format.

The CREATE CAST statement in Figure 7-4 creates an explicit cast that uses the **int_to_float()** function as its cast function.

```
CREATE EXPLICIT CAST (float_type AS int_type
   WITH float_to_int);
```

Figure 7-4
An Explicit Cast from float_type to int_type

The end user can now perform the following cast in an INSERT statement to convert a **float_type** value, **ft_val**, for an **int_type** column, **it_col**:

```
INSERT INTO table1 (it_col) VALUES (ft_value::int_type)
```

Together, the explicit casts in Figure 7-2 on page 7-10 and in Figure 7-4 enable the database server to convert between the **float_type** and **int_type** opaque data types. Each explicit cast provides a cast function that performs one direction of the conversion.

Dropping a Cast

The DROP CAST statement removes the definition for a cast from the database. The database server removes the class definition from the **syscasts** system catalog table. You must be the owner (the person who created the cast) or the DBA to drop its definition from the database.

Warning: *Do not drop the built-in casts, which user **informix** owns. The database server uses built-in casts for automatic conversions between built-in data types. Do not drop support functions for opaque data types that serve as casts if you still want to use the opaque data type in the database.*

The following statements create and then remove casts between the **mytype** and DECIMAL data types:

```
CREATE CAST (decimal AS mytype WITH dec_to_mytype);
CREATE CAST (mytype AS decimal WITH mytype_to_decimal);
...
...
DROP CAST (decimal AS mytype);
DROP CAST (mytype AS decimal);
```

Dropping a cast has no effect on the function associated with the cast. The previous statements do not affect the **dec_to_mytype** or **mytype_to_decimal** functions. Use the DROP FUNCTION statement to remove a function from the database. For information about the syntax of DROP CAST and DROP FUNCTION, refer to the *IBM Informix Guide to SQL: Syntax*.

Chapter

8 Creating User-Defined Aggregates

In This Chapter

This chapter describes how to extend the functionality of aggregates in the database server. An *aggregate* is a function that returns one value for a set of queried rows. The database server provides two ways to extend aggregates:

- Extensions of built-in aggregates

  A *built-in aggregate* is an aggregate that the database server provides, such as COUNT, SUM, or AVG. You can extend the built-in aggregates for use with UDTs.

- User-defined aggregates

  A *user-defined aggregate* is an aggregate that you define to provide an aggregate function that the database server does not provide.

The term *user-defined aggregates* is often used loosely to include both extensions of built-in aggregates and new, user-defined aggregates. The database server manages all aggregates, whether built in or user defined. After you create an extension to the aggregate system, you use all aggregates in the same way, regardless of how the aggregate was created.

The techniques for providing the two types of extensions are different. This chapter provides separate discussions of the two methods for extending aggregates.

For information about how to use aggregates in SELECT statements, refer to the *IBM Informix Guide to SQL: Tutorial*. For information about the syntax of aggregates, refer to the *IBM Informix Guide to SQL: Syntax*.

Extending Existing Aggregates

The database server provides built-in aggregate functions, such as SUM and COUNT, that operate on the built-in data types. You can extend a built-in aggregate so that it can operate on UDTs. To extend a built-in aggregate, you must create UDRs that overload several binary operators.

Overloading Operators for Built-In Aggregates

The following table shows the operators that you must overload for each of the built-in aggregates. For example, if you need only the SUM aggregate for a UDT, you need to overload only the **plus()** operator.

Aggregate	Required Operators	Return Type
AVG	plus(udt, udt), divide(udt, integer)	Return type of divide()
COUNT	-- (no new operators required)	Integer
COUNT DISTINCT	equal(udt,udt)	Boolean
DISTINCT (or UNIQUE)	compare(udt, udt)	Boolean
MAX	greaterthanorequal(udt, udt)	Boolean
MIN	lesthanorequal(udt, udt)	Boolean
RANGE	lessthanorequal(udt, udt), greaterthanorequal(udt, udt), minus(udt, udt)	Return type of minus()

(1 of 2)

Aggregate	Required Operators	Return Type
SUM	plus(udt, udt)	Return type of plus()
STDEV	times(udt, udt), divide(udt, integer), plus(udt, udt), minus(udt, udt), sqrt(udt)	Return type of divide()
VARIANCE	times(udt, udt), divide(udt, integer), plus(udt, udt), minus(udt, udt)	Return type of divide()

(2 of 2)

The database server uses the **compare()** function for indexing as well as for DISTINCT and UNIQUE aggregations. However, the database server calls the **equal()** function to process COUNT DISTINCT. You must write the **compare()** function in C or in Java.

Extending an Aggregate

When you extend a built-in aggregate to include a UDT, you do not use the CREATE AGGREGATE statement because the aggregate itself already exists.

To extend a built-in aggregate

1. Develop support functions to overload the required operators.
2. Register each function with a CREATE FUNCTION statement.

   For more information, refer to "Registering a User-Defined Routine" on page 4-23.

After you register the support functions that overload the binary operators, you can use the built-in aggregates in an SQL statement.

For the syntax of the CREATE FUNCTION statement, see the *IBM Informix Guide to SQL: Syntax*. For more information about how to write overloaded functions, refer to "Overloading Routines" on page 3-13. For information about how to write functions in external languages, refer to the *IBM Informix DataBlade API Programmer's Guide* or the *J/Foundation Developer's Guide*.

Example of Extending a Built-In Aggregate

The following example uses SPL functions to overload the **plus()** and **divide()** operators for a row type, **complex**, that represents a complex number. After you overload the operators, you can use the SUM, AVG, and COUNT operators with **complex**.

```
CREATE ROW TYPE complex(real FLOAT, imag FLOAT);

CREATE FUNCTION plus (c1 complex, c2 complex)
   RETURNING complex;
   RETURN row(c1.real +c2.real, c1.imag +c2.imag)::complex;
END FUNCTION;

CREATE FUNCTION divide (c1 complex, count INT)
   RETURNING complex;
   RETURN row(c1.real/count, c1.imag/count)::complex;
END FUNCTION;
```

You can now use the extended aggregates as follows:

```
CREATE TABLE c_test (a complex, b integer);
INSERT INTO c_test VALUES (ROW(4,8)::complex,14);
INSERT INTO c_test VALUES (ROW(7,9)::complex,3);
...
SELECT SUM(a) FROM c_test;
SELECT AVG(a) FROM c_test;
SELECT COUNT(a) FROM c_test;
```

Creating User-Defined Aggregates

A user-defined aggregate extends the database server by providing information that allows the database server to apply that aggregate to data in the database. To create a user-defined aggregate, write and register support functions that perform the aggregation and then implement the aggregate with the CREATE AGGREGATE statement.

The CREATE AGGREGATE statement provides the following information about the aggregate to the database server:

- The name of the aggregate
- The owner of the aggregate
- The names of the functions that support the aggregate
- Modifiers to the aggregate

For the syntax of the CREATE AGGREGATE statement, see the *IBM Informix Guide to SQL: Syntax*.

You cannot create a user-defined aggregate for any of the following data types:

- Collection data types: LIST, MULTISET, or SET
- Unnamed row types
- Smart-large-object data types: CLOB or BLOB
- Simple-large-object data types: TEXT or BYTE

Support Functions

The CREATE AGGREGATE statement expects information about four support functions. The following table summarizes these support functions. You must provide support functions for each data type that will use the aggregate.

Function Type	Purpose
INIT	Initializes the data structures required for computing the aggregate
ITER	Merges a single (row) value with the previous partial result
COMBINE	Merges one partial result with another partial result, thus allowing parallel execution of the aggregate
FINAL	Converts the partial result into the final value It can perform clean-up operations and release resources.

You can write the support functions in SPL, C, or Java. For information about SPL, refer to the *IBM Informix Guide to SQL: Syntax*. For information about writing functions in external languages, refer to the *IBM Informix DataBlade API Programmer's Guide* or the *J/Foundation Developer's Guide*.

The following CREATE AGGREGATE statement registers the SUMSQ aggregate with support functions named **init_func**, **iter_func**, **combine_func**, and **final_func**. You can register an aggregate even though you have not yet written the support functions.

```
CREATE AGGREGATE sumsq
    (INIT = init_func,
     ITER = iter_func,
     COMBINE = combine_func,
     FINAL = final_func);
```

When you create a user-defined aggregate, you must overload each support function to provide for each data type on which the aggregate will operate. That is, if you create a new aggregate, SUMSQ, whose iterator function is **iter_func**, you must overload the **iter_func** function for each applicable data type. Aggregate names are not case sensitive. When you create and use an aggregate, you can use either uppercase or lowercase.

INIT Function

The **INIT** function initializes the data structures required by the rest of the computation of the aggregate. For example, if you write a C function, the **INIT** function can set up large objects or temporary files for storing intermediate results. The **INIT** function returns the initial result of the aggregate, which is of the state type.

The **INIT** function can take one or two arguments. The first argument must be the same type as the column that is aggregated. The database server uses the type of the first argument to resolve overloaded **INIT** functions.

Ext

The first argument of the **INIT** function is a dummy argument and always has a null value. Therefore, all functions that serve as **INIT** functions must be defined with the HANDLESNULLS modifier. ♦

Omitting the INIT Function

You can omit the **INIT** function for simple binary operators whose state type is the same as the type of the first argument of the aggregate. In that case, the database server uses null as the initial result value.

Using the Optional Second Argument

You can use the optional second argument of the **INIT** function as a setup argument to customize the aggregate computation. For example, you could prepare an aggregate that would exclude the *N* largest and *N* smallest values from its calculation of an average. In that case, the value of *N* would be the second argument of the aggregate expression.

The setup expression must come from the group-by columns because the value of the setup should remain the same throughout the computation of the aggregate.

C

The setup expression cannot be a lone host variable reference. ♦

ITER Function

The iteration function, **ITER**, merges a single value with a partial result and returns a partial result. The **ITER** function does the main job of processing the information from each row that your query selects. For example, for the AVG aggregate, the **ITER** function adds the current value to the current sum and increments the row count by one.

The **ITER** function is required for all user-defined aggregates. If no **INIT** function is defined for a user-defined aggregate, the **ITER** function must explicitly handle nulls.

The **ITER** function obtains the state of the aggregate computation from its state argument.

SPL routines handle null arguments by default. In C and Java functions, you must explicitly handle null values in the **ITER** function and register the function with the HANDLESNULLS modifier.

C

The **ITER** function should not maintain additional states in its FPARAM structure because the FPARAM structure is not shared among support functions. However, you can use the FPARAM structure to cache information that does not affect the aggregate result. ♦

FINAL Function

The **FINAL** function converts the internal result to the result type that it returns to the user. For example, for the AVG aggregate, the **FINAL** function returns the current sum divided by the current row count.

The **FINAL** function is not required for aggregates that are derived from simple binary operators whose result type is the same as the state type and the column type. If you do not define a **FINAL** function, the database server simply returns the final state.

Ext

The **FINAL** function can perform cleanup work to release resources that the **INIT** function allocated. However, it must not free the state itself. ♦

COMBINE Function

The **COMBINE** function merges one partial result with another partial result and returns the updated partial result. For example, for the AVG aggregate, the **COMBINE** function adds the two partial results and adds the two partial counts.

If the aggregate is derived from a simple binary operator whose result type is the same as the state type and the column type, the **COMBINE** function can be the same as the **ITER** function. For example, for the AVG aggregate, the **COMBINE** function adds the current sum and the row count of one partial result to the same values for another partial result and returns the new values.

The database server uses the **COMBINE** function for parallel execution. When a query includes an aggregate, the database server uses parallel execution when the query includes only aggregates. However, the **COMBINE** function might also be used even when a query is not parallelized. For example, when a query contains both distinct and nondistinct aggregates, the database server can decompose the computation of the nondistinct aggregate into subaggregates based on the distinct column values. Therefore, you must provide a **COMBINE** function for each user-defined aggregate.

Parallel aggregation must give the same results as an aggregate that is not computed in parallel. You must write the **COMBINE** function so that the result of aggregating over a set of rows is the same as aggregating over two partitions of the set separately and then combining the results.

Ext

The **COMBINE** function can perform clean-up work to release resources that the **INIT** function allocated. However, it must not free the state arguments. ♦

Resolving the Support Functions

When an SQL statement uses a user-defined aggregate, the database server resolves the support functions to the proper UDRs.

The database server resolves the support functions without a *database owner* name. Therefore, the user-defined function resolution logic attempts the following schemas, respectively: the current user, the schema of the argument types, and the Informix schema, respectively. For more information about routine resolution, refer to "Understanding Routine Resolution" on page 3-11.

Support-Function States

The database server uses the following steps to find the support functions:

1. If the CREATE AGGREGATE statement includes an **INIT** function, resolve the following UDR:

   ```
   init_func (dt_agg, dt_setup)
   ```

   The return type of the **INIT** function establishes a *state type* that the database server uses to resolve the other support functions. If the **INIT** function is omitted, the state type is the data type of the argument of the aggregate.
2. For the **ITER** function, resolve the following UDR:

   ```
   iter_func (state_type, dt_agg)
   ```

   The return type of the **ITER** function should be the state type.
3. For the **COMBINE** function, resolve the following UDR:

   ```
   comb_func (state_type, state_type)
   ```

   The return type of the **COMBINE** function should be the state type.
4. If the **FINAL** function is specified, resolve the following UDR:

   ```
   final_func (state_type)
   ```

   The return type of the user-defined aggregate is the return type of the **FINAL** function. If the **FINAL** function is not specified, the return type is the state type.

The preceding steps use the following variables.

Variable	Description
comb_func	Name of the **COMBINE** function
dt_aggr	Data type of the first argument of the aggregate
dt_setup	Data type of the second, or setup, argument of the aggregate
final_func	Name of the **FINAL** function
init_func	Name of the **INIT** function
iter_func	Name of the **ITER** function
state_type	The state type that the return value of the **INIT** function establishes

Aggregate states should never be null. That is, the support functions should not return a null value. The database server cannot distinguish a null value from the result of aggregating over an empty table. Therefore, although null values do not cause runtime errors, the **COMBINE** function and the **FINAL** function ignore them.

Ext

Using C or Java Support Functions

When you use C or Java to write routines for the support functions, you must consider the treatment of null values. Unless the HANDLESNULLS modifier is present, rows with null values in the column that is aggregated do not contribute to the aggregate computation. If the iteration function, **ITER**, uses HANDLESNULLS, all of the support functions must be declared to handle null values. The initialization function, **INIT**, must always be able to handle null values.

User-defined aggregates are strongly typed. That is, the database server uses the state type information from the support functions to ensure that values are well typed and that their memory is properly managed. With caution, you might be able to use the generic user-defined type *pointer* to avoid creating a new state type.

To create a user-defined aggregate

1. Write the functions that support the aggregate.
2. Register the support function with the CREATE FUNCTION statement.
3. Register the aggregate with the CREATE AGGREGATE statement.

After you register the aggregate, you can use the aggregate in an SQL statement.

For more information about registering a function, refer to "Registering a User-Defined Routine" on page 4-23. For the syntax of the CREATE FUNCTION and CREATE AGGREGATE statements, see the *IBM Informix Guide to SQL: Syntax*.

Example of a User-Defined Aggregate

The following example uses SPL functions to provide the support functions for a new aggregate, SUMSQ, that calculates the sum of squares. After you register the support functions and create the aggregate, you can use the SUMSQ aggregate with any column that has a data type that casts to a float data type.

```
CREATE FUNCTION ssq_init (dummy float)
   RETURNING float;
   RETURN 0;
END FUNCTION;

CREATE FUNCTION ssq_iter (result float, value float)
   RETURNING float;
   RETURN result + value * value;
END FUNCTION;

CREATE FUNCTION ssq_combine(partial1 float, partial2 float)
   RETURNING float;
   RETURN partial1 + partial2;
END FUNCTION;

CREATE FUNCTION ssq_final(final float)
   RETURNING float;
   RETURN final;
END FUNCTION;

CREATE AGGREGATE sumsq WITH
   (INIT = ssq_init,
    ITER = ssq_iter,
    COMBINE = ssq_combine,
    FINAL = ssq_final);
```

Now, for example, you can use SUMSQ with the INTEGER column of the **c_test** table illustrated in "Example of Extending a Built-In Aggregate" on page 8-6.

```
SELECT SUMSQ(b) FROM c_test;
```

Using User-Defined Data Types with User-Defined Aggregates

You cannot use SUMSQ with the **complex** column of the **c_test** table illustrated in "Example of Extending a Built-In Aggregate" on page 8-6 because the **complex** data type does not cast to the FLOAT data type. To use SUMSQ with the complex data type, you must overload the support functions of the SUMSQ aggregate.

```
CREATE FUNCTION ssq_init (dummy complex)
   RETURNING complex;
   RETURN ROW(0,0)::complex;
END FUNCTION;

CREATE FUNCTION ssq_iter (partial complex, c complex)
   RETURNING complex;
   RETURN ROW (
      (partial.real + c.real*c.real - c.imag*c.imag),
      (partial.imag + 2*c.real*c.imag)
      )::complex;
END FUNCTION;

CREATE FUNCTION ssq_combine(p1 complex, p2 complex)
   RETURNING complex;
   RETURN ROW(p1.real + p2.real,
           p1.imag + p2.imag)::complex;
END FUNCTION;

CREATE FUNCTION ssq_final(final complex)
   RETURNING complex;
   RETURN final::complex;
END FUNCTION;
```

When you overload support functions for a user-defined aggregate, you must prepare exactly the same functions as those declared in the CREATE AGGREGATE statement. In this example, that requirement means overloading each of the support functions.

Omitting Support Functions

For completeness, the preceding examples show all four support functions: **INIT**, **ITER**, **COMBINE**, and **FINAL**. Because SUMSQ is a simple aggregate, the examples could have omitted the **INIT** and **FINAL** functions. You could use the following commands to create the SSQ2 aggregate:

```
CREATE FUNCTION ssq2_iter (result float, opr float)
   RETURNING float;
   IF result IS NULL THEN
     LET result = (opr*opr);
   ELSE
     LET result = result + opr*opr;
   END IF
   RETURN result;
END FUNCTION;

CREATE FUNCTION ssq2_combine(partial1 float, partial2 float)
   RETURNING float;
   RETURN partial1 + partial2;
END FUNCTION;

CREATE AGGREGATE ssq2 WITH
   (ITER = ssq2_iter,
    COMBINE = ssq2_combine);
```

Difference Between SUMSQ and SSQ2 Aggregates

The **INIT** function for SUMSQ explicitly initializes the state; that is, the result. Because the SSQ2 aggregate does not include an **INIT** function, the **ITER** function must explicitly handle the case where the result is null.

The behavior of the SSQ2 aggregate is not exactly the same as that of the SUMSQ aggregate. You can use SSQ2 *only* with a column of the FLOAT data type unless you explicitly cast the column to FLOAT. In the following example, the first SELECT statement fails, but the other SELECT statements succeed:

```
CREATE TABLE trial (t INT);
  INSERT INTO trial VALUES (2);
  INSERT INTO trial VALUES (3);
SELECT ssq2(t) FROM trial;           -- fails
SELECT ssq2(t::float) FROM trial;    -- succeeds
SELECT sumsq(t) from trial;          -- succeeds
```

Because the **INIT** function was omitted from the declaration of SSQ2, the aggregate uses the data type of the aggregate argument as its state type. The **ITER** function expects a FLOAT data type. Thus, when the **INIT** function is omitted, the aggregate argument must be a FLOAT data type. For more about the state type, refer to "Resolving the Support Functions" on page 8-11.

Overloading the Support Functions for SSQ2

Because any overloaded functions must be the same as those in the declaration of the aggregate, you must overload **ssq2_iter** and **ssq2_combine** to extend the SSQ2 aggregate to the complex data type.

```
CREATE FUNCTION ssq2_iter (partial complex, c complex)
   RETURNING complex;
   RETURN ROW (
      (partial.real + c.real*c.real - c.imag*c.imag),
      (partial.imag + 2*c.real*c.imag)
      )::complex;
END FUNCTION;

CREATE FUNCTION ssq2_combine(p1 complex, p2 complex)
   RETURNING complex;
   RETURN ROW(p1.real + p2.real,
           p1.imag + p2.imag)::complex;
END FUNCTION;
```

Managing Aggregates

The database server provides tools for managing user-defined or user-extended aggregates and their associated functions.

Parallel Execution of Aggregates

In aggregate-only queries, the database server can break the computation of the aggregate into several pieces and compute each piece in parallel. The database server then uses the **COMBINE** function to combine the partial results from all pieces in a single result value. The database server uses the optimizer to decide when and how to parallelize an aggregate. This action is transparent to the user.

In queries that are not exclusively aggregate, the database server can still compute multiple aggregate results in parallel. In such cases, the database server computes each aggregate result sequentially (without using the **COMBINE** function).

For more information about parallelization and optimization, refer to the *Performance Guide*.

Privileges for User-Defined Aggregates

No privileges are directly associated with user-defined or user-extended aggregates. Instead, you must set the correct privileges for the functions that support the aggregates.

To create a function, you must have RESOURCE or DBA database-level privileges. When you create a function in a database that is not ANSI compliant, any user can use the function. When you create a function in an ANSI-compliant database, you must explicitly grant the Execute privilege on that function, so that users can use the function and thus the related aggregate.

For more information about privileges, refer to the GRANT statement in the *IBM Informix Guide to SQL: Syntax*.

Aggregate Information in the System Catalog

The CREATE AGGREGATE statement registers an aggregate in the **sysaggregates** system catalog table. The person who registers the aggregate with CREATE AGGREGATE is the owner of the aggregate. The **sysaggregates** table does not include information about built-in aggregates.

Both user-extended built-in aggregates and user-defined aggregates require user-defined functions. The system catalog tables **sysprocauth**, **sysprocbody**, and **sysprocedures** record information about the functions that you create, including those that support user-defined aggregates and extensions of built-in aggregates.

For descriptions of the system catalog tables, see the *IBM Informix Guide to SQL: Reference*.

Aggregate Information from the Command Line

The **-g cac agg** option of the **onstat** utility provides information about user-defined aggregates. For information about **onstat**, refer to the *Administrator's Reference*.

