Extending INFORMIX®-Universal Server:

User-Defined Routines

Version 9.1
March 1997
Part No. 000-3803

Published by INFORMIX® Press

Informix Software, Inc.
4100 Bohannon Drive
Menlo Park, CA 94025-1032

Copyright © 1981-1997 by Informix Software, Inc. or their subsidiaries, provided that portions may be copyrighted by third parties, as set forth in documentation. All rights reserved.

The following are worldwide trademarks of Informix Software, Inc., or its subsidiaries, registered in the United States of America as indicated by "®," and in numerous other countries worldwide:

INFORMIX®; INFORMIX®-OnLine Dynamic Server™; DataBlade®

The following are worldwide trademarks of the indicated owners or their subsidiaries, registered in the United States of America as indicated by "®," and in numerous other countries worldwide:

Adobe Systems Incorporated: PostScript®
Sun Microsystems, Inc.: Solaris®

All other marks or symbols are registered trademarks or trademarks of their respective owners.

Documentation Team: Diana Chase, Abby Knott, Dawn Maneval, Patrice O'Neill, Virginia Panlasigui

To the extent that this software allows the user to store, display, and otherwise manipulate various forms of data, including, without limitation, multimedia content such as photographs, movies, music and other binary large objects (blobs), use of any single blob may potentially infringe upon numerous different third-party intellectual and/or proprietary rights. It is the user's responsibility to avoid infringements of any such third-party rights.

RESTRICTED RIGHTS/SPECIAL LICENSE RIGHTS

Software and documentation acquired with US Government funds are provided with rights as follows: (1) if for civilian agency use, with Restricted Rights as defined in FAR 52.227-19; (2) if for Dept. of Defense use, with rights as restricted by vendor's standard license, unless superseded by negotiated vendor license as prescribed in DFAR 227.7202. Any whole or partial reproduction of software or documentation marked with this legend must reproduce the legend.

Table of Contents

Introduction

About This Manual . . . 3
- Organization of This Manual . . . 3
- Types of Users . . . 4
- Software Dependencies . . . 4
- Assumptions About Your Locale . . . 4
- Demonstration Database . . . 5

Major Features . . . 5

Documentation Conventions . . . 5
- Typographical Conventions . . . 6
- Icon Conventions . . . 7

Additional Documentation . . . 8
- On-Line Manuals . . . 8
- Printed Manuals . . . 9
- Error Message Files . . . 9
- Documentation Notes, Release Notes, Machine Notes . . . 10

Compliance with Industry Standards . . . 10

Informix Welcomes Your Comments . . . 11

Chapter 1 Overview of User-Defined Routines

What Is a User-Defined Routine? . . . 1-3

Tasks That You Can Perform with UDRs . . . 1-5
- Encapsulate Multiple SQL Statements . . . 1-6
- Extend Functions for Built-In Data Types . . . 1-7
- Support New Data Types . . . 1-10
- Use as a Triggered Action . . . 1-13
- Restrict Access to a Table . . . 1-14
- Enforce Business Rules . . . 1-14

Ways to Invoke User-Defined Routines . . . 1-15

Chapter 2 Routine Overloading and Routine Resolution

What Is Routine Overloading? 2-3
The Routine Signature 2-4
The Routine-Resolution Process 2-7
Candidate List of Routines 2-8
Precedence List of Data Types 2-9
Built-In SQL Functions That You Can Overload 2-17
Operator-to-Function Binding 2-19

Chapter 3 Designing a User-Defined Routine

Designing the Routine Source 3-3
Naming Your Routine 3-4
Number of Arguments 3-5
Coding Standards 3-5
Shared Object Libraries 3-6
Creating a Shared Library 3-6
Loading a Shared Library into Memory 3-7
Replacing a Shared Object File 3-9
Registering External Routines 3-11
Registering an External Function 3-11
Registering an External Procedure 3-13
Registering an External Routine with Modifiers 3-14
Returning Multiple Values from External Functions 3-15
OUT Parameters and Statement Local Variables 3-15
Iterator Function 3-17
Privileges for Registering a Routine 3-19
Privileges for Executing a Routine 3-19
Privileges on Objects Associated with a Routine 3-21
Executing a Routine as DBA 3-22

Chapter 4 Debugging User-Defined Routines

Invoking a Routine 4-3
Invoking a Function with the EXECUTE statement 4-3
Invoking a Procedure with the EXECUTE statement 4-5
Invoking a Function with the CALL Statement 4-5
Invoking a Procedure with the CALL Statement 4-5
Invoking a Function in an Expression. 4-6
Private Installation 4-8
Installing and Registering DataBlade Modules 4-8
Debugging a DataBlade Module 4-9
Connecting to the Server from a Client 4-9
Loading the DataBlade Module 4-9

Identifying the Server Process 4-10
Starting the Debugger 4-11
Setting Breakpoints 4-11
Symbols in Shared Object Files 4-12

Chapter 5 Performance Considerations

SPL Considerations 5-3
SPL Compilation 5-4
SPL Execution 5-5
SPL Optimization 5-5
Updating Statistics for a UDR 5-7
Choosing a Virtual-Processor Class 5-8
CPU Virtual-Processor Class 5-8
User-Defined Virtual-Processor Class 5-9
Considerations for Parallel Execution of SPL Routines 5-11
Number of Virtual Processors (VPCLASS) 5-12
Memory Considerations 5-12
Stack-Size Considerations 5-12
Setting Stack Sizes for User-Defined Routines 5-13
Virtual-Memory Cache for Routines. 5-13
I/O Considerations 5-14
Isolate System Catalogs 5-15
Balance the I/O Activities 5-15

Index

Introduction

About This Manual 3
Organization of This Manual 3
Types of Users 4
Software Dependencies 4
Assumptions About Your Locale 4
Demonstration Database 5

Major Features 5

Documentation Conventions 5
Typographical Conventions 6
Icon Conventions 7
Comment Icons 7
Feature and Product Icons. 7
Compliance Icons 8

Additional Documentation 8
On-Line Manuals 8
Printed Manuals 9
Error Message Files 9
Documentation Notes, Release Notes, Machine Notes 10

Compliance with Industry Standards 10

Informix Welcomes Your Comments 11

Read this introduction for an overview of the information provided in this manual and for an understanding of the documentation conventions used.

About This Manual

Extending INFORMIX-Universal Server: User-Defined Routines explains how to define your own functions and procedures for use in an INFORMIX-Universal Server database. It describes common considerations for SPL routines and external routines.

Organization of This Manual

This manual includes the following chapters:

- This Introduction provides an overview of the manual and describes the documentation conventions used.
- Chapter 1, "Overview of User-Defined Routines," describes user-defined routines.
- Chapter 2, "Routine Overloading and Routine Resolution," describes how you can define routines with the same name and how Universal Server determines which routine to execute.
- Chapter 3, "Designing a User-Defined Routine," describes how to create and register external user-defined routines.
- Chapter 4, "Debugging User-Defined Routines," describes how to invoke and debug your routines.
- Chapter 5, "Performance Considerations," describes performance considerations for user-defined routines.

Types of Users

This manual is written for the experienced application developer who might be creating application-specific routines for application end-users. This developer should have more understanding of database theory than a client-application developer.

This reader should have a thorough grasp of database theory, UNIX, C, and SQL.

Software Dependencies

This manual assumes that you are using INFORMIX-Universal Server, Version 9.1, as your database server.

In this manual, all instances of Universal Server refer to INFORMIX-Universal Server.

Assumptions About Your Locale

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

This manual assumes that you are using the default locale, **en_us.8859-1**. This locale supports U.S. English format conventions for dates, times, and currency. In addition, this locale supports the ISO 8859-1 code set, which includes the ASCII code set plus many 8-bit characters such as é, è, and ñ.

If you plan to use nondefault characters in your data or your SQL identifiers, or if you want to conform to the nondefault collation rules of character data, you need to specify the appropriate nondefault locale(s). For instructions on how to specify a nondefault locale, additional syntax, and other considerations related to GLS locales, see the *Guide to GLS Functionality*.

Demonstration Database

The DB-Access utility, which is provided with your Informix database server products, includes a demonstration database called **stores7** that contains information about a fictitious wholesale sporting-goods distributor. Sample command files are also included.

Many examples in Informix manuals are based on the **stores7** demonstration database. The **stores7** database is described in detail and its contents are listed in Appendix A of the *Informix Guide to SQL: Reference*.

The script that you use to install the demonstration database is called **dbaccessdemo7** and is located in the **$INFORMIXDIR/bin** directory. For a complete explanation of how to create and populate the demonstration database on your database server, refer to the *DB-Access User Manual*.

Major Features

The Introduction to each Version 9.1 product manual contains a list of major features for that product. The Introduction to each manual in the Version 9.1 *Informix Guide to SQL* series contains a list of new SQL features.

Major features for Version 9.1 Informix products also appear in release notes.

Documentation Conventions

This section describes the conventions that this manual uses. These conventions make it easier to gather information from this and other Informix manuals.

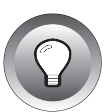
The following conventions are covered:

- Typographical conventions
- Icon conventions

Typographical Conventions

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

Convention	Meaning
KEYWORD	All keywords appear in uppercase letters in a serif font.
italics	Within text, new terms and emphasized words appear in italics. Within syntax diagrams, values that you are to specify appear in italics.
boldface	Identifiers (names of classes, objects, constants, events, functions, program variables, forms, labels, and reports), environment variables, database names, filenames, table names, column names, icons, menu items, command names, and other similar terms appear in boldface.
`monospace`	Information that the product displays and information that you enter appear in a monospace typeface.
KEYSTROKE	Keys that you are to press appear in uppercase letters in a sans serif font.
♦	This symbol indicates the end of feature-, product-, platform-, or compliance-specific information.



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

Icon Conventions

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

Comment Icons

Comment icons identify warnings, important notes, or tips. This information is always displayed in italics.

Icon	Description
	The *warning* icon identifies vital instructions, cautions, or critical information.
	The *important* icon identifies significant information about the feature or operation that is being described.
	The *tip* icon identifies additional details or shortcuts for the functionality that is being described.

Feature and Product Icons

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

Icon	Description
SPL	Identifies information that is valid only if you are using Informix Stored Procedure Language (SPL).
E/C	Identifies information that is specific to the INFORMIX-ESQL/C.

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 feature- or product-specific information.

Compliance Icons

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

Icon	Description
ANSI	Identifies information that is specific to an ANSI-compliant database.

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.

Additional Documentation

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

- On-line manuals
- Printed manuals
- Error message files
- Documentation notes, release notes, and machine notes

On-Line Manuals

A 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 either the installation guide for your product or the installation insert that accompanies the documentation CD.

The documentation set that is provided on the CD describes Universal Server, its implementation of SQL, and its associated application-programming interfaces. For an overview of the manuals in the Universal Server documentation set, see *Getting Started with INFORMIX-Universal Server*.

Printed Manuals

The Universal Server documentation set describes Universal Server, its implementation of SQL, and its associated application-programming interfaces. For an overview of the manuals in the Universal Server documentation set, see *Getting Started with INFORMIX-Universal Server*.

To order printed manuals, call 1-800-331-1763 or send email to moreinfo@informix.com.

