Extending Informix Dynamic Server.2000
# Table of Contents

**Introduction**
- In This Introduction ........................................ 3
- About This Manual ............................................ 3
  - Types of Users ............................................. 3
  - Software Dependencies ..................................... 4
  - Assumptions About Your Locale. .......................... 4
  - Demonstration Databases .................................. 4
- New Features ................................................... 5
  - Extensibility Enhancements ................................. 5
  - Performance Improvements ................................ 5
- Documentation Conventions .................................. 6
  - Typographical Conventions ................................. 6
  - Icon Conventions ........................................... 7
  - Sample-Code Conventions .................................. 10
- Additional Documentation .................................... 10
  - On-Line Manuals ........................................... 11
  - Printed Manuals ........................................... 11
  - On-Line Help ............................................... 11
  - Error Message Documentation .............................. 11
  - Documentation Notes, Release Notes, Machine Notes ... 12
  - Related Reading ............................................ 13
- Compliance with Industry Standards ....................... 14
- Informix Welcomes Your Comments ......................... 14
Chapter 1  
Extending the Database Server

In This Chapter ...................................... 1-3
Creating User-Defined Routines .................. 1-3
   End-User Routines .............................. 1-4
   Routines That Extend Operations ............. 1-5
Extending the Built-In Data Types ............... 1-6
   Building Data Types on Existing Data Types 1-6
   Extending Operators .......................... 1-6
   Building Opaque Data Types ................ 1-7
Routine Management ................................ 1-8

Chapter 2  
Creating a User-Defined Routine

In This Chapter ...................................... 2-3
User-Defined Routines ............................ 2-3
   SPL Routines ................................ 2-5
   External Routines ............................ 2-6
Tasks That You Can Perform with UDRs .......... 2-7
   Extending Data Type Support ............... 2-8
   Creating an End-User Routine .............. 2-19
Using a User-Defined Routine .................... 2-22
   Invoking a User-Defined Routine ........ 2-22
   Creating a User-Defined Routine .......... 2-23
   Managing User-Defined Routines .......... 2-24
   Optimizing a User-Defined Routine ........ 2-25

Chapter 3  
Running a User-Defined Routine

In This Chapter ...................................... 3-3
Invoking a User-Defined Routine ............... 3-3
   Invoking a UDR in an SQL Statement ....... 3-3
   Invoking a UDR in an SPL Routine .......... 3-6
Executing a User-Defined Routine ............... 3-7
   Parsing the SQL Statement ............... 3-7
   Optimizing the SQL Statement ............ 3-8
   Executing an External Routine .......... 3-8
   Executing an SPL Routine ................. 3-11
Understanding Routine Resolution ............... 3-11
   The Routine Signature ..................... 3-12
   Overloading Routines ...................... 3-13
   The Routine-Resolution Process .......... 3-17
   Routine Resolution with User-Defined Data Types 3-22
   Null Arguments in Overloaded Routines ..... 3-27
## Chapter 4  
**Developing a User-Defined Routine**

<table>
<thead>
<tr>
<th>In This Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Planning the Routine</td>
<td>4-3</td>
</tr>
<tr>
<td>Naming the Routine</td>
<td>4-4</td>
</tr>
<tr>
<td>Defining Routine Parameters</td>
<td>4-5</td>
</tr>
<tr>
<td>Returning a Value</td>
<td>4-6</td>
</tr>
<tr>
<td>Returning Values with an Iterator Function</td>
<td>4-11</td>
</tr>
<tr>
<td>Adhering to Coding Standards</td>
<td>4-13</td>
</tr>
<tr>
<td>Writing the Routine</td>
<td>4-14</td>
</tr>
<tr>
<td>Creating a Shared-Object File</td>
<td>4-14</td>
</tr>
<tr>
<td>Registering a User-Defined Routine</td>
<td>4-15</td>
</tr>
<tr>
<td>Setting Privileges for a Routine</td>
<td>4-15</td>
</tr>
<tr>
<td>Registering an SPL Routine</td>
<td>4-17</td>
</tr>
<tr>
<td>Registering an External Routine</td>
<td>4-20</td>
</tr>
<tr>
<td>Reviewing Information about User-Defined Routines</td>
<td>4-26</td>
</tr>
</tbody>
</table>

## Chapter 5  
**Extending Data Types**

<table>
<thead>
<tr>
<th>In This Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Understanding the Data Type System</td>
<td>5-3</td>
</tr>
<tr>
<td>Understanding Data Types</td>
<td>5-5</td>
</tr>
<tr>
<td>Built-In Data Types</td>
<td>5-6</td>
</tr>
<tr>
<td>Extended Data Types</td>
<td>5-8</td>
</tr>
<tr>
<td>Extending the Data Type System</td>
<td>5-13</td>
</tr>
<tr>
<td>Operations</td>
<td>5-13</td>
</tr>
<tr>
<td>Casts</td>
<td>5-14</td>
</tr>
<tr>
<td>Operator Classes</td>
<td>5-14</td>
</tr>
<tr>
<td>Optimizer Information</td>
<td>5-15</td>
</tr>
</tbody>
</table>

## Chapter 6  
**Extending an Operation**

<table>
<thead>
<tr>
<th>In This Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operators and Operator Functions</td>
<td>6-3</td>
</tr>
<tr>
<td>Arithmetic Operators</td>
<td>6-4</td>
</tr>
<tr>
<td>Text Operators</td>
<td>6-5</td>
</tr>
<tr>
<td>Relational Operators</td>
<td>6-5</td>
</tr>
<tr>
<td>Overloading an Operator Function</td>
<td>6-6</td>
</tr>
<tr>
<td>Built-In Functions</td>
<td>6-8</td>
</tr>
<tr>
<td>Built-In Functions That You Can Overload</td>
<td>6-8</td>
</tr>
<tr>
<td>Built-In Functions That You Cannot Overload</td>
<td>6-9</td>
</tr>
<tr>
<td>Overloading a Built-In Function</td>
<td>6-10</td>
</tr>
<tr>
<td>Aggregate Functions</td>
<td>6-10</td>
</tr>
<tr>
<td>End-User Routines</td>
<td>6-11</td>
</tr>
</tbody>
</table>
Creating an Operator Class ........................................ 9-15
Creating a New B-Tree Operator Class .................. 9-16
Creating an Absolute-Value Operator Class ......... 9-18
Defining an Operator Class for Other Secondary Access Methods ........................................ 9-21
Dropping an Operator Class ................................. 9-22

Chapter 10 Creating an Opaque Data Type

In This Chapter ..................................................... 10-3
Opaque Data Types ................................................. 10-3
The Internal Structure ............................................ 10-4
Support Functions ............................................... 10-6
Additional SQL-Invoked Routines ......................... 10-7
Advantages of Opaque Data Types ....................... 10-9
Creating an Opaque Data Type ............................. 10-10
Creating the Internal Structure in C .................... 10-10
Registering the Opaque Data Type with the Database ........................................ 10-14
Granting Privileges for an Opaque Data Type .......... 10-19
Privileges on the Opaque Data Type .................... 10-19
Privileges on the Support Functions ..................... 10-20
Creating SQL-Invoked Functions ......................... 10-21
Operating on Data ............................................... 10-21
Comparing Data ................................................... 10-23
Customizing Access Methods .............................. 10-27
Using the Generic B-Tree ..................................... 10-27
Using Other Access Methods ............................... 10-28
Other Operations on Opaque Data Types ............... 10-29
Accessing an Opaque Data Type ......................... 10-29
Dropping an Opaque Data Type ......................... 10-30
A Fixed-Length Opaque Data Type: circle ................ 10-32
Creating the Fixed-Length Internal Structures ........ 10-32
Writing the circle Support Functions .................. 10-33
Registering the circle Opaque Data Type ............. 10-34
Granting Privileges on the circle Data Type .......... 10-35
Creating SQL-Invoked Functions for the circle Data Type ........................................ 10-36
Providing Indexes for the circle Data Type .......... 10-37
Using the circle Opaque Data Type .................... 10-39
A Varying-Length Opaque Data Type: image .................. 10-39
Creating the Varying-Length Internal Structure .............. 10-40
Writing the image Support Functions ....................... 10-40
Registering the image Opaque Data Type ................... 10-42
Registering Other Support Functions for the image Data Type ... 10-44
Granting Privileges on the image Data Type ................. 10-46
Creating SQL-Invoked Functions for the image Data Type . 10-46
Providing Indexes for the image Data Type .................. 10-47

Chapter 11 Writing Support Functions
In This Chapter ........................................... 11-3
Support Functions for Opaque Data Types .................... 11-3
  Handling the External Representation ...................... 11-5
  Handling Platform-Specific Internal Representations ...... 11-10
  Performing Bulk Copies ................................ 11-14
  Inserting and Deleting Data ................................ 11-19
  Handling Smart Large Objects ............................. 11-21
  Comparing Data ........................................ 11-22
Handling Locale-Sensitive Data ................................ 11-23
  In Input and Output Support Functions .................... 11-24
  In Receive and Send Support Functions .................... 11-25

Chapter 12 Managing a User-Defined Routine
In This Chapter ........................................... 12-3
Assigning Execute Privilege to a Routine .................... 12-3
  Granting and Revoking the Execute Privilege ............. 12-4
  Privileges on Objects Associated with a UDR ............. 12-5
  Executing a UDR as DBA ................................ 12-7
  Using DBA Privileges with Objects and Nested UDRs ..... 12-7
Reloading a User-Defined Routine .......................... 12-9
  Unloading the Shared-Object File ......................... 12-9
  Using the ifx_replace_module() Function ................. 12-10
Altering a User-Defined Routine ................................ 12-12
Dropping a User-Defined Routine .............................. 12-12
## Chapter 13 Improving UDR Performance

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>In This Chapter</td>
<td>13-3</td>
</tr>
<tr>
<td>Optimizing a User-Defined Routine</td>
<td>13-3</td>
</tr>
<tr>
<td>Optimizing an SPL Routine</td>
<td>13-4</td>
</tr>
<tr>
<td>Updating Statistics for an SPL Routine</td>
<td>13-6</td>
</tr>
<tr>
<td>Optimizing Functions in SQL Statements</td>
<td>13-8</td>
</tr>
<tr>
<td>Calculating the Query Plan</td>
<td>13-8</td>
</tr>
<tr>
<td>Specifying Cost and Selectivity</td>
<td>13-9</td>
</tr>
<tr>
<td>Calculating Cost</td>
<td>13-10</td>
</tr>
<tr>
<td>Selectivity and Cost Examples</td>
<td>13-11</td>
</tr>
<tr>
<td>Extending UPDATE STATISTICS</td>
<td>13-12</td>
</tr>
<tr>
<td>Using UPDATE STATISTICS</td>
<td>13-13</td>
</tr>
<tr>
<td>Support Functions for UPDATE STATISTICS</td>
<td>13-13</td>
</tr>
<tr>
<td>Using Negator Functions</td>
<td>13-16</td>
</tr>
<tr>
<td>Using a Virtual-Processor Class</td>
<td>13-18</td>
</tr>
<tr>
<td>Choosing a Virtual-Processor Class</td>
<td>13-19</td>
</tr>
<tr>
<td>Using Virtual Processors with UDRs Written in C</td>
<td>13-21</td>
</tr>
<tr>
<td>Using Virtual Processors with UDRs Written in Java</td>
<td>13-22</td>
</tr>
<tr>
<td>Managing Virtual Processors</td>
<td>13-22</td>
</tr>
<tr>
<td>Using UDRs in Parallel</td>
<td>13-23</td>
</tr>
<tr>
<td>Executing UDRs in Parallel</td>
<td>13-24</td>
</tr>
<tr>
<td>Enabling Parallel UDRs</td>
<td>13-31</td>
</tr>
<tr>
<td>Setting the Number of Virtual Processors</td>
<td>13-34</td>
</tr>
<tr>
<td>Monitoring Parallel UDRs</td>
<td>13-35</td>
</tr>
<tr>
<td>Memory Considerations</td>
<td>13-36</td>
</tr>
<tr>
<td>Stack-Size Considerations</td>
<td>13-36</td>
</tr>
<tr>
<td>Virtual-Memory Cache for Routines</td>
<td>13-37</td>
</tr>
<tr>
<td>I/O Considerations</td>
<td>13-38</td>
</tr>
<tr>
<td>Isolating System Catalog Tables</td>
<td>13-39</td>
</tr>
<tr>
<td>Balancing the I/O Activities</td>
<td>13-39</td>
</tr>
</tbody>
</table>

## Index
In This Introduction ........................................... 3
About This Manual ........................................... 3
   Types of Users ........................................... 3
   Software Dependencies ................................... 4
   Assumptions About Your Locale .................. 4
   Demonstration Databases .............................. 4
New Features .................................................. 5
   Extensibility Enhancements ........................... 5
   Performance Improvements ......................... 5
Documentation Conventions ............................... 6
   Typographical Conventions ......................... 6
   Icon Conventions ...................................... 7
      Comment Icons ................................... 8
      Feature, Product, and Platform Icons .......... 8
      Compliance Icons ................................ 9
   Sample-Code Conventions ......................... 10
Additional Documentation ................................. 10
   On-Line Manuals .................................... 11
   Printed Manuals ..................................... 11
   On-Line Help ......................................... 11
   Error Message Documentation .................... 11
   Documentation Notes, Release Notes, Machine Notes . 12
   Related Reading .................................... 13
Compliance with Industry Standards ..................... 14
Informix Welcomes Your Comments ..................... 14
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 Informix Dynamic Server 2000. 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 manual for a list of supplementary titles.
Software Dependencies

This manual assumes that you are using Informix Dynamic Server 2000, Version 9.2.

Assumptions About Your Locale

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

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

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

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

Demonstration Databases

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

- The **stores_demo** database illustrates a relational schema with information about a fictitious wholesale sporting-goods distributor. Many examples in Informix manuals are based on the **stores_demo** database.

- The **superstores_demo** database illustrates an object-relational schema. The **superstores_demo** database 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 *DB-Access User’s Manual*. For descriptions of the databases and their contents, see the *Informix Guide to SQL: Reference*.

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

### New Features

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

- Extensibility enhancements
- Performance improvements

### Extensibility Enhancements

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

- Enhancements to user-defined routines (UDRs):
  - `GRANT/REVOKE` on UDR external languages
  - `ALTER ROUTINE` statement
  - User-defined aggregates
- Connectivity: Client interfaces (ESQL/C, Informix ODBC Driver, C++, Java, JDBC, GLS) supported in a single package

### Performance Improvements

This manual describes the following performance improvements to Version 9.2 of Dynamic Server for user-defined routines:

- Expensive-function optimization
- Parallel UDRs
- User-defined statistics routines
### 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.

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>KEYWORD</strong></td>
<td>All primary elements in a programming language statement (keywords) appear in uppercase letters in a serif font.</td>
</tr>
<tr>
<td><em>italics</em></td>
<td>Within text, new terms and emphasized words appear in italics.</td>
</tr>
<tr>
<td><strong>italics</strong></td>
<td>Within syntax and code examples, variable values that you are to specify appear in italics.</td>
</tr>
<tr>
<td><strong>boldface</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>monospace</strong></td>
<td>Information that the product displays and information that you enter appear in a monospace typeface.</td>
</tr>
</tbody>
</table>

(1 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.

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEYPRESS</td>
<td>Keys that you are to press appear in uppercase letters in a sans serif font.</td>
</tr>
<tr>
<td>♦</td>
<td>This symbol indicates the end of one or more product- or platform-specific paragraphs.</td>
</tr>
<tr>
<td>→</td>
<td>This symbol indicates a menu item. For example, “Choose Tools→Options” means choose the Options item from the Tools menu.</td>
</tr>
</tbody>
</table>
Icon Conventions

Comment Icons

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

<table>
<thead>
<tr>
<th>Icon</th>
<th>Label</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Warning Icon]</td>
<td>Warning:</td>
<td>Identifies paragraphs that contain vital instructions, cautions, or critical information</td>
</tr>
<tr>
<td>![Important Icon]</td>
<td>Important:</td>
<td>Identifies paragraphs that contain significant information about the feature or operation that is being described</td>
</tr>
<tr>
<td>![Tip Icon]</td>
<td>Tip:</td>
<td>Identifies paragraphs that offer additional details or shortcuts for the functionality that is being described</td>
</tr>
</tbody>
</table>

Feature, Product, and Platform Icons

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

<table>
<thead>
<tr>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>![C Icon]</td>
<td>Identifies information that relates to routines written in C</td>
</tr>
<tr>
<td>![E/C Icon]</td>
<td>Identifies information that relates to ESQL/C</td>
</tr>
<tr>
<td>![EXT Icon]</td>
<td>Identifies information that relates to routines written in external languages, either C or Java.</td>
</tr>
<tr>
<td>![GLS Icon]</td>
<td>Identifies information that relates to the Informix Global Language Support (GLS) feature</td>
</tr>
</tbody>
</table>
**Icon Conventions**

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.

<table>
<thead>
<tr>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>Identifies information that relates to routines written in Java.</td>
</tr>
<tr>
<td>SPL</td>
<td>Identifies information that relates to the Stored Procedure Language</td>
</tr>
<tr>
<td>UNIX</td>
<td>Identifies information that is specific to UNIX platforms</td>
</tr>
<tr>
<td>WIN NT</td>
<td>Identifies information that is specific to the Windows NT environment</td>
</tr>
</tbody>
</table>

These 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 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

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

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

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

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

www.informix.com/answers

Printed Manuals

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

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

On-Line Help

Informix provides on-line help with each graphical user interface (GUI) that displays information about those interfaces and the functions that they perform. Use the help facilities that each GUI provides to display the on-line help.

Error Message Documentation

Informix software products provide ASCII files that contain all of the Informix error messages and their corrective actions.
To read error messages and corrective actions on UNIX, use one of the following utilities.

<table>
<thead>
<tr>
<th>Utility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>finderr</td>
<td>Displays error messages on line</td>
</tr>
<tr>
<td>rofferr</td>
<td>Formats error messages for printing</td>
</tr>
</tbody>
</table>

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

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

**Documentation Notes, Release Notes, Machine Notes**

In addition to printed documentation, the following sections describe the online files that supplement the information in this manual. Please examine these files before you begin using your database server. They contain vital information about application and performance issues.
On UNIX platforms, the following on-line files appear in the 
$INFORMIXDIR/release/en_us/0333 directory.

<table>
<thead>
<tr>
<th>On-Line File</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXTENDDOC_9.2</td>
<td>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.</td>
</tr>
<tr>
<td>SERVERS_9.2</td>
<td>The release notes file describes feature differences from earlier versions of Informix products and how these differences might affect current products. This file also contains information about any known problems and their workarounds.</td>
</tr>
<tr>
<td>IDS_9.2</td>
<td>The machine notes file describes any special actions that you must take to configure and use Informix products on your computer. The machine notes are named for the product described.</td>
</tr>
</tbody>
</table>

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

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

Machine notes do not apply to Windows environments. ♦

Related Reading

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

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

Informix Welcomes Your Comments

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

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

Send electronic mail to us at the following address:

doc@informix.com

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

We appreciate your suggestions.
Extending the Database Server

In This Chapter ......................................... 1-3
Creating User-Defined Routines ..................... 1-3
  End-User Routines .................................. 1-4
  Routines That Extend Operations ................ 1-5
Extending the Built-In Data Types ................... 1-6
  Building Data Types on Existing Data Types ... 1-6
  Extending Operators ............................... 1-6
  Building Opaque Data Types ..................... 1-7
Routine Management .................................. 1-8
In This Chapter

This manual discusses extending Informix Dynamic Server 2000 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 user-defined routines (UDRs) to support the extensions. A routine is a collection of program statements that perform a particular task. A user-defined routine 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, “Creating a User-Defined Routine”
- Chapter 3, “Running a User-Defined Routine”
- Chapter 4, “Developing a User-Defined Routine”
End-User Routines

You can write UDRs in the following languages:

- **Stored Procedure Language (SPL)**
  The SPL language is a 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.

- **The C programming language**
  To write routines in C, you need a C compiler and the Informix ESQL/C compiler.
  For information about writing UDRs in C, refer to the *DataBlade API Programmer’s Manual.*

- **The Java programming language**
  To write routines in Java, you must have J/Foundation. You also must install the Java Development Kit (JDK).
  For information about writing UDRs in Java, refer to *Creating UDRs in Java.*

**Important:** Informix recommends that you use DBDK to develop UDRs in C or Java because DBDK enforces standards that facilitate migration between different versions of the database server.

End-User Routines

End-user routines are routines that the SQL user can include in an SQL statement. Such a routine might be as simple as “increase the price of every item from XYZ Corporation by 5 percent” or something far more complicated. To write these routines, you might need only the first three chapters of this manual plus the language documentation mentioned in the previous section.
Routines That Extend Operations

You can write special-purpose end-user routines that extend the built-in operations of the database. An *operation* is a task that the database server performs on one or more values. The manual discusses 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 an Operation,” discusses general aspects of extending an operation and describes how to extend operator symbols and built-in functions.

- **Aggregates**
  An aggregate produces one value that summarizes some aspect of a selected column; for example, the average or the count. You can create a user-defined aggregate to sum the square of each value in the column. You can also extend existing aggregates, 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.”

- **Operator classes**
  An operator class is the set of functions that is associated with building an index.
  Chapter 9, “Extending an Operator Class,” describes how to create a user-defined operator class and how to extend an existing operator class.
  Chapter 11, “Writing Support Functions,” describes the support functions that an operator class uses.
Extending the 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 the built-in data types in the following ways:

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

Building Data Types on Existing 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 user-defined data types that are based on built-in data types. The Informix Guide to Database Design and Implementation also provides an extensive discussion of UDTs that are based on built-in data types.

Extending Operators

When you build a user-defined data type, you must provide for the operations that the data type uses:

- Operators
  
  If you want to use the basic operator symbols (+, -, and so on) or functions such as cos() and abs() with the user-defined data type, you must write routines that define these operations. 
  
  Chapter 6, “Extending an Operation,” discusses general aspects of extending an operation and describes how to extend operator symbols and built-in functions.
Building Opaque Data Types

- **Casts**
  The database server provides casts for the built-in data types. When you use user-defined data type, you usually need to provide casts for the user-defined data types.

- **Aggregates**
  “Extending Existing Aggregates” on page 8-4 describes how to extend built-in aggregates to support a user-defined data type.
  “Using User-Defined Data Types with User-Defined Aggregates” on page 8-16 describes how to extend a user-defined aggregate so that it supports a user-defined data type.

**Building Opaque Data Types**

An opaque data type is an atomic 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:

- 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:
  - Can you find a picture?
  - Can you say that one data item is inside another?

Chapter 10, “Creating an Opaque Data Type,” describes the basic steps for creating an opaque data type. Chapter 11, “Writing Support Functions,” describes the support functions that an opaque data type uses.
Routine Management

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, Informix recommends that you use the Informix DataBlade Developers Kit 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. Informix provides DataBlade modules that support various special-purpose opaque data types. To find out what DataBlade modules are available, contact your local Informix representative.

Routine Management

The final two chapters of the book discuss the management of UDRs.

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

- Assigning privileges to a UDR
- Loading and unloading a UDR
- Modifying a UDR

Chapter 13, “Improving UDR Performance,” discusses ways that you can optimize the performance of your UDR.
Chapter 2

Creating a User-Defined Routine

In This Chapter .................................. 2-3
User-Defined Routines ............................ 2-3
SPL Routines ...................................... 2-5
External Routines .................................. 2-6
Tasks That You Can Perform with UDRs ............ 2-7
Extending Data Type Support ...................... 2-8
  Cast Functions .................................. 2-9
  End-User Routines ............................... 2-11
  Aggregate Functions ............................ 2-11
  Operator Functions ............................. 2-12
  Operator-Class Functions ...................... 2-14
  Optimization Functions ....................... 2-17
  Opaque Data Type Support Functions .......... 2-18
  Access-Method Purpose Functions ............. 2-18
Creating an End-User Routine ...................... 2-19
  Encapsulating Multiple SQL Statements ....... 2-19
  Creating Triggered Actions .................... 2-20
  Restricting Access to a Table .................. 2-21
  Creating Iterators ............................. 2-21
Using a User-Defined Routine ..................... 2-22
  Invoking a User-Defined Routine .............. 2-22
    Explicit Invocation .......................... 2-22
    Implicit Invocation ......................... 2-23
  Creating a User-Defined Routine .............. 2-23
  Managing User-Defined Routines ............... 2-24
  Optimizing a User-Defined Routine ............ 2-25
In This Chapter

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

- User-defined routines
- Tasks that you can perform with UDRs
- Uses of UDRs

User-Defined Routines

A routine is a collection of program statements that perform a particular task. A user-defined routine (UDR) is a routine that you can define and that can be invoked within an SQL statement or another UDR. 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.
- A user-defined procedure is a routine that optionally accepts a set of arguments but does not return any values. A procedure cannot be used in SQL expressions because it does not return a value.