Dropping an Aggregate

The DROP AGGREGATE statement removes the definition of an aggregate from the database. You must be the owner of the aggregate or the database administrator (DBA) to drop its definition from the database.

If you are the owner or the DBA, the following statement removes the aggregate SUMSQ from the database:

```
DROP AGGREGATE SUMSQ;
```

Dropping an aggregate has no effect on functions that are associated with the aggregate. Use the DROP FUNCTION statement to remove a function from the database.

Chapter

Creating an Opaque Data Type

9

In This Chapter

This chapter provides the following information:

Opaque Data Types

An *opaque data type* is an atomic data type that you define for the database. An opaque data type gets its name from the fact that the database server maintains no information about the internal representation of the data type. Unlike built-in types, for which the database server maintains information about the internal format, the opaque types are encapsulated; that is, the database server has no knowledge of the format of the data within an opaque data type.

When you define an opaque data type, you extend the data type system of the database server. You can use the new opaque data type in the same way as any built-in data type that the database server provides. To define the opaque data type to the database server, you must provide the following information in an external language (C or Java):

- A data structure that defines the internal storage of the opaque data type
- Support functions that allow the database server to interact with this internal structure

- Optional modifiers that specify how the data type should be treated
- Optional additional routines that can be called by other support functions or by end users to operate on the opaque data type

The following sections introduce each of these parts of an opaque data type. For information on how to create these parts, see "Creating an Opaque Data Type" on page 9-8.

The Internal Structure

To create an opaque data type, you must first provide a data structure that stores the data in its internal representation. This data structure is called the *internal structure* of the opaque data type because it is how the data is stored on disk. The support functions that you write operate on this internal structure; the database server never sees the internal structure. You create the internal structure as a data structure in the external language.

You can define an internal structure that supports either a fixed-length opaque data type or a varying-length opaque data type.

A Fixed-Length Opaque Data Type

A *fixed-length opaque data type* has an internal structure whose size is the same for all possible values of the opaque data type. Fixed-length opaque types are useful for data that you can represent in fixed-length fields, such as numeric values.

You provide the size when you register the opaque data type in the database. For more information, see "Data Type Size" on page 9-9.

A Varying-Length Opaque Data Type

A *varying-length opaque data type* has an internal structure whose size might be different for different values of the opaque data type. Varying-length opaque types are useful for storage of multirepresentational data, such as images. For example, image sizes vary from one picture to another. You might store data up to a certain size within the opaque data type and use a smart large object in the opaque data type if the image size exceeds that size.

When you register the opaque data type in the database, you indicate that the size is varying, and you can indicate a maximum size for the internal structure. For more information, see "Data Type Size" on page 9-9.

A *multirepresentational data type* is a varying-length data type that stores data directly in the internal structure of the opaque type if the length of the data is smaller than a specified threshold. If the length of the data is greater than the threshold, the data type stores the value in a smart large object and then stores the smart large object handle in the opaque type.

When you insert a value into a multirepresentational data type, the **assign()** support function determines where the data should be stored. When you delete data, the **destroy()** support function determines whether the data should be removed from the internal structure or from a smart large object. The **update()** and **deepcopy()** functions provide more efficient management for UDTs that contain smart large objects. For more information about these functions, see "Handling Smart Large Objects" on page 10-26. For information about how to use multirepresentational data types, refer to the *IBM Informix DataBlade API Programmer's Guide*.

Support Functions

Support functions provide the basic functionality that the database server needs to interact with your opaque data type. However, you might want to write additional UDRs to provide the following kinds of functions for your opaque data type:

- Operator functions
- Built-in functions
- Aggregate functions
- Statistics-collecting routines
- Selectivity functions
- End-user routines

Operator Functions

An *operator function* is a user-defined function, such as **plus()** or **equal()**, that has a corresponding operator symbol. For an operator function to operate on the opaque data type, you must overload the routine for the opaque data type.

For general information about the operator functions that the database server provides, see "Operators and Operator Functions" on page 6-4. For general information on overloading routines, refer to "Overloading Routines" on page 3-13. For information on how to overload an operator function on an opaque data type, see "Arithmetic and Text Operator Functions for Opaque Data Types" on page 9-19.

Built-In Functions

A *built-in function* is a predefined function, such as **cos()** or **length()**, that the database server provides for use in an SQL expression. The database server supports built-in functions on the built-in data types. For an opaque data type, you must overload the function for the opaque type.

For general information about these built-in functions, see "Built-In Functions" on page 6-7. For information on how to overload a built-in function on an opaque data type, see "Built-in Functions for Opaque Data Types" on page 9-19.

Aggregate Functions

An *aggregate function* returns one value, such as SUM or AVG, for a set of queried rows. You can extend the built-in aggregates to provide for your opaque data types. You can also create new, special-purpose aggregate functions.

For information about how to extend the built-in aggregates, refer to "Extending Existing Aggregates" on page 8-4. For information about how to create new aggregate functions, refer to "Creating User-Defined Aggregates" on page 8-6. For information about how to use aggregate functions, see the Expression segment in the *IBM Informix Guide to SQL: Syntax*.

Statistics-Collecting Routines

The UPDATE STATISTICS statement calls the **statcollect()** function to collect statistics for the optimizer to use. The **statcollect()** function formats information so that the database server can display it.

For more information, refer to "The statcollect() Function" on page 13-14.

End-User Routines

The database server allows you to define SQL-invoked functions or procedures that the end user can use in expressions or SQL statements. These end-user routines provide additional functionality that an end user might need to work with the opaque data type. Examples of end-user routines include:

- Functions that return a particular value in the opaque data type

  Because the opaque data type is encapsulated, an end-user function is the only way that users can access fields of the internal structure.
- Cast functions

  Several of the support functions serve as cast functions between basic data types that the database server uses. You might also write additional cast functions between the opaque data type and other data types (built-in, opaque, or complex) of the database.
- Functions or procedures that perform common operations on the opaque data type

  If an operation or task is performed often on the opaque data type, you might want to write an end-user routine to perform this task.

For more information about end-user routines, see Chapter 4, "Developing a User-Defined Routine."

Advantages of Opaque Data Types

Both an opaque data type and a row data type allow you to define members of the data type. The advantages of creating an opaque data type rather than a row data type are as follows.

- The opaque data type is more compact to store.

  The opaque data type does not have the overhead in the system catalog that a row data type requires.
- The opaque data type is more efficient.

  The support functions of an opaque data type manipulate the internal structure of the opaque data type directly. You do not need to take special steps (DataBlade API calls or SQL dot notation) to extract data from the members as you must do for the fields of a row data type.

Creating an Opaque Data Type

To create an opaque data type, follow these steps:

1. Create the internal structure for the opaque data type.
2. Write and register the support functions.
3. Register the opaque data type in the database with the CREATE OPAQUE TYPE statement.
4. Provide access to the opaque data type and its support functions with the GRANT statement.
5. Write any SQL-invoked functions that are needed to support the opaque data type.
6. Provide any customized secondary-access methods that the opaque data type might need.

The following sections describe each of these steps.

C

Creating the Internal Structure in C

The internal structure of an opaque data type is a C data structure. For the internal structure, use the C **typedef**s that the DataBlade API supplies for those fields whose size might vary by platform. Use of these **typedef**s, such as **mi_integer** and **mi_float**, improves the portability of the opaque data type. For more information on these data types, see the *IBM Informix DataBlade API Programmer's Guide*.

When you create the internal structure, consider the following impacts of the size of this structure:

- The final structure size of the new opaque data type
- The alignment in memory of the opaque data type
- The method for passing the opaque data type to UDRs

You provide this information when you create the opaque data type with the CREATE OPAQUE TYPE statement.

Data Type Size

To save space in the database, lay out internal structures as compactly as possible. The database server stores values in their internal representation, so any internal structure with padding between entries consumes unnecessary space.

The INTERNALLENGTH keyword of the CREATE OPAQUE TYPE statement supplies the final size of the internal structure. This keyword provides the following two ways to specify the size:

- Specify the actual size, in bytes, of the internal structure to define a fixed-length opaque data type.
- Specify the VARIABLE keyword to define a varying-length opaque data type.

A Fixed-Length Opaque Data Type

When you specify the actual size for INTERNALLENGTH, you create a fixed-length opaque data type. The size of a fixed-length opaque data type must match the value that the C-language **sizeof()** directive returns for the internal structure. The maximum internal length for a fixed-length opaque type is 32760 bytes.

On most compilers, the **sizeof()** directive rounds up to the nearest 4-byte size to ensure that pointer match on arrays of structures works correctly. However, you do not need to round up for the size of a fixed-length opaque data type. Instead you can specify alignment for the opaque data type with the ALIGNMENT modifier. For more information, see "Memory Alignment" on page 9-11.

A Varying-Length Opaque Data Type

When you specify the VARIABLE keyword for the INTERNALLENGTH modifier, you create a varying-length opaque data type. The default maximum size for a varying-length opaque data type is 2 kilobytes.

To specify a different maximum size for a varying-length opaque data type, use the MAXLEN modifier. The maximum internal length for a varying-length opaque type is 32740 bytes. When you specify a MAXLEN value, the database server can optimize resource allocation for the opaque data type. If the size of the data for an opaque data type exceeds the MAXLEN value, the database server returns an error. A varying-length opaque data type is also limited to 195 columns within the 32740 byte maximum length.

For example, the following CREATE OPAQUE TYPE statement defines a varying-length opaque data type called **var_type** whose maximum size is 4 kilobytes:

```
CREATE OPAQUE TYPE var_type (INTERNALLENGTH=VARIABLE,
   MAXLEN=4096);
```

Only the last member of the internal structure can be of varying size.

The C data structure for a varying-length opaque type must be stored in an **mi_lvarchar** data structure. For information about **mi_lvarchar**, refer to the *IBM Informix DataBlade API Function Reference*.

Memory Alignment

When the database server passes the data type to a UDR, it aligns opaque-type data on a specified byte boundary. Alignment requirements depend on the C definition of the opaque data type and on the system (hardware and compiler) on which the opaque data type is compiled.

You can specify the memory-alignment requirement for your opaque data type with the ALIGNMENT modifier of the CREATE OPAQUE TYPE statement. The following table summarizes valid alignment values.

ALIGNMENT Value	Meaning	Purpose
1	Align structure on single-byte boundary.	Structures that begin with 1-byte quantities
2	Align structure on 2-byte boundary.	Structures that begin with 2-byte quantities such as **mi_unsigned_smallint**
4	Align structure on 4-byte boundary.	Structures that begin with 4-byte quantities such as float or **mi_unsigned_integer**
8	Align structure on 8-byte boundary.	Structures that contain members of the C double data type

Structures that begin with single-byte characters, **char**, can be aligned anywhere. Arrays of a data type should follow the same alignment restrictions as the data type itself.

For example, the following CREATE OPAQUE TYPE statement specifies a fixed-length opaque data type, called **LongLong**, of 18 bytes that must be aligned on a 1-byte boundary:

```
CREATE OPAQUE TYPE LongLong (INTERNALLENGTH=18, ALIGNMENT=1);
```

If you do not include the ALIGNMENT modifier in the CREATE OPAQUE TYPE statement, the default alignment is a 4-byte boundary.

Parameter Passing

The database server can pass opaque-type values to a UDR in either of the following ways:

- *Pass by value* passes the actual value of the opaque data type to a UDR.
- *Pass by reference* passes a pointer to the value of the opaque data type to a UDR.

By default, the database server passes all opaque types by reference. For the database server to pass an opaque data type by value, specify the PASSEDBYVALUE modifier in the CREATE OPAQUE TYPE statement. Only an opaque data type whose size is 4 bytes or smaller can be passed by value. However, the DataBlade API data type **mi_real**, although only 4 bytes in length, is always passed by reference.

The following CREATE OPAQUE TYPE statement specifies that the **two_bytes** opaque data type be passed by value:

```
CREATE OPAQUE TYPE two_bytes (INTERNALLENGTH=2, ALIGNMENT=2,
   PASSEDBYVALUE);
```

Creating UDT-to-Java Mappings

The routine manager needs a mapping between SQL data values and Java objects to be able to pass parameters to and retrieve return results from a UDR. The SQL to Java data-type mapping is performed according to the JDBC specification. For built-in SQL data types, the routine manager can use mappings to existing JDBC data types.

To create the mapping between a user-defined SQL data type and a Java object

1. Create a user-defined class that implements the **SQLData** interface. (For more information, refer to the JDBC 2.0 specification).
2. Bind this user-defined class to the user-defined SQL data type using the **setUDTExtName** built-in procedure.

Writing and Registering the Support Functions

An opaque data type needs support functions that provide casts for input and output, operator functions, cost functions, selectivity functions, operator-class functions and statistics functions. For more information about these functions, refer to Chapter 10, "Writing Support Functions," and Chapter 11, "Extending an Operator Class."

Registering the Opaque Data Type with the Database

After you create the internal structure and support functions for the opaque data type, use the following SQL statements to register them with the database:

- The CREATE OPAQUE TYPE statement registers an opaque data type as a data type.
- The CREATE FUNCTION statement registers a support function.
- The CREATE CAST statement registers a support function as cast functions.

Registering the Opaque Data Type

To create an opaque data type within a database, you must have the Resource privilege on the database. The CREATE OPAQUE TYPE statement registers an opaque data type with the database. It provides the following information to the database:

- The name and owner of the opaque data type

  The opaque-type name is the name of the data type that SQL statements use. It does not have to be the name of the internal structure for the opaque data type. You might find it useful to create a special prefix to identify the data type as an opaque data type. The opaque-type name must be unique within the name space.
- The size of the opaque data type

  You specify this size information with the INTERNALLENGTH modifier. It indicates whether the data type is a fixed-length or varying-length opaque data type. For more information, see "Creating the Internal Structure in C" on page 9-9.

- The values of the different opaque-type modifiers

  The CREATE OPAQUE TYPE statement can specify the following modifiers for an opaque data type: MAXLEN, PASSEDBYVALUE, CANNOTHASH, and ALIGNMENT. You determine this information when you create the internal structure for the opaque data type. For more information, see "Creating the Internal Structure in C" on page 9-9.

The CREATE OPAQUE TYPE statement stores this information in the **sysxtdtypes** system catalog table. When it stores a new opaque data type in **sysxtdtypes**, the CREATE OPAQUE TYPE statement causes a unique value, called an *extended identifier*, to be assigned to the opaque data type. Throughout the system catalog, an opaque data type is identified by its extended identifier, not by its name. (For more information on the columns of the **sysxtdtypes** system catalog, see the chapter on system catalog tables in the *IBM Informix Guide to SQL: Reference*.)

To register a new opaque data type in a database, you must have the Resource privilege on that database. By default, a new opaque data type has Usage permission assigned to the owner. For information on how to change the permission of an opaque data type, see "Granting Privileges for an Opaque Data Type" on page 9-17.

For more information on the syntax of the CREATE OPAQUE TYPE, CREATE FUNCTION, and CREATE FUNCTION FROM statements, see their descriptions in the *IBM Informix Guide to SQL: Syntax*.

Creating Casts for Opaque Data Types

For each of the support functions in the following table, the database server uses a cast to convert the opaque data type to a particular internal data type.

Support Function	Cast		
	From	**To**	**Type of Cast**
input	LVARCHAR	opaque data type	implicit
output	opaque data type	LVARCHAR	explicit
receive	SENDRECV	opaque data type	implicit

(1 of 2)

Support Function	Cast		
	From	**To**	**Type of Cast**
send	opaque data type	SENDRECV	explicit
import	IMPEXP	opaque data type	implicit
export	opaque data type	IMPEXP	explicit
importbinary	IMPEXPBIN	opaque data type	implicit
exportbinary	opaque data type	IMPEXPBIN	explicit
streamread	STREAM	opaque data type	implicit
streamwrite	opaque data type	STREAM	explicit

(2 of 2)

For the database server to perform these casts, you must create the casts with the CREATE CAST statement. The database server can then call the appropriate support function when it needs to cast opaque-type data to or from the LVARCHAR, SENDRECV, IMPEXP, IMPEXPBIN, or STREAM data types.

The CREATE CAST statement stores information about cast functions in the **syscasts** system catalog table. For more information on the CREATE CAST statement, see the description in the *IBM Informix Guide to SQL: Syntax*. For a description of casting, see the *IBM Informix Guide to SQL: Tutorial*.

Using Non In-Row Storage

An opaque data type can use the following types of non in-row storage:

- Smart large object (BLOB and CLOB)
- Files
- A non in-row storage type that is dependent on the local computer

  For example, this storage type might be a reference to a tape storage system.
- A non in-row storage type that is not dependent on the database server

  For example, this storage type might be a file reference that includes the location of the computer where the user of the reference goes directly to the designated computer, bypassing the database server where the reference is stored.

The routines that support the opaque data type should do the following:

- Include room in the storage handle for location information

  The location information should include the database server name, and, if the data type is dependent on a particular database, the database name.
- Provide routines to set and get the location information from the storage handle to include in the server-send support functions
- Provide support for remote data in the access routines

  For example, the open routine must recognize a reference to a remote database server and access it appropriately.

Granting Privileges for an Opaque Data Type

After you create the opaque data type and register it with the database, use the GRANT statement to define the following privileges on this data type:

- Privileges on the use of the opaque data type
- Privileges on the support functions of the opaque data type

The CREATE OPAQUE TYPE statement creates a new opaque data type with the Usage privilege granted to the owner of the opaque data type and the DBA. To use the opaque data type in an SQL statement, you must have the Usage privilege. The owner can grant the Usage privilege to other users with the USAGE ON TYPE clause of the GRANT statement.

The database server checks for the Usage privilege whenever the opaque-type name appears in an SQL statement (such as a column data type in CREATE TABLE or a cast data type in CREATE CAST). The database server does *not* check for the Usage privilege when an SQL statement:

- Accesses columns of the opaque data type

  The Select, Insert, Update, and Delete table-level privileges determine access to a column.
- Invokes a UDR with the opaque data type as an argument

  The Execute routine privilege determines access to a UDR.

For example, the following GRANT statement assigns the Usage privilege on the **circle** opaque data type to user **dexter**:

```
GRANT USAGE ON TYPE circle TO dexter
```

The **sysxtdtypeauth** system catalog table stores data type-level privileges. This table contains privileges for each opaque and distinct data type that is defined in the database. The table contains one row for each set of privileges granted.

For information about setting the privileges for support functions, refer to "Setting Privileges for Support Functions" on page 10-7.

Creating SQL-Invoked Functions

An *SQL-invoked function* is a user-defined function that an end user can explicitly call in an SQL statement. You might write SQL-invoked functions to extend the functionality of an opaque data type in the following ways:

- Overloading arithmetic or built-in functions to provide arithmetic operations and built-in functions on the opaque data type
- Overloading relational-operator functions to provide comparison operations on the opaque data type
- Writing new end-user routines to provide additional functionality for the opaque data type
- Writing new cast functions to provide additional data conversions to and from the opaque data type

The SQL functions that the database server defines handle the built-in data types. For a UDT to use any of these functions, you can overload the function that handles the UDT. For more information on the details of writing user-defined functions, see Chapter 4, "Developing a User-Defined Routine." For information about overloading functions, refer to "Overloading Routines" on page 3-13.

The database server supports the following types of SQL-invoked functions that allow you to operate on data in expressions of SQL statements:

- Arithmetic and text operator functions
- Built-in functions
- Aggregate functions

The database server also supports the following types of functions that allow you to compare data in expressions of SQL statements:

- SQL operators in a conditional clause
- Relational operator functions

Arithmetic and Text Operator Functions for Opaque Data Types

The database server provides operator functions for arithmetic operators (see "Arithmetic Operators" on page 6-4) and text operators (see "Text Operators" on page 6-5). The operator functions that the database server provides handle the built-in data types. You can overload an operator function to provide the associated operation on your new opaque data type.

If you overload an operator function, make sure you follow these rules:

1. The name of the operator function must match the name of one of the functions that the database server provides. The name is not case sensitive; the **plus()** function is the same as the **Plus()** function.
2. The operator function must handle the correct number of parameters.
3. The operator function must return the correct data type, where appropriate.

Built-in Functions for Opaque Data Types

The database server provides special SQL-invoked functions, called *built-in functions*, that provide some basic mathematical operations. The built-in functions that the database server provides handle the built-in data types. You can overload a built-in function to provide the associated operation on your new opaque data type. If you overload a built-in function, follow these rules:

1. The name of the built-in function must match the name listed in "Built-In Functions That You Can Overload" on page 6-7. However, the name is not case sensitive; the **abs()** function is the same as the **Abs()** function.
2. The built-in function must be one that can be overridden.
3. The built-in function must handle the correct number of parameters, and these parameters must be of the correct data type.
4. The built-in function must return the correct data type, where appropriate.

For more information on built-in functions, see the *IBM Informix Guide to SQL: Syntax*.

Aggregate Functions for Opaque Data Types

You can extend the built-in aggregate functions, such as SUM and AVG, to operate on your opaque data type. You can also create new aggregates. Chapter 8, "Creating User-Defined Aggregates," describes how to extend or create aggregates.

Conditional Operators for Opaque Data Types

The database server supports the following relational operators on an opaque data type in the conditional clause of SQL statements:

- The IS and IS NOT operators
- The IN operator if the **equal()** function has been defined
- The BETWEEN operator if the **compare()** function has been defined

Tip: *The database server also uses the* ***compare()*** *function as the support function for the default B-tree operator class. For more information, see "Extensions of the btree_ops Operator Class" on page 11-9.*

For more information on the conditional clause, see the Condition segment in the *IBM Informix Guide to SQL: Syntax*. For more information on the **compare()** function, see "Comparison Function for Opaque Data Types" on page 9-22.

Relational Operators for Opaque Data Types

The database server provides operator functions for the relational operators listed in "Relational Operators" on page 6-5. The relational-operator functions that the database server provides handle the built-in data types. You can overload a relational-operator function to provide the associated operation on your new opaque data type.