Please provide the following information:

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

Error Message Files

Informix software products provide ASCII files that contain all the Informix error messages and their corrective actions. To read the error messages in the ASCII file, Informix provides scripts that let you display error messages on the screen (**finderr**) or print formatted error messages (**rofferr**). For a detailed description of these scripts, see the Introduction to the *Informix Error Messages* manual.

Documentation Notes, Release Notes, Machine Notes

In addition to printed documentation, the following on-line files, located in the **$INFORMIXDIR/release/en_us/0333** directory, supplement the information in this manual.

On-Line File	Purpose
EXTUDRDOC_9.1	The documentation-notes file describes features that are not covered in this manual or that have been modified since publication.
SERVERS_9.1	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.
IUNIVERSAL_9.1	The machine-notes file describes any special actions that are required to configure and use Informix products on your computer. Machine notes are named for the product described.

Please examine these files because they contain vital information about application and performance issues.

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, on INFORMIX-Universal Server. In addition, many features of Universal Server comply with the SQL-92 Intermediate and Full Level and X/Open SQL CAE (common applications environment) standards.

Informix Welcomes Your Comments

Please tell us what you like or dislike about our manuals. To help us with future versions of our manuals, we want to know about 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

Write to us at the following address:

Informix Software, Inc.
SCT Technical Publications Department
4100 Bohannon Drive
Menlo Park, CA 94025

If you prefer to send email, our address is:

doc@informix.com

Or send a facsimile to the Informix Technical Publications Department at:

415-926-6571

We appreciate your feedback.

Chapter

1

Overview of User-Defined Routines

What Is a User-Defined Routine? 1-3

Tasks That You Can Perform with UDRs 1-5
Encapsulate Multiple SQL Statements 1-6
Simplify Writing Programs 1-6
Improve Performance of Multiple SQL Statements 1-6
Extend Functions for Built-In Data Types 1-7
SQL-Invoked Functions 1-7
Casting functions 1-8
Operator Class Functions 1-8
Support New Data Types 1-10
SQL-Invoked Functions 1-10
Support Functions 1-10
Operator Class Functions 1-11
Casting Functions 1-12
Operator Binding 1-13
Use as a Triggered Action 1-13
Restrict Access to a Table 1-14
Enforce Business Rules 1-14

Ways to Invoke User-Defined Routines 1-15

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

- What is a user-defined routine?
- Tasks that you can perform with UDRs
- Ways to invoke UDRs

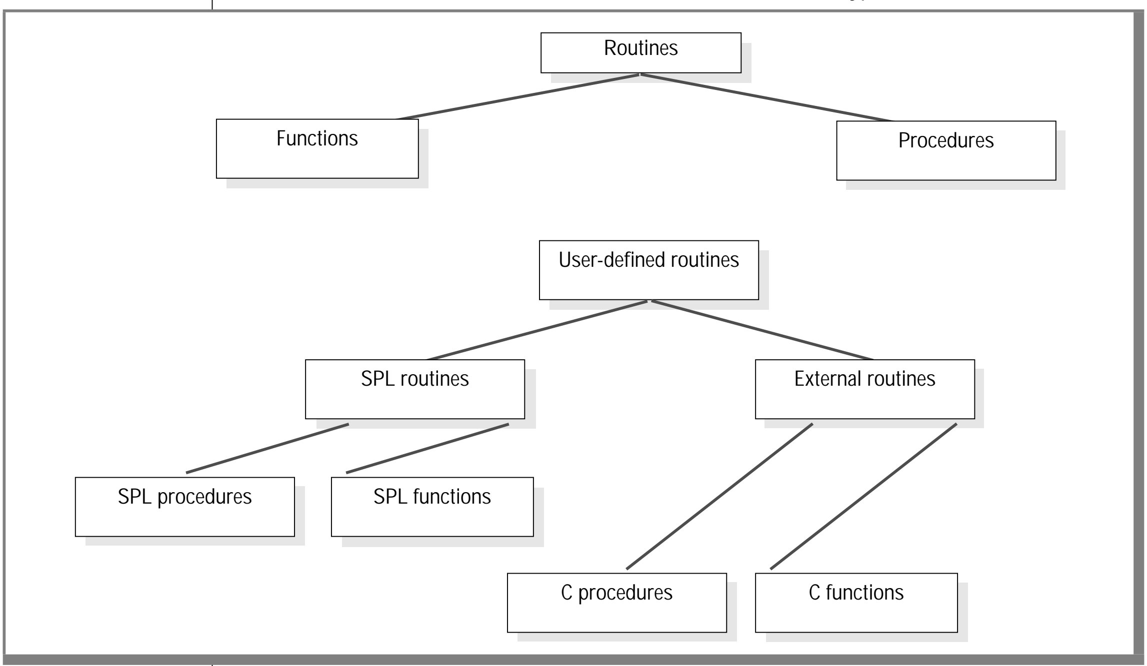
What Is a User-Defined Routine?

A user-defined routine (UDR) is a routine that you create and register in the system catalog tables and that you invoke within an SQL statement or another routine.

A UDR can be either a *function* or a *procedure*. A function is a routine that optionally accepts a set of arguments and returns a set of values. A function can be used in SQL expressions. A procedure is a routine that optionally accepts a set of arguments and does not return any values. A procedure cannot be used in SQL expressions because it does not return a value.

Figure 1-1 shows the different types of user-defined routines.

Figure 1-1
Types of User-Defined Routines



The database server provides the following kinds of routines:

- Informix Stored Procedure Language (SPL) routines

  The body of an SPL routine contains SQL statements and flow control statements for looping and branching. SPL provides the flow control extensions to SQL.

  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 to write an SPL routine, refer to the *Informix Guide to SQL: Tutorial*.

- External routines

  The body of an external routine contains statements from a supported language (for example, C language) other than SPL. External routines typically perform operations on user-defined data types. For more information on how to write an external routine, refer to the *DataBlade API Programmer's Manual*.

SPL routines can execute routines written in C or other external languages, and external routines can execute SPL routines.

Tasks That You Can Perform with UDRs

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

- Encapsulate multiple SQL statements
- Extend functions on built-in data types
- Support new data types
- Create triggered actions for multiple applications
- Restrict who can read data, change data, or create objects

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

- Manipulate large objects
- Facilitate interactive multimedia publication

- Collect data from Internet and Intranet end users
- Search graphical data

Encapsulate Multiple SQL Statements

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

Simplify Writing Programs

An SPL routine can batch frequently performed tasks that require several SQL statements. SPL offers program control statements that extend what SQL can accomplish alone. You can test database variables in an SPL routine and perform the appropriate actions for the values 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.

SPL routines are especially helpful in a client/server environment. If a change is made to application code, it must be distributed to every client computer. An SPL routine resides on the server computer, so the change is made in only one location.

SPL

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. ♦

Improve Performance of Multiple SQL Statements

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 5, "Performance Considerations."

Tip: *Not all the encapsulated SPL that you created as SPL procedures in earlier Informix products has the properties currently associated with 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.*

Extend Functions for Built-In Data Types

Universal Server provides the following types of routines for built-in data types:

- SQL functions
- Casting functions
- Operator class functions

SQL-Invoked Functions

An SQL-invoked function is a routine that end users can specify within an SQL statement to operate on a data type. SQL-invoked functions on built-in data types can be one of the following:

- Operator functions

  An *operator function* is a user-defined function that has a corresponding operator symbol (such as '=' and '+'). These operator symbols are used within expressions in an SQL statement. For more information on operator functions, refer to Chapter 2 of *Extending INFORMIX-Universal Server: Data Types*.

- Built-in functions such as **cos()**, **sin()**, and **tan()**

  Universal Server provides *built-in functions* that provide some basic mathematical operations. Universal 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 data type.

  For more information on the built-in functions, see the Expression segment in Chapter 1 of the *Informix Guide to SQL: Syntax*.

Casting functions

Universal Server provides system-defined casts that perform automatic conversions between certain built-in data types. For more information on these system-defined casts, refer the *Informix Guide to SQL: Reference*.

You cannot create user-defined casts to allow conversions between two built-in data types for which a system-defined cast does not currently exist. For more information on when you would want to write new casting functions, refer to "Casting Functions" on page 1-12.

Operator Class Functions

An *operator class* is the set of operators that Universal 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 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.

Two types of functions make up the operator class for the secondary access method:

- Strategy functions

  Strategy functions are functions that end users can invoke within an SQL statement to operate on a data type. End users can invoke a strategy function by its operator symbol (such as > or =) or by its name (such as **contains** or **within**).

- Support functions

  Support functions are functions that the secondary access method uses internally to build and search the index.

You can write new operator class functions if you want to do the following:

- Use a different ordering scheme than the order provided by the default operator classes

  Universal Server provides the strategy and support functions for the default operator class (sometimes referred to as an *opclass*) for the B-tree index access methods. For more information on the syntax to register operator classes, refer to the CREATE OPCLASS statement in Chapter 1 of the *Informix Guide to SQL: Syntax*.

  If a DataBlade module provides an index access method, it might also provide a default operator class 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.

- Use a set of operators that is different from any existing opclasses with an index access method

  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 strategy functions in the operator class associated with the index.

  For more information on optimizing queries with user-defined routines, refer to "SPL Optimization" on page 5-5.

For more information on operator classes, refer to Chapter 3 of *Extending INFORMIX-Universal Server: Data Types*.

Support New Data Types

When you create a new data type, you must provide the following:

- SQL-invoked functions that users specify explicitly in SQL statements
- Support routines that the database server invokes implicitly to operate on these types
- Cast routines that the server can invoke implicitly or that users can specify explicitly in SQL statements to convert data from one type to another
- Operator class functions if you want to use a different ordering scheme than the order that the default operator class provides.

SQL-Invoked Functions

An SQL-invoked function is a routine that end users can specify within an SQL statement to operate on a data type. SQL-invoked functions can be one of the following:

- Operator functions that have a symbol (such as = and +)
- Built-in functions (such as **cos()**, **sin()**, and **tan()**)
- Function names that you use in an SQL statement to operate on one or more data columns (such as **contains**)

Support Functions

If you define a new opaque data type, you also provide support routines that enable the server to operate on the type. Universal Server requires some routines, and others are optional. The following list shows the standard routines 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
- LessThan routine

- Hash routine
- Assign and Destroy routines

For more information on the support routines for opaque data types, refer to *Extending INFORMIX-Universal Server: Data Types*.

Operator Class Functions

"Operator Class Functions" on page 1-8 explains what an operator class is and describes the functions associated with an operator class.

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

- Order the new data type with the ordering scheme that the default operator class provides.

  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 "What Is Routine Overloading?" on page 2-3.
- Tell the query optimizer which user-defined functions that appear in an SQL statement can be processed with a given secondary access method

  These functions are called the *strategy functions* for the operator class.

For more information on the routines for operator classes, refer to *Extending INFORMIX-Universal Server: Data Types*.

Casting Functions

You can create user-defined casts to perform conversions between most data types, including opaque types, distinct types, row types, and built-in types. For example, you can define casts for any of the following user-defined data types:

- **Opaque data types**. Developers of opaque data types must define cast functions to handle conversions between the internal and external representations of the opaque type. For information about how to create and register casts for opaque data types, see the *Extending INFORMIX-Universal Server: Data Types* manual.
- **Distinct data types**. A routine cannot directly compare a distinct type to its source type. However, Universal 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.