The database server supports UDRs written in the following languages:

- Stored Procedure Language (SPL), a language that is internal to the database server
- External languages such as C or Java

SPL routines can execute routines written in external languages (C or Java), and external routines (routines written in C or Java) can execute SPL routines.
You create the UDR in a language that the database server supports and then register it in the system catalog tables so that the database server can access it. The database server stores information on user-defined routines in the following system catalog tables:

- The `sysprocedures` system catalog table contains information about the user-defined routine, such as its name, owner, and whether it is a user-defined function or user-defined procedure.
- The `sysproauth` system catalog table contains information on which users of the database server can execute a particular user-defined routine.
SPL Routines

An SPL routine is a user-defined routine that is written in Stored Procedure Language (SPL) and SQL that the database server parses, optimizes, and stores in the system catalog tables in executable format. Because routines written in SQL are parsed, optimized as far as possible, and then stored in the system catalog tables in executable format, consider using an SPL routine for SQL-intensive tasks. For more information on how the database server parses an SPL routine, see “Executing an SPL Routine” on page 3-11.

Tip: Not all the encapsulated SPL that you created as SPL procedures in earlier Informix products has the properties currently associated with user-defined procedures. If the SPL routine returns a value, you now refer to it as an SPL function. If the SPL routine does not return a value, you still refer to it as an SPL procedure.

You create and register an SPL routine with one of the following SQL statements.

<table>
<thead>
<tr>
<th>SQL Statement</th>
<th>Type of SPL Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE FUNCTION</td>
<td>SPL function</td>
<td>An SPL routine that returns a value</td>
</tr>
<tr>
<td>CREATE PROCEDURE</td>
<td>SPL procedure</td>
<td>An SPL routine that does not return a value</td>
</tr>
</tbody>
</table>

For more information, see “Registering an SPL Routine” on page 4-17.

These SQL statements contain the SPL code that makes up the SPL routine. SPL provides the flow-control extensions to SQL. The body of an SPL routine contains SQL statements and flow-control statements for looping and branching. The CREATE FUNCTION and CREATE PROCEDURE statements store the code for the body of the routine in the sysprocboby system catalog table of the database. For information on the syntax of SPL statements, see the Informix Guide to SQL: Syntax. For an explanation of how to use SPL statements, refer to the Informix Guide to SQL: Tutorial.
External Routines

An external routine is a user-defined routine that is written in an external language. The body of an external routine contains external-language 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 routine into an executable format.

The database server supports user-defined routines written in the following external languages.

<table>
<thead>
<tr>
<th>External Language</th>
<th>Detailed Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>DataBlade API Programmer’s Manual</td>
</tr>
<tr>
<td>Java</td>
<td>Creating UDRs in Java</td>
</tr>
</tbody>
</table>

The database server stores information on external languages that it supports for user-defined routines 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.

Tip: For the most current information on the languages that the database server supports for user-defined routines, query the sysroutinelangs system catalog table.

You register the external routine with one of the following SQL statements.

<table>
<thead>
<tr>
<th>SQL Statement</th>
<th>Type of External Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE FUNCTION</td>
<td>External function</td>
<td>An external routine that returns a value</td>
</tr>
<tr>
<td>CREATE PROCEDURE</td>
<td>External procedure</td>
<td>An external routine that does not return a value</td>
</tr>
</tbody>
</table>
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.

For more information, see “Registering an External Routine” on page 4-20.

Tasks That You Can Perform with UDRs

You can write user-defined routines to accomplish the following kinds of tasks:

- Extend support for built-in or user-defined data types
- 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 user-defined routines.

<table>
<thead>
<tr>
<th>UDR Task</th>
<th>SPL Routines</th>
<th>C Routines</th>
<th>Java Routines</th>
<th>For More Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Built-in function</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Chapter 6</td>
</tr>
<tr>
<td>Cast function</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Chapter 7</td>
</tr>
<tr>
<td>Cost function</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Chapter 13</td>
</tr>
<tr>
<td>End-user routine</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>page 2-19</td>
</tr>
<tr>
<td>Iterator function</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Chapter 4</td>
</tr>
<tr>
<td>Negator function</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Chapter 13</td>
</tr>
<tr>
<td>Opaque data type support function</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Chapter 10</td>
</tr>
<tr>
<td>Operator function</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Chapter 6</td>
</tr>
<tr>
<td>Operator-class function</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Chapter 9</td>
</tr>
<tr>
<td>Parallelizable UDR</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Chapter 13</td>
</tr>
<tr>
<td>Statistics function</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Chapter 13</td>
</tr>
<tr>
<td>Selectivity function</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Chapter 13</td>
</tr>
<tr>
<td>User-defined aggregate</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes (with some limitations)</td>
<td>Chapter 8</td>
</tr>
</tbody>
</table>

**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.
When you create a user-defined data type, you must provide the following routines:

- End-user routines that users specify explicitly in SQL statements
- Support functions that the database server invokes implicitly to operate on these 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
- Operator-class functions if you want to use a different ordering scheme in an index than the order that the default operator class provides
- Access-method purpose functions

**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 user-defined data types. 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 *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.
Extending Data Type Support

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 user-defined data types:

- **Opaque data types**
  You can create casts to convert a user-defined data type to other data types in the database. Developers of opaque data types must also opaque-type functions that serve as cast functions between the internal and external representations of the opaque type. For more information, see Chapter 10, “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 vice versa. 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 casting between named row types and unnamed row types, see the *Informix Guide to SQL: Tutorial*.

In addition, you might want to define a new cast function to do the following:

- **Ensure that the database server invokes the correct routine for the data type that a user might specify when invoking an overloaded function**
  For more information on casting and routine resolution, refer to “The Routine-Resolution Process” on page 3-17.

- **Ensure that the query optimizer considers an index defined on column that might be specified in the filter of a query**

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 provides special functionality that application users often need. 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 user-defined routine 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 an Operation."

- **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 2-19.

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.
Extending Data Type Support

You can extend an aggregate function for built-in and user-defined data types, 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 user-defined data type, 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 write an aggregate function with the same names as the built-in aggregate that you want to extend. This function implements the aggregate for the user-defined data type
  - 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.”

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.

<table>
<thead>
<tr>
<th>Method of Comparison</th>
<th>Operator Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operator function</td>
<td>equal(value1, value2)</td>
</tr>
<tr>
<td>Operator symbol</td>
<td>value1 = value2</td>
</tr>
</tbody>
</table>

The following sections summarize how you can extend an operator on built-in and user-defined data types. For more information on how to extend operators, refer to Chapter 6, “Extending an Operation.”

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 an Operation.” 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 user-defined data type. When you define the appropriate operator function, operator binding enables SQL statements to use both the function name and its operator symbol on the user-defined data type. 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 MacDonald and MacDonald to appear together on a phone list. The default relational operators (for example, =) for character strings do not achieve this ordering.
To order Mc and Mac in the same way, you can create external functions that treat Mc and Mac the same. You can define an equal() 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 equal(Scottish, Scottish) external function when it compares two Scottish-name values. For more information on routine overloading, 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 user-defined data type, 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 optimizing queries with user-defined routines, refer to “Optimizing a User-Defined Routine” on page 13-3.

The following sections summarize how you can extend an operator class on built-in and user-defined data types. For more information on how to extend operator classes, refer to “Extending an Existing Operator Class” on page 9-9.
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.
  You write operator-class functions that have the same names as those in the existing operator class but implement a new ordering scheme. These new functions change how the existing operators order data when they handle the user-defined data type. The query optimizer can choose an index on this data type when the index uses this existing 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 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 ordering 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 ordering 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.

- Extend the default operator class for an existing secondary access method to redefine the ordering scheme that these operators support.
  You write operator-class functions that have the same names as those in the existing operator class but implement a new ordering scheme. These new functions change how the existing operators order data when they handle the opaque data type. The query optimizer can choose an index on this data type when the index uses this existing operator class and the SQL statement contains one of the operators in this operator class.

- Define a new operator class to provide an entirely new set of operators that operate on the opaque type.
  You write operator-class functions with names that are different from any existing operating-class functions associated with the secondary access method. These functions define the new operators that the query optimizer can recognize as associated with the secondary access method. 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.
Extending Data Type Support

Optimization Functions

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

<table>
<thead>
<tr>
<th>Optimization Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negator function</td>
<td>Specifies the function to use for a NOT condition that involves a Boolean UDR</td>
</tr>
<tr>
<td>Cost function</td>
<td>Specifies the cost factor for execution of a particular UDR</td>
</tr>
<tr>
<td>Selectivity function</td>
<td>Specifies the percentage of rows for which a Boolean UDR is expected to return true</td>
</tr>
<tr>
<td>Parallel UDR</td>
<td>A UDR that can be run in parallel and therefore can be run in parallel queries</td>
</tr>
<tr>
<td>Statistics function</td>
<td>Creates distribution statistics for a user-defined data type</td>
</tr>
</tbody>
</table>

You can extend an optimization function that operates on the following data types:

- **Built-in data types**
  The database server provides optimization functions for the built-in data types. You cannot extend existing optimization for built-in types through optimization functions.

- **User-defined data types**
  You can write an optimization function on any user-defined data type that is registered in the database.

For more information about optimization functions, see Chapter 13, “Improving UDR Performance.”
Extending Data Type Support

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 the support functions for opaque data types, refer to Chapter 11, “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 of 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 of 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 Informix R-Tree Index User’s Guide.

  To associate an operator class with an access method, you must first create the access method, then create the operator class, and finally alter the access method to add the operator class.

  DataBlade modules can provide other primary and secondary access methods. For more information, refer to the DataBlade user guides.
Creating an End-User Routine

Important: The “Virtual-Table Interface Programmer’s Manual” and “Virtual-Index Interface Programmer’s Manual” are intended for Informix DataBlade Partners. For information on how to become a DataBlade Partner, contact your Informix sales representative.

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
- Facilitate interactive multimedia publication

You can write end-user routines in either SPL or external languages.

Encapsulating Multiple SQL Statements

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

Simplifying Programs

A user-defined routine can batch 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 user-defined routine 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 an SPL routine or external routine, so you need not include the same code in each program. The code is stored in only one place, eliminating duplicate code.
Creating an End-User Routine

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 on the database server computer, so the change is made in only one location.

Instead of centralizing database code in client applications, you create SPL 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.

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 *Informix Guide to SQL: Tutorial*.

**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 “Returning Values with an Iterator Function” on page 4-11.

Using a User-Defined Routine

This section summarizes the development process of a user-defined routine.

Invoking a User-Defined Routine

A user-defined routine can be invoked either explicitly or implicitly. The following sections summarize how to invoke a UDR. For more information on how to invoke UDRs, see Chapter 3, “Running a User-Defined Routine.”

Explicit Invocation

You can use the EXECUTE PROCEDURE statement to execute a user-defined procedure (SPL or external) from:

- a user-defined routine (SPL or external).
- DB-Access.
- a client application (such as an ESQL/C application).

You cannot use a procedure in an SQL expression because a procedure does not return a value.

You can execute a user-defined function (SPL or external) in the following ways:

- With the EXECUTE FUNCTION statement:
  - A user-defined routine (SPL or external)
  - DB-Access
  - A client application (such as an ESQL/C application)
- Within an SQL expression (in the SELECT clause or WHERE clause)
Creating a User-Defined Routine

Implicit Invocation

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

<table>
<thead>
<tr>
<th>Implicit Call of UDR</th>
<th>UDR Called</th>
</tr>
</thead>
<tbody>
<tr>
<td>Built-in operator binding</td>
<td>Operator function</td>
</tr>
<tr>
<td>Implicit casting</td>
<td>Implicit cast function</td>
</tr>
<tr>
<td>Opaque-type processing</td>
<td>Opaque-type support functions and statistics functions</td>
</tr>
<tr>
<td>Query processing</td>
<td>Optimization functions and operator-class functions</td>
</tr>
</tbody>
</table>

Creating a User-Defined Routine

The following steps summarize the creation process for a user-defined routine:

- Write the routine source.
  
  The source code can use SQL statements to perform the task of the UDR. The code must be in the language of the UDR:
  
  - For SPL routines, see the Informix Guide to SQL: Syntax.
  - For external routines, refer to the DataBlade API Programmer’s Manual and Creating UDRs in Java.

- Create a shared-object file that contains the executable version of the UDR.
  
  Because external routines are written in a language that is external to the database, you must compile the UDR source code and put it in a shared-object file, which the database server loads when the UDR is invoked.
Managing User-Defined Routines

- Register the user-defined routine.

To make the UDR available to the database server, you register it in the system catalog of the database. Use one of the following SQL statements to register the UDR:

- The CREATE FUNCTION statement registers *user-defined functions*.
- The CREATE PROCEDURE statement registers *user-defined procedures*.

For an SPL routine, the CREATE FUNCTION and CREATE PROCEDURE statements also provide the source code for the routine.

For more information on how to create a user-defined routine, see Chapter 4, “Developing a User-Defined Routine.”

Managing User-Defined Routines

The database server provides support for the following management tasks for a user-defined routine:

- Assigning Execute privilege to a UDR
- Reloading a UDR
- Altering a user-defined routine
- Dropping a user-defined routine

For more information on how to manage a user-defined routine, see Chapter 12, “Managing a User-Defined Routine.”
Optimizing a User-Defined Routine

Once you have created a UDR, you can use several optimization features to improve performance of the routine. These features are as follows:

- For UDRs that return BOOLEAN values:
  - Create a negator function
  - Create a selectivity function
- Execute the UDR in a parallel distributed query (PDQ) statement
- Create a cost function to provide the query optimizer with an estimate of the cost of executing the UDR
- For an opaque data type, create a statistics function that provides statistics information on the opaque type to the query optimizer

For more information on how to optimize a user-defined routine, see Chapter 13, “Improving UDR Performance.”
Running a User-Defined Routine

In This Chapter ........................................ 3-3
Invoking a User-Defined Routine .................. 3-3
   Invoking a UDR in an SQL Statement .......... 3-3
      Invoking a UDR with an EXECUTE Statement. 3-4
      Invoking a User-Defined Function in an Expression 3-5
      Invoking a Function That Is Bound to an Operator 3-6
      Invoking a UDR in an SPL Routine .......... 3-6
Executing a User-Defined Routine ............... 3-7
   Parsing the SQL Statement ..................... 3-7
   Optimizing the SQL Statement ............... 3-8
   Executing an External Routine ............... 3-8
      Loading a Shared-Object File into Memory .. 3-8
      Creating the Routine Sequence ............. 3-9
      Managing Routine Execution .............. 3-10
   Executing an SPL Routine .................... 3-11
Understanding Routine Resolution ............... 3-11
   The Routine Signature ....................... 3-12
      Using ANSI and Non-ANSI Routine Signatures 3-12
      Using the Routine Signature to Perform DBA Tasks 3-13
   Overloading Routines ......................... 3-13
      Creating Overloaded Routines ............. 3-14
      Assigning a Specific Routine Name ........ 3-15
      Specifying Overloaded Routines During Invocation 3-16
   Overloading Built-In SQL Functions .......... 3-16
The Routine-Resolution Process ................. 3-17
   The Routine Signature ....................... 3-17
   Candidate List of Routines .................. 3-18
   Precedence List of Data Types ............... 3-19
   Precedence List for Built-In Data Types .... 3-20
Routine Resolution with User-Defined Data Types . . . . . . . . 3-22
Routine Resolution in a Type Hierarchy . . . . . . . . . . . . . . 3-22
Routine Resolution with Distinct Data Types . . . . . . . . . . . . 3-24
Routine Resolution with Built-In Data Types as Source . . . . . . 3-26
Routine Resolution with Collection Data Types . . . . . . . . . . . . 3-26
Null Arguments in Overloaded Routines . . . . . . . . . . . . . . . . 3-27
In This Chapter

This chapter discusses the following topics:

- How to invoke a user-defined routine
- How a user-defined routine executes
- What routine overloading and routine resolution are

Invoking a User-Defined Routine

This section describes the ways that you can invoke a user-defined routine:

- In an SQL statement
- In an SPL routine, with the CALL statement

Invoking a UDR in an SQL Statement

You can invoke a user-defined routine from within an SQL statement in the following ways:

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

Invoking a UDR with an EXECUTE Statement

For details about the syntax of the EXECUTE FUNCTION and EXECUTE PROCEDURE statements, see the 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 function 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!):

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

The INTO clause must always be present when you invoke a user-defined function with the EXECUTE FUNCTION statement.

Important: You cannot use the EXECUTE FUNCTION statement to invoke a user-defined function that contains an OUT parameter.

Using a SELECT Statement in a Function Argument

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

```sql
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;
```
Invoking a UDR in an SQL Statement

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 another set of parentheses.

**Invoking a Procedure**

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

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

The INTO clause is never present when you invoke a user-defined procedure with the EXECUTE PROCEDURE statement because a procedure does not return a value.

**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 gets invoked automatically. For example, the following query implicitly invokes the appropriate `equal()` function to evaluate the WHERE clause:

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

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

```sql
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:
  ```sql
  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:
  ```sql
  CALL log_compare (arg1, arg2)
  ```
Executing a User-Defined Routine

When you invoke a routine, 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 to break the SQL statement into its syntactic parts
2. Calls the query optimizer to develop a query plan, which efficiently organizes the execution of the parts of the SQL statement
3. Executes the UDR:
   - For external routines, the routine manager executes the UDR in the appropriate external language. The routine manager is the specific part of the database server that manages the execution of user-defined routines.
   - 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.

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 for entire SQL statement. The query optimizer formulates a query plan to fetch the data rows that are required to process a query.

For more information on the query optimizer, see “Optimizing a User-Defined Routine” on page 13-3.

Executing an External Routine

The routine manager performs the following steps to handle execution of external UDRs:

- Loads the shared-object files
- Creates a routine sequence
- Manages the actual execution of the UDR

Loading a Shared-Object File into Memory

To execute a user-defined routine written in an external language, its shared-object file must reside in database server memory. On the first invocation of a user-defined routine, the routine manager loads the shared-object file that contains the UDR into memory. It locates the shared-object file from the path column in the sysprocedures system catalog table.

Use the onstat command-line utility with the -g dll option to view the dynamically loaded libraries in which your user-defined routines reside. For information about the onstat command, refer to the Administrator’s Reference.

Once the routine manager has loaded a shared-object file into memory, this file remains in memory until it is explicitly unloaded or the database server is shut down. For more information, see “Unloading the Shared-Object File” on page 12-9.

When you create an SPL routine or procedure, 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 compiled code. Therefore, the routine manager does not need to load or unload SPL routines.
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 user-defined routine. Every mention of a user-defined routine, implicit or explicit, in an SQL or SPL statement creates a single independent routine sequence. Sometimes, a routine sequence consists of the single call to the user-defined routine, as follows:

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

However, often a user-defined routine 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 running_avg(price) FROM stock_history
WHERE symbol = 'IFMX'
```

All invocations of the `running_avg()` function in this query make up a single routine sequence. In the preceding query, the implicit `equal()` function, which the WHERE clause invokes, is also called for each matching row. This `equal()` function is in its own separate routine sequence. Each individual call to a UDR within a routine sequence is called a routine invocation.

Each invocation logically has a distinct routine instance. In the preceding SELECT statement, a call to `running_avg()` for one matching row in the `stock_history` table is one routine instance.

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
Executing an External Routine

**Important:** This argument information does not include the actual argument values. It only contains information 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.

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 *DataBlade API Programmer’s Manual*.

For a routine written in Java, the `UDREnv` interface provides most of the information that MI_FPARAM provides for a UDR written in C. 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 *Creating UDRs in Java*.

### Managing Routine Execution

Once 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.
Executing an SPL Routine

Unlike an external routine, whose executable code resides in a shared-object file, the executable code for an SPL routine is stored directly in the `sysprocbody` system catalog table of the database. 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.
- The interpreter executes the p-code instructions.

For an SPL routine, the routine-state space is internal to the database server. 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 SQL routine does not have access to the user state of its routine sequence.

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. 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`. 
The Routine Signature

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 user-defined routine.
- avoid corrupting data 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
- 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.
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 statements, the database server stores the routine signature in the sysprocedures system catalog table. For more information, see “Registering a User-Defined Routine” on page 4-15.

**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 shown in Figure 3-1 uses a routine signature.

![Table: 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 user-defined data type.
- 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 and 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):

```sql
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:

```sql
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 statement are equivalent:

```sql
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`:

```sql
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:

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

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

```sql
EXECUTE FUNCTION equal(col1, col2) INTO result
CALL equal(col1, col2) RETURNING result
```

For more information about overloaded operator functions, refer to Chapter 6, “Extending an Operation.”

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 user-defined data type that represents complex numbers. For a complete list of built-in SQL functions that you can overload, see “Built-In Functions” on page 6-8.
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 3-18.
   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 3-19.

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 3-27.
The Routine-Resolution Process

**Candidate List of Routines**

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

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

```sql
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:

```sql
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.
The Routine-Resolution Process

**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 type*, 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 on the precedence of each built-in data type, refer to “Precedence List for Built-In Data Types” on page 3-20.
   
   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 an error message.

-674: Routine routine-name not found.

If the routine-resolution process locates more than one qualifying routine, the database server returns an 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.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Precedence List</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>VARCHAR, LVARCHAR</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>None</td>
</tr>
<tr>
<td>NCHAR</td>
<td>NVARCHAR</td>
</tr>
<tr>
<td>NVARCHAR</td>
<td>None</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>INT, INT8, DECIMAL, SMALLFLOAT, FLOAT</td>
</tr>
<tr>
<td>INT</td>
<td>INT8, DECIMAL, SMALLFLOAT, FLOAT, SMALLINT</td>
</tr>
<tr>
<td>INT8</td>
<td>DECIMAL, SMALLFLOAT, FLOAT, INT, SMALLINT</td>
</tr>
<tr>
<td>SERIAL</td>
<td>INT, INT8, DECIMAL, SMALLFLOAT, FLOAT, SMALLINT</td>
</tr>
<tr>
<td>SERIAL8</td>
<td>INT8, DECIMAL, SMALLFLOAT, FLOAT, INT, SMALLINT</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>SMALLFLOAT, FLOAT, INT8, INT, SMALLINT</td>
</tr>
<tr>
<td>SMALLFLOAT</td>
<td>FLOAT, DECIMAL, INT8, INT, SMALLINT</td>
</tr>
<tr>
<td>FLOAT</td>
<td>SMALLFLOAT, DECIMAL, INT8, INT, SMALLINT</td>
</tr>
<tr>
<td>MONEY</td>
<td>DECIMAL, SMALLFLOAT, FLOAT, INT8, INT, SMALLINT</td>
</tr>
</tbody>
</table>
The Routine-Resolution Process

The following example shows overloaded test functions and a query that invokes the test function. This query invokes the function with a DECIMAL argument, \texttt{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 shown in the precedence list in Figure 3-2.

```
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.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Precedence List</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>None</td>
</tr>
<tr>
<td>DATETIME</td>
<td>None</td>
</tr>
<tr>
<td>INTERVAL</td>
<td>None</td>
</tr>
<tr>
<td>BYTE</td>
<td>None</td>
</tr>
<tr>
<td>TEXT</td>
<td>None</td>
</tr>
</tbody>
</table>

(2 of 2)
Routine Resolution with User-Defined Data Types

The next sections discuss routine resolution when one or more of the arguments in the routine signature are user-defined data types.

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 Informix Guide to Database Design and Implementation.

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:

```sql
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:

```sql
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 3-19.
   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 described in “Precedence List of Data Types” on page 3-19.
   b. Because the second parameter is the `INTEGER` built-in data type, the database server goes to the precedence list shown in Figure 3-2 on page 3-20.
      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)`. 

---

Running a User-Defined Routine 3-23
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 UDRs from their source types when the source is not a built-in data type. For information about distinct data types, refer to “Distinct Data Type” on page 5-11.

When a UDR has distinct types in its parameter list, the database server must resolve which data type to pass to the UDR, 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 routine signature contains a parameter that does not match the distinct data type in the same position of the routine invocation, 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 “Routine Resolution with Casts” on page 2-16.
  - 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.
Routine Resolution with User-Defined Data Types

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

```sql
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;
```

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.

**Important:** The routine-resolution process cannot match two different distinct 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 in 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 shown in:

- An overloaded function `equals(pounds,stones)` to handle the two distinct data types
  
  ```sql
  CREATE FUNCTION equals(pounds, stones) ...
  ```

  The advantage of creating an overloaded `equal()` 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`
  
  ```sql
  CREATE IMPLICIT CAST (pounds AS stones):
  CREATE IMPLICIT CAST (stones AS INT):
  ```
Routine Resolution with User-Defined Data Types

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 the built-in casts provided for the built-in type. For example, suppose you create the following distinct data type and table:

```
CREATE DISTINCT TYPE inches AS FLOAT;
CREATE TABLE test(col1 inches);
```

An SQL user might execute the following sample query:

```
SELECT 4 + col1 FROM test;
```

Although the source data type of the `col1` argument has a built-in cast function to convert FLOAT to INTEGER, this query fails because the distinct type `inches` does not inherit the built-in cast.