If you overload a relational-operator function, make sure you follow these rules:

1. The name of the relational-operator function must match a name listed in "Relational Operators" on page 6-5. However, the name is not case sensitive; the **equal()** function is the same as the **Equal()** function.
2. The relational-operator function must take two parameters, both of the opaque data type.
3. The relational-operator function must be a Boolean function; that is, it must return a BOOLEAN value.

You must define an **equal()** function to handle your opaque data type if you want to allow columns of this data type to be:

- Constrained as UNIQUE or PRIMARY KEY

  For more information on constraints, see the CREATE TABLE statement in the *IBM Informix Guide to SQL: Syntax*.
- Compared with the equal (=) operator in an expression
- Used with the IN operator in a condition

Hashable Data Types

The database server uses a built-in bit-hashing function to produce the hash value for a data type, which means that the built-in hash function can be used only for bit-hashable data types. If your opaque data type is *not* bit hashable, the database server cannot use its built-in hash function for the equality comparison. Therefore, if your data type is not bit-hashable, you cannot use it in the following cases:

- In the GROUP BY clause of a SELECT statement
- In hash joins
- With the IN operator in a WHERE clause
- COUNT DISTINCT aggregates

Nonhashable Data Types

For opaque types that are not bit hashable using the built-in hashing function of the database server, specify the CANNOTHASH modifier in the CREATE OPAQUE TYPE statement.

Hashable data types have the following property: if A = B, then hash(A) = hash(B), which means that A and B have identical bit representations.

Multirepresentational data types are not bit hashable because they store large quantities of data in a smart large object and then store the large object handle in the user-defined type. It is the smart-large-object handle that makes the multirepresentational data type nonhashable. That is, the CREATE OPAQUE TYPE statement for a multirepresentational data type must include the CANNOTHASH modifier.

Comparison Function for Opaque Data Types

The **compare()** function is an SQL-invoked function that sorts the target data type. The database server uses the **compare()** function to execute the following clauses and keywords of the SELECT statement:

- The ORDER BY clause
- The UNIQUE and DISTINCT keywords
- The UNION keyword

The database server also uses the **compare()** function to evaluate the BETWEEN operator in the condition of an SQL statement. For more information on conditional clauses, see the Condition segment in the *IBM Informix Guide to SQL: Syntax*.

The database server provides **compare()** functions that handle the built-in data types. For the database server to be able to sort an opaque data type, you must define a **compare()** function to handle this opaque data type.

If you overload the **compare()** function, make sure you follow these rules:

1. The name of the function must be **compare()**. The name is not case sensitive; the **compare()** function is the same as the **Compare()** function.
2. The function must accept two arguments, each of the data types to be compared.

3. The function must return an integer value to indicate the result of the comparison, as follows:
   - <0 to indicate that the first argument precedes the second argument
   - `0` to indicate that the two arguments are the same
   - >0 to indicate that the first argument comes after the second argument

The **compare()** function is also the support function for the default operator class of the B-tree secondary-access method. For more information, see "Generic B-Tree Index" on page 11-4.

Customizing Access Methods

The database server provides the full implementation of the generic B-tree secondary-access method, and it provides definitions for the R-tree secondary-access method. By default, the CREATE INDEX statement builds a generic B-tree index for the column or user-defined function.

When you create an opaque data type, you must ensure that secondary-access methods exist that support the new data type. Consider the following factors about the secondary-access methods and their support for the opaque data type:

- Does the generic B-tree support the opaque data type?
- If the opaque-type data is spatial, can you use the R-tree index?
- Do other secondary-access methods exist that might better index your opaque-type data?

To create an index of a particular secondary-access method on a column of an opaque data type, the database server must find an operator class that is associated with the secondary-access method. This operator class must specify operations (strategy functions) on the opaque data type as well as the functions that the secondary-access method uses (support functions).

For more information about an operator class and operator-class functions, see "Operator Classes" on page 11-5.

Using the Generic B-Tree

The generic B-tree secondary-access method has a default operator class, **btree_ops**, whose operator-class functions handle indexing for the built-in data types. These operator-class functions have the following functionality for built-in data types:

- They order the data in lexicographical sequence.

  If this sequence is not logical for your opaque data type, you can define operator-class functions for the opaque data type that provide the sequence you need.

- They expect to compare two single, one-dimensional values for a given data type.

  If the opaque data type holds more than one value, but you can define a single value for it, you can define operator-class functions for the opaque data type that compare two of these one-dimensional values. If you cannot define a one-dimensional value for the opaque data type, you cannot use a B-tree index as its secondary-access method.

To provide support for columns and user-defined functions of the opaque data type, you can extend the **btree_ops** operator-class functions so that they handle the new opaque data type. The generic B-tree secondary-access method uses the new operator-class functions to store values of the opaque data type in a B-tree index.

For more information about how to extend the default B-tree operator class, see "Extensions of the btree_ops Operator Class" on page 11-9.

Using Other Access Methods

The way that the generic B-tree secondary-access method orders data is useful for one-dimensional data. When your data type is not one-dimensional, you might need to use some other access method.

For information about the R-tree access method, refer to the *IBM Informix R-Tree Index User's Guide*. For more information on the secondary-access methods that Data Blade modules provide, check the user guide for your DataBlade module.

Indexing Spatial Data

The R-tree secondary-access method is useful for spatial or multidimensional data such as maps and diagrams. To use an R-tree index, you must install a spatial DataBlade module such as the Spatial DataBlade module, Geodetic DataBlade module, or any third-party DataBlade module that implements an R-tree index. For more information, refer to the user documentation for your custom access method.

Indexing Other Types of Data

Your opaque data type might have data that is not optimally indexed by either a generic B-tree or an R-tree. Often, DataBlade modules that define new opaque data types provide their own secondary-access methods for these data types. For information about creating an access method, refer to the *IBM Informix Virtual-Index Interface Programmer's Guide*.

Other Operations on Opaque Data Types

This section describes the following operations that you can perform on opaque data types:

- How to access an opaque data type from a client application
- How to drop an opaque data type from a database

Accessing an Opaque Data Type

After you create the opaque data type, the following client programs can use it once they connect to the database in which it is registered:

- An ESQL/C application that uses SQL statements and an **lvarchar**, **fixed binary**, or **var binary** host variable

  For more information, see the chapter on opaque types in the *IBM Informix ESQL/C Programmer's Manual*.
- A C routine that uses the DataBlade API

  For more information, see the *IBM Informix DataBlade API Programmer's Guide*.

- An SPL UDR

  For more information, see the chapter on SPL in the *IBM Informix Guide to SQL: Tutorial*.
- A client application written in the Java

You can use an opaque data type in any way that you use other data types of the database.

Dropping an Opaque Data Type

You cannot drop an opaque data type if any dependencies on it still exist in the database. Therefore, to drop an opaque data type from a database, you reverse the process of registering the data type, as follows:

1. Remove or change the data type of any columns in the database that have the opaque data type as their data type.

   Use the ALTER TABLE statement to change the data type of database columns. Use the DROP TABLE statement to remove the entire table.
2. The REVOKE statement with the USAGE ON TYPE clause removes one set of privileges assigned to the opaque data type.

   This statement removes the row of the **sysxtdtypeauth** system catalog table that defines the privileges of the opaque data type. Use the statement to drop each set of privileges that have been assigned to the opaque data type.
3. The REVOKE statement with the EXECUTE ON FUNCTION or EXECUTE ON ROUTINE clause removes the privileges assigned to a support function of the opaque data type.

   This statement removes the row of the **sysprocauth** system catalog table that defines the privileges of the opaque data type. Use the statement to drop each set of privileges that have been assigned to a support function. You must drop the privileges for each support function. If you assigned a specific name to the support function, use the SPECIFIC keyword to identify the specific name.

4. The DROP CAST statement drops a cast function for a support function of an opaque data type.

   This statement removes the row of the **syscasts** system catalog table that defines the cast function for a support function. Use the statement to drop each of the casts that you defined. For more information, see "Creating Casts for Opaque Data Types" on page 9-14.

5. The DROP FUNCTION or DROP ROUTINE statement removes a support function of the opaque data type from the current database.

   This statement removes the row of the **sysprocedures** system catalog table that registers a support function. Use the statement to drop each of the support functions that you registered.

6. The DROP TYPE statement removes the opaque data type from the current database.

   This statement removes the row of the **sysxtdtypes** system catalog table that describes the opaque data type. Once you drop an opaque data type from a database, no users of that database can access the data type. You must be the owner of the opaque data type or have DBA privileges to remove the data type.

To use these SQL statements, you must be either the owner of the object that you drop or have DBA privileges. For more information on the syntax of the REVOKE, DROP FUNCTION, DROP ROUTINE, DROP CAST, and DROP TYPE statements, see their descriptions in the *IBM Informix Guide to SQL: Syntax*.

Chapter

Writing Support Functions

10

In This Chapter

This chapter describes the support functions for opaque data types and operator classes.

Writing Support Functions

The support functions for an opaque data type are a set of well-defined, data type specific functions that the database server automatically invokes. Typically, these functions are not explicitly invoked in an SQL statement.

Identifying Support Functions

The following table summarizes the support functions for opaque data types.

Function	Purpose	Reference
input	Converts opaque data from its external text representation to its internal representation. Supports insertion of data in text format into a column of the opaque type. Requires an implicit cast from the LVARCHAR data type to opaque data type.	page 10-9
output	Converts opaque data from its internal representation to its external text representation. Supports selection of data from a column of the opaque type in its external text format. Requires an explicit cast from the opaque data type to LVARCHAR opaque data type.	page 10-11

(1 of 3)

Function	Purpose	Reference
receive	Converts opaque data from its external binary representation on the client computer to its internal representation on the database server computer. Supports insertion of binary data into a column of the opaque type. Requires an implicit cast from the SENDRECV data type to the opaque data type.	page 10-14
send	Converts opaque data from its internal representation on the database server computer to its external binary representation on the client computer. Supports selection of binary data from a column of the opaque type. Requires an explicit cast from the opaque data type to the SENDRECV data type.	page 10-16
import	Performs processing of opaque data for bulk load of text data in a column of the opaque type. Requires an implicit cast from the IMPEXP to the opaque data type.	page 10-19
export	Performs processing of opaque data for bulk unload of text data from a column of the opaque type. Requires an explicit cast from the opaque to the IMPEXP data type.	page 10-19
importbinary	Performs processing of opaque data for bulk load of binary data in a column of the opaque type. Requires an implicit cast from the IMPEXPBIN to the opaque data type.	page 10-21
exportbinary	Performs processing of opaque data for bulk unload of binary data from a column of the opaque type. Requires an explicit cast from the opaque to the IMPEXPBIN data type.	page 10-21
streamread	Converts opaque data from its stream representation to its database server internal representation.	
streamwrite	Converts opaque data from its internal representation on the database server to its stream representation.	
assign	Performs any processing required before the database server stores opaque data to disk. Supports storage of opaque data for INSERT, UPDATE, and LOAD statements.	page 10-23

(2 of 3)

Function	Purpose	Reference
destroy	Performs any processing necessary before the database server removes a row that contains an opaque data type.	page 10-24
lohandles	Returns a list of the embedded large-object handles in the opaque data type.	page 10-26
compare	Supports opaque data types during ORDER BY, UNIQUE, DISTINCT, and UNION clauses, and BETWEEN comparisons. Also supports CREATE INDEX for B-tree indexes.	page 10-28
deepcopy	Supports multirepresentational data types as function return values	page 10-25
update	Supports in-place update on smart large objects	page 10-24

(3 of 3)

Most support functions can have arbitrary names. The database server identifies a support function by the task that it needs to perform. For example, if the client binds a binary value to INSERT, the database server looks for a cast function in the **syscasts** system catalog table that converts the UDT value from its external binary format (SENDRECV) to the opaque data type.

The following functions must be named explicitly: **compare()**, **assign()**, **destroy()**, **update()** and **deepcopy()**. However, the names are not case sensitive. That is, you can name the function **compare()** or **Compare()**.

It is recommended that you give your support functions names that help document the purpose of the function. For example, if your opaque data type is named **sphere**, you might name the receive and send functions **sphere_receive()** and **sphere_send()**.

Whenever possible, you should create the support functions as NOT VARIANT for better performance. For information about variant and non-variant functions, refer to "Returning a Variant or Nonvariant Value" on page 4-7.

Choosing Function Parameters

The following table summarizes the SQL data types for the parameter list and return type of CREATE FUNCTION statements that register support functions.

Support Function	Parameter Type	Return Type	Refer to
input	lvarchar	*opaque data type*	page 10-9
output	*opaque data type*	lvarchar	page 10-11
receive	sendrecv	*opaque data type*	page 10-14
send	*opaque data type*	sendrecv	page 10-14
import	impexp	*opaque data type*	page 10-17
export	*opaque data type*	impexp	page 10-17
importbinary	impexPbin	*opaque data type*	page 10-20
exportbinary	*opaque data type*	impexpbin	page 10-20
assign	*opaque data type*	*opaque data type*	page 10-23
destroy	*opaque data type*	- no return value -	page 10-24
update	*opaque data type, opaque data type*	*opaque data type*	page 10-24
deepcopy	*opaque data type*		page 10-25
lohandles	*opaque data type*	- list of pointers -	page 10-26
compare	*user-defined type, user-defined type*	- integer values to show less than, greater than and equal -	page 10-28

In the preceding table, *opaque data type* is the name of the data type that you specify in the CREATE OPAQUE TYPE statement. For more information, see "Registering the Opaque Data Type" on page 9-13.

When the CREATE FUNCTION statement stores a new support function in **sysprocedures**, it causes the database server to assign a unique value, called a *routine identifier*, to the support function. Throughout the system catalog a support function is identified by its routine identifier, not by its name.

Setting Privileges for Support Functions

The CREATE FUNCTION statement registers a function with the Execute privilege granted to the owner of the support function and the DBA. Such a function is called an *owner-privileged function*.

To execute a support function in an SQL statement, the user must have the Execute privilege. Usually, the default privilege is adequate for support functions that are implicit casts because implicit casts should not generally be called within SQL statements. Support functions that are explicit casts might have the Execute privilege granted so that users can call them explicitly. The owner grants the Execute privilege to other users with the EXECUTE ON clause of the GRANT statement.

The **sysprocauth** system catalog table stores routine-level privileges. This table contains privileges for each UDR and therefore for all support functions that are defined in the database. The table contains one row for each set of privileges granted.

Data Types for Support Functions

The database server provides data types for use with UDTs and UDRs. Although these data types are *predefined* by the database server, the database server treats them as extended data types.

The **sysxtdtypes** system catalog table records extended data types, both predefined and user-defined. This section discusses predefined data types that are specifically used by UDTs and UDRs.

The LVARCHAR Data Type

The database server uses the LVARCHAR data type to transfer the external text representation of an opaque data type between the database server and an application. Although the actual internal, binary representation for the opaque data type might contain nontext types, such as integers or double precision floating-point values, the data in its external text format is an LVARCHAR. The input and output support functions serve as cast functions between the LVARCHAR and opaque data types.

Tip: *When you use LVARCHAR as a column type, the column size is limited to 2 kilobytes. However, when you use LVARCHAR to transport opaque data, the length of the data is limited only by your operating system.*

The DataBlade API provides the **mi_lvarchar** data type to hold the external representation of opaque-type data. For more information, see the *IBM Informix DataBlade API Programmer's Guide*.

E/C

ESQL/C applications use **lvarchar** to transfer the external text representation of an opaque type. The database server implicitly invokes the input and output support functions when it receives an SQL statement that contains an **lvarchar** host variable.

ESQL/C applications use **varbinary** to transfer the external binary representation of an opaque type. ♦

The SENDRECV Data Type

When you create an opaque data type, you must supply support functions that convert the opaque data between its internal representation on the client computer and its internal representation on the database server computer. These functions use the SENDRECV data type as input or output parameters.

Handling the External Representation

Every opaque type has an internal and external representation. The internal representation is the internal structure that you define for the opaque type. (For more information, see "The Internal Structure" on page 9-4.) The external text representation is a character string that is a printable version of the opaque value. The opaque type might also have an external binary representation.

When you define an opaque type, you must supply support functions that convert between the internal and external representations of the opaque type:

- The input function converts from external text representation to internal representation.
- The output function converts from internal to the external text representation.

These support functions do not have to be named *input* and *output*, but they do have to perform the specified conversions. They should be reciprocal functions; that is, the input function should produce a value that the output function accepts as an argument and vice versa. For the database server to execute these support functions automatically, you must provide an implicit cast from LVARCHAR to the user-defined type that invokes the input function. Similarly, you must also provide an explicit cast from the UDT to LVARCHAR that invokes the output function.

The database server raises an error if it cannot find the proper support function to carry out a task. For example, if an application tries to INSERT a value in an external text format, the database server looks for a cast from LVARCHAR to the user-defined type. If that cast does not exist, the database server raises an error.

GLS

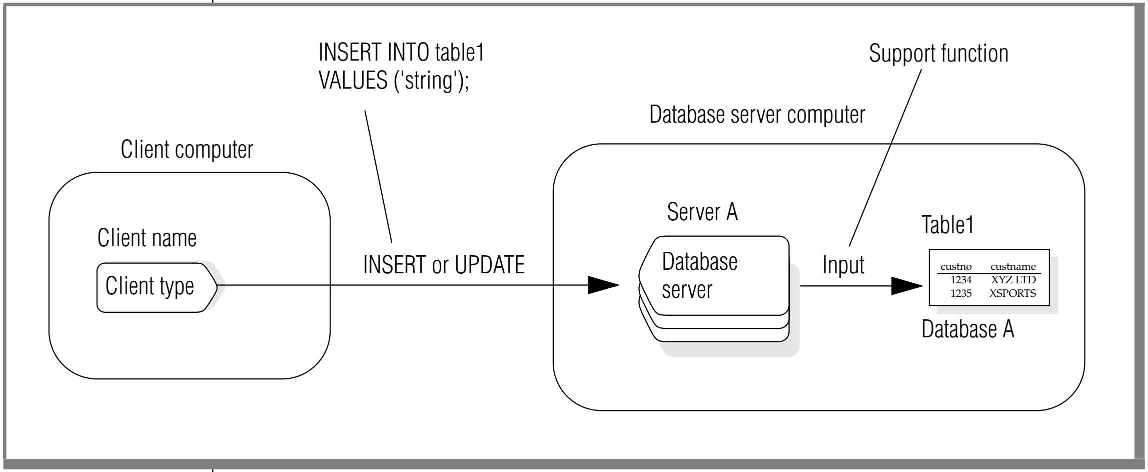
For your opaque data type to accept an external representation on nondefault locales, you must use the IBM Informix GLS API in the input and output functions to access Informix locales from within these functions. For more information, see "Handling Locale-Sensitive Data" on page 10-29. ♦

Input Support Function

The database server calls the input function when it receives the external representation of an opaque type from a client application. For example, when a client application issues an INSERT or UPDATE statement, it can send the text representation of an opaque type to the database server to be stored in an opaque-type column. The database server calls the input function to convert this external representation to an internal representation that it stores on disk.

Figure 10-1 shows when the database server executes the input support function.

Figure 10-1
Execution of the Input Support Function



If the opaque data type is pass by reference, the input support function should perform the following tasks:

- Allocate enough space to hold the internal representation.

  C The function can use the **mi_alloc()** DataBlade API function to allocate the space for the internal structure. ♦
- Parse the input string.

  It must obtain the individual members from the input string and store them in the appropriate fields of the internal structure
- Return a pointer to the internal structure.

If the opaque data type is pass by value, the input support function should perform these same basic tasks but return the actual value in the internal structure instead of a pointer to this structure. You can use pass by value only for opaque types that are less than 4 bytes in length.

C

The input function takes an **mi_lvarchar** value as an argument and returns the internal structure for the opaque type. The following function signature is an input support function for a fixed-length opaque data type whose internal structure is **ll_longlong_t**:

```
ll_longlong_t * ll_longlong_input(mi_lvarchar *extrnl_format);
```

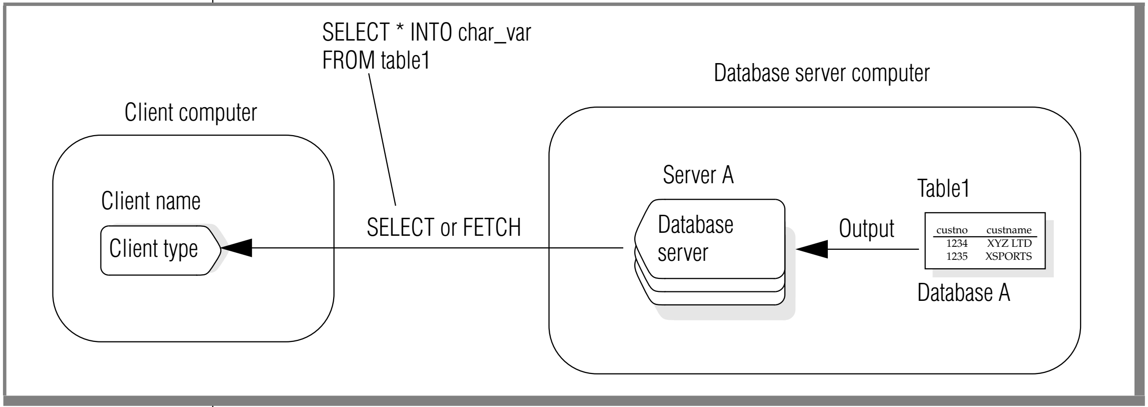
The **ll_longlong_input()** function is a cast function from the LVARCHAR data type to the **ll_longlong_t** internal structure. It must be registered as an implicit cast function with the CREATE IMPLICIT CAST statement. For more information on cast functions, see "Creating Casts for Opaque Data Types" on page 9-14. ♦

Output Support Function

The database server calls the output function when it sends the external representation of an opaque type to a client application. For example, when a client application issues a SELECT or FETCH statement, the application can save the data of an opaque type that it receives from the database server in a character host variable. The database server calls the output function to convert the internal representation that is stored on disk to the external representation that the character host variable requires.

Figure 10-2 shows when the database server executes the output support function.

Figure 10-2
Execution of the Output Support Function



If the opaque data type is pass by reference, the output support function should perform the following tasks:

- Accept a pointer to the internal representation as an argument.
- Allocate enough space to hold the external representation.