  You can create user-defined casts on distinct types. For information and examples that show how to create and use casts on distinct types, see the *Informix Guide to SQL: Tutorial*.
- **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 casting 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 "Routine Resolution with Casts" on page 2-14.
- 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 the *Extending INFORMIX-Universal Server: Data Types* manual.

Operator Binding

Operator binding is the implicit invocation of an *operator function* when an *operator symbol* is used in an SQL statement. Universal Server implicitly maps a built-in operator function name to a built-in operator.

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

To order `Mc` and `Mac` in the same way, you must create external functions that contain code that treats `Mc` and `Mac` the same. You can use the same function names as in the default operator class, **btree_ops**, so that the SQL user does not need to specify one function name to handle regular names and a different function name for Scottish names.

Function overloading is the ability to use the same name for multiple functions to handle different data types. Universal Server uses the external function because the CREATE TABLE statement specifies the Scottish names data type for the column. For more information on function overloading, refer to "What Is Routine Overloading?" on page 2-3.

Use as a Triggered Action

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*.

Restrict Access to a Table

SPL routines offer the ability to restrict access to a table. For example, if an administrator grants insert permissions to a user, that user can insert a row using INFORMIX-Connect, DB-Access, or an application program. This situation could be a problem if an administrator wants to enforce any business rules (see the next section).

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

Enforce 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 first be archived before it is deleted. You can write an SPL routine that accomplishes both tasks and prohibits users from directly accessing the table.

Ways to Invoke User-Defined Routines

You can execute an external procedure and an SPL procedure using the EXECUTE PROCEDURE statement from the following:

- SPL
- DB-Access
- ESQL/C

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

You can execute an external function and an SPL function from the following:

- SPL
- DB-Access
- ESQL/C by using the EXECUTE FUNCTION statement
- An SQL expression (in the SELECT clause or WHERE clause).

The database server can invoke a UDR implicitly for two reasons:

- Built-in operator binding
- Implicit casting

For more information on how to invoke user-defined routines, refer to "Invoking a Routine" on page 4-3.

Chapter

2

Routine Overloading and Routine Resolution

What Is Routine Overloading? 2-3
The Routine Signature 2-4
Specifying a Routine During Creation 2-5
Using the Optional Specific Name 2-6
Specifying Overloaded Routines During Invocation 2-7

The Routine-Resolution Process 2-7
Candidate List of Routines 2-8
Precedence List of Data Types 2-9
Routine Resolution Within a Type Hierarchy 2-10
Routine Resolution with Distinct Data Types 2-11
Precedence List for Built-In Data Types 2-13
Routine Resolution with Casts 2-14
Null Arguments in Overloaded Routines 2-15

Built-In SQL Functions That You Can Overload 2-17

Operator-to-Function Binding 2-19

This chapter discusses the following topics:

- Routine overloading
- Routine resolution
- Specifying arguments in overloaded routines
- Built-in SQL functions that you can overload
- Operator to function binding

What Is Routine Overloading?

Routine overloading refers to the ability to assign one name to multiple routines and specify different types of arguments on which the routines can operate. The advantage of routine overloading is that you do not need to invent a different name for a function that performs the same task for different arguments.

With Universal Server, 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 functions from their source types.

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

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

In this way, you can overload a routine so that you have a customized **area()** routine for every data type that you want to evaluate. When a routine has been overloaded, the database server can execute different routines that have the same name. However, the database server cannot uniquely identify an overloaded routine by its name alone.

The Routine Signature

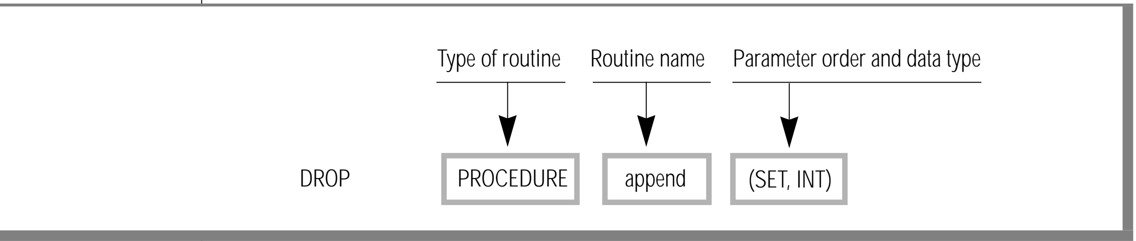
Due to routine overloading, the database server might not be able to uniquely identify a routine by its name alone. The database server uniquely identifies an overloaded routine by its *signature*. The routine signature includes the following information:

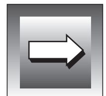
- The type of routine (procedure or function)
- The routine name
- The number of parameters
- The data types of the parameters
- The order of the parameters
- The owner name (ANSI database only)

You use the routine signature in SQL statements when you 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 statement shown in Figure 2-1 uses a routine signature.

Figure 2-1
Example of Routine Signature





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

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 your owner name, the signature includes your owner name as part of the routine name.

ANSI

In an ANSI-compliant database, the routine signature must be unique within the user's name space. The routine name always begins with the owner, in the format <*owner*>.<*routine name*>. ♦

Specifying a Routine During Creation

You establish the routine signature when you create a function or procedure with the CREATE FUNCTION or CREATE PROCEDURE statements. When you create a function or procedure, you also specify the argument names.

For example, the following CREATE FUNCTION statement registers an **equal()** function that was written to take two arguments, **arg1** and **arg2**, of data type `udtype1` and return 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;
```

Using the Optional Specific Name

You can assign a *specific name* to a particular signature of a routine. You can use the unique *specific name* as a shorthand identifier to refer to a particular overloaded variation of a routine. You can use the specific name instead of the full signature in the following SQL statements:

- DROP
- GRANT
- REVOKE
- UPDATE STATISTICS

A specific name can be up to 18 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.

Figure 2-2 shows how to define the specific name **eq_udtype1** in a CREATE FUNCTION statement that creates the **equal()** function.

Figure 2-2

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

Now you can refer to the routine defined in Figure 2-2 with either the routine signature or the specific name. The following sample DDL statement specifies the routine by its signature:

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

The following example specifies the routine defined in Figure 2-2 with the SPECIFIC keyword and the specific name:

```
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 **equal** function for a new data type, **udtype1**:

```
SELECT ... FROM employee
    WHERE equal(col1_udtype1,col2_udtype1) ...
```

You can also invoke the **equal** function as in the following examples:

```
EXECUTE FUNCTION equal(arg1_udtype1,arg2_udtype1) INTO result
CALL equal(arg1_udtype1,arg2_udtype1) RETURNING boolean...
```

For more information on invoking a routine through an SQL expression, the EXECUTE statement, and the CALL statement, refer to "Invoking a Routine" on page 4-3.

Alternatively, you can invoke the **equal** function with an argument on either side of the function symbol. For more information, refer to "Operator-to-Function Binding" on page 2-19.

The Routine-Resolution Process

Routine resolution refers to the process in which the database server determines which routine to execute when you overload a routine. The database server invokes routine resolution implicitly when a routine is invoked by a user or another routine. You need to understand the routine-resolution process to:

- obtain the data results that you expect from a routine.
- avoid corrupting data if the wrong routine executes.
- understand when you need to write an overloaded routine.

When a user or another routine invokes a routine, the database server searches for a signature that matches the routine name and arguments. If the database contains a routine with a matching signature, the database server executes this routine. If no exact match exists, the database server searches for a routine to substitute.

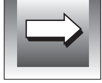
When several arguments are passed to a routine, the database server first tries to match the leftmost argument. The database server checks for a candidate routine that has the same data type as the leftmost argument. If no exact match exists for the first argument, the database server searches the candidate list of routines using a precedence order of data types.

The database server continues matching the arguments from left to right.

Candidate List of Routines

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

- Invoked routine name
- Routine type (function or procedure)
- Number of arguments



Important: *The candidate list contains only routines for which the current session has EXECUTE permission.*

ANSI

In an ANSI-compliant database, the candidate list contains only routines that belong to the current user and the user **informix**. ♦

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 uses the following precedence order to match the data types of arguments in the list of candidate routines:

1. Data type of the argument passed to a routine
2. If an argument passed to a routine is a subtype in a type hierarchy, Universal Server checks up the type hierarchy tree for a routine to execute. For more information, refer to "Routine Resolution Within a Type Hierarchy" on page 2-10.
3. If an argument passed to a routine is a distinct type, Universal Server checks the source type for a routine to execute. If the source type is itself a distinct type, Universal Server checks the source type of that distinct type. For more information, refer to "Routine Resolution with Distinct Data Types" on page 2-11.
4. If an argument passed to a routine is a built-in type, Universal 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 2-13.

   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.

   For more information, refer to "Routine Resolution with Casts" on page 2-14.
6. If an argument passed to a routine is a collection type, Universal Server adds the generic type of the collection to the precedence list for the passed argument.

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.
```

Routine Resolution Within 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 can add additional data attributes and behavior to those inherited from the supertype.

Figure 2-3 shows the CREATE statements to define a sample hierarchy.

Figure 2-3
Sample Data Type Hierarchy

```
CREATE ROW TYPE emp
    (name varchar(30),
     age int,
     salary numeric(10,2));
CREATE ROW TYPE trainee UNDER emp ...
CREATE ROW TYPE student_emp (gpa float) UNDER trainee;
```

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.

For example, suppose you create an overload function, **bonus()**, for the sample type hierarchy in Figure 2-3. You create the **bonus()** function on the root supertype and a subtype and invoke the **bonus()** function with the following statements:

```
CREATE FUNCTION bonus (emp,int) RETURNS numeric(10,2) ...
CREATE FUNCTION bonus(trainee,float) RETURNS numeric(10,2)...
EXECUTE FUNCTION bonus(student_emp, int);
```

The routine-resolution process goes through the following steps:

1. Processes the leftmost argument first.
   a. 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 2-9.
   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 second function, **bonus(trainee,float),** matches the first argument in the routine invocation.
2. The database server processes the second argument next.
   a. Looks for a candidate routine with a second parameter of data type **integer**.

      The matching candidate routine from step 1b above 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 2-9.
   b. Because the second parameter is a built-in data type, the database server goes to the precedence list in Figure 2-5 on page 2-13.

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

Routine Resolution with Distinct Data Types

A *distinct data type* has the same internal storage representation as an existing data type, but it has a different name and cannot be compared to the source type without casting. Distinct types inherit functions from their source types.

When a distinct data type in the argument list does not match the data type of the parameter in the same position of the routine signature, the database server searches for a routine that accepts the source type in the position of that argument. If the source type is itself a distinct type, the database server checks the source type of that distinct type.

The candidate list can contain a routine with an argument that is the source type of the invoked routine argument. However, if the source type is not in the precedence list for that data type, then the routine-resolution process eliminates that candidate.

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

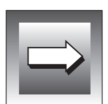
```
CREATE DISTINCT TYPE pounds AS int;
CREATE DISTINCT TYPE stones AS int;
CREATE TABLE test(p pounds, s stones);
```

Figure 2-4 shows a sample query that an SQL user might execute.

Figure 2-4
Sample Distinct Type Invocation

```
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.