**Important:** The routine-resolution process does not consider a routine that contains an argument that can be converted from the source data type by a built-in cast.

To add an INTEGER to the `inches` data type, perform one of the following actions:

- Write and define an implicit cast routine that converts `inches` to INTEGER.
  The routine-resolution process selects the built-in `plus()` function after the `inches` argument is converted to INTEGER.
- Write and define an overloaded `plus()` routine that accepts an INTEGER data type as the first argument and an `inches` data type as the second argument.
- Use explicit casts in the SQL query, as the following sample statements show:
  ```
  SELECT 4 + col1::INT FROM test;
  SELECT 4::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 user-defined routine 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 in SPL or in the C language. (For external functions, you must use the HANDLESNULLS modifier to specify that each function can handle null arguments.)

```
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.
Developing a User-Defined Routine

In This Chapter ............................. 4-3

Planning the Routine ..................... 4-3
  Naming the Routine ..................... 4-4
  Defining Routine Parameters .......... 4-5
    Number of Arguments ................. 4-5
    Declaring Routine Parameters ...... 4-5
  Returning a Value ..................... 4-6
    Returning a Variant or Nonvariant Value 4-7
    Returning Multiple Values with an OUT Parameter 4-8
  Returning Values with an Iterator Function 4-11
  Adhering to Coding Standards .. 4-13

Writing the Routine ...................... 4-14

Creating a Shared-Object File ........... 4-14

Registering a User-Defined Routine .... 4-15
  Setting Privileges for a Routine ..... 4-15
    Database-Level Privilege .......... 4-16
    Language-Level Privilege ........ 4-16
    Routine-Level Privilege .......... 4-17
  Registering an SPL Routine .......... 4-17
  Registering an External Routine .... 4-20
    Registering an External Function 4-21
    Registering an External Procedure 4-21
    Registering an External Routine with Modifiers 4-22
    Registering Parameters and a Return Value 4-25

Reviewing Information about User-Defined Routines .... 4-26
In This Chapter

This chapter describes how to design and create user-defined routines (UDRs). It covers the following topics:

- Planning the routine
- Creating the shared-object file ♦
- Registering the UDR

Although the SQL statements that you use to define SPL routines and external routines are the same, the specific clauses and modifiers that you can use with the SQL statements differ somewhat. For this reason, SPL routines and external routines are often documented separately.

For information about how to create and use SPL routines, see the chapter on SPL routines in the *Informix Guide to SQL: Tutorial*.

Planning the Routine

When you write a user-defined routine, 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 user-defined routine 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?

**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:

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

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

```sql
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 user-defined routine simpler to use.
Defining Routine Parameters

When you invoke a user-defined routine, 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 user-defined routines 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 3-17.

The way that you declare a user-defined routine depends on the language in which you write that routine.
Returning a Value

The parameters in an SPL routine must be declared with SQL data types. For more information, see “Executing an SPL Routine” on page 3-11.

For routines written in an external language (C or Java), you use the syntax of the external 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 4-23.) Therefore, you must ensure that these external data types are compatible with the SQL data types that the routine registration specifies.

For UDRs written in C, 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 in DataBlade API data types in the DataBlade API Programmer’s Manual.

Every UDR written in Java 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 Creating UDRs in Java.

Returning a Value

A common use of a user-defined routine is to return a value to the calling SQL statement. A user-defined routine 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 4-14.
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 contain SQL statements or access external files. For nonvariant functions, the database server might decide at execution time to cache the return values for expensive functions or to parallelize the query. You can create a functional index only on a nonvariant function. You specify a nonvariant function with the NOT VARIANT keywords.

Within the CREATE FUNCTION statement, you can specify VARIANT or NOT VARIANT in either of these places:

- As a routine modifier in the WITH clause
- For external routines, as part of the EXTERNAL NAME clause

If you specify these keywords in both of these locations, the two specifications must be the same.
Returning a Value

Returning Multiple Values with an OUT Parameter

By definition, an external function returns only one value. However, the database server provides the following methods for external routines to return multiple values:

- An OUT parameter and a statement-local variable (SLV) to return multiple values with a single invocation of the function
- An ITERATOR routine modifier to return a value multiple times through the automatic repeated execution of the function by the database server

OUT Parameters and Statement-Local Variables

To create an external function that returns more than one value, perform the following actions:

- Define the UDR to use an OUT parameter.
- Access the returned value within an SQL statement with a statement-local variable (SLV).

Tip: Although this section discusses the use of SLVs in the context of an OUT parameter in an external function, you can use SLVs and OUT parameters in SPL functions.

Defining User-Defined Functions That Have OUT Parameters

An OUT parameter is a function parameter whose argument value the database server passes by reference to the routine. The external function can change the value of this parameter and return a new value through the parameter. Each external function can have at most one OUT parameter.

You can define OUT parameters for the following external routines:

- User-defined routines written in C
  For information about writing database server functions with OUT parameters, see the chapter on writing database server routines in the DataBlade API Programmer's Manual.
- User-defined routines written in Java
  For information about writing database server functions in Java, refer to Creating UDRs in Java.
Returning a Value

To be called in a WHERE clause, a routine must return a BOOLEAN value. Therefore, only user-defined functions can be called in a WHERE clause.

The OUT parameter does not pass any data into the user-defined routine. At the time the function is called, the SLV does not reference any actual value. The database server passes only a pointer to some space where the function can store a value. The external function cannot retrieve values from this reserved space. By contrast, when a pure C-language program passes a pointer to a function, this pointer can point to a valid value, and the function can examine the current value before it modifies and returns the pointer.

Important: In the user-defined function, do not try to read from the address of the OUT parameter pointer because the value is meaningless.

To return a value from an external routine with an OUT parameter

1. Write an external routine that includes the OUT parameter in its parameter list.
   The OUT parameter must be passed by reference. That is, it must be a pointer to the argument value, not the value itself.

   For example, the following declaration of a C-language function allows you to return extra information through the y parameter:

   ```c
   mi_integer my_func(int x, int *y);
   ```

2. Register the user-defined routine with the OUT keyword before the appropriate parameter to indicate that the routine handles an OUT parameter.

   In the CREATE FUNCTION or CREATE PROCEDURE statement, the OUT keyword indicates that the last parameter is passed as a pointer. The OUT parameter must be the last parameter in the parameter list.

   For example, the following statement shows how you might register the `my_func()` function, which uses the y parameter of the function argument to return extra values:

   ```sql
   CREATE FUNCTION my_func(x INT, OUT y INT)
   RETURNING INT
   EXTERNAL NAME "/usr/lib/local_site.so"
   LANGUAGE C
   ```
Returning a Value

Referencing OUT Parameters of User-Defined Functions

A statement-local variable (SLV) transmits an OUT parameter from a user-defined function to other parts of an SQL statement. An SLV is local to the SQL statement; that is, it is only valid for the life of the SQL statement. It provides a temporary name by which to access the OUT parameter value.

In the SQL statement that calls the user-defined function, declare the SLV with the following syntax:

```sql
SLV_name # SLV_type
```

- `SLV_name` is the name of the SLV variable.
- `SLV_type` is the data type of the SLV variable

To obtain OUT parameters from an external routine

1. Declare an SLV to obtain the OUT parameter value when you invoke the external routine in a WHERE clause of an SQL expression.
2. Use the SLV in other parts of the SQL statement to access the OUT parameter value.

For example, the following SELECT statement declares an SLV `y` that is typed as an INTEGER in its WHERE clause and then accesses the `y` SLV in the WHERE clause:

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

For more information on the syntax and use of an SLV, see the description of function expressions within the Expression segment in the Informix Guide to SQL: Syntax.
Returning Values with an Iterator Function

By default, a user-defined function returns only one value; that is, it calculates its return value and returns only once to its calling SQL statement. Although the section “OUT Parameters and Statement-Local Variables” on page 4-8 describes how to return more than one value from a user-defined function, the function still returns all these values at the same time to the 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 User-Defined Routine” 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.

Tip: An iterator function is similar to an SPL function that contains the RETURN WITH RESUME statement.

Creating an Iterator Function

You can write iterator functions in C or in Java:

- 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.
- A Java iterator function uses the UDREnv interface to provide access to the user state.
Registering an Iterator Function

By default, an external function is not an iterator. To define an iterator function, you must register the function with the ITERATOR routine modifier. The following sample CREATE FUNCTION statement shows how to register the function $\text{TopK()}$ as an iterator function in C:

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

Invoking an Iterator Function

You can invoke an iterator function with the EXECUTE FUNCTION statement in 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
Adhering to Coding Standards

The SQL/PSM standard is available for UDR development. In addition, Informix publishes a collection of standards for DataBlade module development. These standards are available from the DataBlade Developers Program. 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: Informix recommends that you use the DataBlade Developers Kit (DBDK), Version 4.0 or later, to help write UDRs. Although it is possible to create a DataBlade without DBDK, it is not desirable. DBDK enforces standards on DataBlade modules that facilitate migration between different versions of the database server.

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 coding your user-defined routine:

- 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 writing UDRs in C, refer to the DataBlade API Programmer’s Manual. For information about writing UDRs in Java, refer to Creating UDRs in Java.

Creating a Shared-Object File

After you create an external routine, you must put it into a shared-object file so that the database server can access it when the routine is invoked. Shared-object modules must be owned by the user who runs the database server. In a production installation, the database server runs as user informix, and shared-object files must be owned by user informix.

Tip: An iterator SPL is not compiled into shared-object files. The database server includes the code to parse, compile, and execute SPL routines.

To create a shared-object module

1. Create a shared-object file and load the object file or files for the external routine or routines.
   Put related routines in the same shared-object file.
2. Put the shared-object file in a directory that user informix can read and set the permissions so that only the owner can write to the shared-object files.

Warning: A shared-object file must have write permission restricted to user informix. No other user can have write permission on a shared-object file. If a shared-object file has write permission set to the public (or any user other than informix), and someone tries to execute a UDR in the shared-object file, the database server issues error -9793 and writes a message in the log file.

C UDRs need to be compiled into shared-object files. The database server loads the shared-object file when the UDR is invoked. For more information about compiling and loading C UDRs, see the DataBlade API Programmer’s Manual.
Once you have created the shared-object module, you can register the external routine. For more information, see “Registering an External Routine” on page 4-18.

## Registering a User-Defined Routine

The database server recognizes the following SQL statements for the registration of user-defined routines 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.

### To register a user-defined routine

1. Ensure 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 routines, the statement specifies the location of the routine code (with an EXTERNAL NAME clause) and registers the routine.

## Setting Privileges for a Routine

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

- Database-level privilege
- Language-level privilege

After you register the UDR, you can assign routine-level-level privileges.
Setting Privileges for a Routine

Database-Level Privilege

Database-level privileges control the ability to extend the database by registering or dropping a user-defined routine. 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 12-7.

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 user-defined routine in a particular UDR language. This privilege needs to be granted by user informix or by another user who been granted 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.
Registering an SPL Routine

The following GRANT statement grants Usage privilege on UDRs written in Java 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 Informix Guide to Database Design and Implementation and the description of the GRANT statement in the Informix Guide to SQL: Syntax

Routine-Level Privilege

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

Registering an SPL Routine

You write an SPL routine in SPL, the internal language of the database server. 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.
Registering an SPL Routine

- 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 4-24
Figure 4-1 shows the parts of a CREATE FUNCTION statement that registers a user-defined function called `abs_eq()`.

<table>
<thead>
<tr>
<th>Routine name</th>
<th>Routine parameter list</th>
<th>Routine body</th>
<th>Routine modifiers (optional)</th>
<th>Return value (functions only)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE FUNCTION abs_eq(arg1 INTEGER, arg2 INTEGER)</td>
<td>RETURNS BOOLEAN WITH (NOT VARIANT)</td>
<td>DEFINE ret BOOLEAN; IF (arg1 &lt; 0) THEN LET arg1 = -arg1; END IF IF (arg2 &lt; 0) THEN LET arg2 = -arg2; END IF IF (arg1 = arg2) THEN LET ret = TRUE; ELSE LET ret = FALSE; END IF; RETURN ret;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When you create an SPL function, you can specify optional routine modifiers that affect how the database server executes the function. SPL procedures 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:

- ITERATOR
- INTERNAL
- NOT VARIANT
- INTERNAL

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.
Registering an External Routine

You can write an external routine in an external language that the database server supports. Once you have a shared-object file for the external routine, you register the routine and its shared-object file in the database.

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

- For UDRs written in C, the location is the pathname of the shared-object module that contains the external routine.
- For UDRs written in Java, the location is the JAR file.

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

The format of the external name often depends on the language in which the UDR is written. You specify the path of the shared-object file in the CREATE FUNCTION (or CREATE PROCEDURE) statement when you register the external routine.
Registering an External Routine

Figure 4-2 shows a CREATE FUNCTION that specifies a shared-object file for a C user-defined function.

**Important:** Before you can register an external routine, you must have Usage permission for the language in which you wrote the routine. For more information on the Usage privilege, see “Language-Level Privilege” on page 4-15.

**Registering an External Function**

To create an external function, write the body of the function in a language other than SPL and then use the CREATE FUNCTION statement to register the function. The RETURNING clause of CREATE FUNCTION specifies the return data type of the external function.

**Important:** To register an external UDR that is a function, you must use the CREATE FUNCTION statement. You cannot use the CREATE PROCEDURE statement to create an external function.

For example, the following CREATE FUNCTION statement registers an external 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;
```

In the preceding example, the END FUNCTION keywords are optional. For more information, see the CREATE FUNCTION statement in the *Informix Guide to SQL: Syntax*.

**Registering an External Procedure**

When you do not want your routine to return a value, you must create the routine as a procedure. To create an external procedure, write the body of the procedure in a language other than SPL and then use the CREATE PROCEDURE statement to register the procedure.
Registering an External Routine

The following example shows how to register an external procedure that is written in C:

```
CREATE PROCEDURE log_compare (arg1 udtype2, arg2 udtype2)
EXTERN NAME /
"/usr/lib/udtype1/lib/libbtype2.so(compare_n_insert)"
LANGUAGE C
END PROCEDURE
```

In the preceding example, the actual external procedure is located in a C routine called `compare_n_insert()`, which is located in the C-language shared library `/usr/lib/udtype1/lib/libbtype2.so`. If the EXTERNAL NAME clause does not specify an entry point within the library, the database server invokes the module at the default entry point, `log_compare()`.

The following example also includes the SPECIFIC keyword to create a function alias, `basetype2_lessthan`. Once you use the SPECIFIC keyword to create a routine alias, you can use that alias in DROP statements.

For information about the CREATE PROCEDURE statement, see the CREATE PROCEDURE statement in the Informix Guide to SQL: Syntax.

Registering an External Routine with Modifiers

When you create an external routine, 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.
## Modifiers in a UDR Written in C

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

<table>
<thead>
<tr>
<th>Routine Modifier</th>
<th>Description</th>
<th>Valid for</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASS</td>
<td>Specifies a virtual-processor class in which to run the UDR</td>
<td>Yes</td>
</tr>
<tr>
<td>COSTFUNC</td>
<td>Specifies the name of the cost function for this UDR</td>
<td>Yes</td>
</tr>
<tr>
<td>HANDLESNULLS</td>
<td>Specifies that the UDR can handle null arguments</td>
<td>Yes</td>
</tr>
<tr>
<td>INTERNAL</td>
<td>Specifies that the UDR is an internal routine; that is, that the routine is not available for use in an SQL or SPL statement</td>
<td>Yes</td>
</tr>
<tr>
<td>ITERATOR</td>
<td>Specifies that the UDR is an iterator function</td>
<td>Yes</td>
</tr>
<tr>
<td>NEGATOR</td>
<td>Specifies that the UDR is a negator function</td>
<td>Yes</td>
</tr>
<tr>
<td>NOT VARIANT</td>
<td>Specifies that all invocations of the UDR with the same arguments return the same value</td>
<td>Yes</td>
</tr>
<tr>
<td>PERCALL_COST</td>
<td>Specifies the cost of execution for the UDR</td>
<td>Yes</td>
</tr>
<tr>
<td>SELCONST</td>
<td>Specifies the selectivity of the UDR</td>
<td>Yes</td>
</tr>
</tbody>
</table>

(1 of 2)
Registering an External Routine

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

```sql
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
```

The HANDLESNULLS modifier indicates that the `basetype2_lessthan()` function (in the shared library `/usr/lib/basetype2/lib/libbtype2.so`) is written so that it can recognize when an SQL null argument is passed.

**Modifiers in a UDR Written in Java**

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

<table>
<thead>
<tr>
<th>Routine Modifier</th>
<th>Type of UDR</th>
<th>External Function</th>
<th>External Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASS</td>
<td>Access to JVP</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>ITERATOR</td>
<td>Iterator function</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

(1 of 2)
Registering an External Routine

<table>
<thead>
<tr>
<th>Routine Modifier</th>
<th>Type of UDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>HANDLESNULLS</td>
<td>UDR that handles SQL null values as arguments</td>
</tr>
<tr>
<td>NEGATOR</td>
<td>Negator function</td>
</tr>
<tr>
<td>PARALLELIZABLE</td>
<td>Parallelizable UDR</td>
</tr>
</tbody>
</table>

(2 of 2)

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:

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

The following CREATE FUNCTION statement registers the `func1` user-defined function:

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

Use the opaque SQL data type, POINTER, to specify a data type for an external 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 an external 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.

<table>
<thead>
<tr>
<th>UDR Information</th>
<th>CREATE Statement Syntax</th>
<th>Column of <code>sysprocedures</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Routine type: function or procedure</td>
<td>FUNCTION or PROCEDURE keyword</td>
<td><code>isproc</code></td>
</tr>
<tr>
<td>Routine name</td>
<td>After FUNCTION or PROCEDURE keyword</td>
<td><code>procname</code></td>
</tr>
<tr>
<td>Owner name</td>
<td>With the following syntax after FUNCTION or PROCEDURE keyword: <code>owner</code>. <code>owner.routine_name</code></td>
<td><code>owner</code></td>
</tr>
<tr>
<td>Specific name</td>
<td>SPECIFIC keyword</td>
<td><code>specificname</code></td>
</tr>
<tr>
<td>Routine parameters</td>
<td>Parameter list</td>
<td><code>numargs, paramstyle, paramtypes</code></td>
</tr>
<tr>
<td>Routine modifiers</td>
<td>WITH clause</td>
<td><code>variant, handlesnulls, percallcost, negator, selfunc, iterator, internal, class, stack, costfunc, selconst, parallelizable</code></td>
</tr>
<tr>
<td>Location of the routine (if it is external)</td>
<td>EXTERNAL NAME</td>
<td><code>externalname</code></td>
</tr>
<tr>
<td>Routine language</td>
<td>LANGUAGE</td>
<td><code>langid</code></td>
</tr>
</tbody>
</table>

The `sysprocedures` system catalog table assigns a `routine identifier` to each row, which uniquely identifies the associated UDR. The `procid` column of `sysprocedures` stores this routine identifier.
The information about the routine type, routine name, routine owner, and routine parameters make up the routine signature. You can use the SPECIFIC keyword of CREATE FUNCTION and CREATE PROCEDURE to assign a specific name to a particular routine signature. For more information, see “Assigning a Specific Routine Name” on page 3-15.

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

In This Chapter .................... 5-3
Understanding the Data Type System ................ 5-3
Understanding Data Types ................ 5-5
   Built-In Data Types .................. 5-6
   Extended Data Types ................. 5-8
      Complex Data Types ................ 5-9
      User-Defined Data Types .......... 5-10
      DataBlade Module Data Types ...... 5-12
Extending the Data Type System ................ 5-13
   Operations .......................... 5-13
   Casts ................................ 5-14
   Operator Classes .................... 5-14
      Providing Additional Operator Classes 5-14
      Extending Operator Classes .......... 5-15
   Optimizer Information ................ 5-15
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
- Extending data types

When you create a new data type or extend an existing data type, you use the user-defined routines that were introduced in Chapter 2, “Creating 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.
Understanding the Data Type System

- 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 method 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 operator class of a 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 user-defined data type, 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.

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

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>Stores binary data in random-access chunks.</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>Stores the Boolean values for true and false.</td>
</tr>
<tr>
<td>BYTE</td>
<td>Stores binary data in chunks that are not random access.</td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>Stores single-byte or multibyte sequences of characters, including letters, numbers, and symbols of fixed length. Collation is code-set dependent.</td>
</tr>
<tr>
<td>CHARACTER(n)</td>
<td>Is a synonym for CHAR.</td>
</tr>
<tr>
<td>CHARACTER VARYING(m,r)</td>
<td>Is an ANSI-compliant version of the VARCHAR data type.</td>
</tr>
<tr>
<td>CLOB</td>
<td>Stores text data in random-access chunks.</td>
</tr>
<tr>
<td>DATE</td>
<td>Stores a calendar date.</td>
</tr>
<tr>
<td>DATETIME</td>
<td>Stores a calendar date combined with the time of day.</td>
</tr>
<tr>
<td>DEC</td>
<td>Is a synonym for DECIMAL.</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>Stores numbers with definable scale and precision.</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>Behaves the same way as FLOAT.</td>
</tr>
<tr>
<td>FLOAT(n)</td>
<td>Stores double-precision floating-point numbers that correspond to the <strong>double</strong> data type in C (on most platforms).</td>
</tr>
<tr>
<td>INT</td>
<td>Is a synonym for INTEGER.</td>
</tr>
</tbody>
</table>

Figure 5-2
Built-In Data Types
### Built-In Data Types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT8</td>
<td>Stores an 8-byte integer value. These whole numbers can be in the range (-(2^{63}-1)) to (2^{63}-1).</td>
</tr>
<tr>
<td>INTEGER</td>
<td>Stores whole numbers from (-(2^{31}-1)) to (2^{31}-1).</td>
</tr>
<tr>
<td>INTERVAL</td>
<td>Stores a span of time.</td>
</tr>
<tr>
<td>LVARCHAR</td>
<td>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.</td>
</tr>
<tr>
<td>MONEY((p,s))</td>
<td>Stores a currency amount.</td>
</tr>
<tr>
<td>NCHAR((n))</td>
<td>Stores single-byte and multibyte sequences of characters, including letters, numbers, and symbols. Collation is locale dependent. For more information, see the <a href="#">Informix Guide to GLS Functionality</a>.</td>
</tr>
<tr>
<td>NUMERIC((p,s))</td>
<td>Is a synonym for DECIMAL.</td>
</tr>
<tr>
<td>NVARCHAR((m,r))</td>
<td>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 <a href="#">Informix Guide to GLS Functionality</a>.</td>
</tr>
<tr>
<td>REAL</td>
<td>Is a synonym for SMALLFLOAT.</td>
</tr>
<tr>
<td>SERIAL</td>
<td>Stores sequential integers; has the same range of values as INTEGER.</td>
</tr>
<tr>
<td>SERIAL8</td>
<td>Stores large sequential integers; has the same range of values as INT8.</td>
</tr>
<tr>
<td>SMALLFLOAT</td>
<td>Stores single-precision floating-point numbers that correspond to the float data type in C (on most platforms).</td>
</tr>
</tbody>
</table>

---

*(2 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
- User-defined data types
  - Opaque data types
  - Distinct data types
- DataBlade module 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 Informix Guide to SQL: Reference.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>Stores whole numbers from (-2^{15}.1) to (2^{15}.1).</td>
</tr>
<tr>
<td>TEXT</td>
<td>Stores text data in chunks that are not random access.</td>
</tr>
<tr>
<td>VARCHAR(m,r)</td>
<td>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.</td>
</tr>
</tbody>
</table>
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 provides.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST($e$)</td>
<td>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, $e$.</td>
</tr>
<tr>
<td>MULTISET($e$)</td>
<td>Stores a collection of values that have no implicit position and allows duplicate values. All elements have the same element type, $e$.</td>
</tr>
</tbody>
</table>
Extended Data Types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Named row type</td>
<td>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.</td>
</tr>
<tr>
<td>ROW</td>
<td>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.</td>
</tr>
<tr>
<td>SET(e)</td>
<td>Stores a collection of values that have no implicit position and does not allow duplicate values. All elements have the same element type, e.</td>
</tr>
</tbody>
</table>

User-Defined Data Types

Figure 5-4 summarizes the user-defined data types that the database server provides.

Data Type | Explanation                                                                                                                                 |
-----------|---------------------------------------------------------------------------------------------------------------------------------------------|
Distinct   | Stored in the same way 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.) |
**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 an Operation.”

