Extending INFORMIX®-Universal Server:

Data Types

Version 9.1
March 1997
Part No. 000-3765

Published by INFORMIX® Press

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

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

Table of Contents

Introduction

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: Data Types explains how to extend existing data types and define new data types for an INFORMIX-Universal Server database. It describes the tasks you must perform to extend operations on data types, to create new casts, to extend operator classes for secondary access methods, and to write opaque data types.

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, "Extending Data Types," introduces the concept of a data type and describes how you can extend data types with Universal Server.
- Chapter 2, "Extending an Operation," describes how you can extend the operations that are available on a data type.
- Chapter 3, "Creating User-Defined Casts," describes how to create a casting function for a data type.
- Chapter 4, "Extending an Operator Class," describes how you can extend an operator class to provide additional functionality for a secondary access method (an index).

- Chapter 5, "Creating an Opaque Data Type," provides the steps for the creation of an opaque data type, a user-defined data type for which you must provide both the internal structure and the functions that operate on this structure.
- Chapter 6, "Writing Support Functions," describes the support functions for an opaque data type and what capabilities they provide for the opaque type.

Types of Users

This manual is written for the database programmer or the DataBlade developer who needs to extend functionality of existing data types or to provide new data types for a Universal Server database.

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

Extending INFORMIX-Universal Server: Data Types provides information on how to use Universal Server features to extend data types:

- Extend operators by writing new SQL functions and registering them with the CREATE FUNCTION statement
- Support data conversion by creating new casts
- Extend operator classes by writing new strategy and support functions, registering new operator classes with the CREATE OPCLASS statement, and building nondefault indexes with the CREATE INDEX statement
- Create opaque data types by creating an internal structure and writing support functions, registering the new opaque type with the CREATE OPAQUE TYPE statement, and granting privileges with the GRANT statement
- Write support functions for an opaque type in the C language

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

Icon	Description
GLS	Identifies information that relates to the Informix Global Language Support (GLS) feature.
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.
X/O	Identifies functionality that conforms to X/Open.

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
EXTDATADOC_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

Extending Data Types

1

This chapter introduces the following concepts about the data type system of INFORMIX-Universal Server:

- What is the extensible data type system of Universal Server?
- How do you extend the behavior of data types that Universal Server already defines?
- How do you define new data types and provide Universal Server with the information it needs about the behavior of these data types?

What Is Data Type Extensibility?

Universal Server provides an extensible data type system that enables you to:

- use the data types that the data type system defines and supports.
- extend the data type system to support additional behavior for data types.

What Does the Data Type System Do?

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

- What layout or *internal structure* can the database server use to store the data type values on disk?

  The database server provides certain data types for which it has determined the internal structure of the associated data on disk.
- Which *operations* (such as multiplication or string concatenation) can the database server apply to values of a particular data type?

  An operation must be defined on the values of a particular data type. Otherwise, the database server does not allow the operation to be performed.
- Which access methods can the database server use for values in columns of this data type?

  An access method defines how to handle:
  - storage and retrieval of a particular data type in a table (a primary access method).

    If the primary access method does not handle a particular data type, the database server cannot access values of that type. For more information on how to define primary access methods, see the *Virtual-Table Interface Programmer's Manual*.
  - storage and retrieve of a particular data type in an index (a secondary access method).

    If the *operator class* of a secondary access method does not handle a particular data type, you cannot use the secondary access method to index the data type values.
- Which *casts* can the database server use to perform data conversion between values of two different data types?

  The database server uses casts to convert data from one data type to another.

The data type system of Universal Server knows how to provide this behavior for the data types that it provides.

Data Types That Universal Server Provides

A data type tells the database server about the *internal structure* of the data type values. Universal Server provides code that understands the internal structure of each of the following data types:

- Built-in data types
- Complex data types

This section summarizes the data types that the data type system of Universal Server provides for you. For a more complete description of the data types that Universal Server supports, see the chapter on data types in the *Informix Guide to SQL: Reference*.

Built-In Data Types

A built-in data type is a fundamental data type that the database server defines. A fundamental data type is atomic; that is, it cannot be broken into smaller pieces, and it serves as the building block for other data types. Figure 1-1 summarizes the built-in data types that Universal Server provides.

Figure 1-1
Built-In Data Types of Universal Server

Data Type	Explanation
BLOB	Stores binary data in random-access chunks.
BOOLEAN	Stores the boolean values for true and false.
BYTE	Stores binary data in chunks that are not random access.
CHAR(*n*)	Stores single-byte or multibyte sequences of characters, including letters, numbers, and symbols of fixed length; collation is code-set dependent.
CHARACTER(*n*)	Is a synonym for CHAR.
CHARACTER VARYING(*m*,*r*)	Stores single-byte and multibyte sequences of characters, including letters, numbers, and symbols of varying length; collation is code-set dependent; is an ANSI-compliant version of the VARCHAR data type.
CLOB	Stores text data in random-access chunks.

(1 of 3)

Data Type	Explanation
DATE	Stores a calendar date.
DATETIME	Stores a calendar date combined with the time of day.
DEC	Is a synonym for DECIMAL.
DECIMAL	Stores numbers with definable scale and precision.
DOUBLE PRECISION	Behaves the same way as FLOAT.
FLOAT(*n*)	Stores double-precision floating-point numbers that correspond to the **double** data type in C (on most platforms).
INT	Is a synonym for INTEGER.
INT8	Stores an 8-byte integer value. These whole numbers can be in the range -9,223,372,036,854,775,87 to 9,223,372,036,854,775,807 (which is -(2^{63}-1) to 2^{63}-1).
INTEGER	Stores whole numbers from –2,147,483,647 to +2,147,483,647 (which is -(2^{31}-1) to 2^{31}-1).
INTERVAL	Stores a span of time.
LVARCHAR	Stores single-byte or multibyte strings of letters, numbers, and symbols of varying length to a maximum of 32 kilobytes; is also the external storage format for opaque data types; collation is code-set dependent.
MONEY(*p,s*)	Stores a currency amount.
NCHAR(*n*)	Stores single-byte and multibyte sequences of characters, including letters, numbers, and symbols; collation is locale dependent. For more information, see the *Guide to GLS Functionality*.
NUMERIC(*p,s*)	Is a synonym for DECIMAL.
NVARCHAR(*m,r*)	Stores single-byte and multibyte sequences of characters, including letters, numbers, and symbols of varying length to a maximum of 255 bytes; collation is locale dependent. For more information, see the *Guide to GLS Functionality*.

(2 of 3)

Data Type	Explanation
REAL	Is a synonym for SMALLFLOAT.
SERIAL	Stores sequential integers; has the same range of values as INTEGER.
SERIAL8	Stores large sequential integers; has the same range of values as INT8.
SMALLFLOAT	Stores single-precision floating-point numbers that correspond to the **float** data type in C (on most platforms).
SMALLINT	Stores whole numbers from –32,767 to +32,767 (which is $-(2^{15}-1)$ to $2^{15}-1$).
TEXT	Stores text data in chunks that are not random access.
VARCHAR(*m*,*r*)	Stores single-byte or multibyte strings of letters, numbers, and symbols of varying length to a maximum of 255 bytes; collation is code-set dependent.

(3 of 3)

For more information on these built-in data types, see their entries in the chapter on data types in the *Informix Guide to SQL: Reference*.

Complex Data Types

A *complex data type* is built from a combination of other data types with an SQL type constructor. An SQL statement can access individual components within the complex type. There are two kinds of complex types:

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

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

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

Figure 1-2 summarizes the complex data types that Universal Server provides.

Figure 1-2
Complex Data Types of Universal Server

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

For more information on these complex data types, see the chapter on data types in the *Informix Guide to SQL: Reference*.

Operations

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

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

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

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

  The SUM and AVG functions are examples of aggregate functions.
- An *end-user routine* is a user-defined routine that end users can use in SQL statements.

  An end-user routine can be either a function (which returns a value) or a procedure (which does not return a value).

Tip: *A casting function is also an operation on a data type; it converts one data type to another. However, because casts have other special features, this manual discusses them separately from data type operations. For more information, see "Casts" on page 1-10.*

The database server provides operator functions, built-in functions, and aggregate functions that handle the data types it provides. For a description of these operations and how to extend them, see Chapter 2, "Extending an Operation."

Operator Classes

A data type tells the database server which *operator class* to associate with the data type values when they are stored in a secondary access method. The secondary access method builds and accesses an index. An operator class associates a group of operators with a secondary access method. When you extend an operator class, you provide additional functions that can be used as filters in queries and for which the database server can use an index.

Universal Server provides an operator class for the built-in secondary access method, a generic B-tree. This operator class handles the data types that the database server provides. For a description of operator classes and how to extend them, see Chapter 4, "Extending an Operator Class."

Casts

A data type tells the database server which *cast* to use to convert the data type value to a different data type. A *cast* performs the necessary operations for conversion from the data type to another data type. When two data types have different internal formats, the database server calls a *casting function* to convert one data type to another. Universal Server provides casts between the built-in data types. For a description of casts and how to extend them, see Chapter 3, "Creating User-Defined Casts."

How Can You Extend the Data Type System?

The data type system of Universal Server is an *extensible* data type system. That is, this data type system is flexible enough to support the following kinds of changes:

- You can extend existing data types.

  You can redefine some of the behavior for the data types that Universal Server provides. For more information, see "Extending Existing Data Types" on page 1-11.

- You can define your own data types.

  You must then define the behavior for these new data types. For more information, see "Creating New Data Types" on page 1-12.

Extending Existing Data Types

You can define the following additional behavior for an existing built-in or complex data type:

- Additional *operations* on existing data types
- New functionality for a secondary access method (an index) with a new *operator class* that supports an existing data type
- Additional *casts* to provide data conversions between existing data types and other data types

Additional Operations

Universal Server defines operations on the data types that it provides. (For a list of these data types, see "Data Types That Universal Server Provides" on page 1-5.) To extend operations in your database, you can write additional routines, called *end-user routines*, that end users can include in their SQL statements. You must register each end-user routine in the database with the CREATE FUNCTION (or CREATE PROCEDURE) statement. For more information, see "How Do You Extend an Operation?" on page 2-11.