  C

  The support function can use the **mi_alloc()** function to allocate the space for the character string. For more information on memory management and the **mi_alloc()** function, refer to the *IBM Informix DataBlade API Programmer's Guide* and the *IBM Informix DataBlade API Function Reference*. ♦
- Create the output string from the individual members of the internal structure.

  The function must build the external representation with the values from the appropriate fields of the internal structure.
- Return a pointer to the character string.

If the opaque data type is pass by value, the output support function should perform the same basic tasks but accept the actual value in the internal structure. You can use pass by value only for opaque types that are 4 bytes or less.

C

The output function takes the internal structure for the opaque type as an argument and returns an **mi_lvarchar** value. The following function signature is for an output support function of an opaque data type whose internal structure is **ll_longlong_t**:

```
mi_lvarchar * ll_longlong_output(ll_longlong_t *intrnl_format);
```

The **ll_longlong_output()** function is a cast function from the **ll_longlong_t** internal structure to the LVARCHAR data type. It must be registered as an explicit cast function with the CREATE EXPLICIT CAST statement. For more information on cast functions, see "Creating Casts for Opaque Data Types" on page 9-14. ♦

Handling the Internal Representation

If a client application that uses an opaque data type executes on a different computer than the database server, the computers involved might have different ways of representing the internal structure of the opaque type. For example, the client computer might use a different byte ordering than the database server computer.

You must supply send and receive support functions, sometimes called *transport functions*, that convert data between the client application and the database server, commonly called *receive* and *send* functions.

You can choose arbitrary names for these support functions. The cast functions that use the functions identify the support functions to the database server.

The receive and send functions support the transfer of opaque types:

- The receive function converts incoming data to the internal representation of the local database server.
- The send function converts outgoing data from the internal representation of the local database server to an appropriate representation for the client application or the external database.

The send and receive functions should be reciprocal functions; that is, the receive function should produce a value that the send function accepts as an argument and the send function should produce a value that the receive function accepts as an argument.

The functions must handle conversions for all platform variations that the client application or external database server might encounter. When the local database server accepts a client connection or connects to a remote database server, it receives a description of the internal representations that the client or the remote database server uses. The database server uses this description to determine which data representation to use in its receive and send support functions.

The IBM Informix DataBlade API provides functions that support conversion between different internal representations of opaque types. The send and receive functions can call DataBlade API routines for each member of the internal structure to convert them to the appropriate representation for the destination platform.

GLS

For an opaque data type to accept an internal representation on nondefault locales, you must use the IBM Informix GLS API in the receive and send functions to access Informix locales from within these functions. For more information, see "Handling Locale-Sensitive Data" on page 10-29. ♦

The Send and Receive Support Functions

The database server uses the send and receive support functions when it passes data to and from a client application.

The SENDRECV Data Type

The SENDRECV data type holds the external binary representation of an opaque data type when it is transferred between the client computer and the database server computer. The SENDRECV data type allows for any possible change in the size of the data when it is converted between the two representations. The receive and send support functions serve as cast functions between the SENDRECV and opaque data type.

E/C

ESQL/C applications do not use the SENDRECV data type. Instead, these applications use **fixed binary** and **var binary** host variables in SQL statements to transfer the internal representation of an opaque type on the client computer. The database server implicitly invokes the receive and send support functions when it receives an SQL statement that contains a **fixed binary** or **var binary** host variable. ♦

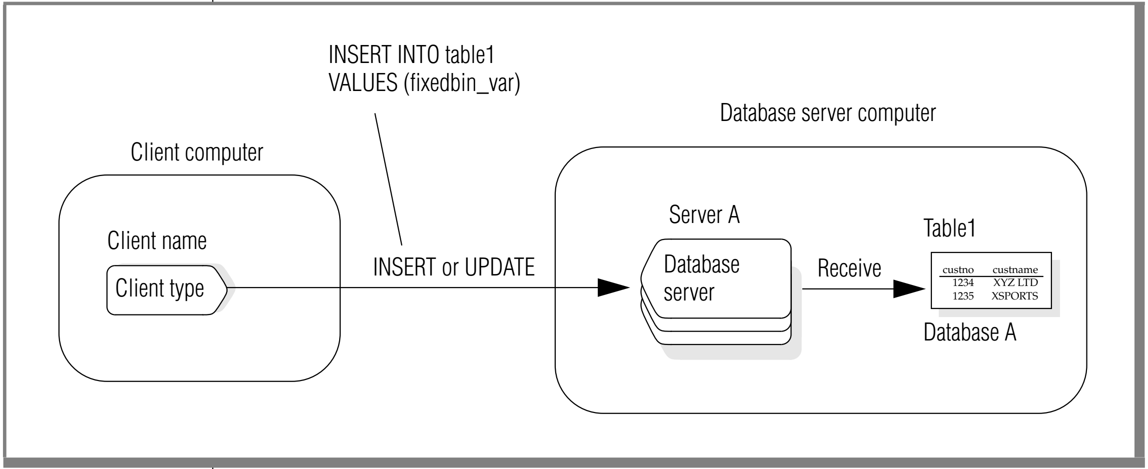
Receive Support Function

The receive support function converts opaque data from its external binary representation on the client computer to its internal representation on the database server computer and provides an implicit cast from the SENDRECV to the opaque data type.

The database server calls the receive function when it receives the external binary representation of an opaque type from a client application. For example, when a client application issues an INSERT or UPDATE statement, it can send the external binary representation of an opaque type to the database server to be stored in a column.

Figure 10-3 shows when the database server executes the receive support function.

Figure 10-3
Execution of the Receive Support Function



The database server calls the receive function to convert the external binary representation of the client computer to the internal representation of the database server computer, where the opaque type is stored on disk.

C

The receive function takes as an argument an **mi_sendrecv** structure (that holds the internal structure on the client computer) and returns the internal structure for the opaque type (the internal representation on the database server computer). The following function signature is for a receive support function of an opaque data type whose internal structure is **ll_longlong_t**:

```
ll_longlong_t * ll_longlong_receive(mi_sendrecv
*client_intrnl_format);
```

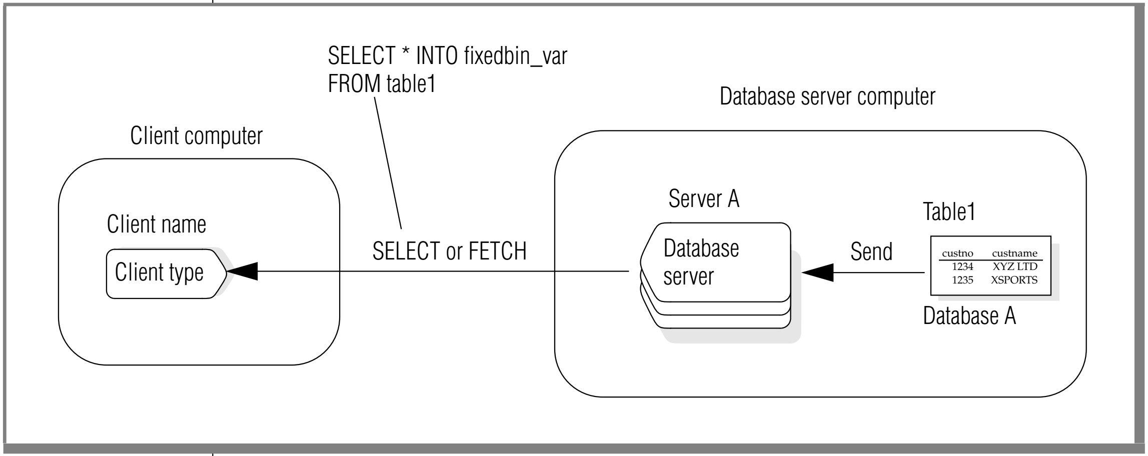
The **ll_longlong_receive()** function is a cast function from the SENDRECV data type to the **ll_longlong_t** internal structure. It must be registered as an implicit cast function with the CREATE IMPLICIT CAST statement. For more information on cast functions, see "Creating Casts for Opaque Data Types" on page 9-14. ♦

Send Support Function

The database server calls the send function when it sends the external binary representation of an opaque type to a client application. For example, when a client application issues a SELECT or FETCH statement, it can save the data of an opaque type that it receives from the database server in a host variable that conforms to the external binary representation of the opaque type.

Figure 10-4 shows when the database server executes the **send** support function.

Figure 10-4
Execution of the Send Support Function



The database server calls the send function to convert the internal representation that is stored on disk to the external binary representation that the client computer uses.

C

The send function takes as an argument the internal structure for the opaque type on the database server computer and returns an **mi_sendrecv** structure that holds the internal structure on the client computer. The following function signature is for a send support function of an opaque data type whose internal structure is **ll_longlong_t**:

```
mi_sendrecv * ll_longlong_send(ll_longlong_t *srvr_intrnl_format);
```

The **ll_longlong_send()** function is a cast function from the **ll_longlong_t** internal structure to the SENDRECV data type. It must be registered as an explicit cast function with the CREATE EXPLICIT CAST statement. For more information on cast functions, see "Creating Casts for Opaque Data Types" on page 9-14. ♦

Performing Bulk Copies

The database server can copy data in and out of a database with a bulk copy operation. In a bulk copy, the database server sends large numbers of column values in a copy file, rather than copying each column value individually. For large amounts of data, bulk copying is far more efficient than moving values individually.

The following Informix utilities can perform bulk copies:

- DB-Access performs bulk copies with the LOAD and UNLOAD statements.
- The **dbimport** and **dbexport** utilities perform bulk copies.
- The High Performance Loader (HPL) performs bulk copies.
- The **pload** utility loads and unloads a database from external files.

The database server can perform bulk copies on binary (internal) or character (external) representations of opaque-type data.

Import and Export Support Functions

The import and export support functions perform any tasks needed to process external text representation of an opaque type for a bulk load and unload. When the database server copies data to or from a database in external text format, it calls the following support functions for every value copied to or from the copy file:

- The import function imports text data by converting from external text representation to the internal format.
- The export function exports text data by converting from the internal format to the external text representation.

These support functions do not have to be named *import* and *export*, but they do have to perform the specified conversions. They should be reciprocal functions; that is, the import function should produce a value that the export function accepts as an argument and vice versa.

The import and export functions can take special actions on the values before they are copied. Typically, only opaque data types that contain smart large objects have import and export functions defined for them. For example, the export function for such a data type might create a file on the client computer, write the smart-large-object data from the database to this file, and send the name of the client file as the data to store in the copy file. Similarly, the import function for such a data type might take the client filename from the copy file, open the client file, and load the large-object data from the copy file into the database. The advantage of this design is that the smart-large-object data does not appear in the copy file; therefore, the copy file grows more slowly and is easier for users to read.

For small opaque data types, you do not usually need to define the import and export support functions. If you do not define import and export support functions, the database server uses the input and output functions, respectively, when it performs bulk copies.

For large opaque data types, the data that the input and output functions generate might be too large to fit in the file or might not represent all of the data in the object. To resolve this problem, you can use the import functions **filetoclob()** and **filetoblob()** and the export function **lotofile()**.

The IMPEXP Data Type

SQL statements support an internal data type called IMPEXP to hold the external representation of an opaque data type for a bulk copy. The IMPEXP data type allows for any possible change in the size of the data when it is converted between the two representations. The import and export support functions serve as cast functions between the IMPEXP and opaque data type.

Import Support Function

The import support function takes as an argument the structure that holds the bulk-copy format of the external representation of the user-defined type and returns the internal structure for the user-defined type.

Any files that the import function reads must reside on the database server computer. If you do not provide an import support function, the database server uses the input support function to import text data.

C

The following function signature is for an import support function of an opaque data type whose internal structure is **ll_longlong_t**:

```
ll_longlong_t * ll_longlong_import(mi_impexp
*extrnl_bcopy_format);
```

The **ll_longlong_import()** function is a cast function from the IMPEXP data type to the **ll_longlong_t** data structure. It must be registered as an implicit cast function with the CREATE IMPLICIT CAST statement. For more information on cast functions, see "Creating Casts for Opaque Data Types" on page 9-14. ♦

Export Support Function

The export function takes as an argument the internal structure for the opaque type and a structure that holds the bulk-copy format of the external representation of the opaque type.

If you do not provide an export support function, the database server uses the output support function to export text data.

C

The following function signature is for an export support function of an opaque data type whose internal structure is **ll_longlong_t**:

```
mi_impexp * ll_longlong_export(ll_long_t *intrnl_bcopy_format);
```

The **ll_longlong_export()** function is a cast function from the **ll_longlong_t** internal structure to the IMPEXP data type. It must be registered as an explicit cast function with the CREATE EXPLICIT CAST statement. For more information on cast functions, see "Creating Casts for Opaque Data Types" on page 9-14. ♦

Importbinary and Exportbinary Support Functions

The importbinary and exportbinary support functions perform any tasks needed to process the external binary representation of an opaque type for a bulk copy, as follows:

- The importbinary function imports binary data by converting from some binary representation to the internal representation.
- The exportbinary function exports binary data by converting from internal representation to some binary representation.

These support functions do not have to be named *importbinary* and *exportbinary*, but they do have to perform the specified conversions. They should be reciprocal functions; that is, the importbinary function should produce a value that the exportbinary function accepts as an argument and conversely. The IBM Informix DataBlade API provides functions that support conversion between different internal representations of opaque types.

For opaque data types that have identical external and internal representations, the import and importbinary support functions can be the same function. Similarly, the export and exportbinary support functions can be the same function.

IMPEXPBIN Data Type

SQL statements support an internal data type called IMPEXPBIN to hold the external binary representation of an opaque data type for a bulk copy. The IMPEXPBIN data type allows for any possible change in the size of the data when it is converted between the two representations. The importbinary and exportbinary support functions serve as cast functions between the IMPEXPBIN and opaque data type.

Importbinary Support Function

The importbinary support function takes as an argument a structure that holds the bulk-copy format of the external binary format of the opaque type and returns the internal structure for the opaque type.

Any files that the import function reads must reside on the database server computer. If you do not provide an importbinary support function, the database server imports the binary data in the database server internal representation of the opaque data type.

C

The following function signature is for an importbinary support function of an opaque data type whose internal structure is **ll_longlong_t**:

```
ll_longlong_t * ll_longlong_importbin(mi_impexpbin
*client_intrnl_bcopy_format);
```

The **ll_longlong_importbin()** function is a cast function from the IMPEXPBIN data type to the **ll_longlong_t** internal structure. It must be registered as an implicit cast function with the CREATE IMPLICIT CAST statement. For more information, see "Creating Casts for Opaque Data Types" on page 9-14. ♦

Exportbinary Support Function

The exportbinary support function takes as an argument the internal structure for the opaque type and returns a structure that holds the bulk-copy format of the external binary representation of the opaque type.

If you do not provide an exportbinary support function, the database server exports the binary data in the external binary representation of the opaque data type.

C

The following function signature is for an exportbinary support function of an opaque data type whose internal structure is **ll_longlong_t**:

```
mi_impexpbin * ll_longlong_exportbin(ll_longlong_t
*srvr_intrnl_bopy_format);
```

The **ll_longlong_exportbin()** function is a cast function from the **ll_longlong_t** internal structure to the IMPEXPBIN data type. It must be registered as an explicit cast function with the CREATE EXPLICIT CAST statement. For more information, see "Creating Casts for Opaque Data Types" on page 9-14. ♦

The Stream Support Functions

The **streamread()** and **streamwrite()** support functions allow the database server to treat opaque data in a stream representation. That is, in a sequential, flattened format. The DataBlade API provides generic functions that handle the transfer of stream data between the database server and other sites or storage media. The *IBM Informix DataBlade API Programmer's Guide* provides detailed information about using generic stream functions.

Important: *These support functions must be named **streamread** and **streamwrite**. The names are case insensitive.*

Inserting and Deleting Data

Some opaque data types might require special processing before they are saved to or removed from disk. The following support functions perform this special processing:

- **assign()**
- **destroy()**
- **update()**
- **deepcopy()**

Important: *These support functions must be named **assign**, **destroy, update,** and **deepcopy**. The names are case insensitive.*

The **assign()** and **destroy()** functions are required for opaque types that include smart large objects or multirepresentational data. If the data is stored in a smart large object, the internal structure of the opaque data type contains the LO handle to identify the location of the data; it does not contain the data itself. The **assign()**, **update()**, and **deepcopy()** support functions decide how and where to store the data, and the **destroy()** support function decides how to remove the data, regardless of where it is stored.

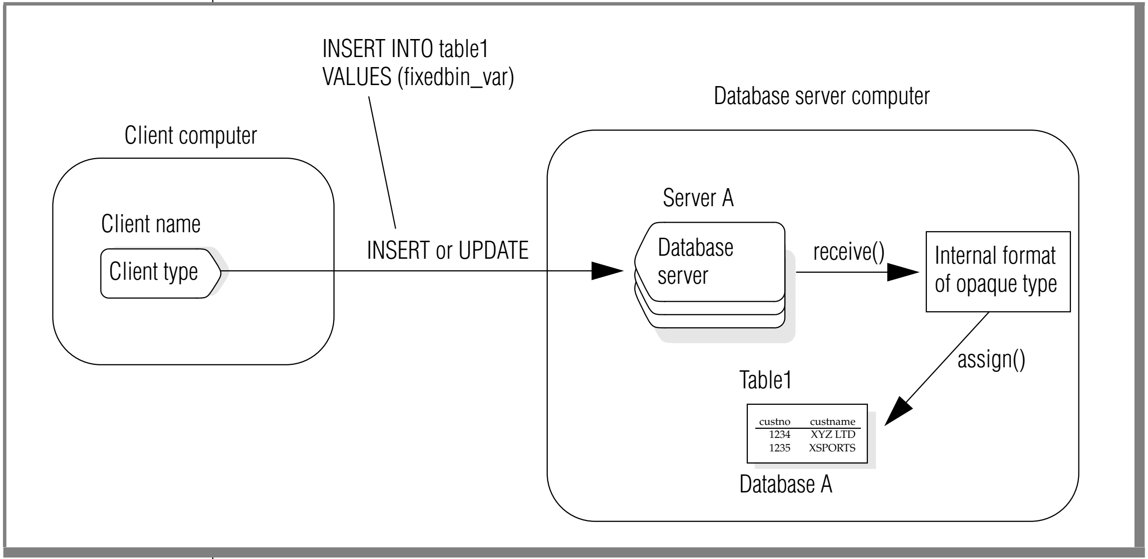
These functions use the **mi_*** memory allocation functions that are documented in the *IBM Informix DataBlade API Function Reference*. For detailed discussions about multirepresentational data types, refer to the DataBlade Developers Corner of the IBM Informix Developer Zone at www.ibm.com/software/data/developer/informix.

The assign() Function

The **assign()** function contains special processing to perform before an opaque data type is inserted into a table. The database server calls the **assign()** function just before it stores the internal representation of an opaque type on disk. For example, when a client application issues an INSERT, UPDATE, or LOAD statement, the database server calls the **assign()** function before it saves the internal representation of an opaque type in a column.

Figure 10-5 shows when the database server executes the **assign()** function.

Figure 10-5
Execution of the assign() Support Function



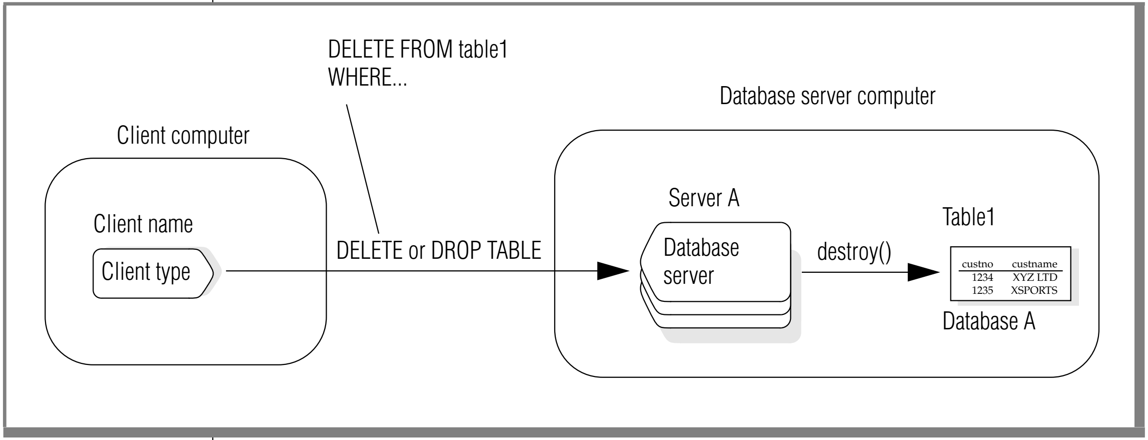
When you INSERT a value of an opaque data type, the **assign()** function takes the opaque data type as an argument, performs whatever additional processing might be required, and returns the final opaque type value for the database server to store in the table.

The destroy() Function

The **destroy()** function performs any processing necessary before the database server removes a row that contains opaque data. The database server calls the **destroy()** function just before it removes the internal representation of an opaque type from disk. For example, when a client application issues a DELETE or DROP TABLE statement, the database server calls the **destroy()** function before it deletes an opaque-type value from a column.

Figure 10-6 shows when the database server executes the **destroy()** function.

Figure 10-6
Execution of the destroy() Support Function



The **destroy()** function takes as an opaque data type. It does not return a value.

The update() Function

The **update()** function allows the database server to handle in-place updates of opaque data type values, improving the performance for an opaque type that has an expensive constructor. For example, an opaque type that contains a smart large object might benefit from an **update()** function. If no **update()** function is present, the database server calls the **assign()** function, which creates an entirely new smart large object, and then calls the **destroy()** function to delete the old smart large object. If the update only changes a few bytes in a large object, this is clearly not efficient.

The **update()** function provides for in-place update of an opaque data type. Like the **assign()** and **destroy()** functions, the **update()** function is an SQL function defined on a given UDT. It takes two arguments, both of the same UDT type, and returns the same UDT type. The first argument is the original value of the user-defined type, and the second argument is the new UDT value. The function must handle NULL.