- 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 11, “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. Once 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 *Informix Guide to SQL: Syntax.*
Extended Data Types

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 *Informix Guide to SQL: Syntax*. For more information, see Chapter 10, “Creating an Opaque Data Type,” and Chapter 11, “Writing Support Functions.”

DataBlade Module Data Types

In addition to the data types that you explicitly define, you can obtain new data types from an Informix DataBlade module. A DataBlade module is a collection of functions that describe special-purpose data types and all of their support functions. A DataBlade module can contain any or all of the previously described data types.

A DataBlade module might provide the data type, including any casts, operations, and secondary access methods, for an application-specific purpose. For example, a DataBlade module might contain the definitions needed for a spherical coordinate system. For more information on available DataBlade modules, consult your Informix 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 (or CREATE PROCEDURE) 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 user-defined routine that end users can use in SQL statements to perform some useful action. An end-user routine can be either a function (which returns a value) or a procedure (which does not return a value).

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 an Operation.”
Casts

A data type tells the database server which cast to use to convert the data type value to a different data 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

A data type tells the database server which operator class to associate with the data type values when they are stored by a secondary 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 only 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 9, "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 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 user-defined data types, you must provide the functions that collect the information. For more information on these functions, see Chapter 13, "Improving UDR Performance."
Extending an Operation
In This Chapter

This chapter describes how to extend an operation. An operation is a task that the database server performs on one or more values. The database server provides the following types of SQL-invoked functions that provide operations within SQL statements:

- Operator symbols (such as +, -, /, and *) and their associated operator functions
- Built-in functions such as \texttt{cos()} and \texttt{abs()}
- Aggregate functions such as \texttt{SUM} and \texttt{AVG}
- End-user routines

These functions handle the built-in data types. For a user-defined data type to use any of these functions, you can write a new function that has the same name but accepts the user-defined data type 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. The database server uses routine resolution to determine which function to execute, based on the data types of the arguments for the function. For more information on routine overloading and routine resolution, refer to "Understanding Routine Resolution" on page 3-11.
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 write a new version of an operator function to change the functionality of an operator or to provide the operator on a data type that is not built into the database server. The SQL user can then use the operator with user-defined data types 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.

The database server provides the following types of operators for expressions in SQL statements:

- Arithmetic operators usually operate on numeric values.
- Text operators operate on character strings.
- Relational operators operate on expressions of numeric and string values.
Arithmetic Operators

The database server provides operator functions for the arithmetic operators that Figure 6-1 shows.

<table>
<thead>
<tr>
<th>Arithmetic Operator</th>
<th>Operator Function</th>
<th>Number of Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ (binary)</td>
<td>plus()</td>
<td>2</td>
</tr>
<tr>
<td>- (binary)</td>
<td>minus()</td>
<td>2</td>
</tr>
<tr>
<td>*</td>
<td>times()</td>
<td>2</td>
</tr>
<tr>
<td>/</td>
<td>divide()</td>
<td>2</td>
</tr>
<tr>
<td>+ (unary)</td>
<td>positive()</td>
<td>1</td>
</tr>
<tr>
<td>- (unary)</td>
<td>negate()</td>
<td>1</td>
</tr>
</tbody>
</table>

Text Operators

The database server also provides operator functions for the text operators that Figure 6-2 shows.

<table>
<thead>
<tr>
<th>Text Operator</th>
<th>Operator Function</th>
<th>Number of Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIKE</td>
<td>like()</td>
<td>2 or 3</td>
</tr>
<tr>
<td>MATCHES</td>
<td>matches()</td>
<td>2 or 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

The database server provides operator functions for the relational operators that Figure 6-3 shows.

<table>
<thead>
<tr>
<th>Relational Operator</th>
<th>Operator Function</th>
<th>Number of Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>equal()</td>
<td>2</td>
</tr>
<tr>
<td>&lt;&gt; and !=</td>
<td>notequal()</td>
<td>2</td>
</tr>
<tr>
<td>&gt;</td>
<td>greaterthan()</td>
<td>2</td>
</tr>
<tr>
<td>&lt;</td>
<td>lessthan()</td>
<td>2</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greaterthanorequal()</td>
<td>2</td>
</tr>
<tr>
<td>&lt;=</td>
<td>lessthanorequal()</td>
<td>2</td>
</tr>
</tbody>
</table>

All operator functions in Figure 6-3 must return a Boolean value. For more information on relational operators, see the Relational Operator segment in the 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 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 the 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. For more information on the circle opaque data type, see “A Fixed-Length Opaque Data Type: circle” on page 10-32.
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 9-14.

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 9-5.

**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 operator function listed in Figures 6-1 through Figure 6-3 on pages 6-5 and 6-6. 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.
Built-In Functions

The database server provides special SQL-invoked functions, called *built-in functions*, that provide some basic mathematical operations.

### Built-In Functions That You Can Overload

Figure 6-4 shows the built-in functions that you can overload.

<table>
<thead>
<tr>
<th>Built-In Function</th>
<th>Number of Parameters</th>
<th>Parameter Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs()</td>
<td>1</td>
<td>Real number</td>
</tr>
<tr>
<td>mod()</td>
<td>2</td>
<td>Integer-number expression, integer-number expression</td>
</tr>
<tr>
<td>pow()</td>
<td>2</td>
<td>Real-number expression, real-number expression</td>
</tr>
<tr>
<td>root()</td>
<td>1 or 2</td>
<td>Real-number expression [ real-number expression]</td>
</tr>
<tr>
<td>round()</td>
<td>1 or 2</td>
<td>Expression [ literal integer]</td>
</tr>
<tr>
<td>sqrt()</td>
<td>1</td>
<td>Real-number expression</td>
</tr>
<tr>
<td>trunc()</td>
<td>1 or 2</td>
<td>Expression [ literal integer]</td>
</tr>
<tr>
<td>exp(), log(), logn()</td>
<td>1</td>
<td>Positive real-number expression</td>
</tr>
<tr>
<td>cos(), sin(), tan()</td>
<td>1</td>
<td>Numeric expression</td>
</tr>
<tr>
<td>acos(), asin(), atan()</td>
<td>1</td>
<td>numeric expression</td>
</tr>
<tr>
<td>atan2()</td>
<td>2</td>
<td>Numeric expression, numeric expression</td>
</tr>
</tbody>
</table>

(1 of 2)
# Built-In Functions That You Cannot Overload

The following table lists the built-in functions that you cannot overload.

<table>
<thead>
<tr>
<th>Built-In Function</th>
<th>Number of Parameters</th>
<th>Parameter Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>hex()</td>
<td>1</td>
<td>Integer expression</td>
</tr>
<tr>
<td>length(),</td>
<td>1</td>
<td>Character string</td>
</tr>
<tr>
<td>char_length(),</td>
<td></td>
<td></td>
</tr>
<tr>
<td>character_length(),</td>
<td></td>
<td></td>
</tr>
<tr>
<td>octet_length()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>extend()</td>
<td>1</td>
<td>Date/time expression</td>
</tr>
</tbody>
</table>

**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.

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

<table>
<thead>
<tr>
<th>Built-In Function</th>
<th>Parameter Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>descr()</td>
<td></td>
</tr>
<tr>
<td>volume()</td>
<td></td>
</tr>
<tr>
<td>family()</td>
<td></td>
</tr>
</tbody>
</table>
Overloading a Built-In Function

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

- The name of the built-in function must match the name that Figure 6-4 lists. The name is case insensitive; the abs() function is the same as the Abs() function.
- The built-in function must be one that you can overload.
- The built-in function must handle the correct number of parameters, and these parameters must be the correct data type.
- The built-in function must return the correct data type, where appropriate.

For more information on the built-in functions, see the Expression segment in the Informix Guide to SQL: Syntax.

Aggregate Functions

The database server provides special SQL-invoked functions, called aggregate functions, that take values that depend on all the rows that the query selects; they return information about these values, not the actual values themselves. The database server provides the following aggregate functions:

- COUNT returns the number of rows that satisfy the WHERE clause of a SELECT statement.
- AVG returns the average of all values that the query selects.
- MAX returns the largest value in the specified column or expression.
- MIN returns the smallest value in the specified column or expression.
- SUM returns the sum of all values in the specified column or expression.
You can create a new aggregate function that operates on built-in data types, or you can extend an existing aggregate function to operate on a new data type. For information on creating aggregate functions, see Chapter 8, “Creating User-Defined Aggregates.”

For information on using aggregate functions, see the Expression segment in the Informix Guide to SQL: Syntax.

---

End-User Routines

The database server provides special SQL-invoked functions and procedures, called end-user routines, that implement tasks you define for end users to use in expressions of SQL statements. These routines provide additional functionality that an end user might need to work with a built-in or user-defined data type. The two kinds of end-user routines are as follows:

- An end-user function returns a value to the SQL statement.
- An end-user procedure performs a task but does not return a value to the SQL statement.

Once you have written the end-user routines, you must register them with a CREATE FUNCTION or CREATE PROCEDURE statement. You must also use the GRANT statement to grant the Execute privilege to those users who have permission to call the end-user routines. For more information on how to write an end-user routine, see Chapter 4, “Developing a User-Defined Routine.”
Creating User-Defined Casts

In This Chapter ........................................... 7-3
Understanding Casts ................................. 7-3
  Built-In Casts ......................................... 7-3
  User-Defined Casts ................................. 7-4
    Opaque Data Types ............................... 7-4
    Distinct Data Types ......................... 7-4
    Named Row Types ............................... 7-4
  Casts That You Cannot Create.................. 7-5
Creating a User-Defined Cast ...................... 7-5
  Choosing the Kind of User-Defined Cast ....... 7-6
    Implicit Cast ................................... 7-6
    Explicit Cast ................................... 7-7
  Choosing the Cast Mechanism .................... 7-8
    Straight Cast ................................... 7-8
    Cast Function .................................. 7-9
  Example of a Cast Function ..................... 7-10
  Defining the Direction of the Cast ............... 7-11
Dropping a Cast ........................................ 7-13
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:

- Built-in casts
- User-defined casts

This chapter describes how to create casts for user-defined data types.

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 example, you can concatenate an integer value with a character value to form a label or a name. In that operation, the integer value is cast to a character value. However, you cannot add a datetime value to an integer because the two types of data are not comparable.

For more information on built-in casts, refer to the chapter on data types in the Informix Guide to SQL: Reference.
User-Defined Casts

A user-defined cast is a cast that you define to perform conversion from one user-defined data type 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 new 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 10-18.

Distinct Data Types

You cannot compare a distinct data type directly to its source data type. However, 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 vice versa. Although a distinct data type inherits the casts and functions of its source data type, the casts and functions that you define on a distinct data type are not available to its source data type. You might also create casts on distinct types to handle conversions between the new distinct data type and other data types in the database.

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 Informix Guide to Database Design and Implementation.

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.
Casts That You Cannot Create

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

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 Informix Guide to SQL: Syntax. For a general discussion of using casts, refer to the Informix Guide to Database Design and Implementation.

Important: The cast that you create must be unique within the database.

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

- The kind of user-defined cast to create
  The CREATE CAST statement specifies whether this cast is implicit or explicit.
Choosing the Kind of User-Defined Cast

The cast mechanism that the database server is to use to perform the data conversion

The CREATE CAST statement can optionally specify the name of the 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 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

The database server supports two kinds of user-defined casts:

- Implicit cast
  The database server automatically invokes an implicit cast to perform conversions between two data types.

- 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 user-defined data types (such as opaque data types, distinct data types, and row types). The database server automatically invokes an implicit cast when it performs the following tasks:

- It passes arguments of one data type to a user-defined routine whose parameters are of another data type.
- It evaluates expressions and needs to operate on two similar data types.
Choosing the Kind of User-Defined Cast

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 user-defined data types (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. The following CREATE CAST statement creates an explicit cast from the percent data type to the INTEGER data type:

```
CREATE EXPLICIT CAST (percent AS INTEGER)
```
Choosing the Cast Mechanism

When you do not specify the IMPLICIT or EXPLICIT keyword, you create an explicit cast because the default is explicit. The following CREATE CAST statement also creates an explicit cast from percent to INTEGER:

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

Choosing the Cast Mechanism

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 user-defined data type `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.
Choosing the Cast Mechanism

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. You can write cast functions in SPL, in C, or in Java.

   If you write an external function, you must write the function in an external file, compile the code, and load it into a shared-object file. If you write an SPL function, you define and register the function with the CREATE FUNCTION statement.

   For information about creating a shared-object file, refer to “Creating a Shared-Object File” on page 4-14.

2. Register the cast function with the CREATE FUNCTION statement.
   
   If you create an SPL function, the CREATE FUNCTION statement contains the actual SPL statements of the cast function and registers the function. If you create an external function, CREATE FUNCTION specifies the name of the shared-object file that contains the compiled code and just registers the function.

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.
Choosing the Cast Mechanism

Example of a Cast Function

For example, suppose you want to compare values of two opaque data types, `int_type` and `float_type`. The CREATE FUNCTION statement in Figure 7-1 creates and registers an SPL function, `int_to_float()`, that takes an `int_type` value as an argument and returns a value of data type `float_type`.

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

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:
   
   ```sql
   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:

   ```sql
   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 “In Input and Output Support Functions” on page 11-24.
Once you create this cast function, you 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.

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

Once 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 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 user-defined data type called percent:

```sql
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:

```sql
INSERT INTO table1 (ft_col) VALUES (it_val::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.

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

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.

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

```sql
INSERT INTO table1 (it_col) VALUES (ft_val::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 of the cast or the DBA to drop its definition from the database.

**Warning:** Do not drop the built-in casts, which the 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.

If you are the owner (the person who created the cast) or the DBA, the following statements remove the casts between the DECIMAL and `percent` data types from the database:

```sql
DROP CAST (decimal AS percent);
DROP CAST (percent AS decimal);
```

Dropping a cast has no effect on the function associated with the cast. 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 *Informix Guide to SQL: Syntax*.
Creating User-Defined Aggregates

In This Chapter .................... 8-3
Extending Existing Aggregates. .......................... 8-4
Overloading Operators for Built-In Aggregates .......................... 8-4
Extending an Aggregate .............................................. 8-5
Example of Extending a Built-In Aggregate ................. 8-6
Creating User-Defined Aggregates ............................. 8-7
Support Functions ....................................................... 8-8
  INIT Function ...................................................... 8-9
  ITER Function ...................................................... 8-9
  FINAL Function ..................................................... 8-10
  COMBINE Function .................................................. 8-10
Resolving the Support Functions. ........................................ 8-11
Support-Function States .............................................. 8-12
Using External Support Functions ................................. 8-13
Preparing a User-Defined Aggregate ............................... 8-14
Example of a User-Defined Aggregate ............................. 8-15
  Using User-Defined Data Types with User-Defined Aggregates . 8-16
  Omitting Support Functions ......................................... 8-17
Managing Aggregates ................................................ 8-19
  Parallel Execution of Aggregates ................................. 8-19
Privileges for User-Defined Aggregates .......................... 8-19
  Aggregate Information in the System Catalog ................. 8-20
  Aggregate Information from the Command Line ................. 8-20
Dropping an Aggregate ................................................ 8-20
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 is provided by the database server, such as COUNT, SUM, or AVG. You can extend the built-in aggregates for use with user-defined data types.

- **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. Once you have created 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 using aggregates in SELECT statements, refer to the *Informix Guide to SQL: Tutorial*. For information about the syntax of aggregates, refer to the *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 user-defined data types. To extend a built-in aggregate, you must create user-defined routines 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 user-defined data type, you need to overload only the plus() operator.

<table>
<thead>
<tr>
<th>Aggregate</th>
<th>Required Operators</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG</td>
<td>plus(udt, udt), divide(udt, integer)</td>
<td>Return type of divide()</td>
</tr>
<tr>
<td>DISTINCT (or UNIQUE)</td>
<td>compare(udt, udt)</td>
<td>Boolean</td>
</tr>
<tr>
<td>COUNT</td>
<td>-- (no new operators required)</td>
<td>Integer</td>
</tr>
<tr>
<td>MAX</td>
<td>greaterthanorequal(udt, udt)</td>
<td>Boolean</td>
</tr>
<tr>
<td>MIN</td>
<td>lesthанorequal(udt, udt)</td>
<td>Boolean</td>
</tr>
<tr>
<td>RANGE</td>
<td>lessthanorequal(udt, udt),</td>
<td>Return type of minus()</td>
</tr>
<tr>
<td></td>
<td>greaterthanorequal(udt, udt),</td>
<td></td>
</tr>
<tr>
<td></td>
<td>minus(udt, udt)</td>
<td></td>
</tr>
</tbody>
</table>
Extending an Aggregate

The database server uses the `compare()` operator for indexing as well as for DISTINCT and UNIQUE aggregations. You must create the `compare()` function as an external function. You cannot use the `compare()` operator with user-defined types that are not hashable. For a description of hashable user-defined types, refer to “Hashable Data Types” on page 10-24.

## Extending an Aggregate

When you extend a built-in aggregate to include a user-defined data type, you do not use the CREATE AGGREGATE statement because the aggregate itself already exists.

### To extend a built-in aggregate

1. Develop 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 4-15.

After you have registered the functions that overload the binary operators, you can use the built-in aggregates in an SQL statement.

---

### Extending an Aggregate

<table>
<thead>
<tr>
<th>Aggregate</th>
<th>Required Operators</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUM</td>
<td><code>plus(udt, udt)</code></td>
<td>Return type of <code>plus()</code></td>
</tr>
<tr>
<td>STDEV</td>
<td><code>times(udt, udt), divide(udt, integer), plus(udt, udt), minus(udt, udt), sqrt(udt)</code></td>
<td>Return type of <code>divide()</code></td>
</tr>
<tr>
<td>VARIANCE</td>
<td><code>times(udt, udt), divide(udt, integer), plus(udt, udt), minus(udt, udt)</code></td>
<td>Return type of <code>divide()</code></td>
</tr>
</tbody>
</table>

(2 of 2)
Example of Extending a Built-In Aggregate

For the syntax of the CREATE FUNCTION statement, see the Informix Guide to SQL: Syntax. For more information about writing overloaded functions, refer to “Overloading Routines” on page 3-13. For information about writing functions in external languages, refer to the DataBlade API Programmer’s Manual or Creating UDRs in Java.

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

For the syntax of the CREATE AGGREGATE statement, see the *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.

<table>
<thead>
<tr>
<th>Function Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>INIT</td>
<td>Initializes the data structures required for computing the aggregate.</td>
</tr>
<tr>
<td>ITER</td>
<td>Merges a single (row) value with the previous partial result.</td>
</tr>
<tr>
<td>COMBINE</td>
<td>Merges one partial result with another partial result, thus allowing parallel execution of the aggregate.</td>
</tr>
<tr>
<td>FINAL</td>
<td>Converts the partial result into the final value. It can perform clean-up operations and release resources.</td>
</tr>
</tbody>
</table>

You can write the support functions in SPL, C, or Java. For information about SPL, refer to the Informix Guide to SQL: Syntax. For information about writing functions in external languages, refer to the DataBlade API Programmer's Manual or Creating UDRs in Java.

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.

```sql
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.
Support Functions

**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.

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.

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.

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.

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.
Support Functions

The ITER function obtains the state of the aggregate computation from its state argument.

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.

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.
Resolving the Support Functions

The COMBINE function is required 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.

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 user-defined routines.

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 user-defined routine:
   \[ \text{init_func (dt_agg, dt_setup)} \]
   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 user-defined routine:
   \[ \text{iter_func (state_type, dt_agg)} \]
   The return type of the ITER function should be the state type.

3. For the COMBINE function, resolve the following user-defined routine:
   \[ \text{comb_func (state_type, state_type)} \]
   The return type of the COMBINE function should be the state type.

4. If the FINAL function is specified, resolve the following user-defined routine:
   \[ \text{final_func (state_type)} \]
   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.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>comb_func</td>
<td>Name of the COMBINE function</td>
</tr>
<tr>
<td>dt_agg</td>
<td>Data type of the first argument of the aggregate</td>
</tr>
<tr>
<td>dt_setup</td>
<td>Data type of the second, or setup, argument of the aggregate</td>
</tr>
<tr>
<td>final_func</td>
<td>Name of the FINAL function</td>
</tr>
</tbody>
</table>

(1 of 2)
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.

**Using External Support Functions**

When you write external 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.
Preparing a User-Defined Aggregate

To create a user-defined aggregate, follow these steps:

1. Write the functions that support the aggregate.
   You can write SPL support functions or external support functions.
2. Register the support function with the CREATE FUNCTION statement.
   For more information, refer to “Registering a User-Defined Routine” on page 4-15.
3. Register the aggregate with the CREATE AGGREGATE statement.

Once you register the aggregate, you can use the aggregate in an SQL statement.

For the syntax of the CREATE FUNCTION and CREATE AGGREGATE statements, see the 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.

```sql
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.

```sql
SELECT SUMSQ(b) FROM c_test;
```
Example of a User-Defined Aggregate

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:

```sql
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:

```sql
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
```
Example of a User-Defined Aggregate

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.

```sql
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 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 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.
Extending an Operator Class

In This Chapter ............................................. 9-3
Using Operator Classes .................................... 9-3
  Secondary Access Methods. .............................. 9-4
    Generic B-Tree Index ............................... 9-4
    R-Tree Index ........................................ 9-5
  Other User-Defined Secondary Access Methods .... 9-5
Operator Classes ........................................... 9-5
  Generic B-Tree Operator Class ...................... 9-6
  R-Tree Index Operator Class ......................... 9-8
Extending an Existing Operator Class .................. 9-9
  Extensions of the btree_ops Operator Class ........ 9-10
  Reasons for Extending btree_ops ..................... 9-12
Creating an Operator Class ............................. 9-15
  Creating a New B-Tree Operator Class .............. 9-16
  Creating an Absolute-Value Operator Class ......... 9-18
  Defining an Operator Class for Other Secondary Access Methods 9-21
Dropping an Operator Class ............................. 9-22
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 user-defined data type, you must extend an operator class.
- User-defined operator classes
  When one of the existing secondary access methods cannot easily index a user-defined data type, 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 secondary access methods and operator classes, 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 user-defined data types. 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 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 user-defined data types, 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 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 user-defined data type 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 *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*. 

---

**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*. 

---

**Extending an Operator Class**

9-5
Operator Classes

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 9-4 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 \texttt{btree} secondary access method, see “Generic B-Tree Index” on page 9-4.

\textbf{B-Tree Strategy Functions}

The \texttt{btree_ops} operator class defines the following strategy functions for the \texttt{btree} access method:

- \texttt{lessthan (<)}
- \texttt{lessthanorequal (<=)}
- \texttt{equal (=)}
- \texttt{greaterthanorequal (>=)}
- \texttt{greaterthan (>)}

These strategy functions are all \textit{operator functions}. That is, each function is associated with an operator symbol; in this case, with a relational-operator symbol. For more information on relational-operator functions, see “Relational Operators” on page 6-6.

\textbf{B-Tree Support Function}

The \texttt{btree_ops} operator class has one support function, a comparison function called \texttt{compare()}. The \texttt{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 \textit{equal} to the second argument
- A value less than 0 when the first argument is \textit{less than} the second argument
- A value greater than 0 when the first argument is \textit{greater than} the second argument

The B-tree secondary access method uses the \texttt{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 \texttt{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.
Operator Classes

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 “Comparing Data” on page 10-23.

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 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 user-defined data type, 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 user-defined data type.
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 user-defined data type, 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.

Once you determine how you want to implement the relational operators for a user-defined data type, 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 user-defined data type in their parameter list.
   
   You can write strategy functions as external functions or as SPL functions. 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 user-defined data type, 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 have already registered the relational-operator functions, you do not need to reregister them as strategy functions.
3. Write an external function for the B-tree support function, \( \texttt{compare()} \), that accepts the user-defined data type in its parameter list. (The \( \texttt{compare()} \) function cannot be in SPL.)

Compile this function and store it in a shared-object file. For more information, refer to Chapter 4, “Developing a User-Defined Routine.”

The \( \texttt{compare()} \) function also provides support for a user-defined data type 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 user-defined data type, 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 9-7. For information on relational operators for an opaque data type, see “Comparing Data” on page 10-23.

After you have registered the support function, you can use the CREATE INDEX statement to create a B-tree index on the column of the table that contains the user-defined data type. 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 user-defined data type.

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. For an example of how to extend the generic B-tree index for an opaque-type column, see “A Fixed-Length Opaque Data Type: circle” on page 10-32.

**Important:** When the database server uses a generic B-tree index to process an ORDER BY clause in a SELECT statement, the database server uses the \( \texttt{btree\_ops} \) support function called \( \texttt{compare()} \). However, the optimizer does not use the B-tree index to perform ORDER BY if the index does not use the \( \texttt{btree\_ops} \) operator class.
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 9-16.

**Reasons for Extending btrees_ops**

The strategy functions of `btrees_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 9-7.) The generic B-tree index handles only the built-in data types. When you write relational-operator functions that handle a new user-defined data type, you extend the generic B-tree so that it can handle the user-defined data type 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 user-defined data type, 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 user-defined data type, 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 the built-in data types. (For more information on the built-in data types, see the chapter on data types in the *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 user-defined data type, you must ensure that the relational-operator functions can compare two values of the user-defined data type. Otherwise, the comparison cannot occur, and the user-defined data type 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. For more information on the `circle` opaque type, see “A Fixed-Length Opaque Data Type: `circle`” on page 10-32.
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.

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 Informix Guide to GLS Functionality.

For some user-defined data types, 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 lexicographical sequence to some other sequence.

**Tip:** When you extend an operator class, you can change the sort order for a user-defined data type. 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 9-16.

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 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 9-10 to extend the btree_ops operator class.
Creating an Operator Class

To support the ScottishName data type

1. Prepare and register external functions for the strategy functions that handle the ScottishName data type: lessThan(), lessThanEqual(), equal(), greaterThan(), and greaterThanEqual().

   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.