New Operator Classes

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

To provide additional sequences in which the B-tree orders values in the index, you might want to create an additional operator class for the generic B-tree. You define *strategy and support functions* that handle each built-in data type that you want to index with the new sequence. You must register this new operating class in the database with the CREATE OPCLASS statement. For more information, see "Extending an Existing Operator Class" on page 4-11.

Additional Casts

Universal Server provides casts between the built-in data types that it provides. You might want to create additional casts to provide data conversion between an existing data type and an extended data type that you create. If the two data types have different internal formats, you must define a *casting function* to perform the data conversion. You must register the casting function with the CREATE FUNCTION statement and create the cast with the CREATE CAST statement before it can be used. For more information on casts, see Chapter 3, "Creating User-Defined Casts."

Creating New Data Types

The extensible data type system of Universal Server allows you to:

- define new data types.
- define the behavior of these new data types to the database server.
  - Which *operations* are supported on the new data types
  - New *operator class* that supports the new data type and provides new functionality for a secondary access method (an index)
  - Additional *casts* to provide data conversions between the new data types and other data types

Data Types That You Provide

You can define the following kinds of data types in Universal Server:

- A user-defined data type:
  - An opaque data type
  - A distinct data type
- A named row type
- A DataBlade module data type

Together these data types are called *extended data types*. Universal Server stores information about extended data types in the **sysxtdtypes** and **sysxtdtypeauth** system catalog tables.

Figure 1-3 summarizes the user-defined data types that Universal Server provides.

Figure 1-3
User-Defined Data Types of Universal Server

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

This section summarizes the kinds of data types that you can define to extend the Universal Server data type system. For more information, see the chapter on data types in the *Informix Guide to SQL: Reference*.

The Opaque Data Type

The opaque data type is a *fundamental* data type. However, unlike the other fundamental data types (the built-in data types), the internal structure of the opaque data type is not known to the database server. Therefore, when you define an opaque type, you must provide the following information:

- You define the *internal structure* of the opaque data type, which provides the format of the data.

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

  You define operator functions, built-in functions, or end-user routines that handle the opaque type.

- You can extend the *operator class* of a secondary access methods so that its strategy and support functions handle the opaque data type.
- You define *casting* functions to provide the data conversions to and from the opaque type.

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

You register an opaque data type with the CREATE OPAQUE TYPE statement. For more information, see Chapter 5, “Creating an Opaque Data Type,” and Chapter 6, “Writing Support Functions.”

A Distinct Data Type

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

- You choose the source data type, which defines the *internal structure* of the distinct data type.

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

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

  The database server automatically creates explicit casts between the distinct type and its source type. However, because these two data types have the same internal format, this cast does not require a casting function. You can write casting functions to support data conversion between the distinct type and other data types in the database or to support implicit casts between the distinct type and its source type.

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

A Named Row Type

A named row type has a group of one or more components called fields. Each field has a name and a data type. The fields of a row type can be any data type *except* SERIAL and SERIAL8. Some exceptions exist for the fields of a row type that are the TEXT or BYTE data type.

Named row types are identified by their names. When you define a named row type, you provide the following information:

- You define the fields of the named row type, which defines the *internal structure* of the data type.

  The database server uses the functions that are associated with each of the field types to determine how to interact with this internal structure.
- You define the *operations* that are valid on the named row type.

  You define operator functions, built-in functions, or end-user routines that handle the distinct type.
- You can extend the *operator class* of a secondary access methods so that its strategy and support functions handle the named row type.
- You provide the data conversions with *casting* functions that convert from or to the named row type.

You create a named row type with the CREATE ROW TYPE statement. Once you create the named row type, you can use the row-type name as a data type anywhere that other data types are valid. In addition, you can use this name to create type and table inheritance. For more information, see the description of this statement in the *Informix Guide to SQL: Syntax*.

DataBlade Module Data Types

In addition to the data types that you explicitly define, you can obtain new data types from an Informix DataBlade module. A DataBlade module might provide the data type, including any casts, operations, and secondary access methods, for an application-specific purpose. For more information on the DataBlade modules available, consult your Informix sales representative or refer to the user guides for the DataBlade modules.

Ways to Extend With New Data Types

You can extend the following data type information for an existing built-in or complex data type:

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

Extending Operations

To implement operations on an extended data type, you define an *SQL-invoked function* that performs the operation. You can write SQL-invoked functions to extend the following operations for an extended data type:

- Create an operator function to implement a particular operator symbol for an extended data type.
- Create a built-in function to implement one of the built-in tasks on an extended data type.
- Create an end-user routine to provide additional functionality for the extended data type.

  An end-user routine can be either a function (which returns a value) or a procedure (which does not return a value).

You must register each SQL-invoked function in the database with the CREATE FUNCTION (or CREATE PROCEDURE) statement. For more information, see "How Do You Extend an Operation?" on page 2-11.

Extending Operator Classes

The default operator class that Universal Server provides for the built-in secondary access method (a generic B-tree) handles only built-in data types and uses the relational operators (<, >, =, and so on) to order values.

You might want to extend an operator class to support an extended data type for the following reasons:

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

To implement an operator class, you define *strategy and support functions* that handle each extended data type you want to index. You must register this new operating class in the database with the CREATE OPCLASS statement. For more information, see Chapter 4, "Extending an Operator Class."

Creating Casts

Universal Server provides casts between the built-in data types that it provides. You might want to create additional casts to provide data conversion between:

- an existing data type and a new extended data type.
- two different extended data types.

If the two data types have the same internal structure, you create the cast with the CREATE CAST statement. If the two data types have different internal structures, you must define a *casting function* to perform the data conversion. You register the casting function with the CREATE FUNCTION statement and create the cast with the CREATE CAST statement. For more information on casts, see Chapter 3, "Creating User-Defined Casts."

Chapter

Extending an Operation

2

An operation is a task that INFORMIX-Universal Server performs on one or more values. Universal Server provides the following types of SQL-invoked functions that provide operations within SQL statements:

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

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

The property called *routine overloading* allows you to create a user-defined function whose name is already defined in the database but whose parameter list is different. All functions with the same name have the same functionality, but they operate on different data types. Universal Server uses *routine resolution* to determine which function to execute, based on the data types of the arguments for the function. For more information on routine overloading and routine resolution, see Chapter 1 of *Extending INFORMIX-Universal Server: User-Defined Routines*.

What Is an Operation?

Universal Server provides the following types of *operations* to operate on data types:

- An *operator function* implements a particular operator symbol.
- A *built-in function* is a predefined function that Universal Server provides for use in SQL statements.
- An *aggregate function* returns a single value for a set of retrieved rows.
- An *end-user routine* is a user-defined routine that end users can use in SQL statements.

Operators and Operator Functions

Universal Server provides the following types of operators for expressions in SQL statements:

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

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

The versions of the operator functions that Universal Server provides handle the built-in data types. You can write a new version of an operator function to change the functionality of an operator or to provide the operator on a data type that is not built into the database server. If you write a new version of an operator function, follow these rules:

- The name of the operator function must match the name that Figure 2-4 or Figure 2-5 lists. However, the name is case insensitive; the **plus()** function is the same as the **Plus()** function.
- The operator function must handle the correct number of parameters.
- The operator function must return the correct data type, where appropriate.

Arithmetic Operators

Universal Server provides operator functions for the arithmetic operators that Figure 2-4 shows.

Figure 2-4
Operator Functions for Arithmetic Operators

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

When an SQL statement contains an arithmetic operator, the database server automatically invokes the associated operator function. All operator functions in Figure 2-4 *except* **positive()** and **negate()** take two arguments; the **positive()** and **negate()** functions take only one argument. All these operator functions can return any data type.

Text Operators

Universal Server also provides operator functions for the text operators that Figure 2-5 shows.

Figure 2-5
Operator Functions for Text Operators

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

When an SQL statement contains a text operator, the database server automatically invokes the associated operator function. The **like()** and **matches()** functions take two or three arguments and must return a Boolean value. The **concat()** function takes two arguments and can return any data type.

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

Relational Operators

Universal Server provides operator functions for the relational operators that Figure 2-6 shows.

Figure 2-6
Operator Functions for Relational Operators

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

When an SQL statement contains a relational operator, the database server automatically invokes the associated operator function. All operator functions in Figure 2-6 take two arguments and must return a Boolean value. For more information on relational operators, see the description of the Relational Operator segment in the *Informix Guide to SQL: Syntax*.

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

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

  For example, you create the **circle** opaque type to implement a circle. A circle is a spatial object that does not have a single value to compare. However, you can define the relational operators on this data type can that use the value of its area: one **circle** is less than a second **circle** if its area is less than the area of the second. For more information on the **circle** opaque type, see "A Fixed-Length Opaque Type: circle" on page 5-30.

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

  For example, suppose you create a data type, **ScottishName**, that holds Scottish names, and you want to order the data type in a different way than the U.S. English collating sequence. You might want the names *McDonald* and *MacDonald* to appear together on a phone list. You can define relational operators for this data type that equate the strings *Mc* and *Mac*. For more information, see "Changing the Sort Order" on page 4-15.

The relational-operator functions are strategy functions for the built-in secondary access method, a generic B-tree. For more information, see "The Generic B-Tree Index" on page 4-4.

Built-In Functions

Universal Server provides special SQL-invoked functions, called *built-in functions*, that provide some basic mathematical operations. Figure 2-7 shows the built-in functions that you can overload.