The following statement registers an **update()** function for the multirepresentational data type **MyUDT**:

```
CREATE FUNCTION Update (MyUDT, MyUDT)
   RETURNS MyUDT
   WITH (HANDLESNULLS, NOT VARIANT)
   EXTERNAL NAME'/usr/lib/extend/blades/MyUDT.so(MyUDT_update)'
   LANGUAGE C;
```

The update function must check for updates that cross the threshold for multirepresentational data. For example, if a large quantity of data is updated to a small quantity, the **update()** routine needs to decrement the smart blob reference count and return the updated value as an in-row object.

The deepcopy() Function

Multirepresentational opaque types typically defer creating a smart large object until the database server calls the **assign()** function. Until **assign()** is called, the opaque type stores a large value in separately allocated memory and stores the pointer to that memory in the data structure of the opaque type.

However, the database server does not know about this additional memory when it copies a return value, so it copies only part of the value. In other words, the database server performs a shallow copy. This means that only allocations with a very high memory duration persist long enough for some query contexts.

The **deepcopy()** support function provides the method for the database server to copy the entire opaque type value and lets the opaque type support routines that use the default memory duration.

Alternatively, you might use some higher memory duration such as PER_STMT_EXEC. However, this strategy increases memory usage significantly because there are cases where using the default memory duration is sufficient. For information about PER_STMT_EXEC, refer to the *IBM Informix DataBlade API Programmer's Guide* and the *IBM Informix DataBlade API Function Reference*.

The **deepcopy()** function should make a copy of the input opaque type using memory allocated from default memory duration and return the copy. The functions that **deepcopy()** can use to allocate memory from default memory duration include **mi_alloc**, **mi_zalloc**, **mi_new_var**, and **mi_var_copy**. It is important to use memory allocated from those functions for the return UDT because the database server prepares the appropriate default memory duration depending on the query context before it invokes **deepcopy()**.

If the input UDT contains pointers to an out-of-row buffer, **deepcopy()** can copy the out-of-row data using memory from **mi_alloc** and store the pointer with that of memory in the copied UDT.

If the input UDT contains a reference to a smart large object, **deepcopy()** should copy the large object handle to the return value, but **deepcopy()** does not need to copy the large object.

Handling Smart Large Objects

If an opaque data type contains an embedded smart large object, you can define an **lohandles()** function for the opaque type. The **lohandles()** support function takes an instance of the opaque type and returns a list of the pointer structures for the smart large objects that are embedded in the data type. You might, for example, use a **lohandles()** function to provide information about which smart large object a given data type value is referencing.

The database server uses the **lohandles()** support function when it must search opaque-type values for references to smart large objects. The database server does *not* automatically call **lohandles()**. To execute this function, you must call it explicitly. You might use **lohandles()** for the following tasks:

- Performing an archive of the database
- Obtaining a reference count for the smart large objects
- Running the **oncheck** utility

A **lohandles()** support function does not perform automatic incrementing and decrementing of the reference count for a smart large object. You must handle the reference count explicitly in the **assign()** and **destroy()** functions, as follows:

- In the **assign()** function, increment the reference count with the DataBlade API function **mi_lo_increfcount()**.
- In the **destroy()** function, increment the reference count with the DataBlade API function **mi_lo_decrefcount()**.

If you define an opaque type that references one or more smart large objects, you must consider defining the following support functions:

- **assign()**
- destroy()
- **update()**
- **deepcopy()**
- An import function
- An export function
- An importbinary function
- An exportbinary function

For more information on **assign()** and **destroy()** support functions, see "Inserting and Deleting Data" on page 10-22. For information on the import, export, importbinary, and exportbinary support functions, see "Performing Bulk Copies" on page 10-17.

Comparing Data

The **compare()** function is an SQL-invoked function that sorts the target data type. The database server uses the **compare()** function in the CREATE INDEX statement and to execute the following components of the SELECT statement:

- The ORDER BY clause
- The UNIQUE and DISTINCT keywords
- The UNION keyword
- The BETWEEN operator

For more information on the SELECT statement, see the *IBM Informix Guide to SQL: Syntax*.

For the database server to be able to sort an opaque type, you must define a **compare()** function that handles the opaque type. The **compare()** function must follow these rules:

1. The name of the function must be **compare()**. However, the name is not case sensitive; the **compare()** function is the same as the **Compare()** function.
2. The function must accept two arguments, each of the data types to be compared.
3. The function must return an integer value to indicate the result of the comparison, as follows:
   - <0 to indicate that the first argument is less than (<) the second argument
   - `0` to indicate that the two arguments are equal (=)
   - >0 to indicate that the first argument is greater than (>) the second argument

The **compare()** function is the support function for the built-in secondary-access method, B-tree. For more information on the built-in secondary-access method, see “Generic B-Tree Index” on page 11-4. For more information on how to customize a secondary-access method for an opaque data type, see “Using Operator Classes” on page 11-3.

GLS

Handling Locale-Sensitive Data

An Informix database has a fixed locale per database. This locale, the *database locale*, is attached to the database at the time that the database is created. In any given database, all character data types (such as CHAR, NCHAR, VARCHAR, NVARCHAR, and TEXT) contain data in the code set that the database locale supports.

However, using the SQL statement SET COLLATION you can specify the collation order to use at runtime, which is independent of the locale used to store data in the database, and lasts for the duration of the session. You can use the **mi_get_db_locale()** function to determine which locale a user has set for the collation order in a session. If the user has not changed the collation, **mi_get_db_locale()** returns the default database locale. See the *IBM Informix Guide to SQL: Syntax* for information about the SET COLLATION statement. See the *IBM Informix DataBlade API Function Reference* for information about the **mi_get_db_locale()** function.

An opaque data type can hold character data. The following support functions provide the ability to transfer opaque-type data between a client application and the database server:

- The input and output support functions provide the ability to transfer the external representation of the opaque type.
- The receive and send support functions provide the ability to transfer the internal representation of the opaque type.

However, the ability to transfer the data between client application and database server is not sufficient to support locale-sensitive data. It does not ensure that the data is correctly manipulated at each end. You must ensure that both sides of the connection handle the locale-sensitive data, as follows:

- At the client side of the connection, the client application must handle the locale-sensitive data for opaque-type columns correctly.

  It must also have the **CLIENT_LOCALE** environment variable set correctly.
- At the database server side of the connection, you must ensure that the appropriate support functions handle the locale-sensitive data.

  In addition, the **DB_LOCALE** and **SERVER_LOCALE** environment variables must be set correctly.

For more information on the **CLIENT_LOCALE**, **DB_LOCALE**, and **SERVER_LOCALE** environment variables, see the *IBM Informix GLS User's Guide*.

To help you write support functions that handle locale-sensitive data, the IBM Informix GLS API is provided. The GLS API is a thread-safe library. This library contains C functions that allow your support functions to obtain locale-specific information from GLS locales, including:

- Functions to manipulate locale-sensitive data in a portable fashion
- Functions to handle single-byte and multibyte character access
- Functions to manipulate other locale-sensitive data, such as the end-user formats of date, time, or monetary data

For an overview of the GLS API, see the *IBM Informix GLS User's Guide*. For a description of the GLS API functions, see the *IBM Informix GLS Programmer's Manual*.

Locale-Sensitive Input and Output Support Functions

The LVARCHAR (and **mi_lvarchar**) data type can hold data in the code set of the client or database locale. This data includes single-byte (ASCII and non-ASCII) and multibyte character data. The LVARCHAR data type holds opaque-type data as it is transferred to and from the database server in its external representation. Therefore, the external representation of an opaque data type can hold single-byte or multibyte data.

However, you must write the input and output support functions to interpret the LVARCHAR data in the correct locale. These support functions might need to perform code-set conversion if the client locale and database locale support different code sets. For more information on code-set conversion, see the *IBM Informix GLS User's Guide*.

Locale-Sensitive Receive and Send Support Functions

The SENDRECV (and **mi_sendrecv**) data type holds the internal structure of an opaque type. This internal structure can contain the following types of locale-sensitive data:

- Character fields that can hold data in the code set of the client or database locale

  This data includes single-byte (ASCII and non-ASCII) and multibyte character data.
- Monetary, date, or time fields that hold a locale-specific representation of the data

The client application has no way of interpreting the fields of the internal structure because an opaque type is encapsulated.

The SENDRECV data type holds opaque-type data as it is transferred to and from the database server in this internal representation. You must write the receive and send support functions to interpret the locale-specific data within the SENDRECV structure.

Chapter

11 Extending an Operator Class

In This Chapter

This chapter describes how to extend the functionality of operator classes. An *operator class* is the set of functions that is associated with a secondary-access method. The database server provides two ways to extend operator classes:

- Extensions of operator classes that the database server provides

  When you want to order the data in a different sequence or provide index support for a UDT, you must extend an operator class.
- User-defined operator classes

  When one of the existing secondary-access methods cannot easily index a UDT, you might need to create a new operator class.

Using Operator Classes

For most situations, when you build an index, you can use the default operators that are defined for a secondary-access method. This section provides a brief introduction to secondary-access methods and operator classes.

For a more detailed discussion of this topic, see the *Performance Guide*.

Secondary-Access Methods

A *secondary-access method*, often called an *index*, is a set of user-defined functions that build, access, and manipulate an index structure. These functions encapsulate index operations, such as how to scan, insert, delete, or update nodes in an index. A secondary-access method describes how to access the data in an index that is built on a column (column index) or on a user-defined function (functional index). Typically, a secondary-access method speeds up the retrieval of a type of data.

The database server provides definitions for the following secondary-access methods in the system catalog tables of each database:

- A generic B-tree
- An R-tree

DataBlade modules can provide additional secondary-access methods for use with UDTs. For more information about secondary-access methods of DataBlade modules, refer to the user guide for each DataBlade module. For more information about R-trees, refer to the *IBM Informix R-Tree Index User's Guide*.

Generic B-Tree Index

In traditional relational database systems, the B-tree access method handles only built-in data types and therefore can compare only two keys of built-in data types. The B-tree index is useful for a query that retrieves a range of data values. To support UDTs, the database server provides an extended version of a B-tree, the *generic* B-tree *index*.

The database server uses the generic B-tree index as the built-in secondary-access method. This secondary-access method is registered in the **sysams** system catalog table with the name **btree**. When you use the CREATE INDEX statement (without the USING clause) to create an index, the database server creates a generic B-tree index. The following statement creates a B-tree index on the **zipcode** column of the **customer** table:

```
CREATE INDEX zip_ix ON customer (zipcode)
```

For more information, see the CREATE INDEX statement in the *IBM Informix Guide to SQL: Syntax*.

R-Tree Index

The database server can support the *R-tree index* for columns that contain spatial data such as maps and diagrams. An R-tree index is most beneficial when queries look for objects that are within other objects or for an object that contains one or more objects.

To use an R-tree index, install a spatial DataBlade module such as the Spatial DataBlade module, Geodetic DataBlade module, or any other third-party DataBlade module that implements an R-tree index.

Other User-Defined Secondary-Access Methods

A DataBlade module can provide a UDT to handle a particular type of data. The module might also provide a new secondary-access method (index) for the new data type that it defines. For example, the Excalibur Text DataBlade module provides an index to search text data. For more information, refer to the *Excalibur Text Search DataBlade Module User's Guide*. For more information on the types of data and functions that each DataBlade module provides, refer to the user guide for the DataBlade module. The **sysams** system catalog table describes the secondary-access methods that exist in your database. For information about **sysams**, see the *IBM Informix Guide to SQL: Reference*.

Operator Classes

An *operator class* is a group of functions that allow the secondary-access method to store and search for values of a particular data type. The query optimizer uses an operator class to determine if an index can process the query with the least cost. For more information on the query optimizer, see the *Performance Guide*.

The operator-class functions fall into the following categories:

- Strategy functions

  The database server uses the *strategy functions* of a secondary-access method to help the query optimizer determine whether a specific index is applicable to a specific operation on a data type. The strategy functions are the operators that can appear in the filter of an SQL statement.

- Support functions

  The database server uses the *support functions* of a secondary-access method to build and access the index. End users do not call these functions directly. When an operator in the filter of a query matches one of the strategy functions, the secondary-access method uses the support functions to traverse the index and obtain the results.

Each secondary-access method has a *default operator class* associated with it. By default, the CREATE INDEX statement associates the default operator class with an index.

The database server stores information about operator classes in the **sysopclasses** system catalog table.

Generic B-Tree Operator Class

The built-in secondary-access method, the generic B-tree, has a single operator class defined in the **sysopclasses** system catalog table. This operator class, called **btree_ops**, is the default operator class for the **btree** secondary-access method.

The database server uses the **btree_ops** operator class to specify:

- The strategy functions to tell the optimizer which filters in a query can use a B-tree index
- The support function to build and search the B-tree index

The CREATE INDEX statement in "Generic B-Tree Index" on page 11-4 shows how to create a B-tree index whose column uses the **btree_ops** operator class. This CREATE INDEX statement does not need to specify the **btree_ops** operator class because **btree_ops** is the default operator class for the **btree** access method.

For more information on the **btree** secondary-access method, see "Generic B-Tree Index" on page 11-4.

B-Tree Strategy Functions

The **btree_ops** operator class defines the following strategy functions for the **btree** access method:

- **lessthan** (<)
- **lessthanorequal** (<=)
- **equal** (=)
- **greaterthanorequal** (>=)
- **greaterthan** (>)

These strategy functions are all *operator functions*. That is, each function is associated with an operator symbol; in this case, with a relational-operator symbol. For more information, see "Relational Operators" on page 6-5.

B-Tree Support Function

The **btree_ops** operator class has one support function, a comparison function called **compare()**. The **compare()** function is a user-defined function that returns an integer value to indicate whether its first argument is equal to, less than, or greater than its second argument, as follows:

- A value of `0` when the first argument is *equal to* the second argument
- A value less than 0 when the first argument is *less than* the second argument
- A value greater than 0 when the first argument is *greater than* the second argument

The B-tree secondary-access method uses the **compare()** function to traverse the nodes of the generic B-tree index. To search for data values in a generic B-tree index, the secondary-access method uses the **compare()** function to compare the key value in the query to the key value in an index node. The result of the comparison determines if the secondary-access method needs to search the next lower level of the index or if the key resides in the current node.

The generic B-tree access method also uses the **compare()** function to perform the following tasks for generic B-tree indexes:

- Sort the keys before building the index
- Determine the linear ordering of keys in a generic B-tree index
- Evaluate the relational operators

The database server uses the **compare()** function to evaluate comparisons in the SELECT statement. To provide support for these comparisons for opaque data types, you must write the **compare()** function. For more information, see "Conditional Operators for Opaque Data Types" on page 9-20.

R-Tree Index Operator Class

The R-tree secondary-access method has an operator class defined in the **sysopclasses** system catalog table. This operator class, called **rtree_ops**, is the default operator class for the **rtree** secondary-access method. The database server defines the default R-tree operator class in the system catalog tables but does *not* provide the operator-class functions to implement this operator class.

To use an R-tree index, install a spatial DataBlade module such as the Spatial DataBlade module, Geodetic DataBlade module, or any other third-party DataBlade module that implements an R-tree index. For more information on R-tree indexes, refer to *IBM Informix R-Tree Index User's Guide*. For more information on the spatial DataBlade modules, consult the appropriate DataBlade module user guide.

Extending an Existing Operator Class

You can define operator-class functions of an operator class only for existing data types. When you create a UDT, you must determine whether you need to create operator-class functions for this data type. The creation of new operator-class functions that have the same names as the existing operator class functions is the most common way to extend an existing operator class.

To extend the functionality of an operator-class function, write a function that has the same name and return value. You provide parameters for the new data type and write the function to handle the new parameters. *Routine overloading* allows you to create many functions, all with the same name but each with a different parameter list. The database server then uses *routine resolution* to determine which of the overloaded functions to use based on the data type of the value. For more information on routine overloading and routine resolution, see Chapter 3, "Running a User-Defined Routine."

To define operator-class functions for a user-defined data type

1. Decide which of the secondary-access methods can support the UDT.
2. Extend the operator classes of the chosen secondary-access method or methods.

   To allow end users to use the user-defined type with the operators that are associated with the secondary-access method, write new strategy and support functions to handle this new data type.

Extensions of the btree_ops Operator Class

Before the database server can support generic B-tree indexes on a UDT, the operator classes associated with the B-tree secondary-access method must be able to handle that data type. The default operator class for the generic B-tree secondary-access method is called **btree_ops**. Initially, the operator-class functions (strategy and support functions) of the **btree_ops** operator class handle the built-in data types. When you define a new data type, you must extend these operator-class functions to handle the data type.

Important: *You cannot extend the* ***btree_ops*** *operator class for the built-in data types.*

After you determine how you want to implement the relational operators for a UDT, you can extend the **btree_ops** operator class so that the query optimizer can consider use of a B-tree index for a query that contains a relational operator.

To extend the default operator class for a generic B-tree index

1. Write functions for the B-tree strategy functions that accept the UDT in their parameter list.

   The relational-operator functions serve as the strategy functions for the **btree_ops** operator class. If you have already defined these relational-operator functions for the UDT, the generic B-tree index uses them as its strategy functions. For example, you might have defined the relational-operator functions when you extended an aggregate for the user-defined type. (See "Example of Extending a Built-In Aggregate" on page 8-6.)

2. Register the strategy functions in the database with the CREATE FUNCTION statement.

   If you already registered the relational-operator functions, you do not need to reregister them as strategy functions.

3. Write a function in C or Java for the B-tree support function, **compare()**, that accepts the UDT in its parameter list. (The **compare()** function cannot be in SPL.)

   The **compare()** function also provides support for a UDT in comparison operations in a SELECT statement (such as the ORDER BY clause or the BETWEEN operator). If you have already defined this comparison function for the UDT, the generic B-tree index uses it as its support function.

4. Register the support functions in the database with the CREATE FUNCTION statement.

   For opaque data types, you might have already defined this function to provide support for the comparison operations in a SELECT statement (such as the ORDER BY clause or the BETWEEN operator) on your opaque data type.

For more information on strategy functions, see "B-Tree Strategy Functions" on page 11-7. For information on relational operators for an opaque data type, see "Conditional Operators for Opaque Data Types" on page 9-20.

After you register the support function, use the CREATE INDEX statement to create a B-tree index on the column of the table that contains the UDT. The CREATE INDEX statement does not need the USING clause because you have extended the default operating class for the default index type, a generic B-tree index, to support your UDT.

The query optimizer can now consider use of this generic B-tree index to execute queries efficiently. For more information on the performance aspects of column indexes, see the *Performance Guide*.

The previous steps extend the default operator class of the generic B-tree index. You could also define a new operator class to provide another order sequence. For more information, see "Creating a New B-Tree Operator Class" on page 11-15.

Reasons for Extending btree_ops

The strategy functions of **btree_ops** are the relational operations that end users can use in expressions. (For a list of the relational operators, see "B-Tree Strategy Functions" on page 11-7.) The generic B-tree index handles only the built-in data types. When you write relational-operator functions that handle a new UDT, you extend the generic B-tree so that it can handle the UDT in a column or a user-defined function. To create B-tree indexes on columns or functions of the new data type, you must write new relational-operator functions that can handle the new data type.

In the relational-operator functions, you determine the following behavior of a B-tree index:

- What single value does the B-tree secondary-access method use to order the index?

  For a particular UDT, the relational-operator functions must compare two values of this data type for the data type to be stored in the B-tree index.

- In what order does the B-tree index sort the values?

  For a particular UDT, the relational-operator functions must determine what constitutes an ordered sequence of the values.

Generating a Single Value for a New Data Type

A B-tree index indexes one-dimensional objects. It uses the relational-operator functions to compare two one-dimensional values. It then uses the relationship between these values to determine how to traverse the B-tree and in which node to store a value.

The relational-operator functions handle built-in data types. (For more information on built-in data types, see the chapter on data types in the *IBM Informix Guide to SQL: Reference*.) The built-in data types contain one-dimensional values. For example, the INTEGER data type holds a single integer value. The CHAR data type holds a single character string. The DATE data type holds a single date value. The values of all these data types can be ordered linearly (in one dimension). The relational-operator functions can compare these values to determine their linear ordering.

When you create a new UDT, you must ensure that the relational-operator functions can compare two values of the UDT. Otherwise, the comparison cannot occur, and the UDT cannot be used in a B-tree index.

For example, suppose you create the **circle** opaque type to implement a circle. A circle is a spatial object that might be indexed best with a user-defined secondary-access method such as an R-tree, which handles multidimensional objects. However, you can use the **circle** data type in a B-tree index if you define the relational operators on the value of its area: one **circle** is less than a second **circle** if its area is less than the area of the second.

Changing the Sort Order

A generic B-tree uses the relational operators to determine which value is less than another. These operators use lexicographical sequence (numeric order for numbers, alphabetic order for characters, chronological order for dates and times) for the values that they order.

GLS

The relational-operator functions use the code-set order for character data types (CHAR, VARCHAR, and LVARCHAR) and a localized order for the NCHAR and NVARCHAR data types. When you use the default locale, U.S. English, code-set order and localized order are those of the ISO 8895-1 code set. When you use a nondefault locale, these two orders might be different. For more information on locales, see the *IBM Informix GLS User's Guide*. ♦

For some UDTs, the relational operators in the default B-tree operator class might not achieve the order that you want. You can define the relational-operator functions for a particular user-defined type so that the sort order changes from a lexicographical sequence to some other sequence.

Tip: *When you extend an operator class, you can change the sort order for a* UDT. *To provide an alternative sort order for all data types that the B-tree handles, you must define a new operator class. For more information, see "Creating a New B-Tree Operator Class" on page 11-15.*

For example, suppose you create an opaque data type, **ScottishName**, that holds Scottish names, and you want to order the data type in a different way than the U.S. English collating sequence. You might want the names *McDonald* and *MacDonald* to appear together on a phone list. This data type can use a B-tree index because it defines the relational operators that equate the strings *Mc* and *Mac*.