```sql
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 cnme_ix index to evaluate the following query:

```sql
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
Creating a New B-Tree Operator Class

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:

```
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 9-4.

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 9-14.) 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.
Creating a New B-Tree Operator Class

To create a new operator class for a generic B-tree index

1. Write external (C or Java) functions for five new 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 9-7.

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 external functions 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 user-defined data types. However, the return type must be integer. For more information on support functions, see “B-Tree Support Function” on page 9-7.

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
Creating an Absolute-Value 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 `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 9-10.

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.

\(-4, 2, -3, -1\)

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.”
Creating an Absolute-Value Operator Class

2. Register the five new strategy functions with the CREATE FUNCTION statement.

The following CREATE FUNCTION statements register the five strategy functions that handle the INTEGER data type:

```sql
CREATE FUNCTION abs_lt(integer, integer)
RETURNS boolean
EXTERNAL NAME '/lib/absbtree.so(abs_less than)'
LANGUAGE C NOT VARIANT;

CREATE FUNCTION abs_lte(integer, integer)
RETURNS boolean
EXTERNAL NAME '/lib/absbtree.so(abs_less than or equal)'
LANGUAGE C NOT VARIANT;

CREATE FUNCTION abs_eq(integer, integer)
RETURNS boolean
EXTERNAL NAME '/lib/absbtree.so(abs_equal)'
LANGUAGE C NOT VARIANT;

CREATE FUNCTION abs_gte(integer, integer)
RETURNS boolean
EXTERNAL NAME '/lib/btree1.so(abs_greater than or equal)'
LANGUAGE C NOT VARIANT;

CREATE FUNCTION abs_gt(integer, integer)
RETURNS boolean
EXTERNAL NAME '/lib/absbtree.so(abs_greater than)'
LANGUAGE C NOT VARIANT;
```

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.

The following CREATE FUNCTION statement registers the support function that handles the INTEGER data type:

```sql
CREATE FUNCTION abs_cmp(integer, integer)
RETURNS integer
EXTERNAL NAME '/lib/absbtree.so(abs_compare)'
LANGUAGE C NOT VARIANT;
```
5. Create the new \texttt{abs\_btree\_ops} operator class for the B-tree secondary access method.

\begin{verbatim}
CREATE OPCLASS abs_btree_ops FOR btree  
  STRATEGIES (abs_lt, abs_lte, abs_eq, abs_gte, abs_gt)  
  SUPPORT (abs_cmp);
\end{verbatim}

You can now create a B-tree index on an INTEGER column and associate the new operator class with this column.

\begin{verbatim}
CREATE TABLE cust_tab  
  (    cust_name varchar(20),    cust_num integer    ...  
);  
CREATE INDEX c_num1_ix  
  ON cust_tab (cust_num abs_btree_ops);
\end{verbatim}

The \texttt{c_num1\_ix} index uses the new operator class, \texttt{abs\_btree\_ops}, for the \texttt{cust\_num} column. An end user can now use the absolute value functions in SQL statements, as in the following example:

\begin{verbatim}
SELECT * FROM cust_tab WHERE abs_lt(cust_num, 7)
\end{verbatim}

In addition, because the \texttt{abs\_lt()} function is part of an operator class, the query optimizer can use the \texttt{c_num1\_ix} index when it looks for all \texttt{cust\_tab} rows with \texttt{cust\_num} values between -7 and 7. A \texttt{cust\_num} value of -8 does \textit{not} satisfy this query.

The default operator class is still available for indexes. The following CREATE INDEX statement defines a second index on the \texttt{cust\_num} column:

\begin{verbatim}
CREATE INDEX c_num2_ix ON cust_tab (cust_num);
\end{verbatim}

The \texttt{c_num2\_ix} index uses the default operator class, \texttt{btree\_ops}, for the \texttt{cust\_num} column. The following query uses the operator function for the default \textit{less than} (<) operator:

\begin{verbatim}
SELECT * FROM cust_tab WHERE lessthan(cust_num, 7)
\end{verbatim}

The query optimizer can use the \texttt{c_num2\_ix} index when it looks for all \texttt{cust\_tab} rows with \texttt{cust\_num} values less than 7. A \texttt{cust\_num} value of -8 \textit{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 "Informix Guide to SQL: Reference."

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 9-16.) 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 on how to create this operator class, see “Creating a New B-Tree Operator Class” on page 9-16.) 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.

Once the preceding conditions are met, 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.
## Creating an Opaque Data Type

In This Chapter .................................................. 10-3

Opaque Data Types ............................................. 10-3
  The Internal Structure ..................................... 10-4
    A Fixed-Length Opaque Data Type .................. 10-5
    A Varying-Length Opaque Data Type ................ 10-5
  Support Functions ........................................ 10-6
  Additional SQL-Invoked Routines ..................... 10-7
    Built-In Functions .................................... 10-7
    Operator Functions .................................. 10-8
    Aggregate Functions .................................. 10-8
    Statistics-Collecting Routines ..................... 10-8
    End-User Routines ..................................... 10-9
  Advantages of Opaque Data Types ..................... 10-9

Creating an Opaque Data Type ............................. 10-10
  Creating the Internal Structure in C ................. 10-10
    Data Type Size ....................................... 10-11
    Memory Alignment ..................................... 10-12
    Parameter Passing .................................... 10-14
  Registering the Opaque Data Type with the Database .. 10-14
    Registering the Opaque Data Type .................. 10-15
    Registering Support Functions ...................... 10-16
    Creating Casts for Opaque Data Types .............. 10-18

Granting Privileges for an Opaque Data Type ........... 10-19
  Privileges on the Opaque Data Type ................ 10-19
  Privileges on the Support Functions ................ 10-20
Extending Informix Dynamic Server 2000

Creating SQL-Invoked Functions ........................................... 10-21
Operating on Data ............................................................... 10-21
  Operator Functions for Opaque Data Types ....................... 10-21
  Built-in Functions for Opaque Data Types ....................... 10-22
Comparing Data ............................................................... 10-23
  Conditions for Opaque Data Types ................................. 10-23
  Relational Operators for Opaque Data Types ................... 10-23
  Comparison Function for Opaque Data Types ................... 10-25

Customizing Access Methods ................................................. 10-27
Using the Generic B-Tree .................................................. 10-27
Using Other Access Methods ............................................. 10-28
  Indexing Spatial Data .................................................. 10-29
  Indexing Other Types of Data ...................................... 10-29

Other Operations on Opaque Data Types ................................ 10-29
  Accessing an Opaque Data Type .................................... 10-29
  Dropping an Opaque Data Type ..................................... 10-30

A Fixed-Length Opaque Data Type: circle .............................. 10-32
  Creating the Fixed-Length Internal Structures .................. 10-32
  Writing the circle Support Functions ............................. 10-33
  Registering the circle Opaque Data Type ....................... 10-34
  Granting Privileges on the circle Data Type ................... 10-35
  Creating SQL-Invoked Functions for the circle Data Type .... 10-36
  Providing Indexes for the circle Data Type .................... 10-37
  Using the circle Opaque Data Type ................................. 10-39

A Varying-Length Opaque Data Type: image ............................ 10-39
  Creating the Varying-Length Internal Structure ................ 10-40
  Writing the image Support Functions .............................. 10-40
  Registering the image Opaque Data Type ......................... 10-42
  Registering Other Support Functions for the image Data Type 10-44
  Granting Privileges on the image Data Type ..................... 10-46
  Creating SQL-Invoked Functions for the image Data Type ...... 10-46
  Providing Indexes for the image Data Type ..................... 10-47
In This Chapter

This chapter provides the following information about opaque data types:

- A definition of an opaque data type
- The steps to create an opaque data type
- The way to access an opaque data type from client applications
- The steps to drop an opaque data type

It then provides two sample opaque data types, a fixed-length and a varying length opaque data type.

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 serves as the internal storage of the opaque data type
- Support functions that allow the database server to interact with this internal structure
- 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 10-10.

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. (See “Support Functions” on page 10-6.) You create the internal structure as a data structure in the external language.

You can define an internal structure that supports either of the following kinds of opaque types:

- A fixed-length opaque data type
- A varying-length opaque data type
Creating an Opaque Data Type

The Internal Structure

**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. For an example of this data type, see “A Fixed-Length Opaque Data Type: circle” on page 10-32.

You provide the size when you register the opaque data type in the database. For more information, see “Data Type Size” on page 10-11.

**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. For an example of this data type, see “A Varying-Length Opaque Data Type: image” on page 10-39.

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 10-11.

A multirepresentational data type is a varying-length data type that is stored in the internal structure of the data type if the length of the data is below a specified threshold and in a smart large object if the length of the data is above the threshold. When you insert a value into this 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. For information about using multirepresentational data types, refer to the DataBlade API Programmer’s Manual.
Support Functions

The support functions of an opaque data type tell the database server how to interact with the internal structure of the opaque data type. (For more information, see “The Internal Structure” on page 10-4). Because an opaque data type is encapsulated, the database server has no knowledge of the internal structure. It is the support functions, which you write, that interact with this internal structure. The database server calls these support functions to perform the interactions.

The following table summarizes the support functions of the opaque data type.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>Converts the opaque-type data from its external representation to its internal representation.</td>
</tr>
<tr>
<td>output</td>
<td>Converts the opaque-type data from its internal representation to its external representation.</td>
</tr>
<tr>
<td>receive</td>
<td>Converts the opaque-type data from its internal representation on the client computer to its internal representation on the server computer.</td>
</tr>
<tr>
<td>send</td>
<td>Converts the opaque-type data from its internal representation on the database server computer to its internal representation on the client computer.</td>
</tr>
<tr>
<td>import</td>
<td>Performs any tasks needed to process an opaque data type when a bulk copy imports the opaque data type in its external representation.</td>
</tr>
<tr>
<td>export</td>
<td>Performs any tasks needed to process an opaque data type when a bulk copy exports the opaque data type in its external representation.</td>
</tr>
<tr>
<td>importbinary</td>
<td>Performs any tasks needed to process an opaque data type when a bulk copy imports the opaque data type in its internal representation.</td>
</tr>
<tr>
<td>exportbinary</td>
<td>Performs any tasks needed to process an opaque data type when a bulk copy exports the opaque data type in its internal representation.</td>
</tr>
</tbody>
</table>

(1 of 2)
Additional SQL-Invoked Routines

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>assign</td>
<td>Performs any tasks needed to process an opaque data type before storing it to disk.</td>
</tr>
<tr>
<td>destroy</td>
<td>Performs any tasks needed to process an opaque data type necessary before the database server removes a row that contains the data type.</td>
</tr>
<tr>
<td>lohandles</td>
<td>Returns a list of the smart large objects embedded in the opaque data type.</td>
</tr>
<tr>
<td>compare</td>
<td>Compares two values of opaque-type data during a sort.</td>
</tr>
</tbody>
</table>

For more information on the support functions for opaque data types, see Chapter 11, “Writing Support Functions.”

Additional SQL-Invoked Routines

The 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 user-defined routines to provide the following kinds of functions for your opaque data type:

- Built-in functions
- Aggregate functions
- Operator functions
- Statistics-collecting routines
- End-user routines

Built-In Functions

A built-in function is a predefined function that the database server provides for use in an SQL expression. The database server supports the built-in functions on the built-in data types. For a built-in function to operate on the opaque data type, you must write a version of the function that accepts the opaque data type in its parameter list.
For general information about these built-in functions, see “Built-In Functions” on page 6-8. For information on how to implement a built-in function on an opaque data type, see “Built-in Functions for Opaque Data Types” on page 10-22.

**Operator Functions**

An *operator function* is a user-defined function that has a corresponding operator symbol. For an operator function to operate on the opaque data type, you must write a version of the function that accepts the opaque data type in its parameter list.

For general information about the operator functions that the database server provides, see “Operators and Operator Functions” on page 6-4. For information on how to implement an operator function on an opaque data type, see “Operator Functions for Opaque Data Types” on page 10-21.

**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 extending the built-in aggregates, refer to “Extending Existing Aggregates” on page 8-4. For information about creating new aggregate functions, refer to “Creating User-Defined Aggregates” on page 8-7. For information about using aggregate functions, see the Expression segment in the *Informix Guide to SQL: Syntax*.

**Statistics-Capturing Routines**

The UPDATE STATISTICS statement calls the `statcollect()` function to collect statistics for the optimizer to use. The `statcollect()` function formats information from `statcollect()` so that the database server can display it. For information about `statcollect()`, 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 of 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 how to write 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 in C or Java.
2. Write the support functions as external functions.
3. Register the opaque data type in the database with the CREATE OPAQUE TYPE statement.
4. Register the support functions of the opaque data type with the CREATE FUNCTION statement.
5. Provide access to the opaque data type and its support functions with the GRANT statement.
6. Write any SQL-invoked functions that are needed to support the opaque data type.
7. Provide any customized secondary access methods that the opaque data type might need.

The following sections describe each of these steps.

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 typedefs that the DataBlade API supplies for those fields whose size might vary by platform. Use of these typedefs, such as mi_integer and mi_float, improves the portability of the opaque data type. For more information on these data types, see the DataBlade API Programmer’s Manual.

The internal structure uniquely names the opaque data type. Informix recommends that you develop a unique prefix (such as the mi_prefix used in the previous paragraph) for the opaque data type. You can use this prefix with each member of the internal structure and the structure itself. For opaque data types, Informix appends the string _t to the structure name. For example, the circle_t data structure holds the values for the circle opaque data type.
Creating an Opaque Data Type

Creating the Internal Structure in C

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 user-defined routines

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.

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 10-12.
For an example of a fixed-length opaque data type, see “A Fixed-Length Opaque Data Type: circle” on page 10-32.

A Varying-Length Opaque Data Type

When you specify the VARIABLE keyword for the INTERNALLENGTH modifier, you create a varying-length opaque data type. By default, the 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. You can specify a maximum length of up to 32 kilobytes. 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.

For example, the following CREATE OPAQUE TYPE statement defines a varying-length opaque data type called \texttt{var\_type} whose maximum size is 4 kilobytes:

\begin{verbatim}
CREATE OPAQUE TYPE var_type (INTERNALLENGTH=VARIABLE,
MAXLEN=4096);
\end{verbatim}

Only the last member of the internal structure can be of varying size.

For an example of a varying-length opaque data type, see “A Varying-Length Opaque Data Type: image” on page 10-39.

Memory Alignment

When the database server passes the data type to a user-defined routine, 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.

<table>
<thead>
<tr>
<th>ALIGNMENT Value</th>
<th>Meaning</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Align structure on single-byte boundary.</td>
<td>Structures that begin with 1-byte quantities</td>
</tr>
<tr>
<td>2</td>
<td>Align structure on 2-byte boundary.</td>
<td>Structures that begin with 2-byte quantities such as \texttt{mi_unsigned_smallint}</td>
</tr>
<tr>
<td>4</td>
<td>Align structure on 4-byte boundary.</td>
<td>Structures that begin with 4-byte quantities such as float or \texttt{mi_unsigned_integer}</td>
</tr>
<tr>
<td>8</td>
<td>Align structure on 8-byte boundary.</td>
<td>Structures that contain members of the C double data type</td>
</tr>
</tbody>
</table>

Structures that begin with single-byte characters, \texttt{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 \texttt{LongLong}, of 18 bytes that must be aligned on a 1-byte boundary:

```sql
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 user-defined routine in either of the following ways:

- **Pass by value** passes the actual value of the opaque data type to a user-defined routine.
- **Pass by reference** passes a pointer to the value of the opaque data type to a user-defined routine.

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:

```sql
CREATE OPAQUE TYPE two_bytes (INTERNALLENGTH=2 PASSEDBYVALUE);
```

Registering the Opaque Data Type with the Database

Once you have created 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 the opaque data type as a data type.
- The CREATE FUNCTION statement registers a support function.
- The CREATE CAST statement registers the support functions as cast functions.

**Important:** These SQL statements register the opaque data type in the current database. For users of another database to have access to the opaque data type, you must run the CREATE OPAQUE TYPE, CREATE FUNCTION, and CREATE CAST statements when this second database is the current database.


**Registering the Opaque Data Type**

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 10-10.

- **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 10-10.

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 *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 10-19.
Registering the Opaque Data Type with the Database

Once you have registered the opaque data type, you can use the data type in SQL statements and in user-defined routines. For more information on the syntax of the CREATE OPAQUE TYPE statement, see the description in the Informix Guide to SQL: Syntax.

Registering Support Functions

Use the CREATE FUNCTION statement to register a support function with the database. The following example shows the CREATE FUNCTION syntax for a support function written in C:

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

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 to indicate that a support function does not return different results with different arguments: NOT VARIANT

When you register a support function with the CREATE FUNCTION syntax for external functions, use the appropriate SQL data types for `parameter_list` and `ret_type`, as follows.

<table>
<thead>
<tr>
<th>Support Function</th>
<th>Parameter Type</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>LVARCHAR</td>
<td>opaque data type</td>
</tr>
<tr>
<td>output</td>
<td>opaque data type</td>
<td>LVARCHAR</td>
</tr>
<tr>
<td>receive</td>
<td>SENDRECV</td>
<td>opaque data type (on the database server computer)</td>
</tr>
<tr>
<td>send</td>
<td>opaque data type (on server computer)</td>
<td>SENDRECV</td>
</tr>
</tbody>
</table>
Registering the Opaque Data Type with the Database

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 10-15.

The CREATE FUNCTION statement stores this information in the sysprocedures system catalog table. When it stores a new support function in sysprocedures, the CREATE FUNCTION statement causes a unique value, called an routine identifier, to be assigned to the support function. Throughout the system catalog, a support function is identified by its routine identifier, not by its name.

By default, a new support function has Execute permission assigned to the owner. For information on how to change the permission of an support function, see “Privileges on the Support Functions” on page 10-20.

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.

For more information on the syntax of the CREATE FUNCTION and CREATE FUNCTION FROM statements, see their descriptions in the Informix Guide to SQL: Syntax.

<table>
<thead>
<tr>
<th>Support Function</th>
<th>Parameter Type</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>import</td>
<td>IMPEXP</td>
<td>opaque data type</td>
</tr>
<tr>
<td>export</td>
<td>opaque data type</td>
<td>IMPEXP</td>
</tr>
<tr>
<td>importbinary</td>
<td>IMPEXPBIN</td>
<td>opaque data type</td>
</tr>
<tr>
<td>exportbinary</td>
<td>opaque data type</td>
<td>IMPEXPBIN</td>
</tr>
</tbody>
</table>
### 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.

<table>
<thead>
<tr>
<th>Support Function</th>
<th>Cast</th>
<th>Type of Cast</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>LVARCHAR</td>
<td>opaque data type</td>
</tr>
<tr>
<td>output</td>
<td>opaque data type</td>
<td>LVARCHAR</td>
</tr>
<tr>
<td>receive</td>
<td>SENDRECV</td>
<td>opaque data type</td>
</tr>
<tr>
<td>send</td>
<td>opaque data type</td>
<td>SENDRECV</td>
</tr>
<tr>
<td>import</td>
<td>IMPEXP</td>
<td>opaque data type</td>
</tr>
<tr>
<td>export</td>
<td>opaque data type</td>
<td>IMPEXP</td>
</tr>
<tr>
<td>importbinary</td>
<td>IMPEXPBIN</td>
<td>opaque data type</td>
</tr>
<tr>
<td>exportbinary</td>
<td>opaque data type</td>
<td>IMPEXPBIN</td>
</tr>
</tbody>
</table>

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, or IMPEXPBIN 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 *Informix Guide to SQL: Syntax*. For a description of casting, see the *Informix Guide to SQL: Tutorial*. 
Granting Privileges for an Opaque Data Type

Once you have created the opaque data type and registered 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

**Important:** The GRANT statement registers the privileges on the opaque data type and support functions in the current database. For users of another database to have access to the opaque data type and its support functions, you must run the GRANT statement when this second database is the current database.

**Privileges on the Opaque Data Type**

To create a new opaque data type within a database, you must have the Resource privilege on the database. 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 user-defined routine with the opaque data type as an argument.
  - The Execute routine privilege determines access to a user-defined routine.
Privileges on the Support Functions

For example, the following GRANT statement assigns the Usage privilege on the circle opaque data type to the user dexter:

```sql
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.

Privileges on the Support Functions

To register a support function within a database, you must have the Resource privilege on the database. The CREATE FUNCTION statement registers the new support 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 user-defined routine and therefore for all support functions that are defined in the database. The table contains one row for each set of privileges granted.
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:

- Writing new versions of arithmetic or built-in functions to provide arithmetic operations and built-in functions on the opaque data type
- Writing new versions of 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 user-defined data type to use any of these functions, you can write a version of 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.”)

Operating on Data

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

Operator Functions for Opaque Data Types

The database server provides the following types of operators for expressions in SQL statements:

- Arithmetic operators usually operate on numeric values.
- Text operators operate on character strings.
Tip: The database server also provides relational operators. For more information on the relational operators and their operator functions, see “Comparing Data” on page 10-23.

The database server provides operator functions for the arithmetic operators (see Figure 6-1 on page 6-5) and text operators (see Figure 6-2 on page 6-5). The versions of the operator functions that database server provides handle the built-in data types. You can write a new version of one of these operator function to provide the associated operation on your new opaque data type.

If you write a new version of an operator function, make sure you follow these rules:

1. The name of the operator function must match the name that Figure 6-1 on page 6-5 or Figure 6-2 on page 6-5 lists. 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. Figure 6-4 on page 6-8 shows the built-in functions that the database server defines. The versions of the built-in functions that database server provides handle the built-in data types. You can write a new version of a built-in function to provide the associated operation on your new opaque data type. If you write a new version of a built-in function, follow these rules:

1. The name of the built-in function must match the name that Figure 6-4 on page 6-8 lists. 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. Figure 6-4 on page 6-8 lists the number and data types of the parameters.
4. The built-in function must return the correct data type, where appropriate.
Comparing Data

The database server 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

Conditions 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 9-10.

For more information on the conditional clause, see the Condition segment in the Informix Guide to SQL: Syntax. For more information on the compare() function, see “Comparison Function for Opaque Data Types” on page 10-25.

Relational Operators for Opaque Data Types

The database server provides operator functions for the relational operators that Figure 6-3 on page 6-6 shows. The versions of the relational-operator functions that the database server provides handle the built-in data types. You can write a new version of a relational-operator function to provide the associated operation on your new opaque data type.
Comparing Data

If you write a new version of a relational-operator function, make sure you follow these rules:

1. The name of the relational-operator function must match the name that Figure 6-3 on page 6-6 lists. 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 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 `equal()` function can compare only bit-hashable data types; that is, a bitwise compare can determine equality. This comparison means that two values are equal if they have the same internal representation. The database server uses a built-in hash function to perform this comparison.

If your opaque data type is not bit hashable, the database server cannot use its built-in hash function for the equality comparison. Therefore, you cannot use the opaque data type in the following cases:

- In the GROUP BY clause of a SELECT statement
- In hash joins
- With the IN operator in a WHERE clause
Comparing Data

Nonhashable Data Types

For opaque types that are not bit hashable, specify the CANNOTHASH modifier in the CREATE OPAQUE TYPE statement.

Bit-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. CANNOTHASH specifies that data type equality cannot be determined by a bit-wise compare.

The following requirements apply to support functions on nonhashable data types:

- The equal() function should be hashable (no CANNOTHASH modifier) to indicate that two values have same internal representation or you must define a hash() function. You must have an equal() function if a column of the data requires the UNIQUE constraint.

Data types whose equal() functions require the support of a hash() function are those in which a bitwise comparison does not guarantee equality (not the same internal representation).