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

Built-In Function	Number of Parameters	Parameter Types
abs()	1	real number
mod()	2	integer-number expression, integer-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-number expression
cos(), sin(), tan()	1	numeric expression
acos(), asin(), atan()	1	numeric expression
atan2()	2	numeric expression, numeric expression
hex()	1	integer expression
length(), char_length(), character_length(), octet_length()	1	character string
user()	0	*None*

(1 of 2)

Built-In Function	Number of Parameters	Parameter Types
current()	0	*None*
dbservername(), sitename()	0	*None*
today()	0	*None*
extend()	1	date / time expression

(2 of 2)

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

cardinality()	month()
date()	trim()
day()	weekday()
dbinfo()	year()
mdy()	

OL/Optical

The following table lists the built-in functions for the INFORMIX-OnLine / Optical product that you *cannot* overload.

descr()	volume()
family()	

♦

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 opaque data type. If you write a new version of a built-in function, make sure you follow these rules:

- The name of the built-in function must match the name that Figure 2-7 lists. However, the name is case insensitive; the **abs()** function is the same as the **Abs()** function.
- The built-in function must be one that you can overload.
- The built-in function must handle the correct number of parameters, and these parameters must be the correct data type.
- The built-in function must return the correct data type, where appropriate.

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

Aggregate Functions

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

- COUNT returns the number of rows that satisfy the WHERE clause of a SELECT statement.
- AVG returns the average of all values that the query selects.
- MAX returns the largest value in the specified column or expression.
- MIN returns the smallest value in the specified column or expression.
- SUM returns the sum of all values in the specified column or expression.

For more information on aggregate functions, see the description of the Expression segment in the *Informix Guide to SQL: Syntax*.

End-User Routines

The database server provides special SQL-invoked functions and procedures, called *end-user routines*, that implement tasks you define for end users to use in expressions of SQL statements. These routines provide additional functionality that an end user might need to work with an existing or new data type. An end-user routine can be either of the following:

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

Once you have written the end-user routines, you must register them in the database as you do support functions. Use the CREATE FUNCTION statement to register end-user functions. Use the CREATE PROCEDURE statement to register end-user procedures. You must also use the GRANT statement to grant the Execute privilege to those users who have permission to call the end-user routines. For more information on how to write an end-user routine, see *Extending INFORMIX-Universal Server: User-Defined Routines*.

How Do You Extend an Operation?

The property called *routine overloading* allows you to create a user-defined function whose name is already defined in the database but whose parameter list is different. All functions with the same name have the same functionality, but they operate on different data types. Universal Server uses *routine resolution* to determine which function to execute, based on the data types of the arguments for the function. For more information on routine overloading and routine resolution, see Chapter 1 of *Extending INFORMIX-Universal Server: User-Defined Routines*.

You can extend an operation on:

- existing data types.
- new extended data types.

On Existing Data Types

The data type system of Universal Server defines operations on the data types that it provides. (For a summary of these data types, see "Data Types That Universal Server Provides" on page 1-5.) To provide additional functionality in your database, you can write *end-user routines*, which end users can include in their SQL statements.

Tip: *You cannot redefine the operator functions, built-in functions, or aggregate functions on the existing data types.*

You can write these end-user routines in either of the following languages:

- Stored Procedure Language (SPL)
- An external language that Universal Server supports (such as C)

Once you have written the end-user routines, register them in the database. Use the CREATE FUNCTION statement to register end-user functions and the CREATE PROCEDURE statement to register end-user procedures. You must also use the GRANT statement to grant the Execute privilege to those users who have permission to call the end-user routines.

For more information on how to write an end-user routine, see *Extending INFORMIX-Universal Server: User-Defined Routines*.

On Extended Data Types

The extensible data type system of Universal Server allows you to define operations on extended data types. (For a summary of these data types, see "Data Types That You Provide" on page 1-12.) To provide additional functionality in your database, you can create user-defined functions to support the following types of operations on an extended data type:

- Operator functions to provide support for operator symbols (such as +, -, /, and *) on an extended data type

  Follow the rules that "Operators and Operator Functions" on page 2-4 describes to name the operator function and to choose the data types of its arguments and its return values.

- Built-in functions such as **cos()** and **abs()** that handle extended data types as their arguments

  Figure 2-7 on page 2-8 lists the built-in functions that you can overload.

- End-user routines that provide additional functionality on the extended data type

  For more information on how to write an end-user routine, see "End-User Routines" on page 2-10.

Tip: *You cannot redefine the aggregate functions to handle extended data types.*

For more information on how to provide these operations for an opaque data type, see "Creating SQL-Invoked Functions" on page 5-19.

Chapter

Creating User-Defined Casts

3

A *cast* is a mechanism that converts a value from one data type to another data type. Casts allow you to make comparisons between values of different data types or substitute a value of one data type when another data type is expected.

To convert a value of one data type to another data type, the database server must have a cast that describes how to perform this conversion. Universal Server supports two kinds of cast:

- System-defined casts

  A *system-defined cast* is a cast that is built into the database server. A system-defined cast performs an automatic conversion between two built-in data types.

- User-defined casts

  A *user-defined cast* is a cast that you define to provide data conversion between two non-built-in data types.

For more information on system-defined casts, see the chapter on data types in the *Informix Guide to SQL: Reference*. This chapter describes how to create user-defined casts for data types in a database.

What Is a User-Defined Cast?

Create a user-defined cast to perform data conversion from one data type to another. You create a user-defined cast with the CREATE CAST statement, which registers the cast in the database.

The database server provides casts between most of the built-in data types. You can create user-defined casts to perform conversions between most data types, including opaque types, distinct types, row types, and built-in types. User-defined casts are generally used to provide type conversions for the following extended data types:

- Opaque data types

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

  For information about how to create and register casts for opaque data types, see "Creating the Casts" on page 5-17.

- Distinct data types

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

  For more information and examples that show how you can create and use casts for distinct types, see the chapter on casting in the *Informix Guide to SQL: Tutorial*.

- Named row types

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

  For information about casting between named row types and unnamed row types, see the chapter on casting in the *Informix Guide to SQL: Tutorial*.

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

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

How Do You Create a User-Defined Cast?

The CREATE CAST statement registers a cast in the current database. It registers a cast in the **syscasts** system catalog table. A cast is owned by the person who registers it with CREATE CAST.

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

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

- The kind of user-defined cast to create

  The CREATE CAST statement specifies whether this cast is implicit or explicit.
- The cast mechanism that the database server is to use to perform the data conversion

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

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

Choosing the Kind of User-Defined Cast

Universal Server supports two kinds of user-defined casts:

- Implicit cast

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

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

Implicit Cast

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

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

Conversion of one data type to another can involve loss of data. Be careful of creating implicit casts for such conversions. The end user has no ability to control when the database server invokes an implicit cast and therefore cannot avoid the loss of data that is inherent to such a conversion.

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

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

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

Explicit Cast

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

- The CAST AS keywords

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

  ```
  WHERE col1 > (CAST percent AS integer)
  ```

- The :: cast operator

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

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

For more information on how to invoke explicit casts, see the *Informix Guide to SQL: Tutorial*.

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

To create an explicit cast, specify the EXPLICIT keyword of the CREATE CAST statement. The following CREATE CAST statement creates an explicit cast from the **percent** data type to the INTEGER data type:

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

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

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

Choosing the Cast Mechanism

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

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

A Straight Cast

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

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

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

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

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

For the syntax of the CREATE CAST statement, see the *Informix Guide to SQL: Syntax*.

A Casting Function

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

To create a cast that has a casting function, follow these steps:

1. Write the casting function.

   The casting function takes the source data type as its argument and returns the target data type. You can write casting functions in SPL (an SPL function) or in C (an external function).

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

2. Register the casting function with the CREATE FUNCTION statement.

   If you create an SPL function, the CREATE FUNCTION statement contains the actual SPL statements of the casting function and registers the function. If you create an external function, CREATE FUNCTION specifies the name of the shared library that contains the compiled code and just registers the function.

3. Register the cast with the CREATE CAST statement.

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

For example, suppose you want to compare values of two opaque data types, **int_type** and **float_type**. The CREATE FUNCTION statement in Figure 3-1 creates and registers an SPL function, **int_to_float()** that takes an **int_type** value as an argument and returns a value of type **float_type**.

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

Figure 3-1
An SPL Function as a Casting Function from int_type to float_type

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

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

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

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

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

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

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

Once you create this casting function, you use the CREATE CAST statement to register the function as a cast. The function cannot be used as a cast until it is registered with the CREATE CAST statement. The CREATE CAST statement in Figure 3-2 creates an explicit cast that uses the **int_to_float()** function as its casting function.

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

Figure 3-2
An Explicit Cast from int_type to a float_type

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

Defining the Direction of the Cast

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

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

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

To provide data conversion between two data types, you must define a cast for each direction of the conversion. For example, the explicit cast in Figure 3-2 enables the database server to convert from the **int_type** opaque type to the **float_type** opaque type. Therefore, the end-user can perform the following cast in an INSERT statement to convert an **int_type** value, **it_val**, for a **float_type** column, **ft_col**:

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

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

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

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

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

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

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

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

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

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

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

The CREATE CAST statement in Figure 3-4 creates an explicit cast that uses the **int_to_float()** function as its casting function.

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

Figure 3-4
An Explicit Cast From float_type To int_type

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

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

Together, the explicit casts in Figure 3-2 on page 3-10 and Figure 3-4 enable the database server to convert between the **float_type** and **int_type** opaque data types. Each explicit cast provides a casting function that performs one direction of the conversion.

Dropping a Cast

The DROP CAST statement removes the definition for a cast from the database. Universal Server removes the class definition from the **syscasts** system catalog table. You must be the owner of the cast or the DBA to drop its definition from the database.

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

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

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

Dropping a cast has no effect on the function associated with the cast. Use the DROP FUNCTION statement to remove a function from the database.

Chapter

Extending an Operator Class

4

This chapter describes how to extend an operator class. It provides the following information:

- A brief description of secondary access methods and operator classes, with some factors to consider when you choose a secondary access method
- Steps for how to extend an existing operator class of an existing secondary access method to support a new user-defined data type
- Steps for how to create a new operator class for a secondary access method to provide an additional sort order
- A description of how to drop an existing operator class

Using an Operator Class

An *operator class* is the set of functions that is associated with a secondary access method. For most situations, use the default operators that are defined for a secondary access method. However, when you want to order the data in a different sequence or provide index support for a user-defined data type, you must extend an operator class.