Important: *The routine-resolution process cannot match two different distinct types.*

The database server chooses the built-in **equals** function when you explicitly cast the arguments, as the following query shows:

```
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 in Figure 2-4:

- An overloaded function **equals(pounds,stones)** to handle the two distinct data types. The advantage of creating an overloaded **equals** function is that the SQL user does not need to know that these are new data types that require explicitly casting.
- Implicit cast functions from the data type **pounds** to **stones** and **stones** to **int.**

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 2-5 lists the precedence for the built-in data types when an argument in the routine invocation does not match the parameter in the candidate list.

Figure 2-5
Precedence of Built-In Data Types

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

Figure 2-6 shows sample overloaded **test** functions and a query that invokes the **test** function. This query invokes the function with a DECIMAL argument, **test(2.0)**. Because a **test** function for a DECIMAL argument does not exist, the routine-resolution process checks for the existence of a **test** function for each data type shown in the precedence list in Figure 2-5 on page 2-13.

Figure 2-6
Example of Data Type Precedence During Routine Resolution

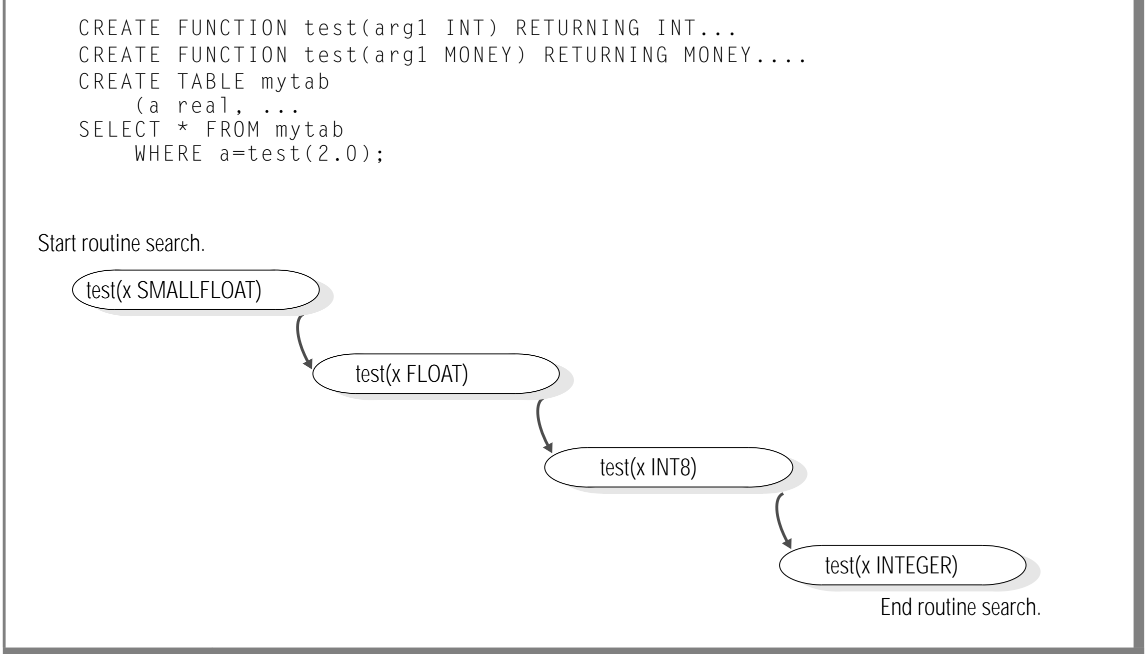


Figure 2-6 shows the order in which the database server performs a search for the overloaded function, **test()**. The database server searches for a qualifying **test()** function that takes a single argument of type INT.

Routine Resolution with Casts

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

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

```
CREATE IMPLICIT CAST (foo AS bar)
CREATE IMPLICIT CAST (bar AS foo)
CREATE FUNCTION g(foo, foo) ...
CREATE FUNCTION g(bar, bar) ...
```

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

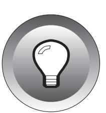
```
EXECUTE FUNCTION g(foo, bar);
```

The database server considers both functions as candidates. The routine-resolution process selects the function **g(foo,foo)** because of the left-to-right rule. The database server executes the second cast, **cast(bar AS foo)**, to convert the second argument before the function **g(foo,foo)** executes.

Because the database server performs implicit casts during routine resolution, a different routine than the one that the user expects might execute. For example, suppose the database has the following two routines named **func():**

```
func(arg1 INT)
func(arg1 DECIMAL)
```

If a user wants to invoke **func(10.0)** but accidentally invokes **func(10)**, the **func()** routine that accepts an INT argument executes instead of the **func()** that accepts a DECIMAL argument. The value returned from the function for INT might corrupt the user's data.



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

Null Arguments in Overloaded Routines

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

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

Suppose you create the following functions in SPL or C language. (If the functions are external functions, you must use the HANDLESNULLS modifier to specify that each function can handle null arguments.)

```
CREATE FUNCTION foo(arg1 INT, arg2 INT) RETURNS BOOLEAN...
CREATE FUNCTION foo(arg1 MONEY, arg2 INT) RETURNS BOOLEAN...
CREATE FUNCTION foo(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 **foo()** function that matches the function argument in the expression:

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

The null value acts as a wildcard for the second argument and matches the second parameter type for each function **foo()** defined. The only **foo()** 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 **foo()** function to invoke. The null value in the first argument matches the first parameter of each function; all three **foo()** functions expect a second argument of type INT.

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

To avoid ambiguity, use null values as arguments carefully.

Built-In SQL Functions That You Can Overload

Universal Server provides special built-in SQL functions that provide some basic mathematical operations. You can overload most built-in SQL functions. For example, you might want to create a **sin()** function on a new data type that represents complex numbers.

Figure 2-7 provides the full list of built-in SQL functions that you can overload. A user might create a function with the same name as one from this table, but its routine signature must be unique in the database (non-ANSI) or the schema (ANSI). For more information on a routine signature, refer to "The Routine Signature" on page 2-4.

Figure 2-7
Built-In SQL Functions That You Can Overload

Built-In SQL Function Names	Number of Parameters	Parameter Types
abs()	1	real number
mod()	2	real number expression, real number expression
pow()	2	real number expression, real number expression
root()	1 or 2	real number expression [, real number expression]
round()	1 or 2	expression, [literal integer]
sqrt()	1	real number expression
trunc()	1 or 2	expression, literal integer
exp(), log(), logn()	1	positive real expression
cos(), sin(),tan()	1	numeric expression

(1 of 2)

Built-In SQL Function Names	Number of Parameters	Parameter Types
asin(), acos(), atan()	1	numeric expression
atan2	2	numeric expression, numeric expression
length(), octet_length(), char_length(), character_length()	1	character string

(2 of 2)

These built-in SQL functions appear as UDRs and have entries in the **sysprocedures** system catalog table. The language for these entries are C or BUILTIN, depending on the internal implementation. Informix strongly recommends that you do not update or delete these entries.

Universal Server does not handle the following built-in SQL functions as UDRs. Therefore, you cannot overload these built-in SQL functions:

- user
- sitename, dbservername
- today
- current
- extend
- trim
- hex
- date
- mdy
- day, month, weekday, year
- dbinfo

In general, Universal Server does not treat these functions as UDRs because the syntax that you use to call them does not conform to the syntax that you use to invoke UDRs.

Operator-to-Function Binding

Operator binding is the implicit invocation of an *operator function* (such as **equal**) when an *operator symbol* (such as =) is used in an SQL statement with user-defined data types.

This implicit mapping of operator function names to built-in operators facilitates writing SQL expressions with user-defined data types. When you overload an operator function, the SQL user does not need to specify anything different in the SQL statement. The SQL user can invoke the routine with an argument on either side of the operator symbol, as is normally done with built-in data types.

For example, the following sample SQL statements show how you can invoke the **equal** function for a new data type, ScottishName, that treats the strings "`Mc`" and "`Mac`" the same:

```
CREATE TABLE employee
    (emp_name ScottishName,
     emp_phone char(10)
    ...
SELECT emp_name, emp_phone FROM employee
    WHERE emp_name = 'McDonald'::ScottishName;
```

The SQL user does not need to invoke the user-defined **equal** function as in the following example:

```
SELECT * FROM employee
    WHERE equal(emp_name,'McDonald'::ScottishName);
```

Figure 2-8 lists the operator symbol and the built-in function name to which the database server maps implicitly.

Figure 2-8
Operator-Symbol To Operator-Function Binding

Operator Symbol	Operator Function	Number of Arguments	Return Type
=	**equal**	2	Boolean
<>,!=	**notequal**	2	Boolean
>	**greaterthan**	2	Boolean

(1 of 2)

Operator Symbol	Operator Function	Number of Arguments	Return Type
<	**lessthan**	2	Boolean
>=	**greaterthanorequal**	2	Boolean
<=	**lessthanorequal**	2	Boolean
like	**like**	2 or 3	Boolean
matches	**matches**	2 or 3	Boolean
-	**minus**	2	Any type
+	**plus**	2	Any type
/	**divide**	2	Any type
*	**times**	2	Any type
||	**concat**	2	Any type
+	**positive**	1	Any type
-	**negate**	1	Any type

(2 of 2)

Chapter

3

Designing a User-Defined Routine

Designing the Routine Source 3-3
Naming Your Routine 3-4
Number of Arguments 3-5
Coding Standards 3-5

Shared Object Libraries 3-6
Creating a Shared Library 3-6
Loading a Shared Library into Memory 3-7
Replacing a Shared Object File 3-8

Registering External Routines 3-10
Registering an External Function. 3-10
Registering an External Procedure 3-12
Registering an External Routine with Modifiers 3-13
Modifiers for External Functions 3-13
Modifiers for External Procedures 3-14

Returning Multiple Values from External Functions 3-14
OUT Parameters and Statement Local Variables 3-14
Referencing OUT Parameters in User-Defined Routines . . . 3-16
Iterator Function 3-16
Writing an Iterator Function 3-17
Registering an Iterator Function. 3-17
Invoking an Iterator Function 3-17

Privileges for Registering a Routine 3-18
Privileges for Executing a Routine 3-18
Granting and Revoking the Execute Privilege 3-19
Privileges on Objects Associated with a Routine 3-20
Executing a Routine as DBA 3-21
Effect of DBA Privileges on Objects and Nested Routines . . . 3-22

This chapter describes how to create external routines. It covers the following topics:

- Designing the routine source
- Writing the routine source
- Creating object libraries
- Registering external routines
- Granting permissions on routines
- Returning multiple values from external functions

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 documented separately.

For information about how to create and use SPL routines, see Chapter 15, "Creating and Using SPL Routines," in the *Informix Guide to SQL: Tutorial*.