- The hash() function must have the following property: if equal(x,y) is true, then equal(hash(x), hash(y)) is also true. The hash() function takes a single argument and returns an integer. GROUP BY, IN, and hash joins and merges use the hash() function.

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 Informix Guide to SQL: Syntax.
Comparing Data

The database server provides versions of the `compare()` function 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 that handles this opaque data type.

If you write a new version of a `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 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

If your opaque data type is not bit hashable, the `compare()` function should generate an error so that the database server does not use the default `compare()` function.

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 9-4.
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 9-5.

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. For more information, see “A Fixed-Length Opaque Data Type: circle” on page 10-32.

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 9-10. For an example of how to extend the `btree_ops` operator class for a fixed-length opaque data type, see “A Fixed-Length Opaque Data Type: circle” on page 10-32.

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 other access methods, refer to the DataBlade API Programmer’s Manual. For more information on the secondary access methods that Data Blade modules provide, check the user guide for your DataBlade module.
Other Operations on Opaque Data Types

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 Virtual-Index Interface Programmer’s Manual.

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

Once you have created 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 Informix ESQL/C Programmer’s Manual.
- An external routine written in C language that uses the DataBlade API
  For more information, see the DataBlade API Programmer’s Manual.
Dropping an Opaque Data Type

An SPL user-defined routine written in SPL that uses SQL statements
For more information, see the chapter on SPL in the Informix Guide to SQL: Tutorial.

A client application written in 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 database, 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 sysxdtypeauth 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 10-18.

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. For more information, see “Registering Support Functions” on page 10-16.

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 Informix Guide to SQL: Syntax.
A Fixed-Length Opaque Data Type: circle

The circle data type is an example of a fixed-length opaque data type written in C. This data type includes an (x,y) coordinate, to represent the center of the circle, and a radius value. This section briefly shows how to create the circle data type.

Creating the Fixed-Length Internal Structures

Figure 10-1 shows the internal data structures for the circle data type.

```c
typedef struct {
  double x;
  double y;
} point_t;

typedef struct {
  point_t center;
  double radius;
} circle_t;
```

The internal representation for circle requires three double values: two double values for the center (x and y in the point_t structure) and one for the radius. Because each double is 8 bytes, the total internal length for the circle_t structure is 24 bytes.
Writing the circle Support Functions

The following table shows possible function signatures for the support functions of the circle data type.

<table>
<thead>
<tr>
<th>Support Function</th>
<th>Function Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>circle_t * circle_input(extrnl_format)</td>
</tr>
<tr>
<td></td>
<td>mi_lvarchar * extrnl_format;</td>
</tr>
<tr>
<td>output</td>
<td>mi_lvarchar * circle_output(intrnl_format)</td>
</tr>
<tr>
<td></td>
<td>circle_t * intrnl_format;</td>
</tr>
<tr>
<td>receive</td>
<td>circle_t * circle_receive(client_intrnl_format)</td>
</tr>
<tr>
<td></td>
<td>mi_sendrecv * client_intrnl_format;</td>
</tr>
<tr>
<td>send</td>
<td>mi_sendrecv * circle_send(srvr_intrnl_format)</td>
</tr>
<tr>
<td></td>
<td>circle_t * srvr_intrnl_format;</td>
</tr>
</tbody>
</table>

The actual code for the circle data type would be written in the C language, compiled, and put in the circle.so shared-object file. For information about shared-object files, refer to “Creating a Shared-Object File” on page 4-14.

Suppose the input and output functions of the circle data type define the external format that Figure 10-2 shows.

The input support function, circle_input(), would accept a string of the form in Figure 10-2, parse the string, and store each value in the circle_t structure. The output support function, circle_output(), would perform the inverse operation: it would take the values from the circle_t structure and build a string of the form in Figure 10-2.
Registering the circle Opaque Data Type

The following example shows the SQL statements that register the circle opaque data type and its input, output, send, and receive support functions in the database:

```
CREATE OPAQUE TYPE circle (INTERNALLENGTH = 24);

CREATE FUNCTION circle_in(txt LVARCHAR) RETURNS circle
  EXTERNAL NAME '/usr/lib/circle.so(circle_input)'
  LANGUAGE C NOT VARIANT;
CREATE IMPLICIT CAST (LVARCHAR AS circle WITH circle_in);

CREATE FUNCTION circle_out(cir circle) RETURNS LVARCHAR
  EXTERNAL NAME '/usr/lib/circle.so(circle_output)'
  LANGUAGE C NOT VARIANT;
CREATE EXPLICIT CAST (circle AS LVARCHAR WITH circle_out);

CREATE FUNCTION circle_rcv(cl_cir SENDRECV) RETURNS circle
  EXTERNAL NAME '/usr/lib/circle.so(circle_receive)'
  LANGUAGE C NOT VARIANT;
CREATE IMPLICIT CAST (SENDRECV AS circle WITH circle_rcv);

CREATE FUNCTION circle_snd(srv_cir circle) RETURNS SENDRECV
  EXTERNAL NAME '/usr/lib/circle.so(circle_send)'
  LANGUAGE C NOT VARIANT;
CREATE EXPLICIT CAST (circle AS SENDRECV WITH circle_snd);
```

The CREATE OPAQUE TYPE statement registers the circle data type, which has the following characteristics:

- It is a fixed-length opaque data type whose size is 24 bytes.
- It is to be aligned on 4-byte boundaries (the default alignment).
- When the opaque data type is passed as an argument to a user-defined function, it is passed by reference (the default parameter-passing mechanism).
Granting Privileges on the circle Data Type

The CREATE FUNCTION statements register the following support functions:

- The input support function is registered in the database as `circle_in()`. Its source code is the `circle_input()` function in the `/usr/lib/circle.so` shared-object file.
- The output support function is registered in the database as `circle_out()`. Its source code is the `circle_output()` function in the `/usr/lib/circle.so` shared-object file.
- The receive support function is registered in the database as `circle_rcv()`. Its source code is the `circle_receive()` function in the `/usr/lib/circle.so` shared-object file.
- The send support function is registered in the database as `circle_snd()`. Its source code is the `circle_send()` function in the `/usr/lib/circle.so` shared-object file.

The CREATE CAST statements create the appropriate casts for the support functions. For more information on these casts, see “Creating Casts for Opaque Data Types” on page 10-18.

Granting Privileges on the circle Data Type

The CREATE OPAQUE TYPE statement in the previous example creates the `circle` data type with the Usage privilege granted to the owner (the person who ran the CREATE OPAQUE TYPE statement). The following GRANT statement allows all users of the database to use the `circle` data type in SQL statements:

```
GRANT USAGE ON TYPE circle TO PUBLIC
```

The CREATE FUNCTION statements register the support functions for the `circle` data type with the Execute privilege granted to the owner (the person who ran the CREATE FUNCTION statements). The following GRANT statements grant the Execute privilege to all users of the database on the support statements that are defined as explicit casts:

```
GRANT EXECUTE ON FUNCTION circle_out TO PUBLIC;
GRANT EXECUTE ON FUNCTION circle_snd TO PUBLIC;
```
Creating SQL-Invoked Functions for the circle Data Type

The circle data type has two end-user functions:

- The `radius()` function obtains the radius value from a `circle_t` structure.
  - The actual code for the `radius()` function is written in the C language, compiled, and put in the `circle.so` shared-object file.
- The `area()` function calculates the area of the circle that a `circle_t` structure describes.
  - The `area()` function is written in SPL, so the CREATE FUNCTION statement includes the actual code.

The following SQL statements register the end-user functions for the `circle` data type with the database:

```sql
CREATE FUNCTION radius(circle) RETURNS FLOAT
  EXTERNAL NAME '/usr/lib/circle.so'
  LANGUAGE C NOT VARIANT;

CREATE FUNCTION area(crcl_val circle) RETURNING FLOAT
  DEFINE pi FLOAT;
  LET pi = 3.1415926;
  RETURN (pi * POWER(RADIUS(circle), 2));
END FUNCTION;
```
Providing Indexes for the circle Data Type

To generate a lexicographical sequence, the B-tree index compares the area of two circle values to determine the order of the values. Because the operator-class functions of the B-tree index do not include information about the circle data type, you must write the following functions so that the B-tree index can order the values correctly:

- The strategy functions of the default operator class for the generic B-tree
  For more information about these functions, see “B-Tree Strategy Functions” on page 9-7.
- The support function of the default operator class for the generic B-tree
  For more information about these functions, see “B-Tree Support Function” on page 9-7.

You must first write the B-tree strategy functions in C. The following table shows the meaning and possible function signatures for the strategy functions that handle the circle data type.

<table>
<thead>
<tr>
<th>B-Tree Strategy Function</th>
<th>Meaning for circle Data Type</th>
<th>Function Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>lessthan()</td>
<td>area(circle1) &lt; area(circle2)</td>
<td>boolean circle_lessthan(circle1, circle2) circle *circle1, *circle2;</td>
</tr>
<tr>
<td>lessthanorequal()</td>
<td>area(circle1) &lt;= area(circle2)</td>
<td>boolean circle_lessthanorequal(circle1, circle2) circle *circle1, *circle2;</td>
</tr>
<tr>
<td>equal()</td>
<td>area(circle1) = area(circle2)</td>
<td>boolean circle_equal(circle1, circle2) circle *circle1, *circle2;</td>
</tr>
<tr>
<td>greaterthan()</td>
<td>area(circle1) &gt; area(circle2)</td>
<td>boolean circle_greaterthan(circle1, circle2) circle *circle1, *circle2;</td>
</tr>
<tr>
<td>greaterthanorequal()</td>
<td>area(circle1) &gt;= area(circle2)</td>
<td>boolean circle_greaterthanorequal(circle1, circle2) circle *circle1, *circle2;</td>
</tr>
</tbody>
</table>
Providing Indexes for the `circle` Data Type

These functions might already have been defined when you provided support for the relational operators on your opaque data type. (For more information, see “Comparing Data” on page 10-23.) In this case, the `lessthан()`, `lessthanorequal()`, `equal()`, `greaterthan()`, and `greaterthanorequal()` functions have already been written, compiled, stored in a shared-object file, and registered in the database. Therefore, the generic B-tree index can already use them as its strategy functions.

However, if you had not yet defined the relational operators for the `circle` data type, you would need to do so to provide support for B-tree indexes on `circle` columns and on user-defined functions that return the `circle` data type. Once you have generated the code for these functions and stored their compiled versions in a shared-object file, you must register them in the database using the `CREATE FUNCTION` statement.

The `compare()` function is the support function for the generic B-tree index. 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, see “Comparing Data” on page 10-23.) In this case, the `compare()` function has already been written, compiled, stored in a shared-object file, and registered in the database. Therefore, the generic B-tree index can use it as its support function for the `circle` data type.

However, if you had not yet defined the `compare()` function for the `circle` data type, you must do so to provide support for B-tree indexes on `circle` columns and user-defined functions that return the `circle` data type. Once you have generated the code for this function and stored its compiled version in a shared-object file, you must register it in the database with the `CREATE FUNCTION` statement.

With the strategy and support functions registered, you are ready to define generic B-tree indexes on the `circle` data type. The code fragment in the next section creates a generic B-tree index called `circle_ix` on the `circle_col` column.

**Tip:** To index the values of the area of the `circle` data type, you could also create a functional index on the `area()` function.
Using the circle Opaque Data Type

The following example shows the SQL statements that create a table called circle_tab that has a column of data type circle and insert several rows into this table:

```
CREATE TABLE circle_tab (circle_col circle);
CREATE INDEX circle_ix ON circle_tab (circle_col);
INSERT INTO circle_tab VALUES ('(12.00, 16.00, 13.00)');
INSERT INTO circle_tab VALUES ('(6.5, 8.0, 9.0)');
```

For information about the CREATE INDEX statement in this example, see “Providing Indexes for the circle Data Type” on page 10-37.

In the Informix ESQL/C Programmer’s Manual, the chapter on opaque types provides examples of how to access the circle opaque data type with a fixed binary host variable.

A Varying-Length Opaque Data Type: image

The image data type is an example of a varying-length opaque data type written in C. The image data type encapsulates an image such as a JPEG, GIF, or PPM file. If the image is less than 2 kilobytes, the data structure for the data type stores the image directly. However, if the image is greater than 2 kilobytes, the data structure stores a reference (an LO-pointer structure) to a smart large object that contains the image data. An image stored in the smart large object can be referenced by multiple rows, but the database server only needs to store a single copy of it in an sbspace.

This section briefly outlines how to create the image data type.
Creating the Varying-Length Internal Structure

The following statements show the internal structure for the image data type in the database:

typedef struct
{
    int   img_len;
    int   img_thresh;
    int   img_flags;
    union
        {
            ifx_lop_t img_lobhandle;
            char    img_data[4];
        }
} image_t;

Writing the image Support Functions

The following table shows possible function signatures for the basic support functions of the image data type.

<table>
<thead>
<tr>
<th>Support Function</th>
<th>Function Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>image_t * image_input(extrnl_format) mi_lvarchar *extrnl_format;</td>
</tr>
<tr>
<td>output</td>
<td>mi_lvarchar * image_output(intrnl_format) image_t *intrnl_format;</td>
</tr>
<tr>
<td>receive</td>
<td>image_t * image_receive(client_intrnl_format) mi_sendrecv *client_intrnl_format;</td>
</tr>
<tr>
<td>send</td>
<td>mi_sendrecv * image_send(srvr_intrnl_format) image_t *srvr_intrnl_format;</td>
</tr>
</tbody>
</table>
Writing the image Support Functions

The `image` data type has an embedded smart large object. Therefore, it has the following additional support functions.

<table>
<thead>
<tr>
<th>Support Function</th>
<th>Function Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>import</td>
<td><code>image_t * image_import(extrnl_bcopy_format)</code></td>
</tr>
<tr>
<td></td>
<td><code>mi_impexp * extrnl_bcopy_format</code></td>
</tr>
<tr>
<td>export</td>
<td><code>mi_impexp * image_export(intrnl_bcopy_format)</code></td>
</tr>
<tr>
<td></td>
<td><code>image_t * intrnl_bcopy_format</code></td>
</tr>
<tr>
<td>import-binary</td>
<td><code>image_t * image_impbin(client_intrnl_bcopy_format)</code></td>
</tr>
<tr>
<td></td>
<td><code>mi_impexpbin * client_intrnl_bcopy_format</code></td>
</tr>
<tr>
<td>export-binary</td>
<td><code>mi_impexpbin * image_expbin(srvr_intrnl_bcopy_format)</code></td>
</tr>
<tr>
<td></td>
<td><code>image_t * srvr_intrnl_bcopy_format</code></td>
</tr>
<tr>
<td>lohandles</td>
<td><code>mi_sendrecv * image_lohandles(intrnl_format)</code></td>
</tr>
<tr>
<td></td>
<td><code>image_t * intrnl_format</code></td>
</tr>
<tr>
<td>assign</td>
<td><code>image_t * image_assign(intrnl_format)</code></td>
</tr>
<tr>
<td></td>
<td><code>image_t * intrnl_format</code></td>
</tr>
</tbody>
</table>

The `assign()` function decides whether to store the image directly in the row or in a separate smart large object. This simple varying-length opaque data type does not require a `destroy()` function.
Registering the image Opaque Data Type

The following example shows the SQL statements that register the image data type and its basic support functions in the database:

```sql
CREATE OPAQUE TYPE image (INTERNALLENGTH = VARIABLE);
CREATE FUNCTION image_in(txt LVARCHAR) RETURNS image
    EXTERNAL NAME '/usr/lib/image.so(image_input)' LANGUAGE C NOT VARIANT;
CREATE IMPLICIT CAST (LVARCHAR AS image WITH image_in);
CREATE FUNCTION image_out(im image) RETURNS LVARCHAR
    EXTERNAL NAME '/usr/lib/image.so(image_output)' LANGUAGE C NOT VARIANT;
CREATE EXPLICIT CAST (image AS LVARCHAR WITH image_out);
CREATE FUNCTION image_rcv(cl_im SENDRECV) RETURNS image
    EXTERNAL NAME '/usr/lib/image.so(image_receive)' LANGUAGE C NOT VARIANT;
CREATE IMPLICIT CAST (SENDRECV AS image) WITH image_rcv);
CREATE FUNCTION image_snd(srv_im image) RETURNS SENDRECV
    EXTERNAL NAME '/usr/lib/image.so(image_send)' LANGUAGE C NOT VARIANT;
CREATE EXPLICIT CAST (image AS SENDRECV WITH image_snd);
```

The CREATE OPAQUE TYPE statement registers the image data type, which has the following characteristics:

- It is a varying-length opaque data type whose maximum size is 2 kilobytes (the default maximum size).
- It is to be aligned on 4-byte boundaries (the default alignment).
- When the opaque data type is passed as an argument to a user-defined function, the data type is passed by reference (the default parameter-passing mechanism).
Registering the image Opaque Data Type

The CREATE FUNCTION statements register the following basic support functions:

- The input support function is registered in the database as `image_in()`. Its source code is the `image_input()` function in the `/usr/lib/image.so` shared-object file.
- The output support function is registered in the database as `image_out()`. Its source code is the `image_output()` function in the `/usr/lib/image.so` shared-object file.
- The receive support function is registered in the database as `image_rcv()`. Its source code is the `image_receive()` function in the `/usr/lib/image.so` shared-object file.
- The send support function is registered in the database as `image_snd()`. Its source code is the `image_send()` function in the `/usr/lib/image.so` shared-object file.

For information about shared-object files, refer to “Creating a Shared-Object File” on page 4-14.

This example also shows the CREATE CAST statements to create the appropriate cast definitions for the input, output, receive, and send support functions. For more information on casts, see “Creating Casts for Opaque Data Types” on page 10-18.
Registering Other Support Functions for the image Data Type

The following example shows the SQL statements that register the other support functions for the image data type in the database:

```
CREATE FUNCTION image_imp(imp_im IMPEXP) RETURNS image
    EXTERNAL NAME '/usr/lib/image.so(image_import)' LANGUAGE C NOT VARIANT;
CREATE IMPLICIT CAST (IMPEXP AS image WITH image_imp);

CREATE FUNCTION image_exp(exp_im image) RETURNS IMPEXP
    EXTERNAL NAME '/usr/lib/image.so(image_export)' LANGUAGE C NOT VARIANT;
CREATE EXPLICIT CAST (image AS IMPEXP WITH image_exp);

CREATE FUNCTION image_impbin(impbin_im IMPEXPBIN) RETURNS image
    EXTERNAL NAME '/usr/lib/image.so(image_importbin)' LANGUAGE C NOT VARIANT;
CREATE IMPLICIT CAST (IMPEXPBIN AS image WITH image_impbin);

CREATE FUNCTION image_expbin(expbin_im image) RETURNS IMPEXPBIN
    EXTERNAL NAME '/usr/lib/image.so(image_exportbin)' LANGUAGE C NOT VARIANT;
CREATE EXPLICIT CAST (image AS IMPEXPBIN WITH image_expbin);

CREATE FUNCTION lohandles(im image) RETURNS POINTER
    EXTERNAL NAME '/usr/lib/image.so(image_lohandles)' LANGUAGE C NOT VARIANT;

CREATE FUNCTION assign(im image) RETURNS image
    EXTERNAL NAME '/usr/lib/image.so(image_assign)' LANGUAGE C NOT VARIANT;
```
Registering Other Support Functions for the image Data Type

The CREATE FUNCTION statements register the following support functions:

- The import support function is registered in the database as `image_imp()`. Its source code is the `image_import()` function in the `/usr/lib/image.so` shared-object file.
- The export support function is registered in the database as `image_exp()`. Its source code is the `image_export()` function in the `/usr/lib/image.so` shared-object file.
- The importbinary support function is registered in the database as `image_impbin()`. Its source code is the `image_importbin()` function in the `/usr/lib/image.so` shared-object file.
- The exportbinary support function is registered in the database as `image_expbin()`. Its source code is the `image_exportbin()` function in the `/usr/lib/image.so` shared-object file.
- The lohandles support function is registered in the database as `lohandles()`, its required name. Its source code is the `image_lohandles()` function in the `/usr/lib/image.so` shared-object file.
- The assign support function is registered in the database as `assign()`, its required name. Its source code is the `image_assign()` function in the `/usr/lib/image.so` shared-object file.

For information about shared-object files, refer to “Creating a Shared-Object File” on page 4-14.

This example also shows the CREATE CAST statements to create the cast definitions for the import, export, importbinary, and exportbinary support functions. For more information on these casts, see “Creating Casts for Opaque Data Types” on page 10-18.
Granting Privileges on the image Data Type

The CREATE OPAQUE TYPE statement in the example in “Registering the image Opaque Data Type” on page 10-42 creates the image data type with the Usage privilege granted to the owner (the person who ran the CREATE OPAQUE TYPE statement). The following GRANT statement allows all users of the database to use the image data type in SQL statements:

```
GRANT USAGE ON TYPE image TO PUBLIC
```

The CREATE FUNCTION statements in the two previous examples register the support functions for the image data type with the Execute privilege granted to the owner (the person who ran the CREATE FUNCTION statements). The following GRANT statements grant the Execute privilege to all users of the database on the support statements that are defined as explicit casts:

```
GRANT EXECUTE ON FUNCTION image_out TO PUBLIC;
GRANT EXECUTE ON FUNCTION image_snd TO PUBLIC;
GRANT EXECUTE ON FUNCTION image_imp TO PUBLIC;
GRANT EXECUTE ON FUNCTION image_impbin TO PUBLIC;
```

Creating SQL-Invoked Functions for the image Data Type

The image data type has the following end-user functions:

- The `getsize()` function returns the size of the image data.
- The `image_zoom()` function zooms the given image to a parametric factor in both x and y dimensions and returns a new image.

The actual code for the end-user functions would be written in the C language and put in the image.so shared-object file. The following SQL statements register the end-user functions for the image opaque data type with the database:

```
CREATE FUNCTION getsize(im image) RETURNS INTEGER
    EXTERNAL NAME '/usr/lib/image.so(image_size)'
    LANGUAGE C NOT VARIANT;

CREATE FUNCTION image_zoom(im image, i integer)
    RETURNS image
    EXTERNAL NAME '/usr/lib/image.so(image_zoom)'
    LANGUAGE C NOT VARIANT;
```
Providing Indexes for the image Data Type

Suppose you decide that the default secondary access method, a generic B-tree index, does not adequately handle data in columns of type `image`. Because the image data type holds spatial data, it is a good candidate for an R-tree index. To use an R-tree index, you must first install a spatial DataBlade module that implements the R-tree index. For more information about R-trees, refer to the user guide for the DataBlade module and to the *Informix R-Tree Index User’s Guide*. 
Writing Support Functions

In This Chapter ........................................ 11-3

Support Functions for Opaque Data Types .......................... 11-3
    Handling the External Representation .................. 11-5
        LVARCHAR Data Type .................................. 11-6
        Input Support Function ................................ 11-6
        Output Support Function ................................ 11-8
Handling Platform-Specific Internal Representations .............. 11-10
    SENDRECV Data Type .................................... 11-11
    Receive Support Function ................................ 11-11
    Send Support Function .................................. 11-13
Performing Bulk Copies .......................................... 11-14
    Import and Export Support Functions .................. 11-14
    Importbinary and Exportbinary Support Functions ....... 11-17
Inserting and Deleting Data ...................................... 11-19
    The assign() Support Function ......................... 11-19
    The destroy() Support Function ....................... 11-20
Handling Smart Large Objects .................................. 11-21
Comparing Data ................................................ 11-22
Handling Locale-Sensitive Data .................................. 11-23
    In Input and Output Support Functions ................. 11-24
    In Receive and Send Support Functions ................ 11-25
In This Chapter

This chapter describes the support functions for the following objects:

- An opaque data type
- An operator class

For an introduction to an opaque data type, see Chapter 10, “Creating an Opaque Data Type.” For an introduction to an operator class, see Chapter 9, “Extending an Operator Class.”