This section provides a brief introduction to a secondary access method and an operator class. For a more detailed discussion of these topics, see the *INFORMIX-Universal Server Performance Guide*. For information on how to extend an operator class, see "Extending an Existing Operator Class" on page 4-11 and "Creating an Operator Class" on page 4-19.

What Is a Secondary Access Method?

The *secondary access method*, often called an *index*, is a set of user-defined functions that build, access, and manipulate an index structure. These functions encapsulate index operations, such as how to scan, insert, delete, or update nodes in an index. A secondary access method describes how to access the data in an index that is built on a column (column index) or on a user-defined function (functional index). Typically, a secondary access method speeds up the retrieval of a type of data in particular ways.

Universal Server provides definitions for the following secondary access methods in the system catalog tables of each database:

- A generic B-tree
- An R-tree

DataBlade modules can provide additional secondary access methods for use with user-defined data types. For more information about secondary access methods of DataBlade modules, refer to the user guide for each DataBlade module.

The Generic B-Tree Index

Universal Server provides the *generic B-tree index* for columns in database tables. In traditional relational database systems, the B-tree access method handles only built-in data types and therefore can compare only two keys of built-in data types. To support user-defined data types, Universal Server provides an extended version of a B-tree, the generic B-tree access method.

Tip: *A B-tree index is useful for a query that retrieves a range of data values. For more information on when you should create a B-tree index on a column, see the "INFORMIX-Universal Server Performance Guide."*

Universal Server uses the generic B-tree as the built-in secondary access method. This secondary access method is registered in the **sysams** system catalog table with the name **btree**. When you use the CREATE INDEX statement (without the USING clause) to create an index, Universal Server creates a generic B-tree index. The CREATE INDEX statement in Figure 4-1 creates a B-tree index on the **zipcode** column of the **customer** table.

```
CREATE INDEX zip_ix ON customer (zipcode)
```

Figure 4-1
Creating a B-Tree Index

For more information, see the CREATE INDEX statement in the *Informix Guide to SQL: Syntax*. For more information on the structure of a B-tree index, and on estimating the size of an B-tree index, refer to the *INFORMIX-Universal Server Performance Guide*.

The R-Tree Index

Universal Server can support the *R-tree index* for columns that contain spatial data such as maps and diagrams.

Tip: *An R-tree index is most beneficial when queries commonly look for objects that are within other objects or for an object that contains one or more objects. For more information on when you should create an R-tree index, see the "INFORMIX-Universal Server Performance Guide."*

Universal Server automatically defines the R-tree access method in the **sysams** system catalog table with the name **rtree**. Universal Server databases define the default **rtree** operator class in the system catalog tables but do *not* provide the functions to implement this secondary access method.

Important: *To use an R-tree index, install a spatial DataBlade module such as the Spatial DataBlade module, Geodetic DataBlade, or any other third-party DataBlade module that implements an R-tree index. For more information on the spatial DataBlade modules, consult the appropriate DataBlade module user guide.*

To create an R-tree index, include the USING clause of the CREATE INDEX statement. For example, if you have one of the spatial DataBlade modules installed, the CREATE INDEX statement in Figure 4-2 creates an R-tree index on the **picture** column of the **photos** table.

```
CREATE INDEX pic_rix ON photos (picture)
USING rtree;
```

Figure 4-2
Creating an Index of a User-Defined Secondary Access Method

For more information, see the CREATE INDEX statement in the *Informix Guide to SQL: Syntax*. For more information on the structure of an R-tree index and for information on estimating the size of an R-tree index, refer to *INFORMIX-Universal Server Performance Guide*.

Other User-Defined Secondary Access Methods

A DataBlade module can provide a user-defined data type to handle a particular type of data. The module might also provide a new secondary access method (index) for the new data type that it defines. For example, the Excalibur Text DataBlade provides an index to search text data. For more information, refer to the *Excalibur Text DataBlade User Guide*. For more information on the types of data and functions each DataBlade module provides, refer to the user guide for each DataBlade module. For more information on how to determine which secondary access methods exist in your database, see the *INFORMIX-Universal Server Performance Guide*.

What Is an Operator Class?

An *operator class* is a group of functions that allow the secondary access method to store and search for values of a particular data type. The query optimizer uses an operator class to determine if an index can process the query with the least cost. For more information on the query optimizer, see the *INFORMIX-Universal Server Performance Guide*.

The operator-class functions fall into the following categories:

- Strategy functions

  Universal Server uses the *strategy functions* of a secondary access method to help the query optimizer determine whether a specific index is applicable to a specific operation on a data type. The strategy functions are the operators that can appear in the filter of an SQL statement.

- Support functions

  Universal Server uses the *support functions* of a secondary access method to build and access the index. These functions are not called directly by end users. When an operator in the filter of a query matches one of the strategy functions, the secondary access method uses the support functions to traverse the index and obtain the results.

Each secondary access method has a *default operator class* that is associated with it. By default, the CREATE INDEX statement associates the default operator class with an index.

Universal Server stores information about operator classes in the **sysopclasses** system catalog table. The database server defines the following operator classes in the system catalog tables of every database:

- The generic B-tree operator class, **btree_ops**
- The R-tree operator class, **rtree_ops**

The Generic B-Tree Operator Class

The built-in secondary access method, the generic B-tree, has a single operator class defined in the **sysopclasses** system catalog table. This operator class, called **btree_ops**, is the default operator class for the **btree** secondary access method.

Universal Server uses the **btree_ops** operator class to specify:

- the strategy functions to tell the optimizer which filters in a query can use a B-tree index.
- the support function to build and search the B-tree index.

The CREATE INDEX statement in Figure 4-1 on page 4-5 shows how to create a B-tree index whose column uses the **btree_ops** operator class. This CREATE INDEX statement does not need to specify the **btree_ops** operator class because **btree_ops** is the default operator class for the **btree** access method.

For more information on the **btree** secondary access method, see "The Generic B-Tree Index" on page 4-4.

The B-Tree Strategy Functions

The **btree_ops** operator class defines the following strategy functions for the **btree** access method:

- **lessthan** (<)
- **lessthanorequal** (<=)
- **equal** (=)
- **greaterthanorequal** (>=)
- **greaterthan** (>)

These strategy functions are all *operator functions*. That is, each function is associated with an operator symbol; in this case, with a relational-operator symbol. For more information on relational-operator functions, see page 2-6.

The B-Tree Support Function

The **btree_ops** operator class has one support function, a comparison function called **compare()**. The **compare()** function is a user-defined function that returns an integer value to indicate whether its first argument is equal to, less than, or greater than its second argument, as follows:

- A value of 0 when the first argument is *equal to* the second argument
- A value less than 0 when the first argument is *less than* the second argument
- A value greater than 0 when the first argument is *greater than* the second argument

The B-tree secondary access method uses the **compare()** function to traverse the nodes of the generic B-tree index. To search for data values in a generic B-tree index, the secondary access method uses the **compare()** function to compare the key value in the query to the key value in an index node. The result of the comparison determines if the secondary access method needs to search the next-lower level of the index or if the key resides in the current node.

The generic B-tree access method also uses the **compare()** function to perform the following tasks for generic B-tree indexes:

- Sort the keys before building the index
- Determine the linear ordering of keys in a generic B-tree index
- Evaluate the relational operators

Universal Server uses the **compare()** function to evaluate comparisons in the SELECT statement. To provide support for these comparisons for opaque data types, you must write the **compare()** function. For more information, see "Comparing Data" on page 5-22.

The R-Tree Index Operator Class

The R-tree secondary access method has an operator class defined in the **sysopclasses** system catalog table. This operator class, called **rtree_ops**, is the default operator class for the **rtree** secondary access method. Universal Server databases define the default R-tree operator class in the system catalog tables but do *not* provide the operator-class functions to implement this operator class.

Important: *To use an R-tree index, install a spatial DataBlade module such as the Spatial DataBlade module, Geodetic DataBlade, or any other third-party DataBlade module that implements an R-tree index. These spatial DataBlade modules might use the* ***rtree_ops*** *operator class, or they might provide their own operator classes for an R-tree index. For more information on the spatial DataBlade modules, consult the appropriate DataBlade module user guide.*

Universal Server uses the **rtree_ops** operator class to specify:

- the strategy functions to tell the optimizer which filters in a query can use an R-tree index.
- the support functions to build and search the R-tree index.

The CREATE INDEX statement in Figure 4-2 on page 4-6 shows how to create an R-tree index whose column uses the **rtree_ops** operator class. This CREATE INDEX statement does not need to specify the **rtree_ops** operator class because **rtree_ops** is the default operator class for the **rtree** access method. This statement assumes that you have one of the spatial DataBlade modules installed.

For more information on the **rtree** secondary access method, see "The R-Tree Index" on page 4-5.

The R-Tree Strategy Functions

The **rtree_ops** operator class defines the following strategy functions for the **rtree** secondary access method:

- **Overlap()** returns a Boolean value to indicate whether the two regions overlap.
- **Equal()** returns a Boolean value to indicate whether the two regions are the same.
- **Contains()** returns a Boolean value to indicate whether one region contains another.
- **Within()** returns a Boolean value to indicate whether one region is within (contained by) another.

When the query optimizer examines a query that contains a column, it checks to see if this column has an R-tree index defined on it. If such an index exists *and* if the query contains one of the strategy functions that the **rtree_ops** operator class supports, the optimizer can choose an R-tree index to execute the query.

The R-Tree Support Functions

The **rtree_ops** operator class contains the following support functions:

- **Union()** determines the union of two regions and returns the resulting region.
- **Size()** returns the size of a region as a floating-point number.
- **Inter()** determines the intersection of two regions and returns the resulting region.

The **rtree** secondary access method, if it is defined in the database, uses these support functions to build and access an R-tree index.

Extending an Existing Operator Class

The operator-class functions of an operator class can be defined only for the data types that are currently defined in a database. When you create a new user-defined data type, you must determine whether you need to create operator-class functions that are defined for this new data type. The creation of new operator-class functions that have the same names as the existing operator class functions is the most common way to extend an existing operator class.