Designing the Routine Source

When you design a user-defined routine, you must consider the following:

- Naming your routine
- Number of arguments
- Coding standards

Naming Your Routine

Choose sensible names for your routines. The routine name should be easy to remember and succinctly describe what the routine does. Because Universal Server supports polymorphism, you can have multiple routines with the same name that take different arguments. This is contrary to programming practice in some high-level languages. For example, a C programmer might be tempted to create functions with the following names that return the larger of their arguments:

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

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

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

Using the naming in the second example allows users to ignore the types of the arguments when they call the routine. They simply remember what it does and let Universal Server choose which supporting routine to call based on the argument types. This makes the user-defined routine simpler to use.

Routine overloading refers to the ability to assign one name to multiple routines and specify different types of arguments on which the routines can operate. For more information on routine determination, refer to "The Routine-Resolution Process" on page 2-7.

You should consider the following questions about routine naming and design:

- Are any of my routines modal?
- 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?

Number of Arguments

User-defined routines should take a small number of arguments and should not include arguments that make them modal. For example, the following statements show alternative routine calls to compute containment of spatial values:

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

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

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

The second example is better in that the routine names clearly explain what computation is performed. Always construct your routines to be nonmodal, as in the second example.

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.

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.

You should ask the following questions:

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

Shared Object Libraries

This section contains information about creating, loading, and replacing shared object libraries with the database server.

Creating a Shared Library

When you create external routines, you put them into a shared library that the database server can access.

Shared object files must be owned by the user ID that runs the database server. In a production installation, Universal Server runs as user **informix** and shared object files are owned by user **informix**.

To create a shared library

1. Compile the source file into an object file with the C compiler.

   Include any necessary header files that the file needs, such as the library where the DataBlade API functions reside.

   The following sample compile command can be used on Solaris systems:

   ```
   /compilers/bin/cc -I $INFORMIXDIR/incl -I $INFORMIXDIR/incl/esql -c abs.c
   ```

2. Create a shared object library (**.so** file extension on Solaris systems) and load the object file(s) for the external routine(s).

   Put related routines into the same shared library. The following sample command loads the object file into the shared library:

   ```
   /compilers/bin/cc -K abs.o -o abs.so
   ```

3. Put the shared library in a directory that is readable by user **informix** and set the permissions to 755 or 775 so that only the owner can write to the shared object files.

   ```
   chmod 755 /usr/code/abs.so
   ```

   If a shared object file has write permission set to `all`, and someone tries to execute a routine in the shared object file, the database server issues error -9793 and writes a message in the log file.

4. Specify the path of the shared library in the CREATE FUNCTION (or CREATE PROCEDURE) statement when you register the external routine.

```
CREATE FUNCTION abs_eq(integer, integer)
   RETURNS boolean
   EXTERNAL NAME
      '/usr/code/abs.so(abs_equal)'
   LANGUAGE C NOT VARIANT;
```

Loading a Shared Library into Memory

When an application or the database server invokes one of the UDRs in a shared library, the database server performs the following tasks:

1. If the routine is overloaded, determines which UDR to execute, based upon the arguments specified in the routine invocation

   For more information on routine resolution, refer to "The Routine-Resolution Process" on page 2-7.
2. Locates the shared library that contains the UDR from the **path** column in the **sysprocedures** system catalog table
3. Loads this shared library into memory, if the library is not already loaded

Use the **onstat** command-line utility with the **-g dll** option to view the dynamically loaded libraries in which your user-defined routines reside. Once the database server has loaded a shared library into memory, this library remains in memory until one of the following situations occurs:

- All functions in the shared library are dropped with the DROP FUNCTION, DROP ROUTINE, or DROP PROCEDURE statement.

  Once all user-defined routines are dropped, the database server automatically unloads the shared library from memory.
- The database server is shut down.

  All memory that the database server uses is released when the database server shuts down.

Replacing a Shared Object File

You can explicitly replace a shared library without bringing down the database server. You use one of the following user-defined functions that Informix provides to upgrade a shared library:

- To replace a loaded shared library with a new version that has the same name and location

  ```
  ifx_reload_module
      ("module path", "language name")
  ```

  In this syntax, module path is a character string that lists the full path of the shared library, and language name is either 'c' or 'spl'.

  For example, to reload a new version of the **circle.so** shared library that resides in the **/usr/app/opaque_types** directory, you can use the EXECUTE FUNCTION statement to execute the **ifx_reload_module()** function as follows:

  ```
  EXECUTE FUNCTION
  ifx_reload_module
      ("/usr/apps/opaque_types/circle.so", "c")
  ```

- To replace a loaded shared library with a new version that has a new name or location

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

  In this syntax, <old module path> and <new module path> are each character strings that list the full path of a shared library, and language name is either 'c' or 'spl'.

  For example, to replace the **circle.so** shared library that resides in the **/usr/app/opaque_types** directory with one that resides in the **/usr/app/shared_libs** directory, you can use the EXECUTE FUNCTION statement to execute the **ifx_replace_module()** as follows:

  ```
  EXECUTE FUNCTION
  ifx_replace_module("/usr/apps/opaque_types/circle.so",
      "/usr/apps/shared_libs/circle.so", "c")
  ```

  This **ifx_replace_module** function updates the **sysprocedures** system catalog with the new name or location.

These functions return one of the following integer values:

- Zero indicates success.
- A negative value indicates an error message number.

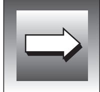
You can also execute these functions in a SELECT statement. For example, the following SELECT statement executes the **ifx_reload_module()** statement:

```
SELECT
ifx_reload_module("/usr/apps/opaque_types/circle.so", "c")
    FROM customer
    WHERE customer_id = 100;
```

If you do not want the shared library replaced multiple times with this SELECT statement, ensure that the SELECT statement returns only one "row" of values.

E/C

When you execute these functions from within an INFORMIX-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*. ♦



Important: *Do not overwrite a shared object file on disk while it is loaded in memory because you might cause the database server to stop functioning when the overwritten module is accessed or unloaded. Use the* ***ifx_replace_module*** *function to replace the shared object file.*

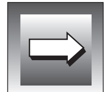
To unload a module without restarting the database server, you must drop all routines in the module using the SQL DROP statement. After you drop all routines in the module and all instances of the routines finish executing, the database server removes the module from the memory map and records a message in the log file to indicate that the module is unloaded. After the module is unloaded, you can replace the shared object file and re-create its user-defined routines in the database.

Registering External Routines

Routines can be functions or procedures. You use different SQL statements to register each type of routine.

Registering an External Function

When you want a routine to return a value, you must create that routine as a function. To create an external function, write the body of the function in a language other than SPL, then use the CREATE FUNCTION statement to register the 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.*

The following example registers an external function of the name **equal** in the database server and specifies the library where the function object module is stored. The **equal()** function takes two arguments of the data type **udtype1** and returns a BOOLEAN value.

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

When you register an external function, the END FUNCTION keywords are optional. Registration for an external routine requires two special clauses that help the database server locate the routine:

- The EXTERNAL NAME clause specifies the path to the C library where the function is stored. By default, the entry point in that library is the routine name that follows the keywords CREATE FUNCTION. You can specify a different entry point with the EXTERNAL NAME clause. To specify an entry point in the EXTERNAL NAME clause, put the appropriate C module name in parentheses after the library file.

  For more information about the EXTERNAL NAME clause, see the External Reference segment in Chapter 1 of the *Informix Guide to SQL: Syntax*.

  In the preceding example, the entry point **udtype1_equal()** replaces the default entry point, **equal()**. The C routine **udtype1_equal()** is invoked whenever an **equal()** function with two arguments of **udtype1** data type is called.

  If the creator of the routine does not indicate an entry point in the EXTERNAL NAME clause as shown in the following example, the database server searches the C library for a routine named **equal()** (whose parameters match the arguments of the calling function):

  ```
  EXTERNAL NAME "/usr/lib/udtype1/lib/libbtype1.so()"
  ```

- The required LANGUAGE clause specifies the language in which the body of the function is written.

For more information, see the CREATE FUNCTION statement in Chapter 1 of the *Informix Guide to SQL: Syntax*.

Registering an External Procedure

When you do not want your routine to return a value, you must create that the routine as a procedure. To create an external procedure, write the body of the procedure in a language other than SPL, then use the CREATE PROCEDURE statement to register the procedure. The following example shows how to register an external procedure:

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

The EXTERNAL NAME and LANGUAGE clauses work the same way for external procedures as is explained for external functions on page 3-10.

In the preceding example, the actual body of the procedure is named **compare_n_insert** and is located in the C-language 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 Chapter 1 of the *Informix Guide to SQL: Syntax*.

Registering an External Routine with Modifiers

When you create an external routine, you can specify optional modifiers that help optimize how the database server executes the routine. The following example shows how to use the WITH clause to specify a set of modifiers when you create an external function:

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

Following the WITH keyword, the modifiers that you want to specify are enclosed within parentheses and separated by commas. The **handlesnulls** modifier indicates that the **basetype2_lessthan()** function (in the library **/usr/lib/basetype1/lib/libbtype1.so)** is written so that it can recognize when an SQL NULL argument is passed.

External functions support a different set of modifiers than external procedures.

Modifiers for External Functions

The database server supports the following modifiers for external functions:

- CLASS
- HANDLESNULLS
- INTERNAL
- ITERATOR
- STACK
- VARIANT and NOT VARIANT

For more information on the syntax of these modifiers, see the CREATE FUNCTION statement in Chapter 1 of the *Informix Guide to SQL: Syntax*.

Modifiers for External Procedures

The database server supports the following modifiers for external procedures:

- HANDLESNULLS
- CLASS
- STACK
- INTERNAL

For a list of the supported modifiers for external procedures, see the CREATE PROCEDURE statement in Chapter 1 of the *Informix Guide to SQL: Syntax.*

Returning Multiple Values from External Functions

The database server provides two methods for external functions to return multiple values:

- OUT keyword and Statement Local Variable to return multiple values with a single invocation of the function
- ITERATOR 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

The database server supports external functions that return more than one value. Use the OUT parameter in the CREATE FUNCTION statement to pass a pointer to the address to which the function can write extra values.

To return extra values from an external function with the OUT parameter

1. Write an external function that returns more than one value to the caller.

   The function must accept an extra parameter that is a pointer to the address to which the function can write. This pointer must be the last parameter.

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

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

2. Register the function with the OUT keyword to indicate that the function returns extra values.

   The OUT keyword indicates that the last parameter passes a pointer.

   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:

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

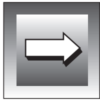
3. Create a Statement Local Variable (SLV) to use the OUT parameter when you invoke the external function in an SQL expression.

   Each SLV gives a variable name to one of the values dereferenced from the OUT pointer.

   To use an OUT parameter in SQL expressions, you create a Statement Local Variable (SLV). The SLV is statement-local because it provides a temporary name that a single statement can manipulate. An SQL statement uses each SLV to transmit the output from a single function to other parts of the SQL statement.