Support Functions for Opaque Data Types

The support functions for an opaque data type are a set of well-defined, type-specific functions that the database server automatically invokes. Typically, these functions are not explicitly invoked in an SQL statement. The following table summarizes the support functions of an opaque data type.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>Converts the opaque-type data from its external to its internal representation. Supports insertion of text data into a column of the opaque type. Is an implicit cast from the LVARCHAR to opaque data type.</td>
<td>page 11-6</td>
</tr>
<tr>
<td>output</td>
<td>Converts the opaque-type data from its internal to external representation. Supports selection of text data from a column of the opaque type. Is an explicit cast from the opaque to LVARCHAR opaque data type.</td>
<td>page 11-8</td>
</tr>
</tbody>
</table>

(1 of 3)
## Support Functions for Opaque Data Types

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>receive</td>
<td>Converts the opaque-type data from its internal representation on the client computer to its internal representation on the database server computer. Supports insertion of binary data in a column of the opaque type. Is an implicit cast from the SENDRECV to the opaque data type.</td>
<td>page 11-11</td>
</tr>
<tr>
<td>send</td>
<td>Converts the opaque-type data from its internal representation on the database server computer to its internal representation on the client computer. Supports selection of binary data from a column of the opaque type. Is an explicit cast from the opaque to the SENDRECV data type.</td>
<td>page 11-13</td>
</tr>
<tr>
<td>import</td>
<td>Performs processing of opaque type for bulk load of text data in a column of the opaque type. Is an implicit cast from the IMPEXP to the opaque data type.</td>
<td>page 11-16</td>
</tr>
<tr>
<td>export</td>
<td>Performs processing of opaque type for bulk unload of text data from a column of the opaque type. Is an explicit cast from the opaque to the IMPEXP data type.</td>
<td>page 11-16</td>
</tr>
<tr>
<td>importbinary</td>
<td>Performs processing of opaque type for bulk load of binary data in a column of the opaque type. Is an implicit cast from the IMPEXPBIN to the opaque data type.</td>
<td>page 11-17</td>
</tr>
<tr>
<td>exportbinary</td>
<td>Performs processing of opaque type for bulk unload of binary data from a column of the opaque type. Is an explicit cast from the opaque to the IMPEXPBIN data type.</td>
<td>page 11-18</td>
</tr>
<tr>
<td>assign</td>
<td>Performs any processing required before the database server stores the opaque-type data to disk. Supports storage of opaque type for INSERT, UPDATE, and LOAD statements.</td>
<td>page 11-19</td>
</tr>
<tr>
<td>destroy</td>
<td>Performs any processing necessary before the database server removes a row that contains opaque-type data. Supports deletion of opaque type for DELETE and DROP TABLE statements.</td>
<td>page 11-20</td>
</tr>
</tbody>
</table>
Handling the External Representation

Every opaque type has its 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 10-4.) The external representation is a character string that is a printable version of the opaque value.

When you define an opaque type, you must supply the following support functions that convert between the internal and external representations of the opaque type:

- The input function converts from external to internal representation.
- The output function converts from internal to external 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 your opaque data type to accept an external representation on non-default locales, you must use the 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 11-23.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>lohandles</td>
<td>Returns a list of the embedded large-object handles in the opaque data type.</td>
<td>page 11-21</td>
</tr>
<tr>
<td>compare</td>
<td>Supports sort of opaque-type data during ORDER BY, UNIQUE, DISTINCT, and UNION clauses and CREATE INDEX operations.</td>
<td>page 11-22</td>
</tr>
</tbody>
</table>
Handling the External Representation

**LVARCHAR Data Type**

SQL statements use the LVARCHAR data type to hold the external representation of an opaque data type. This data type supports varying-length strings whose length is greater than 256 bytes. The input and output support functions serve as cast functions between the LVARCHAR and opaque data type.

*Tip:* The DataBlade API provides the `mi_lvarchar` data type to hold the external representation of opaque-type data. For more information, see the "DataBlade API Programmer’s Manual."

ESQL/C applications use `lvarchar` or other character-based host variables in SQL statements to transfer the external 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.

For more information on how to use SQL statements to register support functions, see “Registering Support Functions” on page 10-16.

**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 character 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.
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
  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.
Handling the External Representation

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 an opaque data type whose internal structure is ll_longlong_t:

```c
ll_longlong_t * ll_longlong_input(extrnl_format)
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 10-18.

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 11-2 shows when the database server executes the output support function.

Figure 11-2
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
  The support function can use the _mi_alloc_ DataBlade API function to allocate the space for the character string. For more information on memory management and the _mi_alloc_ function, refer to the *DataBlade API Programmer’s Manual.*
- Create the output string from the individual members of the internal structure
  It 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 less than 4 bytes in length.

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(intrnl_format)
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 10-18.
Handling Platform-Specific Internal Representations

If a client application that uses an opaque data type executes on a different computer than the database server, it might have a different way 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. For cases where the internal representation of an opaque type might differ on the client and database server computers, you must supply the receive and send support functions that convert between the internal representation on the client computer and that on the database server computer.

These functions must handle conversions for all platform variations that the client applications might encounter. When the client application establishes a connection with the database server, it sends a description of the internal representations that the client computer uses. One representation is the same for all client applications that run on the same architecture. The database server uses this description to determine which client representation to use in its receive and send support functions.

The Informix DataBlade API provides functions that support conversion between different internal representations of opaque types. Your send and receive functions can call these DataBlade API routines for each member of the internal structure to convert them to the appropriate representation for the destination platform.

For your opaque data type to accept an internal representation on non-default locales, you must use the 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 11-23.

The receive and send functions support the binary transfer of opaque types. They convert between the internal representation of an opaque type on a client computer and the internal representation on the database server computer.

- The receive function converts from the client internal representation to the database server internal representation.
- The output function converts from database server internal representation to the client internal representation.
These support functions do not have to be named receive and send, but they do have to perform the specified conversions. They should be reciprocal functions; that is, the receive function should produce a value that the send function accepts as an argument and vice versa.

**SENDRECV Data Type**

SQL statements support an internal data type called SENDRECV to hold the internal 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.

For more information on how to use SQL statements to register support functions, see “Registering Support Functions” on page 10-16.

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 database server calls the receive function when it receives the internal 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 internal representation of an opaque type to the database server to be stored in a column.

ESQL/C applications use the fixed binary and var binary host variables to send the internal representation of an opaque data type.
Handling Platform-Specific Internal Representations

Figure 11-3 shows when the database server executes the receive support function.

The database server calls the receive function to convert the internal representation of the client computer to the internal representation of the database server computer, where the opaque type is stored on disk.

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(client_intrnl_format)
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 10-18.
Handling Platform-Specific Internal Representations

**Send Support Function**

The database server calls the send function when it sends the internal 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 internal representation of the opaque type.

ESQL/C applications use the fixed binary and var binary host variables to receive the internal representation of an opaque data type.

Figure 11-4 shows when the database server executes the send support function.

The database server calls the send function to convert the internal representation that is stored on disk to the internal representation that the client computer uses.

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(srvr_intrnl_format)
ll_longlong_t *srvr_intrnl_format;
```
Performing Bulk Copies

The $ll_{\text{longlong}}$ function is a cast function from the $ll_{\text{longlong}}$ internal structure to the SENDRECV data type. It must be registered as an explicit cast function with the CREATE EXPPLICIT CAST statement. For more information on cast functions, see “Creating Casts for Opaque Data Types” on page 10-18.

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 the external representation of an opaque type for a bulk copy. When the database server copies data to or from a database in external 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 to internal representation.
- The export function exports text data by converting from internal to external 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 would 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 would 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.

For more information on how to use SQL statements to register support functions, see “Registering Support Functions” on page 10-16.
Performing Bulk Copies

Import

The import 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.

The following function signature is for an import support function of an opaque data type whose internal structure is `ll_longlong_t`:

```c
ll_longlong_t * ll_longlong_import(extrnl_bcopy_format)
    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 10-18.

Export

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.

The following function signature is for an export support function of an opaque data type whose internal structure is `ll_longlong_t`:

```c
mi_impexp * ll_longlong_export(intrnl_bcopy_format)
    ll_longlong_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 10-18.
Performing Bulk Copies

Importbinary and Exportbinary Support Functions

The importbinary and exportbinary support functions perform any tasks needed to process the internal (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 vice versa. The 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 internal 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.

For more information on how to use SQL statements to register support functions, see “Registering Support Functions” on page 10-16.

Importbinary

The importbinary function takes as an argument a structure that holds the bulk-copy format of the internal 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.

The following function signature is for an importbinary support function of an opaque data type whose internal structure is `ll_longlong_t`:

```c
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 on cast functions, see “Creating Casts for Opaque Data Types” on page 10-18.

`Exportbinary`

The exportbinary function takes as an argument the internal structure for the opaque type and returns a structure that holds the bulk-copy format of the internal representation of the opaque type.

If you do not provide an exportbinary support function, the database server exports the binary data in the internal representation of the opaque data type.

The following function signature is for an exportbinary support function of an opaque data type whose internal structure is `ll_longlong_t`:

```c
mi_impexpbin * ll_longlong_exportbin(srvr_intrnl_bcopy_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 on cast functions, see “Creating Casts for Opaque Data Types” on page 10-18.
Inserting and Deleting Data

Some opaque data types might require special processing before they are saved to or removed from disk. For the database server to perform this special processing on an opaque type, you can create the following support functions:

- The `assign()` support function contains the special processing to perform before an opaque data type is inserted into a table.
- The `destroy()` support function contains the special processing to perform before an opaque data type is deleted from a table.

A common use of the `assign()` and `destroy()` support functions is for an opaque data type that contains spatial or multirepresentational data. Such a data type might provide a choice of how to store the data: inside the internal structure or, for very large objects, in a smart large object. If the data is stored in a smart large object, the internal structure of the opaque data type contains the LO-pointer structure to identify the location of the data; it does not contain the data itself. The `assign()` support function can make the decision of how to store the data, and the `destroy()` support function can decide how to remove the data, regardless of where it is stored. For an example of such an opaque type, see “A Varying-Length Opaque Data Type: image” on page 10-39.

These support functions must be named `assign` and `destroy`, but the names are case insensitive. They must perform the tasks that should occur before the opaque data type is stored to or removed from disk. The `assign()` and `destroy()` functions are required for opaque types that have smart large objects and for multirepresentational data.

The `assign()` Support Function

The database server calls the `assign()` support 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.
**Inserting and Deleting Data**

Figure 11-5 shows when the database server executes the **assign()** function.

*Figure 11-5*

**Execution of the assign() Support Function**

When you store a value of an opaque data type, the **assign()** function takes as an argument the internal structure for the opaque data type and returns the data that is actually stored in the table.

**The destroy() Support Function**

The database server calls the **destroy()** support 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 11-6 shows when the database server executes the `destroy()` function.

The `destroy()` function takes as an argument the internal structure for the opaque data type.

**Handling Smart Large Objects**

If an opaque data type contains an embedded smart large object, you must define an `lohandles()` support 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.

The database server uses the `lohandles()` support function when it must search opaque-type values for references to smart large objects. Examples of this search include the following:

- Performing an archive of the database
- Obtaining a reference count for the smart large objects
- Running the `oncheck` utility
Comparing Data

If you define an opaque type that references one or more smart large objects, you must also consider defining the following support functions:

- assign()
- destroy()
- 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 11-19. For information on the import, export, importbinary, and exportbinary support functions, see “Performing Bulk Copies” on page 11-14.

Comparing Data

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 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

If your opaque type is not bit hashable, the `compare()` function should generate an error so that the database server does not use the default `compare()` function.

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 9-4. For more information on how to customize a secondary access method for an opaque data type, see “Using Operator Classes” on page 9-3.

---

**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.

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.
In Input and Output Support Functions

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 *Informix Guide to GLS Functionality*.

To help you write support functions that handle locale-sensitive data, Informix provides the Informix GLS API. 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 *Informix Guide to GLS Functionality*. For a description of the GLS API functions, see the *Informix GLS Programmer’s Manual*.

In 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 Informix Guide to GLS Functionality.

In 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.
# Managing a User-Defined Routine

In This Chapter ........................................... 12-3

Assigning Execute Privilege to a Routine ......................... 12-3
  Granting and Revoking the Execute Privilege .................. 12-4
  Privileges on Objects Associated with a UDR .................. 12-5
  Executing a UDR as DBA .................................. 12-7
  Using DBA Privileges with Objects and Nested UDRs ........... 12-7

Reloading a User-Defined Routine ................................ 12-9
  Unloading the Shared-Object File ............................ 12-9
  Using the ifx_replace_module() Function ...................... 12-10

Altering a User-Defined Routine ................................ 12-12

Dropping a User-Defined Routine ................................ 12-12
In This Chapter

This chapter describes how to manage user-defined routines. It includes the following topics:

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

Assigning Execute Privilege to a Routine

The Execute privilege enables users to invoke a user-defined routine. 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 Execution 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.
- 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 Execute privilege on a user-defined routine, use the EXECUTE ON clause of the GRANT and REVOKE statements. The database server stores privileges for user-defined routines in the `sysprocauth` system catalog table.

User-defined routines have the following GRANT and REVOKE requirements for 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 user-defined data type and the GRANT statement to enable user `mary` to execute this variation of the `equal()` function:

```sql
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 user-defined routine named `equal()`.

For more information, see the GRANT and REVOKE statements in the 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
- User-defined data types
- Other routines executed by the routine

When the owner of a routine grants the Execute privilege, some privileges on objects automatically accompany the Execute privilege. A GRANT EXECUTE ON statement confers to the grantee any table-level privileges that the grantor received from a GRANT statement that contained the WITH GRANT keywords.

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.
Privileges on Objects Associated with a UDR

**Figure 12-1** 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 the user `tony` executes the CREATE PROCEDURE statement to register the SPL `promo()` procedure. The user `marty` executes the `promo()` routine 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 privileges 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 tony registers the promo() routine from on page 12-6 with the DBA keyword, as follows:

```
CREATE DBA PROCEDURE promo()
  EXTERNAL NAME create_mo_catalog
END PROCEDURE;
```

Although tony owns the routine, if marty runs it, 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:

```
strcpy(statement, "CREATE INDEX stmt");
EXEC SQL
create cluster index c_clust_ix on catalog (catalog_num);
```

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 `dbspace_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 follows 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.
Reloading a User-Defined Routine

To reload a user-defined routine explicitly, reload its shared-object file. The following methods reload a shared-object file without bringing down the database server:

- Unload and reload the shared-object module by dropping and reregistering all routines in the module.
- Replace a shared-object module with the ifx_replace_module() function.

Unloading the Shared-Object File

Shared-object files are also unloaded when the database server is shut down. All memory that the database server uses, including memory for shared-object modules, is released when the database server shuts down.

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 user-defined routine. These statements remove the registration information about the user-defined routine from the system catalog tables.

The database server removes the shared-object file from the memory map after both of the following conditions occur:

- You drop all routines in the module.
- All instances of the routines finish executing.

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 user-defined routines in the database.
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 the shared object file. The surest way to unload a shared-object file is to drop all routines that the module contains.

**Tip:** SPL p-code is not compiled into shared-object modules. Therefore, the Routine Manager does not need to unload SPL routines.

### Using the `ifx_replace_module()` Function

You can use the `ifx_replace_module()` user-defined function, which Informix provides, to replace a loaded shared-object module with a new version that has a new name. Use the following syntax for `ifx_replace_module()`:

```c
ifx_replace_module ("old module path", "new module path", 
                   "language name")
```

- **old module path**  A character string that lists the full path of the current name of the shared-object module.
- **new module path**  A character string that lists the full path of the new name of the shared-object module.
- **language name**  A character string that specifies the name of the language in which the UDRs within the shared-object module are written.

  - For a DataBlade API shared-object library, `language name` must be "c".
  - For a Java db-applet shared object, `language name` must be "dbapplet".

For more information on the syntax of the `ifx_replace_module()` function, see the description of Function Expressions within the Expression segment in the *Informix Guide to SQL: Syntax*. 

---

---

**Extending Informix Dynamic Server 2000**

---

---
Using the `ifx_replace_module()` Function

For example, to replace the `circle.so` shared 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:

```sql
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 function returns 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:

```sql
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.

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 *Informix ESQL/C Programmer's Manual*. ♦
Altering a User-Defined Routine

You can use the ALTER FUNCTION, ALTER PROCEDURE, and ALTER ROUTINE statements to change the routine modifiers or pathname name 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.

For information on these SQL statements, refer to their description in the Informix Guide to SQL: Syntax.

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.

For information on these SQL statements, refer to their description in the Informix Guide to SQL: Syntax.
Improving UDR Performance

In This Chapter ........................................ 13-3

Optimizing a User-Defined Routine .................. 13-3
  Optimizing an SPL Routine .......................... 13-4
    Optimization Levels ............................... 13-4
    Automatic Optimization ............................ 13-5
  Updating Statistics for an SPL Routine .......... 13-6

Optimizing Functions in SQL Statements ............ 13-8
  Calculating the Query Plan ......................... 13-8
  Specifying Cost and Selectivity .................... 13-9
    Constant Cost and Selectivity Values .......... 13-9
    Dynamic Cost and Selectivity Values ......... 13-10
  Calculating Cost .................................. 13-10
    Selectivity and Cost Examples .................. 13-11

Extending UPDATE STATISTICS ....................... 13-12
  Using UPDATE STATISTICS .......................... 13-13
    Support Functions for UPDATE STATISTICS ..... 13-13
      The stat Data Type ............................. 13-13
      The statcollect() Function ..................... 13-14
      The statprint() Function ....................... 13-15
      Example of User-Defined Statistics Functions 13-15

Using Negator Functions ............................... 13-16

Using a Virtual-Processor Class ....................... 13-18
  Choosing a Virtual-Processor Class ............... 13-19
    CPU Virtual-Processor Class ..................... 13-19
    User-Defined Virtual-Processor Class .......... 13-20
  Using Virtual Processors with UDRs Written in C 13-21
  Using Virtual Processors with UDRs Written in Java 13-22
Managing Virtual Processors ........................................... 13-22
Adding and Dropping Virtual Processors .......................... 13-22
Monitoring Virtual-Processor Classes ............................. 13-23

Using UDRs in Parallel ..................................................... 13-23
Executing UDRs in Parallel ............................................. 13-24
   Execution of a UDR in a Query Expression ..................... 13-25
   FastPath Execution of a UDR in a DataBlade API ............... 13-28
   Implicit UDR Execution of a User-Defined Aggregate ........ 13-29
   Implicit UDR Execution of a Comparison Operator ........... 13-29
   Implicit Execution of an Assign UDR ......................... 13-30
   Execution of a Comparison UDR for Sort ....................... 13-30
   Execution of a UDR by an Index on a UDT column .......... 13-30

Enabling Parallel UDRs .................................................. 13-31
   Specifying the PARALLELIZABLE Modifier ...................... 13-31
   Writing PDQ Thread-Safe UDRs ................................. 13-31
   Turning On PDQ and Reviewing Other Configuration Parameters 13-32
   Step-By-Step Procedure to Enable Parallel UDRs .......... 13-33
   Setting the Number of Virtual Processors ................. 13-34
   Monitoring Parallel UDRs ....................................... 13-35

Memory Considerations ................................................. 13-36
Stack-Size Considerations .......................................... 13-36
Virtual-Memory Cache for Routines ............................... 13-37
   The sysprocedures System Catalog Table .................... 13-37
   UDR Cache .................................................. 13-38

I/O Considerations .................................................... 13-38
Isolating System Catalog Tables ................................ 13-39
Balancing the I/O Activities ..................................... 13-39
In This Chapter

This chapter describes performance considerations for user-defined routines (UDRs) and includes the following topics:

- Optimization of SQL statements with UDRs
- Selection of a virtual-processor class
- Considerations for parallel execution of SPL routines
- Memory considerations
- I/O considerations

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.”

This section covers the following UDR-specific optimization topics:

- How SPL routines are optimized
- How updating statistics affects UDRs
- How to write optimization UDRs
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.

<table>
<thead>
<tr>
<th>SET OPTIMIZATION Statement</th>
<th>Algorithm Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET OPTIMIZATION HIGH</td>
<td>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.</td>
</tr>
<tr>
<td>SET OPTIMIZATION LOW</td>
<td>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.</td>
</tr>
</tbody>
</table>

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`, `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.
Updating Statistics for an SPL Routine

- **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 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 *Informix Guide to SQL: Reference*. For more information about the UPDATE STATISTICS statement, refer to the *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.

For some functions, specifying the same cost for all invocations of the function is sufficient. For example, finding the absolute value of -68327 is no more time-consuming than finding the absolute value of -4. In these cases, a constant value that estimates the cost is appropriate.
**Calculating Cost**

---

**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 execute a function to compute 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 user-defined data type (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 13-12.

---

**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:

   ```sql
   SET EXPLAIN ON;
   SELECT count(*) from bigtable;
   ```

   Time the query.
Selectivity and Cost Examples

2. Let \( \text{secost} \) be the cost the optimizer assigned for the scan. Read the \texttt{sqexplain.out} file to get \( \text{secost} \).
   For information about \texttt{sqexplain.out}, refer to the Performance Guide.

3. Let \( \text{sacost} \) be the time required to complete the SQL statement.

4. Execute and time your function. Let \( \text{facost} \) be the actual time required to execute the function once.
   The cost of executing the function once can be approximated as follows:
   \[
   (\left(\frac{\text{secost}}{\text{sacost}}\right) \times \text{facost})
   \]
   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 \texttt{contains\_sel()} to determine the selectivity of the \texttt{contains()} function.

```sql
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:

```sql
CREATE FUNCTION expensive(c cust int)
RETURNING boolean WITH(percall_cost=50,selconst=.1)
EXTERNAL NAME "/ix/9.2/exp_func.so" LANGUAGE c;

CREATE FUNCTION cheap(c cust int)
RETURNING boolean WITH(percall_cost=1,selconst=.1)
EXTERNAL NAME "/ix/9.2/exp_func.so" LANGUAGE c;
```
Extending UPDATE STATISTICS

When both of these functions are in one SQL statement, the optimizer executes the `cheap0` function first because of the lower cost. The following SET EXPLAIN output, which lists `cheap0` first in the Filters: line, shows that indeed the optimizer did execute `cheap0` 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 DataBlade Developers Kit.

---

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 user-defined data types, 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 `statcollect0` 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 user-defined data types 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 user-defined data types.

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 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 user-defined data type, you must create these functions.

The stat Data Type

The `statcollect()` and `statprint()` functions use an SQL data type called `stat`. 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()`:
Support Functions for UPDATE STATISTICS

- The `statdata` field should contain the histogram for the distribution. User-defined data types 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.
Support Functions for UPDATE STATISTICS

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 collected by the `statcollect()` function 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 collected by the `statcollect()` function.

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 in the `\%INFORMIXDIR\dbdk\examples\Types\dapi\Statistics\Box\src\c` directory, after you install the DataBlade Developers 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 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 or as external functions. The following example shows CREATE FUNCTION statements that specify negator functions:

```sql
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.

Tip: The database server implements its own threads to schedule client-application requests. Therefore, this implementation is consistent across all platforms. 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.

<table>
<thead>
<tr>
<th>Virtual-Processor Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Central processing (the primary VP class, which controls client-application requests)</td>
</tr>
<tr>
<td>AIO</td>
<td>Asynchronous disk I/O</td>
</tr>
<tr>
<td>SHM</td>
<td>Shared-memory network communication</td>
</tr>
<tr>
<td>JVP</td>
<td>Special VP class for execution of UDRs written in Java</td>
</tr>
<tr>
<td>User-defined</td>
<td>Special VP class for additional types of processing</td>
</tr>
</tbody>
</table>

The following section provides information about how to choose a virtual-processing class for a user-defined routine. 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.

<table>
<thead>
<tr>
<th>Virtual-Processor Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU VP</td>
<td>VP for execution of SPL routines. Default VP for execution of C UDR. The UDR must be well behaved to run in the CPU VP.</td>
</tr>
<tr>
<td>User-defined VP</td>
<td>VP for execution of C UDR that has some ill-behaved characteristics. Also called an Extension VP (EXT VP)</td>
</tr>
<tr>
<td>Java VP (JVP)</td>
<td>VP for execution of Java UDR. This VP class contains the Java Virtual Machine (Java VM).</td>
</tr>
</tbody>
</table>

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.

**Tip:** For general information on CPU virtual processors, see the “Administrator’s Guide.”

The CPU VP class is the default VP class for a user-defined routine. 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.
Choosing a Virtual-Processor Class

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:

```sql
CREATE FUNCTION getTotal(order_num, state_code)
RETURNS MONEY
...
END FUNCTION
```

You *cannot* run an SPL routine in a user-defined VP.

By default, a C UDR runs in the CPU VP class. Generally, user-defined routines 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

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.

**User-Defined Virtual-Processor Class**

For DataBlade or external routines, you can designate a user-defined class of virtual processors, called user-defined VPs, to run the routine.

**Important**: 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. Thus, you should use the user-defined VP with caution.
You cannot run an SPL routine in a user-defined VP. SPL routines always run in the CPU VP.

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.

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:
   \[
   \text{CLASS = vpclass\_name}
   \]
   In this syntax, vpclass\_name is the name of the user-defined VP class that you have configured in the database server.

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:
   \[
   \text{VPCLASS newvp, num=3# New VP class for testing}
   \]
   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.

Important: If you create a new virtual-processor class, you must remove the SINGLE\_CPU\_VP parameter from the ONCONFIG file.
When you configure a new class of user-defined virtual processors to run user-defined routines, you must ensure that the name of the class agrees with the name that you assigned to the CLASS routine modifier in the CREATE FUNCTION or CREATE PROCEDURE statement. The class name is not case sensitive.

When you register functions or procedures with the CREATE FUNCTION or CREATE PROCEDURE statement, you can reference any user-defined VP class that you like. The VP class need not exist when the external routine is registered. However, if you try to use a routine that refers to a user-defined VP class, 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 DataBlade API Programmer’s Manual. For information on the VPCLASS configuration parameter, see the Administrator’s Reference.

You cannot run a Java UDR in a user-defined VP. Java routines always run in a Java VP. You must specify the following CLASS routine modifier when you register a Java UDR:

```
CLASS = jvp
```

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 on-line. 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.

Using UDRs in Parallel

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 parallel database query 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 following UDRs can be execute in parallel if they are part of a parallel database query and PDQPRIORITY is turned on:

- UDRs written in C that call only DataBlade API functions that are PDQ thread-safe can execute in parallel.
- UDRs written in Java 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 user-defined data types, such as the following operators that are used for a generic B-tree index:
    - less than (<)
    - less than or equal (<=)
    - equal (=)
    - greater than or equal (>=)
    - greater than (>)

UDRs can execute in parallel in the following situations if they are part of a parallel database query 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 executed by a generic B-tree index
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 parallel database query that contains two UDRs:

```sql
SELECT c_udr1(tabid) FROM tab
WHERE tablename = c_udr2(3) AND
  tabid > 100;
```
Executing UDRs in Parallel

If the table \texttt{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 \texttt{tab} is performed by multiple scan threads in parallel. Each scan thread fetches a row from a fragment of \texttt{tab}.
- Each scan thread also evaluates the WHERE condition in parallel. Each scan thread executes the UDR \texttt{c\_udr2()} in parallel.
- Each scan thread also executes the UDR \texttt{c\_udr1()} in the select list in parallel.

Parallel UDR in a Join

The following sample query contains a join between table \texttt{t1} and \texttt{t2}:

\begin{verbatim}
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;
\end{verbatim}

If the tables \texttt{t1} and \texttt{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 \texttt{t1} is performed by multiple scan threads in parallel. Each scan thread fetches a row from a fragment of \texttt{t1} and executes the UDR \texttt{c\_udr1()} in parallel.
- The scan of table \texttt{t2} is performed by multiple scan threads in parallel. Each scan thread fetches a row from a fragment of \texttt{t2} and executes the UDR \texttt{c\_udr2()} in parallel.
- The join of tables \texttt{t1} and \texttt{t2} is performed by multiple join threads in parallel. Each join thread fetches a row from a fragment of \texttt{t2} and executes the UDR \texttt{c\_udr()} in parallel.
Executing UDRs 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.
Executing UDRs 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 user-defined routine area, and then execute the following sample query:

```sql
INSERT INTO cir_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.