To order the data type in this way, write the relational-operator functions so that they implement this new order. For the strings *Mc* and *Mac* to be equal, you must define the relational-operator functions that:

- Accept the opaque data type, **ScottishName**, in the parameter list
- Contain code that equates *Mc* and *Mac*

The following steps use the steps described in "Extensions of the btree_ops Operator Class" on page 11-9 to extend the **btree_ops** operator class.

To support the ScottishName data type

1. Prepare and register the strategy functions that handle the **ScottishName** data type: **lessthan()**, **lessthanorequal()**, **equal()**, **greaterthan()**, and **greaterthanorequal()**.

   For more information, refer to Chapter 4, "Developing a User-Defined Routine."
2. Prepare and register the external function for the **compare()** support function that handles the **ScottishName** data type.

You can now create a B-tree index on a **ScottishName** column:

```
CREATE TABLE scot_cust
(
   cust_id integer,
   cust_name ScottishName
   ...
);
CREATE INDEX cname_ix
   ON scot_cust (cust_name);
```

The optimizer can now choose whether to use the **cname_ix** index to evaluate the following query:

```
SELECT * FROM scot_cust
WHERE cust_name = 'McDonald'::ScottishName
```

Creating an Operator Class

For most indexing, the operators in the default operator class of a secondary-access method provide adequate support. However, when you want to order the data in a different sequence than the default operator class provides, you can define a new operator class for the secondary-access method.

The CREATE OPCLASS statement creates an operator class. It provides the following information about the operator class to the database server:

- The name of the operator class
- The name of the secondary-access method with which to associate the functions of the operator class
- The names and, optionally, the parameters of the strategy functions
- The names of the support functions

The database server stores this information in the **sysopclasses** system catalog table. You must have the Resource privilege for the database or be the DBA to create an operator class.

The database server provides the default operator class, **btree_ops**, for the generic B-tree access method. The following CREATE OPCLASS statement creates a new operator class for the generic B-tree access method. You must list the strategy functions in the order shown:

```
CREATE OPCLASS new_btree_ops FOR btree
   STRATEGIES (lessthan, lessthanorequal, equal,
      greaterthanorequal, greaterthan)
   SUPPORT(compare);
```

For more information, see "Generic B-Tree Index" on page 11-4.

You might want to create a new operator class for:

- The generic B-tree secondary-access method

  A new operator class can provide an additional sort order for all data types that the B-tree index can handle.
- Any user-defined secondary-access methods

  A new operator class can provide additional functionality to the strategy functions of the operator class.

Creating a New B-Tree Operator Class

To traverse the index structure, the generic B-tree index uses the sequence that the relational operators define. By default, a B-tree uses the lexicographical sequence of data because the default operator class, **btree_ops**, contains the relational-operator functions. (For more information on this sequence, see "Changing the Sort Order" on page 11-12.) For a generic B-tree to use a different sequence for its index values, you can create a new operator class for the **btree** secondary-access method. You can then specify the new operator class when you define an index on that data type.

When you create a new operator class for the generic B-tree index, you provide an additional sequence for organizing data in a B-tree. When you create the B-tree index, you can specify the sequence that you want a column (or user-defined function) in the index to have.

To create a new operator class for a generic B-tree index

1. Write functions for the B-tree strategy functions that accept the appropriate data type in their parameter list.

   The B-tree secondary-access method expects five strategy functions; therefore, any new operator class must define exactly five. The parameter data types can be built in or user defined. However, each function *must* return a Boolean value. For more information on strategy functions, see "B-Tree Strategy Functions" on page 11-7.

2. Register the new strategy functions in the database with the CREATE FUNCTION statement.

   You must register the set of strategy functions for each data type on which you are supporting the operator class.

3. Write the external function for the new B-tree support function that accepts the appropriate data type in its parameter list.

   The B-tree secondary-access method expects one support function; therefore, any new operator class must define only one. The parameter data types can be built-in or UDTs. However, the return type must be integer. For more information on support functions, see "B-Tree Support Function" on page 11-7.

4. Register the new support function in the database with the CREATE FUNCTION statement.

   You must register a support function for each data type on which you are supporting the operator class.

5. Create the new operator class for the B-tree secondary-access method, **btree**.

   When you create an operator class, specify the following in the CREATE OPCLASS statement:

   - After the OPCLASS keyword, the name of the new operator class
   - In the FOR clause, **btree** as the name of the secondary-access method with which to associate the operator class
   - In the STRATEGIES clause, a parenthetical list of the names of the strategy functions for the operator class

     You registered these functions in step 2. You must list the functions in the order that the B-tree secondary-access method expects: the first function is the replacement for **lessthan()**, the second for **lessthanorequal()**, and so on.
   - In the SUPPORT clause, the name of the support function to use to search the index

     You registered this function in step 4. It is the replacement for the **compare()** function.

   For more information on how to use the CREATE OPCLASS statement, refer to the *IBM Informix Guide to SQL: Syntax*.

These steps create the new operator class of the generic B-tree index. You can also extend the default operator class to provide support for new data types. For more information, see "Extensions of the btree_ops Operator Class" on page 11-9.

To use the new operator class, specify the name of the operator class after the column or function name in the CREATE INDEX statement.

Creating an Absolute-Value Operator Class

As an example, suppose you want to define a new ordering for integers. The lexicographical sequence of the default B-tree operator class orders integers numerically: -4 < -3 < -2 < -1 < 0 < 1 < 2 < 3. Instead, you might want the numbers -4, 2, -1, -3 to appear in order of absolute value.

```
-1, 2, -3, -4
```

To obtain the absolute-value order, you must define external functions that treat negative integers as positive integers. The following steps create a new operator class called **abs_btree_ops** with strategy and support functions that provide the absolute-value order:

1. Write and register external functions for the new strategy functions: **abs_lessthan()**, **abs_lessthanorequal()**, **abs_equal()**, **abs_greaterthan()**, and **abs_greaterthanorequal()**.

   For more information, refer to Chapter 4, "Developing a User-Defined Routine."
2. Register the five new strategy functions with the CREATE FUNCTION statement.
3. Write the C function for the new support function: **abs_compare()**.

   Compile this function and store it in the **absbtree.so** shared-object file.
4. Register the new support function with the CREATE FUNCTION statement.
5. Create the new **abs_btree_ops** operator class for the B-tree secondary-access method.

You can now create a B-tree index on an INTEGER column and associate the new operator class with this column:

```
CREATE TABLE cust_tab
(
   cust_name varchar(20),
   cust_num integer
   ...
);
CREATE INDEX c_num1_ix
   ON cust_tab (cust_num abs_btree_ops);
```

The **c_num1_ix** index uses the new operator class, **abs_btree_ops**, for the **cust_num** column. An end user can now use the absolute value functions in SQL statements, as in the following example:

```
SELECT * FROM cust_tab WHERE abs_lt(cust_num, 7)
```

In addition, because the **abs_lt()** function is part of an operator class, the query optimizer can use the **c_num1_ix** index when it looks for all **cust_tab** rows with **cust_num** values between -7 and 7. A **cust_num** value of `-8` does *not* satisfy this query.

The default operator class is still available for indexes. The following CREATE INDEX statement defines a second index on the **cust_num** column:

```
CREATE INDEX c_num2_ix ON cust_tab (cust_num);
```

The **c_num2_ix** index uses the default operator class, **btree_ops**, for the **cust_num** column. The following query uses the operator function for the default *less than* (<) operator:

```
SELECT * FROM cust_tab WHERE lessthan(cust_num, 7)
```

The query optimizer can use the **c_num2_ix** index when it looks for all **cust_tab** rows with **cust_num** values less than 7. A **cust_num** value of `-8` *does* satisfy this query.

Defining an Operator Class for Other Secondary-Access Methods

You can also define operator classes for user-defined secondary-access methods. A *user-defined secondary-access method* is one that a database developer has defined to implement a particular type of index. These access methods might have been defined in the database by a DataBlade module.

Tip: *You can examine the **sysams** system catalog table to determine which secondary-access methods your database defines. For information on the columns of the **sysams** system catalog table, see the "IBM Informix Guide to SQL: Reference."*

When you define an operator class on a user-defined secondary-access method, you provide support and strategy functions just as you do when you create an operator class on the generic B-tree index. You must be careful to conform to any operator class requirements of the user-defined secondary-access class. Before you implement an operator class for a user-defined secondary-access method, consult the documentation for the method.

You perform the same steps to define an operator class on a user-defined secondary-access method as you use to define an operator class on the generic B-tree index. (See "Creating a New B-Tree Operator Class" on page 11-15.) The only difference is that to create the index, you must specify the name of the user-defined secondary-access method in the USING clause of the CREATE INDEX statement.

Dropping an Operator Class

The DROP OPCLASS statement removes the definition for an operator class from the database. The database server removes the operator-class definition from the **sysopclasses** system catalog table. You must be the owner of the operator class or the DBA to drop its definition from the database.

You must remove all dependent objects before you can drop the operator class. For example, suppose you have created a new operator class called **abs_btree_ops** for the generic B-tree index. (For more information, see "Creating a New B-Tree Operator Class" on page 11-15.) To drop the **abs_btree_ops** operator class from the database, you must first ensure that:

- You are the owner (the person who created the operator class) or the DBA.
- No indexes are currently defined that use the **abs_btree_ops** operator class.

  If such indexes exist, you must first remove them from the database.

After you meet the preceding conditions, the following statement removes the definition of **abs_btree_ops** from the database:

```
DROP OPCLASS abs_btree_ops RESTRICT
```

The RESTRICT keyword is required in the DROP OPCLASS syntax.

Chapter

12 Managing a User-Defined Routine

In This Chapter

This chapter describes how to manage UDRs. It includes the following topics:

Assigning the Execute Privilege to a Routine

The Execute privilege enables users to invoke a UDR. You might invoke the UDR from the EXECUTE or CALL statements or from a function in an expression. By default, the following users have Execute privilege, which enables them to invoke a UDR:

- Any user with the DBA privilege can execute any routine in the database.
- If the routine is registered with the qualified CREATE DBA FUNCTION or CREATE DBA PROCEDURE statements, only users with the DBA privilege have the Execute privilege for that routine by default.
- If the database is not ANSI compliant, user **public** (any user with Connect database privilege) automatically has the Execute privilege to a routine that is not registered with the DBA keyword.

ANSI

- In an ANSI-compliant database, the procedure owner and any user with the DBA privilege can execute the routine without receiving additional privileges. ♦

Granting and Revoking the Execute Privilege

To control the Execute privilege on a UDR, use the EXECUTE ON clause of the GRANT and REVOKE statements. The database server stores privileges for UDRs in the **sysprocauth** system catalog table.

UDRs have the following GRANT and REVOKE requirements for the Execute privilege:

- The DBA can grant the Execute privilege to or revoke it from *any* routine in the database.
- The creator of a routine can grant or revoke the Execute privilege on that particular routine. The creator forfeits the ability to grant or revoke by including the AS *grantor* clause with the GRANT EXECUTE ON statement.
- Another user can grant the Execute privilege if the owner applied the WITH GRANT keywords in the GRANT EXECUTE ON statement.

A DBA or the routine owner must explicitly grant the Execute privilege to non-DBA users for the following conditions:

- A routine in an ANSI-compliant database
- A database with the **NODEFDAC** environment variable set to `yes`
- A routine that was registered with the DBA keyword

An owner can restrict the Execute privilege on a routine even though the database server grants that privilege to **public** by default. To do so, issue the REVOKE EXECUTE ON...PUBLIC statement. The DBA and owner can still execute the routine and can grant the Execute privilege to specific users, if applicable.

A user might receive the Execute privilege accompanied by the WITH GRANT option authority to grant the Execute privilege to other users. If a user loses the Execute privilege on a routine, the Execute privilege is also revoked from all users to whom that user granted the Execute privilege.

The following example shows an **equal()** function defined for a UDT and the GRANT statement to enable user **mary** to execute this variation of the **equal()** function:

```
CREATE FUNCTION equal (arg1 udtype1, arg2 udtype1)
RETURNING BOOLEAN
EXTERNAL NAME "/usr/lib/udtype1/lib/libbtype1.so(udtype1_equal)"
LANGUAGE C
END FUNCTION;

GRANT EXECUTE ON equal(udtype1, udtype1) to mary
```

User **mary** does not have permission to execute any other UDR named **equal()**.

For more information, see the GRANT and REVOKE statements in the *IBM Informix Guide to SQL: Syntax*.

Privileges on Objects Associated with a UDR

The database server checks the existence of any referenced objects and verifies that the user who invokes the UDR has the necessary privileges to access the referenced objects. For example, if a user executes a UDR that updates data in a table, the user must have the Update privilege for the table or columns referenced in the UDR.

A routine can reference the following objects:

- Tables and columns
- UDTs
- Other routines executed by the routine

In the course of routine execution, the owner of the routine, *not* the user who runs the routine, owns any unqualified objects that the routine creates. The database server verifies that the objects exist and that the UDR owner has the necessary privileges to access them. The user who executes the UDR runs with the privileges of the owner of the UDR.

The following example shows an SPL procedure called **promo()** that creates two tables, **hotcatalog** and **libby.mailers**:

```
CREATE PROCEDURE promo()

   CREATE TABLE hotcatalog
   (
      catlog_num INTEGER
      cat_advert VARCHAR(255, 65)
      cat_picture BLOB
   ) PUT cat_picturein sb1;

   CREATE TABLE libby.maillist
   (
      cust_num INTEGER
      interested_in SET(catlog_num INTEGER)
   );
END PROCEDURE;
```

Suppose user **tony** executes the CREATE PROCEDURE statement to register the SPL **promo()** procedure. User **marty** executes the **promo()** procedure with an EXECUTE PROCEDURE statement, which creates the table **hotcatalog**. Because no owner name qualifies table name **hotcatalog**, the routine owner (**tony)** owns **hotcatalog**. By contrast, the qualified name **libby.maillist** identifies **libby** as the owner of **maillist**.

Executing a UDR as DBA

If a DBA creates a routine using the DBA keyword, the database server automatically grants the Execute privilege only to other users with the DBA privilege. However, a DBA can explicitly grant the Execute privilege on a DBA routine to a non-DBA user.

When a user executes a routine that was registered with the DBA keyword, that user assumes the privileges of a DBA for the duration of the routine. If a user who does not have the DBA privilege runs a DBA routine, the database server implicitly grants a temporary DBA privilege to the invoker. Before the database server exits from a DBA routine, it implicitly revokes the temporary DBA privilege.

Using DBA Privileges with Objects and Nested UDRs

Objects created in the course of running a DBA routine are owned by the user who executes the routine unless a statement in the routine explicitly names someone else as the owner. For example, suppose that user **tony** registers the **promo()** routine on page 12-6, but includes the DBA keyword:

```
CREATE DBA PROCEDURE promo()
...
END PROCEDURE;
```

Although user **tony** owns the routine, if user **marty** runs it, user **marty** owns table **hotcatalog**. User **libby** owns **libby.maillist** because her name qualifies the table name, making her the table owner.

A called routine does not inherit the DBA privilege. If a DBA routine executes a routine that was created without the DBA keyword, the DBA privileges do not affect the called routine.

If a routine that is registered without the DBA keyword calls a DBA routine, the caller must have Execute privileges on the called DBA routine. Statements within the DBA routine execute as they would within any DBA routine.

The following example demonstrates what occurs when a DBA and non-DBA routine interact. Procedure **dbspace_cleanup()** executes procedure **cluster_catalog()**. Procedure **cluster_catalog()** creates an index. The C-language source for **cluster_catalog()** includes the following statements:

```
strcopy(statement, "CREATE INDEX stmt");
ret = mi_exec(conn,
"create cluster index c_clust_ix on catalog(catalog_num)",
MI_QUERY_NORMAL);
```

DBA procedure **dbspace_cleanup()** invokes the other routine with the following statement:

```
EXECUTE PROCEDURE cluster_catalog(hotcatalog)
```

Assume **tony** registered **dbspace_cleanup()** as a DBA procedure, and **cluster_catalog()** is registered without the DBA keyword, as follows:

```
CREATE DBA PROCEDURE dbspace_cleanup(loc CHAR)
   EXTERNAL NAME ...
   LANGUAGE C
END PROCEDURE
CREATE PROCEDURE cluster_catalog(catalog CHAR)
   EXTERNAL NAME ...
LANGUAGE C
END PROCEDURE
GRANT EXECUTION ON dbspace_cleanup(CHAR) to marty;
```

User **marty** runs **dbpace_cleanup()**. Index **c_clust_ix** is created by a non-DBA routine. Therefore **tony**, who owns both routines, also owns **c_clust_ix**. By contrast, **marty** owns index **c_clust_ix** if **cluster_catalog()** is a DBA procedure, as in the following registering and grant statements:

```
CREATE PROCEDURE dbspace_cleanup(loc CHAR)
   EXTERNAL NAME ...
   LANGUAGE C
END PROCEDURE
CREATE DBA PROCEDURE cluster_catalog(catalog CHAR)
   EXTERNAL NAME ...
LANGUAGE C
END PROCEDURE
GRANT EXECUTION ON cluster_catalog(CHAR) to marty;
```

The **dbspace_cleanup()** procedure need not be a DBA procedure to call a DBA procedure.

Modifying a User-Defined Routine

To modify a UDR, you might need to drop and then reregister the routine and its support functions and reload the files that hold the executable version of the routine with new executable files. However you can make some changes in place. ALTER FUNCTION and ALTER PROCEDURE let you modify some attributes of a routine without dropping the routine.

C

Modifying a C UDR

When the database server shuts down, it releases all memory that it has reserved, including memory for shared-object modules.

To unload a shared-object module from memory without restarting the database server, you must drop *all* routines that the shared library contains. Use the SQL DROP ROUTINE, DROP FUNCTION, or DROP PROCEDURE statement to unregister a UDR. These statements remove the registration information about the UDR from the system catalog tables.

Removing Routines from the Shared Library

The following conditions cause the database server to remove the shared-object file from the memory map:

- You drop all routines in the module.
- All instances of the routines finish executing.
- You explicitly call **ifx_unload_module**.

Once these conditions are true, the database server automatically unloads the shared-object file from memory. It also puts a message in the log file to indicate that the shared object is unloaded. Once the shared object is unloaded, you can replace the shared-object file on disk and reregister its UDRs in the database.

You can use the **onstat** utility to verify that a module actually was unloaded:

```
onstat -g dll
```

Do not overwrite a shared-object file on disk while it is loaded in memory because you might cause the database server to generate an error when the overwritten module is accessed or unloaded. Use the **ifx_replace_module()** function to replace a loaded shared object file with a new version. For information on the **ifx_replace_module()** function, see the description of Function Expressions within the Expression segment in the *IBM Informix Guide to SQL: Syntax*.

For example, to replace the **circle.so** shared DataBlade API library that resides in the **/usr/apps/opaque_types** directory with one that resides in the **/usr/apps/shared_libs** directory, you can use the EXECUTE FUNCTION statement to execute the **ifx_replace_module()**, as follows:

```
EXECUTE FUNCTION
   ifx_replace_module("/usr/apps/opaque_types/circle.so",
      "/usr/apps/shared_libs/circle.so", "c")
```

The **ifx_replace_module()** function updates the **sysprocedures** system catalog with the new name or location. This functions return one of the following integer values:

- Zero indicates success.
- A negative value indicates an error message number.

You can also execute the **ifx_replace_module()** function in a SELECT statement, as follows:

```
SELECT
   ifx_replace_module("/usr/apps/opaque_types/circle.so",
      "/usr/apps/shared_libs/circle.so", "c")
   FROM customer
   WHERE customer_id = 100
```

If you do not want the shared library replaced multiple times with this SELECT statement, ensure that the SELECT statement returns only one row of values.

E/C

When you execute these functions from within an ESQL/C application, you must associate the EXECUTE FUNCTION statement with a function cursor. For more information on writing ESQL/C applications, refer to the *IBM Informix ESQL/C Programmer's Manual*. ♦

Modifying a Java UDR

To modify a Java UDR, you can use the SQL/J **replace_jar** method. For example, the following command replaces the **.jar** file in the database with a new copy:

```
execute procedure replace_jar(
"file:/d:/informix/extend/Zip.1.0/Zip.jar", "ZipJar");
```

Altering a User-Defined Routine

You can use the ALTER FUNCTION, ALTER PROCEDURE, and ALTER ROUTINE statements to change the routine modifiers or pathname of a previously defined UDR. These statements let you modify characteristics that control how the function executes. You can also add or replace related UDRs that provide alternatives for the optimizer, which can improve performance.

Dropping a User-Defined Routine

You can use the DROP FUNCTION, DROP PROCEDURE, and DROP ROUTINE statements to drop a previously defined UDR. These statements remove the text and executable versions of the routine from the database.

You cannot drop a UDR that is in use by some database function, such as the definition of an opaque data type, a cast, a user-defined aggregate, an operator class, or an access method.

For information on these SQL statements, refer to their description in the *IBM Informix Guide to SQL: Syntax*.

Chapter

Improving UDR Performance

13

In This Chapter

This chapter describes performance considerations for UDRs and includes the following topics:

Optimizing a User-Defined Routine

The query optimizer decides how to perform a query. A *query plan* is a specific way a query might be performed. A query plan includes how to access the table or tables included in the query, the order of joining tables, and the use of temporary tables. The query optimizer finds all feasible query plans. The optimizer estimates the cost to run each plan and then selects the plan with the lowest cost estimate.

Tip: *For more information on query optimization, refer to the "Performance Guide."*

Optimizing an SPL Routine

During SPL optimization, the query optimizer evaluates the possible query plans and selects the query plan with the lowest cost. The database server puts the selected query plan for each SQL statement in an execution plan for the SPL routine. The database server optimizes each SQL statement within the SPL routine and includes the selected query plan in the *execution plan*.

Optimization Levels

The current *optimization level* set in an SPL routine affects how the SPL routine is optimized. The SQL statement, SET OPTIMIZATION, sets the optimization level, which in turn determines the algorithm that the query optimizer uses, as follows.