To extend the functionality of an operator-class function, write a function that has the same name and return value. You provide parameters for the new data type and write the function to handle the new parameters. *Routine overloading* allows you to create many functions, all having the same name but each having a different parameter list. Universal Server then uses *routine resolution* to determine which of the overloaded functions to use based on the data type of the value. For more information on routine overloading and routine resolution, see *Extending INFORMIX-Universal Server: User-Defined Routines*.

When you create a user-defined data type, you must perform the following steps:

1. Decide which of the secondary access methods can support the user-defined data type.

   For descriptions of the B-tree and R-tree secondary access methods, see "What Is a Secondary Access Method?" on page 4-4.

2. Extend the operator classes of the chosen secondary access method(s).

   To allow end users to use the user-defined type with the operators that are associated with the secondary access method, write new strategy functions that handle the new data type. You also need to write new support functions to handle this new data type.

Universal Server provides the B-tree and R-tree secondary access methods for indexes. The following sections describe how to extend the default operator classes of each of these secondary access methods.

Extending the btree_ops Operator Class

Before the database server can support generic B-tree indexes on a user-defined data type, the operator classes associated with the B-tree secondary access method must be able to handle that data type. The default operator class for the generic B-tree secondary access method is called **btree_ops**. Initially, the operator-class functions (strategy and support functions) of the **btree_ops** operator class handle the built-in data types. When you define a new data type, you must also extend these operator-class functions to handle the data type.

Important: *You cannot extend the **btree_ops** operator class for the built-in data types.*

How to Extend btree_ops

Once you determine how you want to implement the relational operators for a user-defined data type, you can extend the **btree_ops** operator class so that the query optimizer can consider use of a B-tree index for a query that contains a relational operator.

To extend the default operator class for a generic B-tree index

1. Write functions for the B-tree strategy functions that accept the user-defined data type in their parameter list.

   You can write strategy functions as C functions (external functions) or as SPL functions. The relational-operator functions serve as the strategy functions for the **btree_ops** operator class. If you have already defined these operator functions for the user-defined data type, the generic B-tree index uses them as its strategy functions.

   For more information on how to implement relational operators for an opaque data type, see “Relational Operators for Opaque Data Types” on page 5-23. For more information on these strategy functions, see “The B-Tree Strategy Functions” on page 4-8.

2. Register the strategy functions in the database with the CREATE FUNCTION statement.

   If you have already registered the relational-operator functions, you do not need to reregister them as strategy functions.

3. Write a C function for the B-tree support function, **compare()**, that accepts the user-defined data type in its parameter list.

   The **compare()** function also provides support for a user-defined data type in comparison operations in a SELECT statement (such as the ORDER BY clause or the BETWEEN operator). If you have already defined this comparison function for the user-defined data type, the generic B-tree index uses it as its support function.

   For more information on how to implement a comparison function for an opaque data type, see "Conditions for Opaque Data Types" on page 5-22. For more information on this support function, see "The B-Tree Support Function" on page 4-8.

4. Register the support function in the database with the CREATE FUNCTION statement.

   For opaque data types, you might have already defined this function to provide support for the comparison operations in a SELECT statement (such as the ORDER BY clause or the BETWEEN operator) on your opaque type. For more information, see "Comparing Data" on page 5-22.

5. Use the CREATE INDEX statement to create a B-tree index on the column of the table that contains the user-defined data type.

   The CREATE INDEX statement does not need the USING clause because you have extended the default operating class for the default index type, a generic B-tree index, to support your user-defined data type.

The query optimizer can now consider use of this generic B-tree index to execute queries efficiently. For more information on the performance aspects of column indexes, see the *INFORMIX-Universal Server Performance Guide*. For an example of how to extend the generic B-tree index for an opaque-type column, see "A Fixed-Length Opaque Type: circle" on page 5-30.

Important: *When Universal Server uses a generic B-tree index to process an ORDER BY clause in a SELECT statement, the database server uses the **btree_ops** support function called **compare()**. However, the optimizer does not use the B-tree index to perform ORDER BY if the index does not use the **btree_ops** operator class.*

These steps extend the default operator class of the generic B-tree index. You could also define a new operator class to provide another order sequence. For more information, see "Defining a New B-Tree Operator Class" on page 4-20.

Reasons to Extend btree_ops

The strategy functions of **btree_ops** are the relational operations that end users can use in expressions. (For a list of the relational operators, see "The B-Tree Strategy Functions" on page 4-8.) By default, a generic B-tree index can handle only the built-in data types. When you write relational-operator functions that handle a new user-defined data type, you extend the generic B-tree so that it can handle the user-defined data type in a column or a user-defined function. To be able to create B-tree indexes on columns or functions of the new data type, you must write new relational-operator functions that can handle the new data type.

In the relational-operator functions, you determine the following behavior of a B-tree index:

- What single value does the B-tree secondary access method order in the index?

  For a particular user-defined data type, the relational-operator functions must compare two values of this data type for the data type to be stored in the B-tree index.
- In what order does the B-tree index sort the values?

  For a particular user-defined data type, the relational-operator functions must determine what constitutes an ordered sequence of the values.

Generating a Single Value for a New Data Type

A B-tree index is intended to index one-dimensional objects. It uses the relational-operator functions to compare two one-dimensional values. It then uses the relationship between these values to determine how to traverse the tree and in which node to store a value.

By default, the relational-operator functions handle the built-in data types. (For more information on the built-in data types, see the chapter on data types in the *Informix Guide to SQL: Reference*.) The built-in data types contain one-dimensional values. For example, the INTEGER data type holds a single integer value; the CHAR data type holds a single character string; the DATE data type holds a single date value. The values of all these data types can be ordered linearly (in one dimension). The relational-operator functions can compare these values to determine their linear ordering.

When you create a new user-defined data type, you must ensure that the relational-operator functions can compare two values of the user-defined data type. Otherwise, the comparison cannot occur, and the user-defined data type cannot be used in a B-tree index.

For example, suppose you create the **circle** opaque type to implement a circle. A circle is a spatial object that might be indexed best with a user-defined secondary access method such as an R-tree that handles two-dimensional objects. However, this data type can be used in a B-tree index if it defines the relational operators on the value of its area: one **circle** is less than a second **circle** if its area is less than the area of the second. For more information on the **circle** opaque type, see "A Fixed-Length Opaque Type: circle" on page 5-30.

Changing the Sort Order

A generic B-tree uses the relational operators to determine which value is less than another. These operators use lexicographical sequence (numeric order or numbers, alphabetic order for characters, chronological order for dates and times) for the values they order.

GLS

The relational-operator functions use the code-set order for character data types (CHAR, VARCHAR, and LVARCHAR) and a localized order for the NCHAR and NVARCHAR data types. When you use the default locale, U.S. English, code-set order and localized order are those of the ISO 8895-1 code set. When you use a nondefault locale, these two orders might be different. For more information on locales, see the *Guide to GLS Functionality*. ♦

For some user-defined data types, the relational operators in the default B-tree operator class might not achieve the order that you want. You define the relational-operator functions for a particular user-defined type so that they change from lexicographical sequence to some other sequence.

Tip: *When you extend an operator class, you can change the sort order for a user-defined data type. To provide an alternative sort order for all data types that the B-tree handles, you must define a new operator class. For more information, see "Defining a New B-Tree Operator Class" on page 4-20.*

For example, suppose you create an opaque data type, **ScottishName**, that holds Scottish names, and you want to order the data type in a different way than the U.S. English collating sequence. You might want the names *McDonald* and *MacDonald* to appear together on a phone list. This data type can use a B-tree index because it defines the relational operators that equate the strings *Mc* and *Mac*.

To order the data type in this way, write the relational-operator functions so that they implement this new order. For the strings *Mc* and *Mac* be equal, you must define the relational-operator functions that:

- accept the opaque data type, **ScottishName**, in the parameter list.
- contain code that equates *Mc* and *Mac*.

The following steps use the steps in "How to Extend btree_ops" on page 4-12 to extend the **btree_ops** operator class to support the **ScottishName** data type:

1. Write C functions for the strategy functions that handle the **ScottishName** data type: **lessthan()**, **lessthanorequal()**, **equal()**, **greaterthan()**, **greaterthanorequal()**.

   Compile these functions and store them in the **btree.so** shared library and put this shared library in the **/apps/lib** directory.

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

   The following CREATE FUNCTION statements register the strategy functions that handle the **ScottishName** opaque data type:

```
CREATE FUNCTION lessthan (ScottishName, ScottishName)
    RETURNS boolean
    EXTERNAL NAME
        '/apps/lib/btree.so(scottish_lessthan)"
    LANGUAGE C NOT VARIANT;
```

```
CREATE FUNCTION lessthanorequal(ScottishName,
      ScottishName)
   RETURNS boolean
   EXTERNAL NAME
      '/apps/lib/btree.so(scottish_lessthanorequal)'
   LANGUAGE C NOT VARIANT;

CREATE FUNCTION equal(ScottishName, ScottishName)
   RETURNS boolean
   EXTERNAL NAME
      '/apps/lib/btree.so(scottish_equal)'
   LANGUAGE C NOT VARIANT;

CREATE FUNCTION
   greaterthanorequal(ScottishName, ScottishName)
   RETURNS boolean
   EXTERNAL NAME
   '/apps/lib/btree.so(scottish_greaterthanorequal)'
   LANGUAGE C NOT VARIANT;

CREATE FUNCTION greaterthan(ScottishName,
      ScottishName)
   RETURNS boolean
   EXTERNAL NAME
   '/apps/lib/btree.so(scottish_greaterthan)'
   LANGUAGE C NOT VARIANT;
```

3. Write the C function for the **compare()** support function that handles the **ScottishName** data type.

   Compile this function and store it in the **btree.so** shared library.