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:

```sql
SELECT grp, uda(udt_col) FROM tab GROUP BY grp;
```

If the data type of column udt_col is a user-defined data type (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:

```sql
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.
Executing UDRs in Parallel

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 parallel database query.

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

C or Java 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
Enabling Parallel UDRs

If a UDR written in C or Java 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 tuning 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:
   ```sql
   CREATE TABLE natural_number (x integer)
   FRAGMENT BY round robin
   IN dbspace1, dbspace2;
   ```

2. Write a function that is PDQ thread safe.
   For example, the following C prototype shows a function that takes an integer and determines if it is a prime number:
   ```c
   mi_boolean is_prime_number (x mi_integer);
   ```
   For more information on writing 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:
   ```sql
   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:
   ```sql
   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 user-defined routine.

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.

On Windows NT, 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 parallel database queries 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:

- Stack-size considerations for external routines
- The virtual-memory cache for SPL and external routines

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.

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. For more information on the DataBlade API, refer to the DataBlade API Programmer’s Manual.

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
```
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 user-defined routine, 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 user-defined routine, 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)
I/O Considerations

- 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 user-defined routines and triggers in the following system catalog tables:

- `sysprocbody`
- `sysprocedures`
- `sysproclan`
- `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.
Index

A
abs() built-in function 6-8
acos() built-in function 6-8
Active set 4-11
Aggregate function
   built-in 2-12, 8-3
   definition of 5-13, 6-10
   dropping 8-20
   extending built-in functions 8-4
   for an opaque type 10-8
   implicit execution 13-24
   overloading operators 8-4
ALTER FUNCTION
   statement 3-15, 4-16
ALTER PROCEDURE
   statement 3-15, 4-16
ALTER ROUTINE statement 3-15, 4-16, 12-12
ALTER TABLE statement 10-30
ANSI compliance
   icon Intro-9
   level Intro-14
   routine signature 3-13
Argument. See Routine argument.
Arithmetic operator
   description of 6-5
   for an opaque type 10-22
asin() built-in function 6-8
assign() support function
   description of 10-7, 11-19
   for smart large objects 11-22
   summary of 11-4
atan2() built-in function 6-8
atan() built-in function 6-8

B
B-tree. See Generic B-tree.
Built-in aggregate 2-12, 8-3
Built-in cast 7-3, 7-8, 7-13
Built-in data type 5-12
   description of 5-6
   routine resolution with 3-20
Built-in function
   definition of 5-13, 6-8
   for an opaque type 10-7, 10-22
   list you can overload 6-8
   list you cannot overload 6-9
   overloadable 3-16

C
CALL statement 3-6, 3-11
Cast
   built-in 7-3, 7-8, 7-13
   by a support function 10-18, 10-31
   CAST AS keywords 7-7
   routine resolution with 3-18
   types of 7-3
See also User-defined cast.
CAST AS keywords 7-7
Cast (::) operator 7-7
error message files Intro-11
machine notes Intro-13
on-line help Intro-11
on-line manuals Intro-11
printed manuals Intro-11
related reading Intro-13
release notes Intro-13
DROP AGGREGATE statement 8-20
DROP CAST statement 7-13, 10-31
DROP FUNCTION statement 3-15, 4-16, 10-31, 12-9
DROP OPCLASS statement 9-22
DROP PROCEDURE statement 3-15, 4-16
DROP ROUTINE statement 3-15, 4-16, 10-31, 12-9
DROP TYPE statement 10-31

E
End-user routine
definition of 5-13, 6-11
for an opaque data type 10-9
privileges on 6-11
Environment variable,
NODEFDAC 12-4
en_us.8859-1 locale Intro-4
equal() function 6-6, 9-7, 10-23, 10-24, 13-24, 13-29
Error message files Intro-11
EXVP. See User-defined virtual processor.
EXECUTE FUNCTION statement invoicing a user-defined function 3-4
invoicing an iterator function 4-12
OUT parameters with 3-4
SPL statements 3-11
Execute privilege 4-17, 12-3, 12-4
DBA keyword, effect of 12-7
objects referenced by a routine 12-5
EXECUTE PROCEDURE statement invoicing a user-defined procedure 3-4
SPL statements 3-11
Execution plan 3-11, 4-17, 13-4

F
Feature icons Intro-8
Features of this product, new Intro-5
Find Error utility Intro-12
finderr utility Intro-12
Functional index 9-4

G
Generic B-tree
default operator class 9-6, 9-10
extending 9-10, 9-16
new operator class 9-16
strategy functions 9-7
Global Language Support (GLS) Intro-4
API 11-24
GRANT statement
eample using signature 3-15, 12-5
eample using specific name 3-15
Execute privilege 6-11, 10-20, 12-4
Resource privilege 9-16, 10-15
Usage privilege 4-16, 10-19
greaterthinearual() function 6-6, 9-7, 13-24, 13-29
greaterthan() function 6-6, 9-7, 13-24

H
HANDLESNULLS routine
modifier 4-25
description 4-23
overloaded routines 3-27
statcollect() function 13-15
hex() built-in function 6-9

I
Icons
compliance Intro-9
feature Intro-8
Important Intro-8
platform Intro-8
product Intro-8
Tip Intro-8
Warning Intro-8
fx_replace_module() function 12-10
IMPEXP data type
casting from 10-18
casting to 10-18
description of 11-15
parameter in import function 10-17
return value in export function 10-17
IMPEXPBIN data type casting from 10-18
casting to 10-18
description of 11-17
parameter in importbinary function 10-17
return value in exportbinary function 10-17
Implicit cast 7-6
Import support function as cast function 10-18
description of 10-6, 11-16
for smart large objects 11-22
parameter type 10-17
return type 10-17
summary of 11-4
Important paragraphs, icon for Intro-8
Importbinary support function as cast function 10-18
description of 10-6, 11-17
for smart large objects 11-22
parameter type 10-17
return type 10-17
summary of 11-4
INDEX operator 10-24
Index
description of 9-4
See also Generic B-tree; R-tree index; Secondary access method.
Industry standards, compliance with Intro-14
Informix GLS API 11-24
informix user 4-16
informix user account 3-18, 4-14
INFORMIXDIR/bin directory Intro-5
Input support function as cast function 7-10, 7-12, 10-18
description of 10-6, 11-6
locale-sensitive data 11-24
parameter type 10-16
return type 10-16
summary of 11-3
INTERNAL routine modifier 4-23
ISO 8859-1 code set Intro-4
Iterator function
active set 4-11
creating 4-11
definition of 4-11
invoking 4-12
registering 4-12
writing 2-21
ITERATOR routine modifier 2-22, 4-12, 4-23, 4-24

L
Language
external 2-6, 4-26
SPL 2-5
types of 2-3
length() built-in function 6-9
lessthanorequal() function 6-6, 9-7, 13-24
lessthan() function 6-6, 9-7, 13-24
like() function 6-5
Locale Intro-4
default Intro-4
en_us.8859-1 Intro-4
logn() built-in function 6-8
log() built-in function 6-8
lohandles() support function
description of 10-7
summary of 11-5
LVARCHAR data type
casting from 10-18
casting to 10-18
definition of 11-6
locale-sensitive data 11-24
parameter to input function 10-16
return value of output function 10-16

M
Machine notes Intro-13
matches() function 6-5
Message file for error messages Intro-11
minus() function 6-5
mi_lvchar data type
definition of 11-6
locale-sensitive data 11-24
mi_real data type 10-14
miSending data type
locale-sensitive data 11-25
mi_unsigned_integer data type 10-13
mi_unsigned_smallint data type 10-13
Modal routines 4-5
mod() built-in function 6-8
Multirepresentational data 10-5
Multirepresentational data 10-5

N
Name space, DataBlade-module objects 4-13
Named row type
description 5-10
routine-resolution precedence 3-19
subtype 3-22
supertype 3-22
type hierarchy 3-14, 3-22
user-defined cast 7-4
negate() function 6-5
Negator function 2-17, 13-16
NEGATOR routine modifier 4-23, 4-25
New features of this product Intro-5
NODEFDAC environment variable 12-4
Noncursor function 4-11
Nonvariant function 4-7
NOT VARIANT routine modifier 4-7, 4-23
notequal() function 6-6
Null value
aggregate initialization 8-9
as wildcard argument 3-27
in statcollect() function 13-14
in support functions 8-13
See also HANDLESNULL routine modifier.
octet_length() built-in function 6-9
On-line help Intro-11
On-line manuals Intro-11
onmode utility 13-22
onstat utility 3-8, 13-23, 13-36
Opaque data type
  as parameter 10-14
  bit-hashable 10-24, 10-26, 11-23
  casts for 7-4
  comparing 10-23, 11-22
  constraints on 10-24
  definition of 1-7, 5-12, 10-3
  dropping 10-30
  example, circle 10-32
  example, image 10-39
  extended identifier 10-15
  fixed-length 10-5, 10-11
  in sysxtdtypeauth table 10-20
  indexes on 10-27
  internal structure 10-4
  locale-specific data 11-23
  memory alignment of 10-12
  pass by reference 11-7, 11-9
  pass by value 11-7, 11-9
  privileges 10-19, 10-30
  registering 10-15
  structure size 10-11
  system catalog table 10-20
  varying-length 10-12
Operator
  aggregate function 6-10
  built-in function 6-8
  cast function. See Cast.
  end-user routine 6-11
  operator 6-4
  operator function 6-4
Operator function
  arithmetic 10-21
  definition of 5-13, 6-4
  for an opaque data type 10-8, 10-21
  relational 6-6, 10-23
  text 6-5, 10-21
Optimization level 13-4
OUT parameter 3-4, 4-8
Output support function
  as cast function 7-10, 7-12, 10-18
  description of 10-6, 11-8
  locale-sensitive data 11-24
  parameter type 10-16
  return type 10-16
  summary of 11-3
Owner. See Routine owner.
Parallel database queries 13-28
Parallel UDRs
  description 2-17
  enabling 13-33
  execution of 13-24
  GROUP BY 13-27
  INSERT 13-28
  joins 13-26
  scans 13-26
  SELECT list 13-27, 13-28
  sorts 13-30
  when use 13-25
PARALLELIZABLE routine
  modifier 4-25
Parameter. See Routine parameter.
P-code 13-38
  definition 4-18
  size of 13-37
SPL routine 3-11
See also SPL p-code.
PERCALL_COST routine
  modifier 4-23
Platform portability 4-13
plus() function 6-5
POINTER data type 4-25
Polymorphism 3-14, 4-4
positive() function 6-5
Pound (#) sign, SLV indicator 4-10
pow() built-in function 6-8
Printed manuals Intro-11
Privilege
  database-level 4-16
  DBA 4-16, 12-7
  Execute 4-17, 12-3, 12-4
  for a user-defined routine 4-15
  on a support function 10-20, 10-30
  on an end-user routine 6-11
  on an opaque type 10-15, 10-19, 10-30
  on an operator class 9-16
  Resource 4-16
  routine-level 4-17, 10-20, 12-3
  type-level 10-20
Product icons Intro-8
Program group
  Documentation notes Intro-13
  Release notes Intro-13
Query optimizer 2-14, 3-8, 13-6
Query parser 3-7, 3-17
Query plan 3-8, 13-3
Receive support function
  as cast function 10-18
  description of 10-6, 11-11
  locale-sensitive data 11-25
  parameter type 10-16
  return type 10-16
  summary of 11-4
Related reading Intro-13
Relational operator
  description of 6-6
  for an opaque type 10-23
  Release notes Intro-13
  Resource privilege 4-16
  Return value. See Routine return value.
REVOKE statement
  Execute privilege 10-30, 12-4
  Usage privilege 4-16, 10-30
using specific name 3-15
rofferr utility Intro-12
root() built-in function 6-8
round() built-in function 6-8
Routine 1-3, 2-3
Routine argument
definition 4-5
determining number of 4-5
distinct data type as 3-24
in routine resolution 3-17
in routine-state space 3-9
modal 4-5
named row type as 3-22
not matching parameter data
type 3-23
registering 4-26
wildcard 3-27
Routine identifier
derscription 10-17
in sysprocedures 4-26
Routine instance 3-9
Routine invocation, definition 3-9
Routine manager
creating a routine sequence 3-9
loading a shared-object file 3-8
managing routine execution 3-10
role of 3-8
stack space and 13-36
Routine modifier
CLASS 4-23, 4-24, 13-19, 13-21
COSTFUNC 4-23
eXternal routine 4-22
HANDLESNULLS 4-23, 4-25
INTERNAL 4-23
ITERATOR 2-22, 4-12, 4-23, 4-24
NEGATOR 4-23, 4-25
NOT VARIANT 4-7, 4-23
PARALLELIZABLE 4-25
PERCALL_COST 4-23
SELCONST 4-23
SELFUNC 4-24
specifying 4-26
STACK 4-24, 13-37
VARIANT 4-7, 4-24
Routine name
ANSI-compliant 3-13
candidate routines 3-18
choosing 4-4
component of routine
signature 3-12
not ANSI-compliant 3-12
overloaded 3-11, 3-13
registering 4-26
specific. See Specific routine name.
Routine overloading
assigning specific routine
name 3-15
built-in SQL functions 3-16
definition 3-11
description of 3-14, 4-4
in operator binding 2-14
invoking overloaded routine 3-16
using 6-3, 9-9
Routine owner
ANSI-compliant database 3-13
component of routine
signature 3-12
database not ANSI
compliant 3-12
in specific routine name 3-15
registering 4-26
Routine parameter
component of routine
signature 3-12
declaring 4-5
definition 4-5
determining number of 4-5
overloaded 3-13
registering 4-26
Routine resolution
candidate list 3-17
candidate list of routines 3-18
definition 3-11, 3-17
description of 3-17
effect of inheritance 3-22
effect of null value argument 3-27
order of arguments 3-23
precedence list 3-17
precedence list of data types 3-19
purpose 6-3, 9-9
type hierarchy in 3-22
understanding 3-11
Routine return value
in routine-state space 3-9
multiple values 4-8
nonvariant 4-7
using 4-6
Routine sequence, definition 3-9
Routine signature
ANSI-compliant 3-13
components of 3-12, 4-27
description 3-12
in routine resolution 3-11, 3-17
not ANSI-compliant 3-12
registering 3-13
uniqueness of 3-12
using 3-13
Routine type 3-12, 3-18, 4-26
Routine-level privilege 4-17, 12-3
Routine-state space 3-9
Row type, named. See Named row
type.
R-tree index
default operator class 9-8
uses of 9-5

S
Sample-code conventions Intro-10
Scans, parallel 13-35
Secondary access method
defined by database server 9-4
defining new operator
classes 9-21
definition of 9-4
user-defined 9-8
SELCONST routine modifier 4-23
SELECT statement
DISTINCT keyword 10-25, 11-22
GROUP BY clause 10-24
ORDER BY clause 10-25, 11-22
UNION keyword 10-25, 11-22
UNIQUE keyword 10-25, 11-22
Selectivity function 2-17
SELFUNC routine modifier 4-24
Send support function
as cast function 10-18
description of 10-6, 11-13
locale-sensitive data 11-25
parameter type 10-16
return type 10-16
summary of 11-4
SENDBACK data type
casting from 10-18
casting to 10-18
definition of 11-11
locale-sensitive data 11-25
parameter in receive
function 10-16
return value in send
function 10-16
SERVER_LOCALE environment
variable 11-24
SET EXPLAIN
parallel scans 13-35
serial scans 13-35
SET OPTIMIZATION
statement 13-4
Shared library. See Shared-object
file.
Shared-object file
creating 2-23, 4-14
loading 3-8
ownership of 4-14
permissions on 4-14
reloading 12-9
replacing 12-10
unloading 12-9
Signature. See Routine signature.
SIGUSR1 signal 13-34
SINGLE_CPU_VP configuration
parameter 13-21
sin() built-in function 6-8
SLV. See Statement-local variable
(SLV).
Smart large object 10-5, 11-15
Software dependencies Intro-4
SPECIFIC keyword 3-15, 4-27
Specific routine name 4-27
assigning 3-15
definition 3-15
registering 4-26
SPL function
description of 2-5
OUT parameter 4-8
SPL procedure, description of 2-5
SPL routine
caching sysprocedures 13-37
default virtual-processor
class 13-20
definition of 2-5
dependency list 3-11, 4-17
description of 2-5
designing 2-23, 4-3
executing 3-11
execution plan 13-4
FOREACH loop 4-12
invoking a UDR 3-6
optimization level 13-4
optimizing 13-4
p-code 3-11, 4-18
registering 4-17
SPL cache 13-38
updating statistics 13-6
with RETURN WITH
RESUME 4-11
SQL code Intro-10
SQL data type
built-in 3-20
collection 3-26
IMPEXP 11-15
IMPEXPBIN 11-17
in registration 4-25
LVARCHAR 11-6
POINTER 4-25
SENDRECV 11-11
SQL statement
execution plan 4-17
invoking a UDR 3-3
optimizing 3-8, 4-17, 13-3, 13-4
parsing 3-7, 4-17
query optimizer 3-8
query parser 3-7
query plan 3-8
routine sequence 3-9
specific name 3-15
statement-local variable 4-10
where invalid 4-7
SQL-invoked function
aggregate functions 6-10
built-in functions 6-8
cast functions 7-9
definition of 6-3
derender routines 6-11
opaque data types 10-21
operator functions 6-4
sqrt() built-in function 6-8
STACK routine modifier 4-24,
13-37
Stack space 13-36
STACKSIZE configuration
parameter 13-36
Statement-local variable (SLV)
declaring 4-10
definition of 4-10
OUT parameters and 4-10
referencing in function 4-10
scope of 4-10
Statistics function 2-17
stores_demo database Intro-4
Strategy function
description of 9-6
for generic B-tree 9-7
Structured Query Language (SQL).
See SQL statement.
Subtype. See Named row type.
superstores_demo database Intro-4
Supertype. See Named row type.
Support function
as casts 10-18
assign() 10-7, 11-4
bulk copy 11-14
compare() 10-7, 11-5
description of 9-6
destroy() 10-7, 11-4
dropping 10-31
export 10-6, 11-4
exportbinary 10-6, 11-4
for generic B-tree 9-7
import 10-6, 11-4
importbinary 10-6, 11-4
Informix GLS API 11-24
input 10-6, 11-3
lhandles() 10-7, 11-5
output 10-6, 11-3
privileges 10-20, 10-30
receive 10-6, 11-4
registering 10-16
routine identifier 10-17
send 10-6, 11-4
sysams system catalog table 9-21
syscasts system catalog table 7-5,
7-13, 10-18, 10-31
syscolumns system catalog
table 13-6
sysindices system catalog
table 13-6
syslanguauth system catalog
table 2-6
sysopclasses system catalog
table 9-22
sysprocauth system catalog
table 2-4, 4-18, 10-20, 10-30,
12-4, 13-38
sysprocbody system catalog
table 2-5, 3-11, 4-18, 4-27, 13-38
sysprocedures system catalog
table cached in memory 13-37
candidate routines 3-18
columns of 4-26
contents of 13-37
description 2-4, 10-31
I/O considerations 13-38
path column 3-8, 12-11
routine identifier 4-26
support functions 10-17
UDR information 4-18
sysproplan system catalog
table 4-18, 4-27, 13-5, 13-7, 13-38
sysroutinelangs system catalog
table 2-6
systables system catalog
table 13-6
System catalog table
isolating 13-39
sysams 9-21
syscasts 7-5, 10-18
syscolumns 13-6
sysindices 13-6
sylangauth 2-6
sylangauth 2-4, 4-18, 10-20, 12-4,
13-38
sysprocbody 2-5, 3-11, 4-18, 4-27,
13-38
sysprocedures 2-4, 4-18, 4-27,
13-37
sysproplan 4-18, 4-27, 13-5, 13-7,
13-38
sysroutinelangs 2-6
systables 13-6
systrigbody 13-38
systriggers 13-38
sysxtdtypeauth 5-8, 10-20
sysxtdtypes 5-8, 10-15, 13-38
System requirements
database Intro-4
software Intro-4
systrigbody system catalog
table 13-38
systriggers system catalog
table 13-38
sysxtdtypeauth system catalog
table 5-8, 10-20, 10-30
sysxtdtypes system catalog
table 5-8, 10-15, 10-31

T

tan() built-in function 6-8
Text operator
description of 6-5
for an opaque type 10-22
Thread
definition 13-18
insert 13-28, 13-30
join 13-26
primary 13-35
secondary 13-35
sort 13-30
sqlexec 13-35
Thread-safe UDRs 13-31, 13-32
times() function 6-5
Tip icons Intro-8
Triggered action statements 2-20
trunc() built-in function 6-8
Type hierarchy 3-14, 3-19, 3-22
Type inheritance 3-22

U

UDR. See User-defined routine.
UNIX operating system
default locale for Intro-4
UPDATE STATISTICS
statement 3-15, 13-6
User-defined aggregate
definition of 2-12, 8-3
dropping 8-20
parallel execution 13-25, 13-29
User-defined cast
cast function 7-9
creating 7-5
definition of 7-4
dropping 7-13
implicit 7-6
kinds of 7-6
straight 7-8
See also Cast.
routine sequence for 3-9
routine-level privileges 4-17, 12-3
routine-state space 3-9, 3-10
signature. See Routine signature.
tasks of 2-7
unloading 12-9
updating statistics 13-6
user state 3-10
well-behaved 13-20, 13-21
wildcard argument 3-27
writing 2-23
See also User-defined function;
    User-defined procedure.
User-defined virtual processor
    adding 13-22
    dropping 13-22
Users, types of Intro-3

V
Variant function 4-7
VARIANT routine modifier 4-7,
    4-24
Virtual processor (VP)
    choosing for UDR 13-19
    classes 13-18
    CPU 13-19
    definition of 13-18
    monitoring 13-23
    setting number of 13-34
    user-defined 13-20
    using 13-18
    VP classes 13-19
    VPCLASS configuration
        parameter 13-21, 13-34
    VP. See Virtual processor (VP).

W
Warning icons Intro-8
Wildcard, argument for a
    routine 3-27
Windows NT
    default locale for Intro-4
X
X/Open compliance level Intro-14