SET OPTIMIZATION Statement	**Algorithm Used**
SET OPTIMIZATION HIGH	Invokes a sophisticated, cost-based strategy that examines all reasonable query plans and selects the best overall alternative For large joins, this algorithm can incur more overhead than desired. In extreme cases, you can run out of memory.
SET OPTIMIZATION LOW	Invokes a strategy that eliminates unlikely join strategies during the early stages, which reduces the time and resources spent during optimization However, when you specify a low level of optimization, the optimal strategy might not be selected because it was eliminated from consideration during early stages of the algorithm.

For SPL routines that remain unchanged or change only slightly, you might want to set the SET OPTIMIZATION statement to `HIGH` when you create the routine. This optimization level stores the best query plans for the routine. Then set optimization to `LOW` before you execute the routine. The routine then uses the optimal query plans and runs at the more cost-effective rate if reoptimization occurs.

Automatic Optimization

When you create an SPL routine, the database server attempts to optimize the SQL statements within the routine at that time. If the tables cannot be examined at compile time (they might not exist or might not be available), the creation does not fail. In this case, the database server optimizes the SQL statements the first time that the SPL routine executes. The database server stores the optimized execution plan in the **sysprocplan** system catalog table for use by other processes.

The database server uses the dependency list to keep track of changes that would cause reoptimization the next time that an SPL routine executes. The database server reoptimizes an SQL statement the next time that an SPL routine executes after one of the following situations:

- Execution of any data definition language (DDL) statement (such as ALTER TABLE, DROP INDEX, or CREATE INDEX) that might alter the query plan
- Alteration of a table that is linked to another table with a referential constraint (in either direction)
- Execution of UPDATE STATISTICS FOR TABLE for any table involved in the query

  The UPDATE STATISTICS FOR TABLE statement changes the version number of the specified table in **systables.**

The database server updates the **sysprocplan** system catalog table with the reoptimized execution plan.

Updating Statistics for an SPL Routine

The database server stores statistics about the amount and nature of the data in a table in the **systables**, **syscolumns**, and **sysindices** system catalog tables. The statistics that the database server stores include the following information:

- Number of rows
- Maximum and minimum values of columns
- Number of unique values
- Indexes that exist on a table, including the columns and functional values that are part of the index key

The query optimizer uses these statistics to determine the cost of each possible query plan. Run UPDATE STATISTICS to update these values whenever you have made a large number of changes to the table.

The UPDATE STATISTICS statement can have no modifying clauses or several modifying clauses, as in the following statements:

```
UPDATE STATISTICS FOR TABLE tablename
UPDATE STATISTICS FOR ROUTINE routinename
```

Execution of UPDATE STATISTICS affects optimization and changes the system catalog in the following ways:

- No UPDATE STATISTICS statement

  If you do not execute UPDATE STATISTICS after the size or content of a table changes, no SQL statements within the SPL routine are reoptimized. The next time a routine executes, the database server reoptimizes its execution plan if any objects that are referenced in the routine have changed.
- UPDATE STATISTICS

  When you specify no additional clauses, the database server reoptimizes SQL statements in *all* SPL routines and changes the statistics for *all* tables.
- UPDATE STATISTICS FOR TABLE

  When you specify the FOR TABLE clause without a table name, the database server changes the statistics for all tables and does not reoptimize any SQL statements in SPL routines.

- UPDATE STATISTICS FOR TABLE *table name*

  When you specify a table name in the FOR TABLE clause, the database server changes the statistics for the specified table. The database server does not reoptimize any SQL statements in SPL routines.

- UPDATE STATISTICS...

  When you specify one of the following clauses, the database server reoptimizes SQL statements in all SPL routines. The database server does not update the statistics in the system catalog tables.

  - FOR FUNCTION
  - FOR PROCEDURE
  - FOR ROUTINE

- UPDATE STATISTICS... *routine name*

  When you include a routine name in one of the following clauses, the database server reoptimizes SQL statements in the named routine. The database server does not update the statistics in the system catalog tables.

  - FOR FUNCTION *routine name*
  - FOR PROCEDURE *routine name*
  - FOR ROUTINE *routine name*

After the database server reoptimizes SQL statements, it updates the **sysprocplan** system catalog table with the reoptimized execution plan. For more information about **sysprocplan**, refer to the *IBM Informix Guide to SQL: Reference*. For more information about the UPDATE STATISTICS statement, refer to the *IBM Informix Guide to SQL: Syntax*.

Optimizing Functions in SQL Statements

The optimizer by itself cannot evaluate the cost of executing a function in an SQL statement because of the possibility of complex logic, user-defined types, and so on. Because some functions can be expensive to execute, the creator of the function should provide information about the cost and selectivity of the function to help in optimizing the SQL statement.

For example, the following SQL statement includes two functions:

```
SELECT * FROM T WHERE expensive(t1) and cheap(t2);
```

If the cheap() function is less expensive to execute than the expensive() function, the optimizer should place the cheap() function first in the execution plan.

The UDRs discussed in the following sections appear in the WHERE or HAVING clause of an SQL statement. These UDRs return a value of `TRUE` or `FALSE`.

Calculating the Query Plan

The optimizer computes the cost for all possible plans and then chooses the lowest-cost plan. Cost includes the number of disk accesses, the number of network accesses, and the amount of work in memory to access rows and sort data.

Selectivity is also a factor in the total cost. Selectivity is the percentage of rows that pass the filter. The optimizer expresses the selectivity as a number from 0 to 1, which represents the percentage of rows in the table that pass the filter.

The larger the selectivity value, the less likely that a row will disqualify the filter. Therefore, the database server generally evaluates a UDR with a smaller selectivity value before it evaluates a UDR with a larger selectivity value. Similarly, the database server generally evaluates a lower cost UDR before a higher cost one. The ultimate order of UDR filter evaluation depends on a combination of the cost and selectivity of the UDR.

For more information on how the optimizer calculates the query plan, refer to the *Performance Guide*.

Specifying Cost and Selectivity

You can provide the cost and selectivity of the function to the optimizer. The database server uses cost and selectivity together to determine the best path.

To provide the cost and selectivity for a function, include modifiers in the CREATE FUNCTION statement. You can include the cost and selectivity values in the CREATE FUNCTION statement or calculate the values with functions called during the optimization phase.

If you do not specify your own cost and selectivity values for a function, the database server uses a default selectivity of `0.1` and a default cost of `0`. Because the default cost and selectivity are low, the database server considers a UDR with default cost and selectivity inexpensive to execute and will most likely execute that UDR before other UDRs in the WHERE clause.

The database server assigns a **cost** of `0` to all built-in functions, such as SIN and DATE.

Constant Cost and Selectivity Values

The following modifiers specify a cost or selectivity value when you execute the CREATE FUNCTION statement. The cost or selectivity value does not change for each invocation of the function:

- percall_cost=*integer*

  The percall_cost modifier specifies the cost of executing the function once. The *integer* value is a number.
- selconst=*float*

  The selconst modifier specifies the selectivity of a function. The *float value* is a floating-point number between `0` and `1` that represents the fraction of the rows for which you expect the routine to return `TRUE`.

Dynamic Cost and Selectivity Values

In some cases, the cost and selectivity of a function can vary significantly, depending upon the input to the function. If the input can change the optimization, use the following modifiers, which specify a function that computes the cost and selectivity at runtime:

- costfunc=***CostFunction***

  This modifier specifies the name of a function, ***CostFunction***, that the optimizer executes to find the cost of executing your function one time.

- selfunc=***SelectivityFunction***

  This modifier specifies the name of a function, ***SelectivityFunction***, that the optimizer executes to find the selectivity of your function.

You write these cost and selectivity functions to provide the optimizer with enough information about your function to create the best query plan.

The selectivity functions for a UDT might need statistics about the nature of the data in the UDT column. The database server does not generate distributions or maximum and minimum value statistics for a UDT. You need to write and register user-defined statistics functions to generate and store statistics for a UDT in the system catalog tables, in the same locations as statistics stored for built-in data types. For more information about user-defined statistics, refer to "Extending UPDATE STATISTICS" on page 13-13. For information about writing these functions, refer to the *IBM Informix DataBlade API Programmer's Guide*.

Calculating Cost

The cost you specify for a function must be compatible with the cost that the optimizer calculates for other parts of the SQL statement. The following formula is one method to approximate the costing algorithm that the optimizer uses:

1. Execute the following SQL statements from DB-Access, where BIGTABLE is any large table:

   ```
   SET EXPLAIN ON;
   SELECT count(*) from bigtable;
   ```

   Time the query.

2. Let *secost* be the cost the optimizer assigned for the scan. Read the sqexplain.out file to get *secost*.

   For information about **sqexplain.out**, refer to the *Performance Guide*.

3. Let *satime* be the time required to complete the SQL statement.

4. Execute and time your function. Let *facost* be the actual time required to execute the function once.

   The cost of executing the function once can be approximated as follows:

   ```
   ((secost/satime)*facost)
   ```

   Truncate the calculated cost to an integer value.

Selectivity and Cost Examples

The following example creates a function that determines if a point is within a circle. When an SQL statement contains this function, the optimizer executes the function **contains_sel()** to determine the selectivity of the **contains()** function.

```
CREATE FUNCTION contains(c circle, p point)
RETURNING boolean WITH(selfunc=contains_sel)
EXTERNAL NAME "$USERFUNCDIR/circle.so" LANGUAGE C;
```

The following example creates two functions, each with cost and selectivity values:

```
CREATE FUNCTION expensive(cust int)
RETURNING boolean WITH(percall_cost=50,selconst=.1)
EXTERNAL NAME "/ix/9.4/exp_func.so" LANGUAGE c;

CREATE FUNCTION cheap(cust int)
RETURNING boolean WITH(percall_cost=1,selconst=.1)
EXTERNAL NAME "/ix/9.4/exp_func.so" LANGUAGE C;
```

When both of these functions are in one SQL statement, the optimizer executes the **cheap()** function first because of the lower cost. The following SET EXPLAIN output, which lists **cheap()** first in the **Filters:** line, shows that indeed the optimizer did execute **cheap()** first:

```
QUERY:
------
select * from customer
where expensive(customer_num)
and cheap(customer_num)
Estimated Cost: 8
Estimated # of Rows Returned: 1
  1) informix.customer: SEQUENTIAL SCAN
        Filters: (lsuto.cheap(informix.customer.customer_num )AND
lsuto.expensive(informix.customer.customer_num ))
```

For an example of a C function that calculates a cost dynamically, refer to the **\%INFORMIXDIR\dbdk\examples\Types\dapi\Statistics\Box\src\c** directory after you install the DBDK.

Extending UPDATE STATISTICS

The UPDATE STATISTICS statement collects statistics about the data in your database. The optimizer uses these statistics to determine the best path for an SQL statement.

For SQL statements that use UDTs, the optimizer can call custom selectivity and cost functions. (For more information on creating selectivity and cost functions, refer to "Optimizing Functions in SQL Statements" on page 13-8.) Selectivity and cost functions might need to use statistics about the nature of the data in a column. When you create the **statcollect()** function that collects statistics for a UDT, the database server executes this function automatically when a user runs the UPDATE STATISTICS statement with the MEDIUM or HIGH keyword.

Using UPDATE STATISTICS

The syntax of UPDATE STATISTICS is the same for UDTs as for built-in data types. Because the data distributions provide the optimizer with equivalent statistics, the database server does not calculate **colmin** and **colmax** for UDTs.

The **statcollect()** function executes once for every row that the database server scans during UPDATE STATISTICS. The number of rows that the database server scans depends on the mode and the confidence level. Executing UPDATE STATISTICS in HIGH mode causes the database server to scan all rows in the table. In MEDIUM mode the database server chooses the number of rows to scan based on the confidence level. The higher the confidence level, the higher the number of rows that the database server scans. For general information about UPDATE STATISTICS, refer to the *IBM Informix Guide to SQL: Syntax*.

The statistics that the database server collects might require a smart large object for storage. The configuration parameter SBSSPACENAME specifies an sbspace for storing this information. If SBSSPACENAME is not set, the database server might not be able to collect the specified statistics.

Support Functions for UPDATE STATISTICS

The **statcollect()** and **statprint()** functions support the collection of statistics. If you want UPDATE STATISTICS to generate statistics for a UDT, you must create these functions.

The stat Data Type

The statcollect() and statprint() functions use an SQL data type called stat.

C

The corresponding C language structure is called mi_statretval. For an exact description of mi_statretval, see the libmi header file.

Most of the information in mi_statretval is manipulated internally. However, two fields must be filled in by statcollect():

- The statdata field should contain the histogram for the distribution. UDTs are stored in a multirepresentational format.
- The szind field should be set to either MI_MULTIREP_SMALL or MI_MULTIREP_LARGE. ♦

The statcollect() Function

When you run UPDATE STATISTICS, the database server calls the appropriate **statcollect()** function for each column that the database server scans.

The **statcollect()** function takes four arguments:

- The first argument is of the same data type as the UDT for which the statcollect() function is called.

  The database server uses this argument to resolve the function and to pass in values.

  The first time the database server invokes this function, it sets this parameter to null. On subsequent invocations, this argument contains the column value.
- The second argument is a double-precision value that indicates the number of rows that the database server must scan to gather the statistics.

- The third argument is a double-precision value that is the resolution specified by the UPDATE STATISTICS statement. The resolution value specifies the bucket size for the distribution. However, you might choose to ignore this parameter if it does not make sense for your UDT.
- The fourth argument is an MI_FPARAM structure that the database server uses to pass information to the UDR as well as a place to store state information.

On the first call to **statcollect()**, MI_FPARAM contains a SET_INIT value. Check for this value in statcollect() and perform any initialization operations, such as allocating memory and initializing values.

On subsequent calls to **statcollect()**, MI_FPARAM contains a SET_RETONE value. At this point, **statcollect()** should read the column value from the first argument and place it in your distribution structure.

After all rows have been processed, the last call to **statcollect()** puts a value of SET_END in MI_FPARAM. For this final call, **statcollect()** should put the statistics in the **stat** data type and perform any memory deallocation.

You must declare the **statcollect()** function with HANDLESNULLS, but the function itself can ignore nulls if desired.

Allocate any memory used across multiple invocations of **statcollect()** from the PER_COMMAND pool and free it as soon as possible. Any memory not used across multiple invocations of **statcollect()** should be allocated from the PER_ROUTINE pool.

The statprint() Function

The **statprint()** function converts the statistics data that the **statcollect()** function collects to an LVARCHAR value that the database server can use to display information. The **dbschema** utility executes the **statprint()** function.

The **statprint()** function has two arguments. The first argument is a dummy argument of the required data type. The database server uses this argument to resolve the function. The first time the database server executes this function, it sets the first parameter to null.

The second argument is a value of the **stat** data type. The **stat** data type is a multirepresentational data type that the database server uses to store data that the **statcollect()** function collects.

The **statprint()** function must take the histogram, which is stored in multi-representational form, and convert it to a printable form.

After you register the functions, make sure those with DBA privilege or the table owner can execute the **statcollect()** and **statprint()** UDRs.

Example of User-Defined Statistics Functions

For examples of **statprint()** and **statcollect()** functions written in C, refer to the **\%INFORMIXDIR\dbdk\examples\Types\dapi\Statistics\Box\src\c** directory, after you install the DataBlade Developer's Kit.

Using Negator Functions

A *negator function* takes the same arguments as its companion function, in the same order, but returns the Boolean complement. That is, if a function returns `TRUE` for a given set of arguments, its negator function returns `FALSE` when passed the same arguments, in the same order. In certain cases, the database server can process a query more efficiently if the sense of the query is reversed; that is, if the query is, "Is x greater than y?" instead of, "Is y less than or equal to x?"

The NEGATOR modifier of the CREATE FUNCTION statement names a companion function, a negator function, to the current function. When you provide a negator function, the optimizer can use a negator function instead of the function you specify when it is more efficient to do so. If a function has a negator function, any user who executes the function must have the Execute privilege on both the function and its negator. In addition, a function must have the same owner as its negator function.

You can write negator functions in SPL, C, or Java. The following example shows CREATE FUNCTION statements that specify negator functions:

```
CREATE ROW TYPE complex(real FLOAT, imag FLOAT);

CREATE FUNCTION equal (c1 complex, c2 complex)
   RETURNING BOOLEAN WITH (NEGATOR = notequal)
   DEFINE a BOOLEAN;
   IF (c1.real = c2.real) AND (c1.imag = c2.imag) THEN
      LET a = 't';
   ELSE
      LET a = 'f';
   END IF;
   RETURN a;
END FUNCTION;


CREATE FUNCTION notequal (c1 complex, c2 complex)
   RETURNING BOOLEAN WITH (NEGATOR = equal)
   DEFINE a BOOLEAN;
   IF (c1.real != c2.real) OR (c1.imag != c2.imag) THEN
      LET a = 't';
   ELSE
      LET a = 'f';
   END IF;
   RETURN a;
END FUNCTION;
```

Using a Virtual-Processor Class

A *virtual process* is a process that the database server uses to execute queries and perform other tasks, such as disk I/O and network management. A small number of virtual processors (VPs) can carry out tasks on behalf of many client applications because the database server breaks the client-application requests into pieces called *threads*. The VP can schedule the individual threads internally for processing. Therefore, VPs are multithreaded processes because they can run multiple concurrent threads.

The database server implements its own threads to schedule client-application requests. These threads are not the same as operating-system threads, which multithreaded operating systems provide.

Virtual processors are grouped into virtual-processor classes, or *VP classes*. All VPs in a particular VP class handle the same type of processing. The database server supports the following VP classes.

Virtual-Processor Class	Description
CPU	Central processing (the primary VP class, which controls client-application requests)
AIO	Asynchronous disk I/O
SHM	Shared-memory network communication
JVP	Special VP class for execution of Java UDRs
User-defined	Special VP classes for additional types of processing

For general information about virtual processors, see the *Administrator's Guide*.

Choosing a Virtual-Processor Class

The database server supports the following classes of virtual processors for the execution of a UDR.

Virtual-Processor Class	Description
CPU VP	Required VP for execution of SPL routines Default VP for execution of C UDRs. A UDR must be well behaved to run in the CPU VP.
User-defined VP	VP for execution of C UDR that has some ill-behaved characteristics
JVP	VP for execution of Java UDR This VP class contains the Java Virtual Machine (Java VM).

The database server defines the CPU VP and the JVP classes.

CPU Virtual-Processor Class

The CPU virtual-processor class is the primary VP class of the database server. It runs the following kinds of threads:

- All session threads

  Session threads process requests from the SQL client applications.
- Some internal threads

  Internal threads perform services internal to the database server.

The CPU VP class is the default VP class for a UDR. You do not need to specify the CLASS routine modifier in the CREATE FUNCTION or CREATE PROCEDURE statement to have the UDR execute in the CPU VP class.

SPL

SPL routines must *always* run in the CPU VP. Therefore, you do not need to specify the CLASS routine modifier for an SPL routine. The following CREATE FUNCTION statement registers the **getTotal()** SPL routine, which runs in the CPU VP:

```
CREATE FUNCTION getTotal(order_num, state_code)
   RETURNS MONEY
...
END FUNCTION
```

You *cannot* run an SPL routine in a user-defined VP. ♦

C

By default, a C UDR runs in the CPU VP class. Generally, UDRs perform best in the CPU VP class because threads do not have to migrate among operating-system processes during query execution. However, to run in the CPU VP, the C UDR must be *well behaved*; that is, it must adhere to the following programming requirements:

- Preserves concurrency of the CPU VP:
  - Yields the CPU VP for intense calculations
  - Does not perform blocking operating-system calls
- Is thread safe:
  - Does not modify static or global data
  - Does not allocate local resources
  - Does not modify the global VP state
- Does not make unsafe operating-system calls

Important: *Use the CPU VP with caution. If a UDR contains errors or does not adhere to these guidelines, this routine might affect the normal processing of other user queries.*

You can relax some of these programming requirements if you run your C UDR in a user-defined VP class. For more information, see "User-Defined Virtual-Processor Class" on page 13-20. ♦

C

User-Defined Virtual-Processor Class

For routines written in C, you can designate a user-defined class of virtual processors, called user-defined VPs, to run the routine.

Use of user-defined VPs can result in lower performance because queries normally execute in the CPU VP, and the query thread must migrate to the user-defined VP to evaluate external routines.

Java

JVM Virtual-Processor Class

Java routines *always* run in a Java VP. When you register a Java, you can specify the following CLASS routine modifier for legibility, but it is not required:

```
CLASS = jvp
```

C

Using Virtual Processors with UDRs Written in C

To run in the CPU VP class, a C UDR must be well behaved; that is, it must adhere to special programming requirements. Running in a user-defined VP relaxes some, but not all, of the programming requirements of a well-behaved routine. For example, these routines can issue direct file-system calls that block further processing by the virtual processor until the I/O is complete. Because virtual processors are not CPU virtual processors, however, the normal processing of user queries is not affected. However, they still cannot perform local resource allocations because they might migrate among the VPs.

Tip: *The DataBlade Developers corner of the IBM Informix Developer Zone (www.ibm.com/software/data/developer/informix) has a detailed article about data safety when using operating-system functions with user-defined VPs.*

To assign a C UDR to a user-defined VP class

1. When you register an external function or procedure, assign it to a class of virtual processors with the CLASS routine modifier of the CREATE FUNCTION or CREATE PROCEDURE statement.

   The CLASS routine modifier specifies the virtual-processor class with the following syntax:

   ```
   CLASS = vpclass_name
   ```

   In this syntax, *vpclass_name* is the name of the user-defined VP class that you have configured in the database server. The class name is not case sensitive.
2. Configure new user-defined virtual-processor classes in the ONCONFIG file with the VPCLASS configuration parameter.

   The following sample ONCONFIG entry creates the user-defined VP class **newvp**:

   ```
   VPCLASS newvp,num=3  # New VP class for testing
   ```

   The **num** option specifies the number of virtual processors that the database server starts. For the **newvp** virtual-processor class, the database server initially starts three virtual processors.

The VP class need not exist when the routine is registered. However, when you execute the routine, the class must exist and have virtual processors assigned to it. If the class does not have any virtual processors, you receive an SQL error.

For more information on how to choose a virtual-processor class for a C UDR, see the *IBM Informix DataBlade API Programmer's Guide*. For information on the VPCLASS configuration parameter, see the *Administrator's Reference*.