   To show that a variable is an SLV, follow the variable with a # sign and its data type. For example, the following statement has an SLV *y* that is typed as an INT:

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



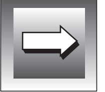
Important: *An SLV is valid only for the life of a single statement.*

Referencing OUT Parameters in User-Defined Routines

Each SLV argument in an SQL statement has a unique name. You can use the SLV name to dereference a return value multiple times within that statement. While each function can pass only one OUT, a single SQL statement in the calling routine can invoke multiple functions with OUT parameters. For example, the following partial statement receives pointers from two functions with OUT parameters, which are referenced with the SLV names **out1** and **out2**:

```
SELECT... WHERE func_2(x, out1 # INT) < 100
    AND (out1 = 12 or out1 = 13)
    AND func_3(a, out2 # FLOAT) = "SAN FRANCISCO"
    AND out2 = 3.14159;
```

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



***Important:** The function that receives the pointer should not try to read from this address because the value is meaningless.*

If the function that sources the SLV is not executed in an iteration of the statement, the SLV has a value of NULL. SLV values do not persist across iterations of the statement. At the start of each iteration, the SLV value is set to NULL.

Because an SLV shares the name space with procedure variables and columns names of the table involved in the statement, a priority has been established to determine which one takes precedence in ambiguous situations. Procedure variables have the highest precedence, and SLVs have the lowest precedence.

Iterator Function

An *iterator function* returns more than one row of values. The database server automatically invokes the function repeatedly, one for each row of return values. An iterator function is similar to an SPL function that contains the RETURN WITH RESUME statement.

Writing an Iterator Function

When you write an iterator function, you use DataBlade API functions (such as **mi_fp_setisdone()** and **mi_fp_request()**) to handle each return row. For more information on writing an iterator function, refer to the *DataBlade API Programmer's Manual*.

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 modifier.

The following sample CREATE FUNCTION statement shows how to register the function **TopK** as an iterator function:

```
CREATE FUNCTION TopK(integer, integer)
RETURNS integer not null
WITH (ITERATOR)
AS 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 one of the following methods:

- With the EXECUTE FUNCTION statement from:
    - DB-Access
    - In a prepared cursor in ESQL/C
    - 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
    - In an SPL FOREACH loop

Privileges for Registering a Routine

To register a routine in the database, a qualified user issues a CREATE FUNCTION or CREATE PROCEDURE statement. 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 statement.

  For an explanation of the DBA keyword, see "Executing a Routine as DBA" on page 3-21.

- A non-DBA user needs the Resource privilege to create an external routine. The creator has owner privileges on the routine.

  A user who does not have the DBA privilege cannot use the DBA keyword to register the routine.

  A DBA must grant the Resource privilege required for any other user to create a routine. The DBA can revoke the Resource privilege, preventing the revoke from creating further routines.

A DBA and the routine owner can cancel the registration with the DROP FUNCTION or DROP PROCEDURE statement.

Privileges for Executing a Routine

The Execute privilege enables users to invoke a routine. The routine might be invoked by the EXECUTE or CALL statements or by using a function in an expression. The following users have a default Execute privilege, which enables them to invoke a routine:

- By default, 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 a default Execution privilege for that routine.

- If the database is not ANSI compliant, user **public** (any user with Connect database privilege) automatically has the Execute privilege to a routine that is not registered with the DBA keyword.

ANSI

- In an ANSI-compliant database, the procedure owner and any user with the DBA privilege can execute the routine without receiving additional privileges. ♦

Granting and Revoking the Execute Privilege

Routines have the following GRANT and REVOKE requirements:

- The DBA can grant or revoke the Execute privilege to 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 Execution 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 created with the DBA keyword

An owner can restrict the Execution privilege on a routine even though the database server grants that privilege to public by default. To do this, issue the REVOKE EXECUTION ON….PUBLIC statement. The DBA and owner still can 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 who were granted the Execute privilege by that user.

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:

```
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 Chapter 1 of the *Informix Guide to SQL: Syntax*.

Privileges on Objects Associated with a Routine

The database server checks the existence of any referenced objects and verifies that the user invoking the routine has the necessary privileges to access the referenced objects. For example, if a user executes a routine that updates data in a table, the user must have the Update privilege for the table or columns referenced in the routine.

Objects referenced by a routine include:

- 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.

The owner of the routine, and not the user who runs the routine, owns the unqualified objects created in the course of executing the routine.

Figure 3-1 shows an SPL procedure called **promo()** that creates two tables, **hotcatalog** and **libby.mailers**.

Figure 3-1
CREATE PROCEDURE Example

```
CREATE PROCEDURE promo()

    CREATE TABLE hotcatalog
    (
        catlog_num INTEGER
        cat_advert VARCHAR(255, 65)
        cat_picture BLOB
    ) PUT cat_picturein sb1;

    CREATE TABLE libby.mailers
    (
        cust_num INTEGER
        interested_in SET(catlog_num INTEGER)
    );
END PROCEDURE;
```

Suppose the user **tony** executes the CREATE PROCEDURE statement in Figure 3-1 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 Routine 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. A DBA can, however, 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 exiting a DBA routine, the database server implicitly revokes the temporary DBA privilege.

Effect of DBA Privileges on Objects and Nested Routines

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 Figure 3-1 on page 3-21 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, then **marty** owns table **hotcatalog**. User **libby** owns **libby.maillist** because her name qualifies the table name, making her the table owner.

A called routine does not inherit the DBA privilege. If a DBA routine executes a routine that was created without the DBA keyword, the DBA privileges do not affect the called routine.

If a routine that is registered without the DBA keyword calls a DBA routine, the caller must have Execute privileges on the called DBA routine. Statements within the DBA routine execute as they would within any DBA routine.

The following example demonstrates what occurs when a DBA and non-DBA routine interact. Procedure **dbspace_cleanup()** executes procedure **cluster_catalog()**. Procedure **cluster_catalog()** creates an index. The C-language source for **cluster_catalog()** includes the following statements:

```
strcopy(statement, “CREATE INDEX stmt”);
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 **dbpace_cleanup()**. Index **c_clust_ix** is created by a non-DBA routine, so **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.

Chapter

4

Debugging User-Defined Routines

Invoking a Routine 4-3
Invoking a Function with the EXECUTE statement 4-3
Invoking a Procedure with the EXECUTE statement 4-5
Invoking a Function with the CALL Statement 4-5
Invoking a Procedure with the CALL Statement 4-5
Invoking a Function in an Expression 4-6
Explicitly Invoking a Function in an Expression 4-6
Implicitly Invoking a Function That Is Bound to an Operator . 4-6
Implicitly Invoking a Function for Casting 4-7

Private Installation 4-8

Installing and Registering DataBlade Modules 4-8

Debugging a DataBlade Module 4-9
Connecting to the Server from a Client. 4-9
Loading the DataBlade Module 4-9
Identifying the Server Process. 4-10
Starting the Debugger 4-11
Setting Breakpoints 4-11
Symbols in Shared Object Files 4-12

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

- Invoking UDRs
- Installing a private server
- Installing and registering a UDR
- Debugging a UDR

Invoking a Routine

This section describes the following methods that you can use to invoke a routine:

- Use the EXECUTE statement.
- Use the CALL statement.
- Invoke a routine in an expression.

Invoking a Function with the EXECUTE statement

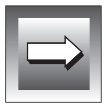
You can use the EXECUTE statement with the FUNCTION keyword to execute a function from one of the following:

- SPL
- An ESQL/C program
- DB-Access

For example, suppose **result** is a function variable of type BOOLEAN. The following EXECUTE statement invokes the **equal()** function:

```
CREATE FUNCTION equal (arg1 udtype2, arg2 udtype2)
    RETURNING BOOLEAN;
    SPECIFIC udtype2_lessthan
    WITH (HANDLESNULLS,
    NOT VARIANT)
    EXTERNAL NAME "/usr/lib/udtype2.so(udtype2_lessthan)"
    LANGUAGE C
    END FUNCTION;
EXECUTE FUNCTION equal (arg1, arg2) INTO result
```

The INTO clause must always be present when you invoke a function using an EXECUTE statement.



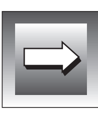
Important: *You cannot use the EXECUTE statement to invoke a function that has an OUT parameter.*

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

```
CREATE ROW TYPE emp
    (name varchar(30),
     emp_num int,
     salary numeric(10,2));
CREATE ROW TYPE trainee UNDER emp ...
CREATE FUNCTION func1 (trainee) RETURNS row ...
```

The following EXECUTE 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 function invoked with the EXECUTE statement, ensure that you enclose the query in another set of parentheses.*

Invoking a Procedure with the EXECUTE statement

You can use the EXECUTE statement with the PROCEDURE keyword to execute a procedure from within an SPL or ESQL/C program or DB-Access. The following EXECUTE statement invokes the **log_compare()** function:

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

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

Invoking a Function with the CALL Statement

You can use the CALL statement to invoke a function from within an SPL program only. The following statement invokes the **equal()** function:

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

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

You cannot use the CALL statement to invoke a function that has an OUT parameter.

You cannot execute the CALL statement from within an ESQL/C program or the DB-Access utility.

Invoking a Procedure with the CALL Statement

You can use the CALL statement to invoke a procedure from within an SPL program only. The following CALL statement invokes the **log_compare()** procedure:

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

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

Invoking a Function in an Expression

You can invoke a function in an expression either explicitly or implicitly. An external procedure cannot be used in an expression because an external procedure does not return a value. This section describes how you can invoke functions explicitly and implicitly in an expression. The examples in the following sections use the **new_type1** and **new_type2** distinct types and the **tab_1** table. Figure 4-1 shows the syntax that creates the distinct types and table.

Figure 4-1

```
CREATE DISTINCT TYPE new_type1 AS DOUBLE PRECISION;
CREATE DISTINCT TYPE new_type2 AS INT;

CREATE TABLE tab_1
(
    col_1 new_type1,
    col_2 new_type1,
    col_3 new_type2
);
```

Explicitly Invoking a Function in an Expression

You can invoke a function in an expression when the function returns a single value. Suppose an equal() function exists to evaluate the equality of **new_type1** and **new_type2** values. The following query invokes the appropriate **equal()** function in the WHERE clause of the SELECT statement:

```
SELECT *
FROM tab_1
WHERE equal (col_1, col_3)
```

Implicitly 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 **new_type1** and returns a Boolean. If the equal operator (=) is used for comparisons between **col_1** and **col_2**, the **equal()** function gets invoked automatically. For example, the following query implicitly invokes the appropriate **equal()** function to evaluate the WHERE clause:

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

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

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

Implicitly Invoking a Function for Casting

Suppose you create the following external function to cast a value of **newtype2** to **newtype1**:

```
CREATE FUNCTION ntype2_to_ntype1 (arg1 newtype2)
RETURNING newtype1
EXTERNAL NAME "/usr/lib/btype1/lib/libntype2.so"
LANGUAGE C
END FUNCTION;
```

Suppose also that you register the function as an implicit cast, as follows:

```
CREATE IMPLICIT CAST (newtype2 AS newtype1 WITH
ntype2_to_ntype1)
```

The following query automatically invokes the cast to convert **new_type2** values to **newtype1** before it invokes the **equal()** function:

```
SELECT *
FROM tab
WHERE col_1 = col_3
```

The previous query is equivalent to the following statement:

```
SELECT *
FROM new_table
WHERE equal (col_1, CAST(col_3 AS newtype1))
```

Private Installation

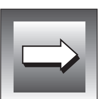
A private installation provides support for users who are developing DataBlade modules and user-defined routines. It allows a developer to test code extensions to Universal Server without affecting performance or the work of other users.

Normally, the utility that starts Universal Server, **oninit**, runs with the privileges of the user **informix**. Therefore, permitting a non-privileged user to create new functions, dynamically link them with the server, and execute them violates security. A private installation allows a developer to run Universal Server without the privileges of the user **informix** or the privileges of the user **root** to facilitate debugging DataBlade modules and user-defined routines.

A private installation does not affect a conventional installation of Universal Server. For information about creating a private-installation version of Universal Server, refer to the *INFORMIX-Universal Server Installation Guide*.

Installing and Registering DataBlade Modules

Installing a DataBlade module places the files for the modules in a subdirectory of the **INFORMIXDIR/extend** directory. Registering a DataBlade module adds the module to a database.



Important: *To debug a DataBlade module, the shared object file must be compiled with the* `-g` *compiler option so that debugging symbols are available to the debugger.*

For DataBlade modules that you create with the DataBlade Developers Kit, you can copy the directory tree that BladePack generates into the module subdirectory under **INFORMIXDIR/extend**. Then you use BladeManager to register the DataBlade module in your test database. For instructions on running BladeManager, refer to the *BladeManager User's Guide*.

To install and register other DataBlade modules, such as the DataBlade modules included with Universal Server, refer to the instructions that accompany them.

Debugging a DataBlade Module

To debug your DataBlade module, use a debugger that can attach to the active server process and access the symbol tables of dynamically loaded shared object files. On Solaris, the **debugger** utility meets these criteria.

To attach to the server process, do the following:

- Connect to the server from a client application, such as DB-Access.
- Identify the server process that is executing the DataBlade module code.
- Start the debugger on the server process.

The following sections describe these steps.

Connecting to the Server from a Client

To connect to Universal Server, choose a client tool that allows you to submit ad hoc queries. On Windows NT, you can use INFORMIX-SQL Editor. On UNIX, you can use DB-Access.

For example, from UNIX, execute the following command, where *database* is a database in which you registered the DataBlade module that you want to debug:

```
dbaccess database
```

Loading the DataBlade Module

Before you can attach to the server process with the debugger, you need to load the shared object file for your DataBlade modules into the server address space. With the shared object file loaded, you can set breakpoints on the entry points for the module and examine local storage provided by the module functions.

To load the DataBlade module into the server address space, execute one of its functions. One technique is to define a routine in the DataBlade module that you use to load the DataBlade module. The routine can be as simple as the following:

```
mi_integer mod_load_bld()
{
    return 0;
}
```

To prevent name conflicts, substitute the object prefix assigned to your DataBlade module in place of *mod*.

To load the DataBlade module shared object file, execute the following command in your client application:

```
select mod_load_bld()from informix.systables where tabid=1;
```

Identifying the Server Process

To find the CPU or EXT virtual process in which your DataBlade module is loaded, execute the **onstat** command, as follows:

```
onstat -g glo
```

Figure 4-2 shows the last section of the output of this **onstat** command.

```
Individual virtual processors:
 vp    pid       class      usercpu    syscpu     total
 1     3544      cpu        3.75       0.96       4.71
 2     3545      adm        0.05       0.03       0.08
 3     3546      lio        0.04       0.07       0.11
 4     3547      pio        0.05       0.03       0.08
 5     3548      aio        0.04       0.04       0.08
 6     3549      msc        0.39       0.19       0.58
 7     3550      aio        0.09       0.10       0.19
 8     3551      aio        0.03       0.07       0.10
 9     3552      aio        0.02       0.07       0.09
 10    3553      aio        0.04       0.04       0.08
 11    3554      aio        0.03       0.05       0.08
                 tot        4.53       1.65       6.18
```

Figure 4-2
onstat -g glo *Command Output*

Find the CPU or EXT virtual processor that you want to debug and record its process ID for the next step.

Starting the Debugger

To start the debugger, enter the following command at the shell prompt, where *process_id* is the PID of the CPU or EXT virtual process:

```
debugger - process_id
```

This command starts the debugger on the server virtual-processor process without starting a new instance of the virtual processor.

You can set breakpoints, examine the stack, resume execution, or carry out any other normal **debugger** command. For more information about available **debugger** commands, see the **debugger** manual page.

Setting Breakpoints

You can set breakpoints in any function with an entry point known to **debugger**, which includes internal server functions and your DataBlade module functions. Universal Server is compiled with debugging support turned off, so local storage and line number information is not available for server routines. However, because you compiled the DataBlade module for debugging, you can see line number information and local storage for your functions.

The Universal Server routine that calls your DataBlade module functions is called **udr_execute()**. You can set a breakpoint in this routine as follows:

```
stop in udr_execute
cont
```

When you enter a command in the client that calls one of your DataBlade module functions, the debugger stops in the **udr_execute()** routine. Then you can step through your function. Because your DataBlade module is compiled with debugging support, you can view the local variables and stack for your functions.

Symbols in Shared Object Files

Undefined symbols in a shared object file are resolved with the main Universal Server module when the file is loaded. If a symbol is missing, the load fails on the first execution of the user-defined routine, and a message is written in the log file.

Symbols defined in two different shared object files are distinct entities and do not resolve against each other.

A symbol defined in a shared object file and the Universal Server main module behaves in one of two ways:

- If the symbol referenced in the shared object file is in the same source file that references it, the debugger accesses the symbol in the shared object file, as expected.
- If the shared object file includes more than one source file and a cross-file symbol reference exists, the symbol is resolved to the Universal Server main module.

Chapter

Performance Considerations

5

SPL Considerations. 5-3
SPL Compilation 5-4
SPL Execution 5-5
SPL Optimization 5-5
Optimization Levels. 5-5
When Optimization Occurs Automatically 5-6
Updating Statistics for a UDR 5-7

Choosing a Virtual-Processor Class 5-8
CPU Virtual-Processor Class 5-8
User-Defined Virtual-Processor Class 5-9
Defining an Extension Virtual-Processor Class. 5-9
Configuring an Extension Virtual-Processor Class 5-10
Using the noyield Option 5-10
Adding and Dropping User-Defined Virtual Processors in
On-Line Mode 5-11

Considerations for Parallel Execution of SPL Routines 5-11
Number of Virtual Processors (VPCLASS) 5-12

Memory Considerations 5-12
Stack-Size Considerations 5-12
Setting Stack Sizes for User-Defined Routines 5-13
Virtual-Memory Cache for Routines 5-13
System Catalog Cache 5-13
SPL Cache 5-14

I/O Considerations. 5-14
Isolate System Catalogs 5-15
Balance the I/O Activities 5-15

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

- SPL considerations
- Choosing a virtual processor class
- Considerations for parallel execution of SPL routines
- Memory considerations
- I/O considerations

SPL Considerations

This section describes the compilation, execution, and optimization processes of an SPL routine.

Tip: *Not all the encapsulated SPL that you created as SPL procedures in earlier Informix products has the properties currently associated with 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.*

SPL Compilation

Universal Server compiles an SPL routine when you execute the CREATE PROCEDURE or CREATE FUNCTION. The following activities occur in the compilation process of the SPL routine:

- Parse and optimize, if possible, all SQL statements

  The database server puts the SQL statements into 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 "SPL Optimization" on page 5-5.
- Build 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.
- Parse SPL statements and convert to pcode

  The term *pcode* refers to pseudocode that an interpreter executes quickly
- Convert the pcode, 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**.
- Store general information about the procedure in the **sysprocedures** system catalog table
- Store permissions for the procedure in the **sysprocauth** system catalog table

SPL Execution

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

- Retrieves the pcode, 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 ROUTINE, 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 pcode instructions.

SPL Optimization

A *query plan* is a specific way that a query might be performed. A query plan includes how to access tables, the order of joining tables, and the use of temporary tables. 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 into an execution plan for the SPL routine.

Optimization Levels

The current optimization level set in an SPL routine affects how the SPL routine is optimized.

The algorithm that a SET OPTIMIZATION HIGH statement invokes is 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.

The alternative algorithm that a SET OPTIMIZATION LOW statement invokes eliminates unlikely join strategies during the early stages, which reduces the time and resources spent during optimization. However, when you specify a low level of optimization, the optimal strategy might not be selected because it was eliminated from consideration during early stages of the algorithm.

For SPL routines that remain unchanged or change only slightly, you might want to set the SET OPTIMIZATION statement to `HIGH` when you create the routine. This optimization level stores the best query plans for the routine. Then set optimization to `LOW` before you execute the routine. The routine then uses the optimal query plans and runs at the more cost-effective rate if reoptimization occurs.

When Optimization Occurs Automatically

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 a UDR

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.

If you do not run UPDATE STATISTICS after the size or content of any 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 procedure have changed.

The various clauses on the UPDATE STATISTICS statement influence re-optimization in the following ways:

- UPDATE STATISTICS

  When you specify no clauses, the database server reoptimizes SQL statements in all SPL routines and changes the statistics for all tables.
- 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.This statement does not reoptimize any SQL statements in SPL routines.
- UPDATE STATISTICS FOR TABLE

  When you specify the FOR TABLE clause without a table name, the database server changes the statistics for all tables and does not reoptimize any SQL statements in SPL routines.

- UPDATE STATISTICS FOR routine statistics *name*

  When you specify one of the following clauses with a name, the database server reoptimizes SQL statements in only the SPL routine listed:

  - FOR FUNCTION *routine name* clause
  - FOR PROCEDURE *routine name* clause
  - FOR ROUTINE *routine name* clause

  When you specify one of these clauses, the database server does not update the statistics in the system catalog tables.

- UPDATE STATISTICS FOR routine statistics

  When you specify one of the following clauses without a name, the database server reoptimizes SQL statements in all SPL routines:

  - FOR FUNCTION clause
  - FOR PROCEDURE clause
  - FOR ROUTINE clause

  When you specify one of these clauses, the database server does not update the statistics in the system catalog tables.

The database server updates the **sysprocplan** system catalog table with the reoptimized execution plan.

Choosing a Virtual-Processor Class

You must choose the virtual-processor (VP) class in which to run the user-defined routine. User-defined routines run in the CPU VP by default.

CPU Virtual-Processor Class

Generally, user-defined routines perform best in the CPU VP because threads do not have to migrate among operating-system processes during query execution.

However, CPU VP execution requires additional programming. You must make sure that your code adheres to the following guidelines:

- Yields the CPU on a regular basis to other threads
- Does not use blocking operating-system calls
- Does not allocate local resources
- Does not modify the global VP state

User-Defined Virtual-Processor Class

You can designate user-defined classes of virtual processors, called Extension (EXT) VPs, to run user-defined routines.

Running in an Extension VP relaxes some, but not all, of the programming requirements. These routines can perform blocking operations, but they still cannot perform local resource allocations because they might migrate among the VPs.

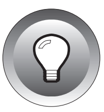
Routines that run in a user-defined virtual processor class need not yield the processor, and they might issue direct file-system calls that block further processing by the virtual processor until the I/O is complete. Because virtual processors are not CPU virtual processors, however, the normal processing of user queries is not affected.

This option results in lower performance because queries normally execute in the CPU VP, and the query thread must migrate to the EXT VP to evaluate user-defined routines. Thus, you should use the EXT VP with caution.

Defining an Extension Virtual-Processor Class

You might want to define a user-defined class of virtual processors to run DataBlade or user-defined routines. When you register a user-defined routine or function, you assign it to a class of virtual processors with the CLASS parameter of the CREATE FUNCTION statement. For example, the following CREATE FUNCTION statement registers the user-defined routine, **GreaterThanEqual** and specifies that calls to this routine should be executed by the user-defined VP class named **new**:

```
CREATE FUNCTION GreaterThanEqual(ScottishName, ScottishName)
    RETURNS boolean
    WITH CLASS = new
```



***Tip:** You can use the CREATE FUNCTION statement to create routines or functions that reference any user-defined class that you like, and the class need not exist when the function is created. However, if you try to use a function that refers to a user-defined 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.*

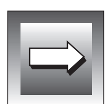
Configuring an Extension Virtual-Processor Class

You configure new virtual-processor classes in 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 in the CREATE FUNCTION statement.

The following example creates the user-defined class **new**, for which the database server starts three virtual processors initially:

```
VPCLASS new,num=3   # New VP class for testing
```

The database server executes user-defined routines that you register with the `CLASS = 'new'` clause in the **new** virtual-processor class. The class name is not case sensitive. It appears in the **onstat -g glo** output as a new process.



***Important:** If you create a new virtual-processor class, you must remove the SINGLE_CPU_VP parameter from the ONCONFIG file.*

Using the noyield Option

The **noyield** option causes a user-defined class of virtual processors to run user-defined routines in a way that gives the routine exclusive use of the virtual-processor class. In other words, user-defined routines that use a **noyield** virtual-processor class run serially, and each routine runs to completion before the next one begins.

You should assign a user-defined routine to a nonyielding class of virtual processors if the routine has not been designed or coded to handle the concurrency issues of multiprocessing. For example, if a user-defined routine uses global variables and makes calls to database services, such as the smart-large-object interface, you should assign it to a nonyielding class of user-defined virtual processors. Similarly, a call to a database can cause the processing thread to yield for I/O operations. Yielding allows another thread to run the same code and change the states of variables that the original thread assumes to be stable when it resumes processing.

This option is ignored for pre-defined virtual-processor classes such as CPU, AIO, and so on.

The following example specifies a user-defined class of virtual processors called **new**, that run in **noyield** mode:

```
VPCLASS new,noyield,num=1
```

Adding and Dropping User-Defined Virtual Processors in On-Line Mode

You can add or drop virtual processors in a user-defined class while Universal Server is on-line. Use **onmode -p** to add virtual processors to the class. The following command adds three virtual processors to the **new** class:

```
onmode -p +3 new
```

For more information on how to assign a user-defined routine to either CPU or user-defined classes of virtual processors, refer to the CREATE FUNCTION statement, and specifically to the CLASS parameter, in the *Informix Guide to SQL: Syntax*.

Considerations for Parallel Execution of SPL Routines

When you invoke an SPL routine with the EXECUTE or CALL statements, the SQL statements within an SPL routine can take advantage of parallel processing. The parallel database query (PDQ) feature of Universal Server executes a single query with multiple threads in parallel. Another Universal Server 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 contained in fragmented tables.

The features of PDQ allow Universal 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.

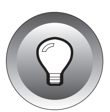
For more information on the PDQ feature, refer to the *INFORMIX-Universal Server Administrator's Guide*. For more information on the performance implications of PDQ, refer to the *INFORMIX-Universal Server Performance Guide*.

Number of Virtual Processors (VPCLASS)

The dynamic, multithreaded nature of a Universal Server virtual processor can perform parallel processing. Virtual processors of the CPU class can run multiple session threads, working in parallel, for an SQL statement contained within an SPL routine.

You can increase the number of CPU virtual processors by setting configuration parameters 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 Universal Server communication mechanism uses the SIGUSR1 signal. When you are debugging, you must avoid SIGUSR1 to prevent server processes from hanging.*

Memory Considerations

This section describes stack-size considerations and the virtual-memory cache for routines.

Stack-Size Considerations

The database server allocates local storage in user-defined routines from shared memory. As a result, a user-defined routine must not consume excessive stack space, either through large local-variable declarations or through excessively long call chains or recursion. A function 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 allocated by a thread must be in shared memory. Otherwise, the memory is not visible when the thread moves from one VP to another.

The database server dynamic function manager 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.

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 interfaces, memory visibility is not a problem. For more information on the DataBlade API, refer to the *DataBlade API Programmer's Manual*.

Setting Stack Sizes for User-Defined Routines

When you specify a stack size for a user-defined routine, the server allocates the specified amount of memory for every execution iteration of the routine. If a routine does not need a stack larger than 32 kilobytes, do not specify a stack size.

Virtual-Memory Cache for Routines

Universal Server caches the following items in the virtual portion of the database server shared memory:

- For SPL routines and external routines, information in the **sysprocedures** system catalog table
- For SPL routines only, the executable form of the routine

System Catalog Cache

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 virtual portion of shared memory for the database server. The database server uses this information in shared memory if it is present for subsequent sessions that invoke the SPL routine.

The database server keeps this **sysprocedures** system catalog information in the virtual memory cache 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 pcode for the routine
- Number of arguments
- Data types of parameters
- Type of routine (function or procedure)
- Location of external routine
- Virtual processor class in which the routine executes

SPL 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 pcode into 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 this executable format of an SPL routine in the virtual-memory cache on a *most recently used* basis.

I/O Considerations

The database server stores user-defined routines and triggers in the following system catalog tables:

- **sysprocbody**
- **sysprocedures**
- **sysprocplan**
- **sysprocauth**
- **systrigbody**
- **systriggers**

These system catalog tables can grow very large with heavy use of user-defined routines in a database. You can tune the key system catalogs as you would any heavily utilized data tables. To improve performance, use the following methods:

- Isolate system catalogs.
- Balance the I/O activities.

Isolate System Catalogs

If your database server has multiple physical disks available, you can isolate your system catalogs 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.

Balance the I/O Activities

If you have a large number of user-defined routines 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 user-defined routine 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 one-half of your user-defined routines 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 user-defined routines. The last half of the routines should spill into the second chunk.
6. Drop the temporary table.

Index

A

ANSI compliance
  icon Intro-8
  level Intro-10
ANSI-compliant database
  routine resolution 2-8
  routine signature 2-5
Argument
  data type does not match parameter data type 2-11, 2-12
  data-type cast 2-15
Argument list
  distinct data type in 2-11
  modal 3-5
  order of 2-8
  prefered format 3-5
  size 3-5

C

Cast
  effect on routine resolution 2-15
Coding standards 3-5
Comment icons Intro-7
Compliance
  icons Intro-8
  with industry standards Intro-10
CPU virtual processor
  adding and dropping in on-line mode 5-11
CREATE FUNCTION
  EXTERNAL NAME clause 3-11
  OUT parameter 3-14
CREATE FUNCTION statement 5-9
  and VPCLASS 5-10
  registering a function 3-10
  routine overloading 2-3
  SPECIFIC keyword with specific name 2-6
CREATE PROCEDURE statement
  SPECIFIC keyword with specific name 2-6

D

Data type hierarchy, description of 2-10
DataBlade API
  memory management routines 5-13
DataBlade module
  debugging 4-8
  loading into server address space 4-9
dbaccessdemo7 script Intro-5
Debugging
  starting the debugger 4-11
  steps for 4-9
Default locale Intro-4
Demonstration database Intro-5
Distinct data type
  description of 2-11
  routine resolution and 2-11
Documentation conventions
  icon Intro-7
  typographical Intro-6
Documentation notes Intro-10

Documentation, types of
documentation notes Intro-10
error message files Intro-9
machine notes Intro-10
on-line help Intro-9
on-line manuals Intro-8
printed manuals Intro-9
release notes Intro-10

E

en_us.8859-1 locale Intro-4
Error message files Intro-9
Execute privilege
DBA keyword, effect of 3-21
objects referenced by a routine 3-20
External function
registering 3-10
External procedure
registering 3-12
External routine
library entry point 3-11
module library, specifying 3-11
registering, with modifiers 3-13
specifying library location 3-11

F

Feature icons Intro-7
Features, product Intro-5
Function
invoking
in an expression 4-6
with CALL statement 4-5
with EXECUTE statement 4-3
Function overloading 1-13

G

Global Language Support (GLS) Intro-4
GRANT statement
example using signature 2-6, 3-20
example using specific name 2-6

I

Icons
comment Intro-7
compliance Intro-8
feature Intro-7
product Intro-7
Industry standards, compliance with Intro-10
INFORMIXDIR/bin directory Intro-5
ISO 8859-1 code set Intro-4

L

Locale Intro-4

M

Machine notes Intro-10
Major features Intro-5
Modality, description of 3-5

N

Name space, shared by DataBlade module objects 3-5
NODFDAC
effect on privileges granted to public 3-19
Null value
argument for an overloaded routine 2-16

O

On-line help Intro-9
On-line manuals Intro-8
Operator class
definition 1-8
OUT parameter 3-14
reading values of 3-16

P

Platform portability
coding standards for 3-5
Polymorphism 3-4
Printed manuals Intro-9
Private installation 4-8
Procedure
invoking
with CALL statement 4-5
with EXECUTE statement 4-5
Product icons Intro-7

Q

Query plan
description 5-5

R

Release notes Intro-10
Return value
multiple values 3-14
Routine
wildcard argument 2-16
Routine name
ANSI-compliant 2-5
Routine overloading
description 2-3, 3-4
routine signature 2-4
Routine resolution
ANSI-compliant database 2-8
candidate list 2-8
description of 2-7
distinct type 2-12
effect of inheritance 2-10, 2-11
effect of null value argument 2-15
order of arguments 2-11
order of casting 2-15
precedence list 2-8
type hierarchy 2-10
Routine signature
ANSI-compliant 2-5
description 2-4
use in CREATE statement 2-5
use in SQL statements 2-4

S

Shared object file
  loading into server address space 4-9
Signature
  components of 2-4
SLV
  referencing return values with 3-16
Software dependencies Intro-4
SPECIFIC keyword 2-6
Specific name
  creating 2-6
  naming conventions 2-6
Stack space
  used by DataBlade module routines 5-12
Statement local variable (SLV) 3-15
  precedence of 3-16
stores7 database Intro-5
Strategy functions
  definition 1-8
Structured Query Language (SQL)
  CREATE FUNCTION statement 2-4, 2-5
  CREATE PROCEDURE statement 2-5
  statements using routine signature 2-4
  statements using specific name 2-6
Subtype and supertype 2-10
Support functions
  definition 1-8

T

Trigger
  definition of 1-13
Triggered action
  statements 1-13
Type hierarchy
  description of 2-10

U

Universal Server
  dynamic function manager 5-13
User-defined routine (UDR)
  description of 1-3
  invoking 4-3
  routine overloading, description of 3-4
  routine resolution 2-7

V

Virtual processors
  choosing for user-defined routine 5-8
VPCLASS parameter
  user-defined classes 5-9

W

Wildcard
  argument for a routine 2-16

X

X/Open compliance
  level Intro-10

Symbols

sign, SLV indicator 3-15