Managing Virtual Processors

You can use the **onmode** and **onstat** utilities to manage virtual processors. For additional information about **onmode** and **onstat**, refer to the *Administrator's Reference*.

Adding and Dropping Virtual Processors

You can add or drop virtual processors in a user-defined VP class or in the CPU VP class while the database server is online. Use **onmode -p** to add virtual processors to the class. For example, the following command adds two virtual processors to the **newvp** class:

```
onmode -p +2 newvp
```

Monitoring Virtual-Processor Classes

You can monitor VPs with the **onstat** utility. The **-g glo** option prints information about global multithreading such as CPU use of virtual processors and total number of sessions. A user-defined VP class appears in the **onstat -g glo** output as a new process.

Parallel UDRs

The parallel database query (PDQ) feature executes a single query with multiple threads in parallel. Another feature, table fragmentation, allows you to store the parts of a table on different disks. PDQ delivers maximum performance benefits when the data that is being queried is in fragmented tables.

PDQ features allow the database server to distribute the work for one aspect of an SQL statement among several processors. For example, if an SQL statement requires a scan of several parts of a table that reside on different disks, multiple scans can occur simultaneously.

A PDQ is a query that the database server processes with PDQ techniques when the optimizer chooses parallel execution. When the database server processes a query with PDQ, it first divides the query into subplans. The database server then allocates the subplans to a number of threads that process the subplans in parallel. Because each subplan represents a smaller amount of processing time when compared to the original query and because each subplan is processed simultaneously with all other subplans, the database server can drastically reduce the time that is required to process the query.

For more information on the PDQ feature, refer to the *Administrator's Guide*. For more information on the performance implications of PDQ, refer to the *Performance Guide*.

Executing UDRs in Parallel

The database server can execute the following UDRs in parallel if they are part of a PDQ and PDQPRIORITY is turned on:

- C UDRs that call only DataBlade API functions that are PDQ thread-safe can execute in parallel.
- Java UDRs that call only DataBlade API functions that are PDQ thread-safe can execute in parallel.

  For more information, refer to "Writing PDQ Thread-Safe UDRs" on page 13-31.
- Built-in function UDR

  Examples of built-in function UDRs include overloaded operators for UDTs, such as the following operators that are used for a generic B-tree index:
  - **lessthan** (<)
  - **lessthanorequal** (<=)
  - **equal** (=)
  - **greaterthanorequal** (>=)
  - **greaterthan** (>)

UDRs can execute in parallel in the following situations if they are part of a PDQ and PDQPRIORITY is turned on:

- A UDR used as an expression in a query
- DataBlade API FastPath executing a UDR
- Implicit UDR execution when evaluating a user-defined aggregate on a column of a user-defined type
- Implicit UDR execution for overloading of comparison operator
- Assign UDR executed implicitly
- Comparison UDR execution for sort
- A UDR that a generic B-tree index executes

A UDR cannot execute the following SQL statements in parallel:

- Singleton execution with the EXECUTE FUNCTION statement in either DB-Access or ESQL/C
- INSERT INTO *tablename* EXECUTE *udr()*
- FOREACH EXECUTE *udr()* END FOREACH
- OPEN CURSOR EXECUTE *udr()*
- Remote UDR execution

Execution of a UDR in a Query Expression

One way to execute UDRs is as an expression in a query. You can take advantage of parallel execution if the UDR is in an expression in one of the following parts of a query:

- WHERE clause
- SELECT list
- GROUP BY list
- Overloaded comparison operator
- User-defined aggregate
- HAVING clause
- Select list for parallel insertion statement

- Generic B-tree index scan on multiple index fragments provided that the compare function used in the B-tree index scan is parallelizable
- Virtual Table Interface (VTI) or Virtual Index Interface (VII) fragments, provided that all **am_purpose** functions for the VTI or VII are all parallelizable

Parallel UDR in WHERE Clause

The following example is a typical PDQ that contains two UDRs:

```
SELECT c_udr1(tabid) FROM tab
   WHERE tabname = c_udr2(3) AND
      tabid > 100;
```

If the table **tab** has multiple fragments and the optimizer decides to run the select statement in parallel, the following operations can execute in parallel:

- The scan of table **tab** is performed by multiple scan threads in parallel. Each scan thread fetches a row from a fragment of **tab**.
- Each scan thread also evaluates the WHERE condition in parallel. Each scan thread executes the UDR **c_udr2()** in parallel.
- Each scan thread also executes the UDR **c_udr1()** in the select list in parallel.

Parallel UDR in a Join

The following sample query contains a join between table **t1** and **t2**:

```
SELECT t1.x, t2.y
   FROM t1, t2
   WHERE t1.x = t2.y AND
      c_udr(t1.z, t2.z, 3) > 5 AND
      c_udr1(t1.u) > 4 AND
      c_udr2(t2.u) < 5;
```

If the tables **t1** and **t2** have multiple fragments and the optimizer decides to run the select statement in parallel, the following operations can execute in parallel:

- The scan of table **t1** is performed by multiple scan threads in parallel. Each scan thread fetches a row from a fragment of **t1** and executes the UDR **c_udr1()** in parallel.
- The scan of table **t2** is performed by multiple scan threads in parallel. Each scan thread fetches a row from a fragment of **t2** and executes the UDR **c_udr2()** in parallel.
- The join of tables **t1** and **t2** is performed by multiple join threads in parallel. Each join thread fetches a row from a fragment of **t2** and executes the UDR **c_udr()** in parallel.

Parallel UDR in the Select List

If you use a UDR in the select list and do not specify a WHERE clause, the UDR can execute in parallel if any of the following conditions are true:

- The GROUP BY clause is specified in the query.
- The ORDER BY clause is specified in the query.
- An aggregate such as MIN, MAX, AVG is specified in the query.
- The query is a parallel INSERT...SELECT statement.
- The query is a SELECT...INTO statement.

The next section shows a sample query with a UDR in the select list and no WHERE clause.

Parallel UDR with GROUP BY

The following sample query contains a GROUP BY clause. This sample query has a UDR in the select list and no WHERE clause.

```
SELECT c_udr1(tabid), COUNT(*)
   FROM t1 GROUP BY 1;
```

If the optimizer decides to run the SELECT statement in parallel, the following operations can execute in parallel:

- The scan of table **t1** is performed by multiple scan threads in parallel. The table **t1** has multiple fragments. Each scan thread fetches a row from a fragment of **t1**.
- Multiple threads execute the UDR **c_udr2()** in parallel to process the GROUP BY clause. If table **t1** is unfragmented, the GROUP BY operation can still execute in parallel even though the scan operation does not execute in parallel.

Parallel UDR in Select List for Parallel Insert

The following sample query is a parallel INSERT statement. Suppose you create an opaque data type **circle**, create a table **cir_t** that defines a column of type circle, create a UDR **area**, and then execute the following sample query:

```
INSERT INTO cirt_t_target
SELECT circle, area(circle)
   FROM cir_t
   WHERE circle > "(6,2,4)";
```

In this query, the following operations can execute in parallel:

- The expression `circle > "(6,2,4)"` in the WHERE clause

  If the table **cir_t** is fragmented, multiple scans of the table can execute in parallel, one scan on each fragment. Then multiple > comparison operators can execute in parallel, one operator per fragment.
- The UDR **area(circle)** in the select list

  If the table **cir_t** has multiple fragments, multiple **area** UDRs can execute in parallel, one UDR on each fragment.
- The INSERT into **cir_t_target**

  If the table **cir_t_target** has multiple fragments, multiple INSERT statements can execute in parallel, one on each fragment.

C

FastPath Execution of a UDR in a DataBlade API

A C UDR can use the following DataBlade API calls to invoke a UDR directly:

- **mi_routine_get()**
- **mi_routine_exec()**

DataBlade API FastPath execution of a UDR executes in parallel as long as the UDR is parallelizable and calls only DataBlade API functions that are PDQ thread safe.

Implicit UDR Execution of a User-Defined Aggregate

A user-defined aggregate (UDA) can execute in parallel as long as the UDR is parallelizable and calls only DataBlade API functions that are PDQ thread safe.

For example, suppose you create a UDA named **uda** and use it in the following SQL query:

```
SELECT grp, uda(udt_col) FROM tab GROUP BY grp;
```

If the data type of column **udt_col** is a UDT whose aggregation requires a UDR call, the following operations can execute in parallel:

- Each group thread executes the aggregation UDR **uda** in parallel.
- If the GROUP BY column **grp** is a UDT column, the **equal()** UDR function on the UDT column executes in parallel by the scan thread for the hash repartitioning on the GROUP BY keys.
- If the table **tab** is fragmented, multiple scan threads can read the table in parallel.

Implicit UDR Execution of a Comparison Operator

When you create opaque data types, you can create overloaded routines for comparison operators such as **equal** (=) or **greaterthanorequal** (>=).

The following sample query selects rows using a filter on the UDT column:

```
SELECT * FROM tab WHERE udt_col = "xyz";
```

The database server converts the comparison operator = to call the **equal** UDR on the **udt_col** column. If the table **tab** is fragmented, the following operations can execute in parallel:

- Multiple scans of the table can execute in parallel, one scan on each fragment.
- Multiple = comparison operators can execute in parallel, one operator per fragment of table **tab**.

Implicit Execution of an Assign UDR

When you create opaque data types, you create the support function **assign()** to insert, update, or load the UDT data in the table.

The following sample SQL statement inserts data in a UDT column:

```
INSERT INTO tab (udtcol) SELECT udtcol FROM t1;
```

If the table **tab** has multiple fragments and the **udtcol** data type has an **assign()** function, each insert thread that inserts a fragment of table **tab** executes the **assign()** UDR in parallel.

The support function **destroy()** for a UDT does not execute in parallel because the **destroy()** UDR is called during a DELETE statement that is not executed in parallel.

Execution of a Comparison UDR for Sort

When you create opaque data types, you create the support function **compare()** to sort the UDT data during ORDER BY, UNIQUE, DISTINCT, and UNION clauses and CREATE INDEX operations.

```
SELECT udtcol FROM t ORDER BY 1;
```

If the **udtcol** column has a comparison UDR that is parallelizable and the client enables parallel sort, each sort thread participating in the parallel sort for the ORDER BY clause executes the comparison UDR in parallel.

Execution of a UDR by an Index on a UDT column

The database server supports indexing on a UDT column. Therefore, index build, search, and recovery require execution of UDRs that operate on UDT columns.

Currently, the database server does not support fragmentation by expression on UDT columns. As a result, the index built on a UDT column by the database server is not fragmented because index fragmentation makes sense only if the fragmentation is based on expression.

Enabling Parallel UDRs

By default, a UDR does not execute in parallel. To enable parallel execution of UDRs, you must take the following actions:

- Specify the PARALLELIZABLE modifier in the CREATE FUNCTION or ALTER FUNCTION statement.
- Ensure that the UDR does not call non-PDQ thread-safe functions.
- Turn on PDQ.
- Use the UDR in a PDQ.

Specifying the PARALLELIZABLE Modifier

When you register a UDR, you must specify the PARALLELIZABLE modifier in the CREATE FUNCTION or ALTER FUNCTION statement. However, an SPL routine is not parallelizable even if it is declared as parallelizable.

Writing PDQ Thread-Safe UDRs

External-language UDRs can execute in parallel as long as they are PDQ thread-safe DataBlade API functions.

The following DataBlade API function categories are PDQ thread safe:

- Data handling

  Exception in this category: collection manipulation functions (**mi_collection_***) are not PDQ thread safe.
- Session, thread, and transaction management
- Function execution
- Memory management
- Exception handling
- Callbacks
- Miscellaneous

If an external-language UDR calls a non-PDQ thread-safe function that was created with the PARALLELIZABLE modifier, the database server aborts the query and issues the following error message:

```
-7422 Can not issue DAPI function %s in a secondary
PDQ thread.
```

The database server substitutes the name of the DataBlade API function for the %s string in this error message.

Turning On PDQ and Reviewing Other Configuration Parameters

Parallel execution of queries is turned off by default. To turn on parallel execution, use one of the following methods:

- Set the environment variable **PDQPRIORITY** greater than 0.
- Execute the SQL statement SET PDQPRIORITY.

The PDQ configuration parameters have the same effect on parallel UDRs as on regular PDQ queries. For example, the DS_MAX_SCANS parameter specifies the maximum number of scan threads that the database server can execute concurrently.

For information on how to tune the PDQ configuration parameters, refer to the *Performance Guide*.

Step-By-Step Procedure to Enable Parallel UDRs

The following procedure includes examples for the tasks described in the previous sections.

To enable parallel UDRs

1. Create a fragmented table and load data into it.

   For example, the following SQL statement creates a fragmented table:

   ```
   CREATE TABLE natural_number (x integer)
      FRAGMENT BY round robin
      IN dbspace1, dbspace2;
   ```

2. Write a function that is PDQ thread safe.

   C

   For example, the following C prototype shows a function that takes an integer and determines if it is a prime number:

   ```
   mi_boolean is_prime_number (x mi_integer);
   ```

   ♦

   For more information on how to write PDQ thread-safe functions, refer to "Writing PDQ Thread-Safe UDRs" on page 13-31.

3. Register the function as an external UDR and specify the PARALLELIZABLE keyword.

   For example, the following SQL statement registers the **is_prime_number** UDR:

   ```
   CREATE FUNCTION is_prime_number (x integer)
      RETURNS boolean
      WITH (parallelizable)
      EXTERNAL NAME "$USERFUNCDIR/math.udr"
      LANGUAGE C;
   ```

4. Turn on PDQ and execute the UDR in a query.

   The following sample SQL statements turn on PDQ and execute the UDR in a query:

   ```
   SET PDQPRIORITY 100;
   SELECT x FROM natural_number
      WHERE is_prime_number(x)
      ORDER BY x;
   ```

   The database server scans each fragment of the table **natural_number** with multiple scan threads executing in parallel. Each scan thread executes the UDR **is_prime_number()** in parallel.

Setting the Number of Virtual Processors

The dynamic, multithreaded nature of a virtual processor allows it to perform parallel processing. Virtual processors of the CPU class can run multiple session threads, working in parallel, for an SQL statement contained within a UDR.

You can increase the number of CPU virtual processors with the VPCLASS configuration parameter in the ONCONFIG file. For example, the following parameter specifies that the database server should start four virtual processors for the **cpu** class:

```
VPCLASS cpu,num=4
```

Tip: *Debugging is more difficult when you have more than one CPU because threads can migrate between processes. The database server communication mechanism uses the SIGUSR1 signal. When you are debugging, you must avoid SIGUSR1 to prevent database server processes from hanging.*

Java

On Windows, all virtual processors share the same process space. Therefore, you do not need to start multiple instances of Java VPs to execute Java UDRs in parallel. On UNIX, the database server must have multiple instances of JVPs to parallelize Java UDR calls. Because the Java Virtual Machines that are embedded in different VPs do not share states, you *cannot* store global states with Java class variables. All global states *must* be stored in the database to be consistent. ♦

Monitoring Parallel UDRs

When PDQ is turned on, the SET EXPLAIN output shows whether the optimizer chose to execute a query in parallel. If the optimizer chose parallel scans, the output shows `PARALLEL`. If PDQ is turned off, the output shows `SERIAL`.

You can monitor the parallel execution of PDQs and parallel UDRs with the following options of the **onstat** utility:

- **onstat -g ath**

  This option shows the threads currently executing for each session. Each session has a primary (**sqlexec)** thread. If the query is executing in parallel, **onstat -g ath** shows secondary threads, such as **scan** and **sort**.

- **onstat -g mem**

  This option shows pool sizes allocated to sessions. This output can provide hints about how much memory the UDR uses.

- **onstat -g ses**

  This option shows the number of threads allocated and the amount of memory used for each session. This output can also provide hints about how much memory the UDR uses.

For more information on interpreting the output of these **onstat** options, refer to the *Performance Guide*.

Memory Considerations

As you create a UDR, consider ways to minimize its memory usage. This section describes the following memory considerations for UDRs:

- Memory durations for external routines
- Stack-size considerations for external routines
- The virtual-memory cache for SPL and external routines

C

Memory Durations for C UDRs

Because a C UDR executes in the memory space of the database server, its dynamic memory allocations can increase the memory usage of the database server. For this reason, it is very important that a UDR release its dynamically allocated memory as soon as it no longer needs to access this memory.

To help ensure that unneeded memory is freed, the database server associates a memory duration with memory allocation made from its shared memory. The database server automatically reclaims shared memory based on its memory duration.

To provide a duration that is safe for return values, use a memory duration that lasts the life of an SQL statement. It is recommended that you use the following memory duration instead of the PER_STATEMENT duration:

- PER_STMT_EXEC

  The PER_STMT_EXEC memory duration helps improve overall database server performance because it does not hold memory as long as the PER_STATEMENT duration.
- PER_STMT_PREP

  Use the PER_STMT_PREP memory duration when you want memory to be held for the life of a prepared statement.

For more information on these memory durations and using **onstat** utility options to monitor memory usage of C UDRs, refer to the *IBM Informix DataBlade API Programmer's Guide*.

EXT

Stack-Size Considerations

The database server allocates local storage in external routines from shared memory. This local storage is called the thread *stack*. The stack has a fixed length. Therefore, an external routine must not consume excessive stack space, either through large local-variable declarations or through excessively long call chains or recursion.

Warning: *An external routine that overruns the shared-memory region allocated for its stack overwrites adjacent shared memory, with unpredictable and probably undesirable results.*

In addition, any nonstack storage that a thread allocates must be in shared memory. Otherwise, the memory is not visible when the thread moves from one VP to another.

The routine manager of the database server guarantees that a large stack region is available to a thread *before* it calls a user-defined function, so stack exhaustion is generally not a problem.

C

For C UDRs, you can dynamically allocate stack space. In addition, the DataBlade API provides memory-management routines that allocate space from shared memory rather than from process-private memory. If you use the DataBlade API, memory visibility is not a problem. ♦

By default, the routine manager allocates a stack size for a UDR with the size that the STACKSIZE configuration parameter specifies. If STACKSIZE is not set, the routine manager uses a default stack size of 32 kilobytes. To determine how much stack space a UDR requires, monitor the UDR from the system prompt with the following **onstat** utility:

```
onstat -g sts
```

Use the **onstat -g sts** option to obtain information on stack-size use for each thread. The output includes the following fields:

- The thread ID
- The maximum stack size configured for each thread
- The maximum stack size that the thread uses

You can use the output of the threads that belong to user sessions to determine if you need to alter the maximum stack size configured for a user session. To alter the maximum stack size for all user sessions, change the value of the STACKSIZE configuration parameter. To alter the maximum stack size for a single user session, change the value of the **INFORMIXSTACKSIZE** environment variable. For more information, see the configuration parameter STAGEBLOB in the *Administrator's Reference* and the environment variable **INFORMIXSTACKSIZE** in the *IBM Informix Guide to SQL: Reference*.

For more information on the **onstat** utility and the **-g sts** option, see the *Administrator's Reference*.

If the stack size is not sufficient for your UDR, you can specify its stack size with the STACK routine modifier in the WITH clause of the CREATE FUNCTION or CREATE PROCEDURE statement. When you specify a stack size for a UDR, the database server allocates the specified amount of memory for *every* routine invocation of the routine. If a routine does not need a larger stack, do *not* specify a stack size.

Virtual-Memory Cache for Routines

The database server caches the following items in the virtual portion of the database server shared memory:

- For SPL routines and other UDRs, information in the **sysprocedures** system catalog table
- For SPL routines only, the executable form of the routine in the UDR cache

The sysprocedures System Catalog Table

When any session requires the use of an SPL routine for the first time, the database server reads the **sysprocedures** system catalog tables and stores the information in the buffer pool in shared memory. The database server uses this information in shared memory if it is present for subsequent sessions that invoke the UDR.

The database server keeps the **sysprocedures** system catalog information in the buffer pool on a *most recently used* basis.

The **sysprocedures** table includes the following information:

- Name of routine
- Compiled size (in bytes) of return values
- Compiled size (in bytes) of p-code for the routine
- Number of arguments
- Data types of parameters
- Type of routine (function or procedure)
- Location of external routine
- Virtual-processor class in which the routine executes

UDR Cache

When any session requires the use of an SPL routine for the first time, the database server reads the system catalog tables to retrieve the code for the SPL routine. The database server converts the p-code to an executable form. The database server caches this executable form of the SPL routine in the virtual portion of shared memory.

The database server keeps the executable format of an SPL routine in the UDR cache on a *most recently used* basis.

You can monitor the UDR cache with the **-g prc** option of the **onstat** utility. For more information on **onstat -g prc** and adjusting the size of the UDR cache with the PC_POOLSIZE configuration parameter, refer to the *Performance Guide*.

I/O Considerations

The database server stores UDRs and triggers in the following system catalog tables:

- **sysprocbody**
- **sysprocedures**
- **sysprocplan**
- **sysprocauth**
- **systrigbody**
- **systriggers**

These system catalog tables can grow large with heavy use of UDRs in a database. You can tune the key system catalog tables as you would any heavily utilized data tables. To improve performance, use the following methods:

- Isolate system catalog tables.
- Balance the I/O activities.

Isolating System Catalog Tables

If your database server has multiple physical disks available, you can isolate your system catalog tables on a single device and place the tables for your application in a separate dbspace that resides on a different device. This separation reduces contention for the same device.

Balancing the I/O Activities

If you have a large number of UDRs that span multiple extents, you can spread the system catalog tables across separate physical devices (chunks) within the same dbspace to balance the I/O activities.

To spread user-defined routine catalogs across devices

1. Create the dbspace for the UDR system catalog tables with several chunks. Create each chunk for the dbspace on a separate disk.
2. Use the CREATE DATABASE statement with the IN dbspace clause to isolate the system catalog tables in their own dbspace.
3. Load approximately half of your UDRs with the CREATE PROCEDURE or CREATE FUNCTION statement.
4. Create a temporary table in the dbspace with an extent size large enough to use the remainder of the disk space in the first chunk.
5. Load the remainder of the UDRs. The last half of the routines should spill into the second chunk.
6. Drop the temporary table.

Notices

Trademarks

Index