4. Register the new **compare()** support function with the CREATE FUNCTION statement.

```
CREATE FUNCTION compare(ScottishName, ScottishName)
   RETURNS integer
   EXTERNAL NAME
      '/apps/lib/btree.so(scottish_compare)'
   LANGUAGE C NOT VARIANT
```

5. You can now create a B-tree index on a **ScottishName** column.

```
CREATE TABLE scot_cust
(
   cust_id integer,
   cust_name ScottishName
   ...
);
CREATE INDEX cname_ix
   ON scot_cust (cust_name);
```

The optimizer can now choose whether to use the **cname_ix** index to evaluate the following query:

```
SELECT * FROM scot_cust
WHERE cust_name = 'McDonald'::ScottishName
```

Extending the rtree_ops Operator Class

To be able to use an R-tree index, you must first install a spatial DataBlade module such as the Spatial DataBlade module, Geodetic DataBlade, or any other third-party DataBlade module that implements an R-tree index. These spatial DataBlade modules either:

- extend the default R-tree operator class, **rtree_ops**.
- define a new R-tree operator class.

Important: *These spatial DataBlade modules implement the R-tree operator-class functions. If you do not install one of these DataBlade modules, you must create versions of the R-tree strategy and support functions that support your user-defined data type.*

If an existing R-tree operator class does not support your user-defined data type, you can try to extend the operator-class functions to handle this data type. The ability to extend this operator class depends on whether your user-defined data type can logically be indexed by an R-tree index.

For more information on the R-tree secondary access method, see page 4-9.

Creating an Operator Class

For most indexing situations, the operators in the default operator class of a secondary access method provide adequate support. However, when you want to order the data in a different sequence than the default operator class provides, you can define a new operator class for the secondary access method.

The CREATE OPCLASS statement creates an operator class. It provides the following information about the operator class to the database server:

- The name of the operator class
- The name of the secondary access method with which to associate the functions of the operator class
- The names and, optionally, the parameters of the strategy functions
- The names of the support functions

Universal Server stores this information in the **sysopclasses** system catalog table. You must have the Resource privilege for the database or be the DBA to create an operator class.

Universal Server provides the default operator class, **btree_ops**, for the generic B-tree access method. The following CREATE OPCLASS statement creates the **btree_ops** operator class:

```
CREATE OPCLASS btree_ops FOR btree
    STRATEGIES (lessthan, lessthanorequal, equal,
        greaterorequal, greaterthan)
    SUPPORT(compare);
```

For more information, see "The Generic B-Tree Operator Class" on page 4-7.

You might want to create a new operator class for:

- the generic B-tree secondary access method.

  A new operator class can provide an additional sort order for all data types that the B-tree index can handle.
- any user-defined secondary access methods.

  A new operator class can provide additional functionality to the strategy functions of the operator class.

Defining a New B-Tree Operator Class

To traverse the index structure, the generic B-tree index uses the sequence that the relational operators define. By default, a B-tree uses the lexicographical sequence of data because the default operator class, **btree_ops**, contains the relational-operator functions. (For more information on this sequence, see "Changing the Sort Order" on page 4-15.) To have a generic B-tree use a different sequence for its index values, you can create a new operator class for the **btree** secondary access method. You can then specify the new operator class when you define an index on that data type.

When you create a new operator class for the generic B-tree index, you provide an additional sequence for organizing data in a B-tree. When you create the B-tree index, you can specify the sequence that you want a column (or user-defined function) in the index to have.

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

1. Write C functions for five new B-tree strategy functions that accept the appropriate data type in their parameter list.

   The B-tree secondary access method expects five strategy functions; therefore, any new operator class must define exactly five. The parameter data types can be built-in or user-defined data types. However, each function *must* return a Boolean value. For more information on strategy functions, see page 4-8.

2. Register the new strategy functions in the database with the CREATE FUNCTION statement.

   You must register the set of strategy functions for each data type on which you are supporting the operator class.

3. Write C functions for the new B-tree support function that accepts the appropriate data type in its parameter list.

   The B-tree secondary access method expects one support function; therefore, any new operator class must define only one. The parameter data types can be built-in or user-defined data types. However, the return type must be integer. For more information on support functions, see page 4-8.

4. Register the new support function in the database with the CREATE FUNCTION statement.

   You must register a support function for each data type on which you are supporting the operator class.

5. Create the new operator class for the B-tree secondary access method, **btree**.

   When you create an operator class, specify the following in the CREATE OPCLASS statement:

   - ❑ After the OPCLASS keyword, the name of the new operator class
   - ❑ In the FOR clause, **btree** as the name of the secondary access method with which to associate the operator class
   - ❑ In the STRATEGIES clause, a parenthetical list of the names of the strategy functions for the operator class

     You registered these functions in step 2. They must be listed in the order that the B-tree secondary access method expects: the first function is the replacement for **lessthan()**, the second for **lessthanorequal()**, and so on.
   - ❑ In the SUPPORT clause, the name of the support function to use to search the index.

     You registered this function in step 4. It is the replacement for the **compare()** function.

   For more information on how to use the CREATE OPCLASS statement, refer to the *Informix Guide to SQL: Syntax*.

6. To create a generic B-tree index that uses the new operator class, specify the name of this operator class in the CREATE INDEX statement.

   By default, the CREATE INDEX statement uses the default operator class for columns and functions in a generic B-tree index. To use a new operator class instead, specify the new operator class name after the column or function name.

These steps create the new operator class of the generic B-tree index. You could also extend the default operator class to provide support for new data types. For more information, see "Extending the btree_ops Operator Class" on page 4-12.

As an example, suppose you want to define a new ordering for integers. The lexicographical sequence of the default B-tree operator class orders integers numerically: -4 < -3 < -2 < -1 < 0 < 1 < 2 < 3. Instead, you might want the numbers -4, 2, -1, -3 to appear in order of absolute value:

```
-1, 2, -3, -4
```

To obtain the absolute-value order, you must define external functions that treat negative integers as positive integers. The following steps create a new operator class called **abs_btree_ops** with strategy and support functions that provide the absolute-value order:

1. Write C functions for the new strategy functions: **abs_lessthan()**, **abs_lessthanorequal()**, **abs_equal()**, **abs_greater()**, and **abs_greaterthanorequal()**.

   Compile these functions and store them in the **absbtree.so** shared library.

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

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

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

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

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

   CREATE FUNCTION abs_gte(integer, integer)
      RETURNS boolean
      EXTERNAL NAME
         '/lib/btree1.so(abs_greaterthanorequal)'
      LANGUAGE C NOT VARIANT;
   ```

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

3. Write the C function for the new support function: **abs_compare()**.

   Compile this function and store it in the **absbtree.so** shared library.

4. Register the new support function with the CREATE FUNCTION statement.

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

   ```
   CREATE FUNCTION abs_cmp(integer, integer)
       RETURNS integer
       EXTERNAL NAME '/lib/absbtree.so(abs_compare)'
       LANGUAGE C NOT VARIANT;
   ```

5. Create the new **abs_btree_ops** operator class for the B-tree secondary access method.

   ```
   CREATE OPCLASS abs_btree_ops FOR btree
       STRATEGIES (abs_lt, abs_lte, abs_eq, abs_gte,
           abs_gt)
       SUPPORT (abs_cmp);
   ```

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

   ```
   CREATE TABLE cust_tab
   (
       cust_name varchar(20),
       cust_num integer
       ...
   );
   CREATE INDEX c_num1_ix
       ON cust_tab (cust_num abs_btree_ops);
   ```

The **c_num1_ix** index uses the new operator class, **abs_btree_ops**, for the **cust_num** column. An end user can now use the absolute value functions in their SQL statements, as in the following example statement:

```
SELECT * FROM cust_tab WHERE abs_lt(cust_num, 7)
```

In addition, because the **abs_lt()** function is part of an operator class, the query optimizer can use the **c_num1_ix** index when it looks for all **cust_tab** rows with **cust_num** values between -7 and 7. A **cust_num** value of -8 does *not* satisfy this query.

The default operator class is still available for indexes. The following CREATE INDEX statement defines a second index on the **cust_num** column:

```
CREATE INDEX c_num2_ix ON cust_tab (cust_num);
```

The **c_num2_ix** index uses the default operator class, **btree_ops**, for the **cust_num** column. The following query uses the operator function for the default *less than* (<) operator:

```
SELECT * FROM cust_tab WHERE lessthan(cust_num, 7)
```

The query optimizer can use the **c_num2_ix** index when it looks for all **cust_tab** rows with **cust_num** values less than 7. A **cust_num** value of -8 *does* satisfy this query.

Defining an Operator Class for Other Secondary Access Methods

You can also define operator classes for user-defined secondary access methods. A *user-defined secondary access method* is one that a database developer has defined to implement a particular type of index. These access methods might have been defined in the database by a DataBlade module.

Tip: *You can examine the **sysams** system catalog table to determine which secondary access methods your database defines. For more information on how to determine the secondary access methods available in your database, see the "INFORMIX-Universal Server Performance Guide."*

You perform the same steps to define an operator class on a user-defined secondary access method as you use to define an operator class on the generic B-tree index (see page 4-20). The only difference is that to create the index, you must specify the name of the user-defined secondary access method in the USING clause of the CREATE INDEX statement.

For example, suppose that your system implements the functions of the **rtree** secondary access method.

Important: *To use an R-tree index, install a spatial DataBlade module such as the Spatial DataBlade module, Geodetic DataBlade, or any other third-party DataBlade module that implements an R-tree index. These spatial DataBlade modules might use the **rtree_ops** operator class, or they might provide their own operator classes for an R-tree index. For more information on the spatial DataBlade modules, consult the appropriate DataBlade module user guide.*

You can create a new operator class, **rtree2_ops**, for the **rtree** secondary access method. The new operator class must have the same number of strategy and support functions as the default operator class for **rtree**, **rtree_ops**. For more information on the **rtree_ops** operator class, see page 4-9.

The following steps create the **rtree2_ops** operator class:

1. Write C functions for the new strategy functions: **rtree2_overlap()**, **rtree2_equal()**, **rtree2_contains()**, and **rtree2_within()**.

   These functions must have the same return values as their counterparts in the **rtree_ops** operator class. Compile these functions, store them in the **rtree2.so** shared library, and put this shared library in the **/apps/lib** directory.

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

   The following CREATE FUNCTION statements register the four strategy functions that handle the **polygon** opaque data type:

```
CREATE FUNCTION Overlap2(polygon, polygon)
    RETURNS boolean
    EXTERNAL NAME
        '/apps/lib/rtree2.so(rtree2_overlap)'
    LANGUAGE C NOT VARIANT;

CREATE FUNCTION Equal2(polygon, polygon)
    RETURNS boolean
    EXTERNAL NAME
        '/apps/lib/rtree2.so(rtree2_equal)'
    LANGUAGE C NOT VARIANT;

CREATE FUNCTION Contains2(polygon, polygon)
    RETURNS boolean
    EXTERNAL NAME
        '/apps/lib/rtree2.so(rtree2_contains)'
    LANGUAGE C NOT VARIANT;

CREATE FUNCTION Within2(polygon, polygon)
    RETURNS boolean
    EXTERNAL NAME
        '/apps/lib/rtree2.so(rtree2_within)'
    LANGUAGE C NOT VARIANT;
```

3. Write the C functions for the new support functions: **rtree2_union()**, **rtree2_size()**, and **rtree2_inter()**.

   These functions must have the same return values as their counterparts in the **rtree_ops** operator class. Compile these functions and store them in the **rtree2.so** shared library.

4. Register the four new support functions with the CREATE FUNCTION statement.

   The following CREATE FUNCTION statements register the support functions that handle the **polygon** opaque data type:

```
CREATE FUNCTION Union2(polygon, polygon)
    RETURNS polygon
    EXTERNAL NAME '/lib/rtree2.so(rtree2_union)'
    LANGUAGE C NOT VARIANT;

CREATE FUNCTION Size2(polygon, polygon)
    RETURNS float
    EXTERNAL NAME
        '/lib/rtree2.so(rtree2_size)'
    LANGUAGE C NOT VARIANT;

CREATE FUNCTION Inter2(polygon, polygon)
    RETURNS polygon
    EXTERNAL NAME '/lib/rtree2.so(rtree2_inter)'
    LANGUAGE C NOT VARIANT;
```

5. Create the new **rtree2_ops** operator class for the **rtree** secondary access method.

```
CREATE OPCLASS rtree2_ops FOR rtree
    STRATEGIES (Overlap2, Equal2, Contains2, Within2)
    SUPPORT (Union2, Size2, Inter2);
```

6. You can now create an R-tree index on a **polygon** column and associate the new operator class with this column.

```
CREATE TABLE region_tab
(
    region_id integer,
    region_space polygon
    ...
);
CREATE INDEX reg_spc_rix
    ON region_tab (region_space rtree2_ops)
    USING rtree;
```

The optimizer can now choose whether to use the **reg_spc_rix** index to evaluate the following query:

```
SELECT * FROM region_tab WHERE Contains2(region_space) = 17
```

Dropping an Operator Class

The DROP OPCLASS statement removes the definition for an operator class from the database. Universal Server removes the operator-class definition from the **sysopclasses** system catalog table. You must be the owner of the operator class or the DBA to drop its definition from the database.

You must remove all dependent objects before you can drop the operator class. For example, suppose you have created a new operator class called **abs_btree_ops** for the generic B-tree index. (For more information on how to create this operator class, see "Defining a New B-Tree Operator Class" on page 4-20.) To drop the **abs_btree_ops** operator class from the database, you must first ensure that:

- you are the owner (the person who created the operator class) or the DBA.
- no indexes are currently defined that use the **abs_btree_ops** operator class.

  If such indexes exist, you must first remove them from the database.

Once the preceding conditions are met, the following statement removes the definition of **abs_btree_ops** from the database:

```
DROP OPCLASS abs_btree_ops RESTRICT
```

The RESTRICT keyword is required in the DROP OPCLASS syntax.

Chapter

Creating an Opaque Data Type

5

This chapter provides the following information about opaque data types:

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

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

What Is an Opaque Data Type?

An *opaque data type* is an atomic data type that you define for the database. An opaque type gets its name from the fact that the database server knows nothing about the internal representation of the type. Unlike built-in types, for which knows the internal format, the opaque types are encapsulated; that is, Universal Server has no knowledge of the format of the data within an opaque type.

When you define an opaque data type, you extend the data type system of the database server. The new opaque data type can be used in the same way as any built-in data type that the database server provides. To define the opaque data type to the database server, you must provide the following information in the C language:

- A data structure that serves as the internal storage of the opaque data type
- Support functions that allow the database server to interact with this internal structure
- Optional additional routines that can be called by other support functions or by end users to operate on the opaque data type

The following sections introduce each of these parts of an opaque data type. For information on how to create these parts, see "How Do You Create an Opaque Data Type?" on page 5-9.

The Internal Structure

To create an opaque type, you must first provide a data structure that stores the data in its internal representation. This data structure is called the *internal structure* of the opaque data type because it is how the data is stored on disk. The support functions that you write (see "Support Functions" on page 5-5) operate on this internal structure; the database server never sees it. You create the internal structure as a C-language data structure.

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

- A fixed-length opaque type
- A varying-length opaque type

A Fixed-Length Opaque Type

A *fixed-length opaque type* has an internal structure whose size is the same for all possible values of the opaque type. Fixed-length opaque types are useful for data that you can represent in fixed-length fields, such as numeric values. For an example of this data type, see "A Fixed-Length Opaque Type: circle" on page 5-30.

You provide the size when you register the opaque type in the database. For more information, see "Final Structure Size" on page 5-10.

A Varying-Length Opaque Type

A *varying-length opaque type* has an internal structure whose size might be different for different values of the opaque type. Varying-length opaque types are useful for storage of multirepresentational data, such as images. For example, image sizes vary from one picture to another. You might store data up to a certain size within the opaque type and use a smart large object in the opaque type if the image size exceeds that size. For an example of this data type, see "A Varying-Length Opaque Type: image" on page 5-38.

When you register the opaque type in the database, you indicate that the size is varying, and you can indicate a maximum size for the internal structure. For more information, see "Final Structure Size" on page 5-10.

Support Functions

The *support functions* of an opaque data type tell the database server how to interact with the internal structure (see "The Internal Structure" on page 5-4) of the opaque type. Because an opaque type is encapsulated, the database server has no knowledge of the internal structure. It is the support functions, which you write, that interact with this internal structure. The database server calls these support functions to perform the interactions.

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

Function	Purpose
input	Converts the opaque-type data from its external representation to its internal representation.
output	Converts the opaque-type data from its external representation to its internal representation.
receive	Converts the opaque-type data from its internal representation on the client computer to its internal representation on the server computer.

(1 of 2)

Function	Purpose
send	Converts the opaque-type data from its internal representation on the server computer to its internal representation on the client computer.
import	Performs any tasks needed to process an opaque type when a bulk copy imports the opaque type in its external representation.
export	Performs any tasks needed to process an opaque type when a bulk copy exports the opaque type in its external representation.
importbinary	Performs any tasks needed to process an opaque type when a bulk copy imports the opaque type in its internal representation.
exportbinary	Performs any tasks needed to process an opaque type when a bulk copy exports the opaque type in its internal representation.
assign()	Performs any tasks needed to process an opaque type before storing it to disk.
destroy()	Performs any tasks needed to process an opaque type necessary before the database server removes a row that contains the type.
lohandles()	Returns a list of the smart large objects embedded in the opaque type.
compare()	Compares two values of opaque-type data during a sort.

(2 of 2)

For more information on the opaque-type support functions, see Chapter 6, "Writing Support Functions."

Additional SQL-Invoked Routines

The support functions provide the basic functionality that the database server needs to interact with your opaque data type. However, you might want to write additional user-defined routines to provide the following kinds of functions for your opaque data type:

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

Built-In Functions

A *built-in function* is a predefined function that Universal Server provides for use in an SQL expression. Universal Server supports the built-in functions on the built-in data types. To have a built-in function operate on the opaque data type, you must write a version of the function that accepts the opaque data type in its parameter list.

For general information about these built-in functions, see "Built-In Functions" on page 2-8. For information on how to implement a built-in function on an opaque type, see "Built-In Functions for Opaque Data Types" on page 5-21.

Operator Functions

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

For general information about the operator functions that Universal Server provides, see "Operators and Operator Functions" on page 2-4. For information on how to implement an operator function on an opaque type, see "Operator Functions for Opaque Data Types" on page 5-20.

Aggregate Functions

An *aggregate function* returns one value for a set of queried rows. Universal Server supports only the following aggregate functions on an opaque data type:

- COUNT
- MIN
- MAX

For any other aggregate function to work with your opaque type, you must create your own version of the aggregate function. For more information on aggregate functions, see the Expression segment in the *Informix Guide to SQL: Syntax*.

End-User Routines

Universal Server allows you to define SQL-invoked functions or procedures that the end user can use in expressions of SQL statements. These end-user routines provide additional functionality that an end user might need to work with the opaque data type. Examples of end-user routines include the following:

- Functions that return a particular value in the opaque data type

  Because the opaque type is encapsulated, an end-user function is the only way users can access fields of the internal structure.
- Casting functions

  Several of the support functions serve as casting functions between basic data types that the database server uses. You might also write additional casting functions between the opaque type and other data types (built-in, opaque, or complex) of the database.
- Functions or procedures that perform common operations on the opaque data type

  If an operation or task is performed often on the opaque type, you might want to write an end-user routine to perform this task.

For more information about how to write end-user routines, see "Creating End-User Routines" on page 5-25.

How Do You Create an Opaque Data Type?

To create a new opaque data type, follow these steps:

1. Create the internal structure (a C data structure) for the opaque data type.
2. Write the support functions as C-language functions.
3. Register the opaque data type in the database with the CREATE OPAQUE TYPE statement.
4. Register the support functions of the opaque data type with the CREATE FUNCTION statement.
5. Provide access to the opaque data type and its support functions with the GRANT statement.
6. Write any SQL-invoked functions that are needed to support the opaque data type.
7. Provide any customized secondary access methods that the opaque data type might need.

The following sections describe each of these steps.

Creating the Internal Structure

The internal structure of an opaque data type is a C data structure. For the internal structure, use the C **typedef**s that the DataBlade API supplies for those fields whose size might vary by platform. Use of these **typedef**s, such as **mi_integer** and **mi_float**, improves the portability of the opaque data type. For more information on these data types, see Chapter 1 of the *DataBlade API Programmer's Manual*.

The internal structure uniquely names the opaque data type. Informix recommends that you develop a unique prefix for the opaque data type. You can prepend this prefix to each member of the internal structure and to the structure itself. A convention that Informix follows for opaque data types is to append the string **_t** to the structure name. For example, the **circle_t** data structure holds the values for the **circle** opaque type.

When you create the internal structure, consider the following impacts of the size of this structure:

- The final structure size of the new opaque data type
- How the opaque type should be aligned in memory
- How the opaque type is to be passed to user-defined routines by the database server

You provide this information when you create the opaque data type with the CREATE OPAQUE TYPE statement.

Final Structure Size

To save space in the database, you should lay out internal structures as compactly as possible. The database server stores values in their internal representation, so any internal structure with padding between entries consumes unnecessary space.

You supply the final size of the internal structure with the INTERNAL-LENGTH keyword of the CREATE OPAQUE TYPE statement. This keyword provides the following two ways to specify the size:

- Specifying the actual size, in bytes, of the internal structure defines a fixed-length opaque data type.
- Specifying the VARIABLE keyword defines a varying-length opaque data type.

A Fixed-Length Opaque Data Type

When you specify the actual size for INTERNALLENGTH, you create a fixed-length opaque type. The size of a fixed-length opaque type must match the value that the C-language **sizeof()** directive returns for the internal structure.

On most compilers, the **sizeof()** directive rounds up to the nearest four-byte size to ensure that pointer match on arrays of structures works correctly. However, you do not need to round up for the size of a fixed-length opaque data type. Instead you can specify alignment for the opaque type with the ALIGNMENT modifier. For more information, see "Memory Alignment" on page 5-11.

For an example of a fixed-length opaque type, see "A Fixed-Length Opaque Type: circle" on page 5-30.

A Varying-Length Opaque Data Type

When you specify the VARIABLE keyword for the INTERNALLENGTH modifier, you create a varying-length opaque type. By default, the maximum size for a varying-length opaque type is 2 kilobytes.

To specify a different maximum size for a varying-length opaque type, use the MAXLEN modifier. You can specify a maximum length of up to 32 kilobytes. When you specify a MAXLEN value, Universal Server can optimize resource allocation for the opaque type. If the size of the data for an opaque type exceeds the MAXLEN value, Universal Server returns an error.

For example, the following CREATE OPAQUE TYPE statement defines a varying-length opaque type called **var_type** whose maximum size is 4 kilobytes:

```
CREATE OPAQUE TYPE var_type (INTERNALLENGTH=VARIABLE,
    MAXLEN=4096);
```

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

For an example of a varying-length opaque type, see "A Varying-Length Opaque Type: image" on page 5-38.

Memory Alignment

When Universal Server passes the data type to a user-defined routine, it aligns opaque-type data on a specified byte boundary. Alignment requirements depend on the C definition of the opaque type and on the system (hardware and compiler) on which the opaque data type is compiled.

You can specify the memory-alignment requirement for your opaque type with the ALIGNMENT modifier of the CREATE OPAQUE TYPE statement. The following table summarizes valid alignment values.

ALIGNMENT Value	Meaning	Purpose
1	Align structure on single-byte boundary	Structures that begin with 1-byte quantities
2	Align structure on 2-byte boundary	Structures that begin with 2-byte quantities such as **mi_unsigned_smallint**
4	Align structure on 4-byte boundary	Structures that begin with 4-byte quantities such as **float** or **mi_unsigned_integer**
8	Align structure on 8-byte boundary	Structures that contain members of the C **double** data type

Structures that begin with single-byte characters, **char**, can be aligned anywhere. Arrays of a data type should follow the same alignment restrictions as the data type itself.

For example, the following CREATE OPAQUE TYPE statement specifies a fixed-length opaque type, called **LongLong**, of 18 bytes that must be aligned on a 1-byte boundary:

```
CREATE OPAQUE TYPE LongLong (INTERNALLENGTH=18, ALIGNMENT=1);
```

If you do not include the ALIGNMENT modifier in the CREATE OPAQUE TYPE statement, the default alignment is a 4-byte boundary.

Parameter Passing

Universal Server can pass opaque-type values to a user-defined routine in either of the following ways:

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

By default, the database server passes all opaque types by reference. To have the database server pass an opaque type by value, specify the PASSEDBYVALUE modifier in the CREATE OPAQUE TYPE statement. Only an opaque type whose size is 4 bytes or smaller can be passed by value. However, the DataBlade API data type **mi_real**, although only 4 bytes in length, is always passed by reference.

The following CREATE OPAQUE TYPE statement specifies that the **two_bytes** opaque type be passed by value:

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

Writing the Support Functions

Support functions are user-defined functions that you write in the C language. These user-defined functions are called *external functions*. Once you code the support functions, you compile them and place the compiled versions in a shared library on the server computer. You provide the pathname for these shared libraries in the CREATE FUNCTION statement when you register the support functions (see "Registering Support Functions" on page 5-15). You must ensure that the database server can access the shared libraries at runtime.

For general information on how to write external functions, see *Extending INFORMIX-Universal Server: User-Defined Routines*. For more information on the purpose of opaque-type support functions, see Chapter 6, "Writing Support Functions."

Registering the Opaque Type with the Database

Once you have created the internal structure and support functions for the opaque data type, use the following SQL statements to register them with the database:

- The CREATE OPAQUE TYPE statement registers the opaque data type as a data type.
- The CREATE FUNCTION statement registers a support function.
- The CREATE CAST statement registers the support functions as casting functions.

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

Registering the Opaque Data Type

The CREATE OPAQUE TYPE statement registers an opaque type with the database. It provides the following information to the database:

- The name and owner of the opaque data type

  The opaque-type name is the name of the data type that SQL statements use. It does not have to be the name of the internal structure for the opaque type. You might find it useful to create a special prefix to identify the data type as an opaque type. The opaque-type name must be unique within the name space.

- The size of the opaque data type

  You specify this size information with the INTERNALLENGTH modifier. It indicates whether the type is a fixed-length or varying-length opaque type. For more information, see "Creating the Internal Structure" on page 5-9.

- The values of the different opaque-type modifiers

  The CREATE OPAQUE TYPE statement can specify the following modifiers for an opaque type: MAXLEN, PASSEDBYVALUE, CANNOTHASH, and ALIGNMENT. You determine this information when you create the internal structure for the opaque data type. For more information, see "Creating the Internal Structure" on page 5-9.

The CREATE OPAQUE TYPE statement stores this information in the **sysxtdtypes** system catalog table. When it stores a new opaque type in **sysxtdtypes**, the CREATE OPAQUE TYPE statement causes a unique value, called an *extended identifier*, to be assigned to the opaque type. Throughout the system catalog, an opaque data type is identified by its extended identifier, not by its name. (For more information on the columns of the **sysxtdtypes** system catalog, see the chapter on system catalog tables in the *Informix Guide to SQL: Reference*.)

To register a new opaque data type in a database, you must have the Resource privilege on that database. By default, a new opaque type has Usage permission assigned to the owner. For information on how to change the permission of an opaque type, see "Granting Privileges for an Opaque Data Type" on page 5-18.

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

Registering Support Functions

Use the CREATE FUNCTION statement to register a support function with the database. Because support functions must be written in an external language (currently the C language), the CREATE FUNCTION statement that registers a support function has the syntax that Figure 5-1 shows.

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

Figure 5-1
Syntax of the CREATE FUNCTION Statement for Support Functions

This SQL statement provides the following information to the database:

- The name, *func_name*, and owner of the support function
- An optional specific name for the support function (not shown)
- The data types of the parameters, *parameter_list*, and return value, *ret_type*, of the support function
- The location, *pathname*, of the source code for the support function

- The language in which the support function is written (C language is current required): LANGUAGE C
- The routine modifier to indicate that a support function does not return different results with different arguments: NOT VARIANT

When you register a support function with the CREATE FUNCTION syntax that Figure 5-1 on page 5-15 shows, use the appropriate SQL data types for *parameter_list* and *ret_type*, as follows.

Support Function	Parameter Type	Return Type
input	LVARCHAR	opaque data type
output	opaque data type	LVARCHAR
receive	SENDRECV	opaque data type (on server computer)
send	opaque data type (on server computer)	SENDRECV
import	IMPEXP	opaque data type
export	opaque data type	IMPEXP
importbinary	IMPEXPBIN	opaque data type
exportbinary	opaque data type	IMPEXPBIN

In the preceding table, opaque data type is the name of the data type that you specify in the CREATE OPAQUE TYPE statement. For more information, see "Registering the Opaque Data Type" on page 5-14.

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

By default, a new support function has Execute permission assigned to the owner. For information on how to change the permission of an support function, see "Privileges on the Support Functions" on page 5-19.

E/C

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

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

Creating the Casts

For each of the support functions in the following table, Universal Server uses a companion cast definition to convert the opaque data type to a particular internal data type.

Support Function	**Cast**		
	From	**To**	**Type of Cast**
input	LVARCHAR	opaque data type	implicit
output	opaque data type	LVARCHAR	explicit
receive	SENDRECV	opaque data type	implicit
send	opaque data type	SENDRECV	explicit
import	IMPEXP	opaque data type	implicit
export	opaque data type	IMPEXP	explicit
importbinary	IMPEXPBIN	opaque data type	implicit
exportbinary	opaque data type	IMPEXPBIN	explicit

For the database server to perform these casts, you must create the casts with the CREATE CAST statement. The database server can then call the appropriate support function when it needs to cast opaque-type data to or from the LVARCHAR, SENDRECV, IMPEXP, or IMPEXPBIN data types.

The CREATE CAST statement stores information about casting functions in the **syscasts** system catalog table. For more information on the CREATE CAST statement, see the description in the *Informix Guide to SQL: Syntax*. For a description of casting, see the *Informix Guide to SQL: Tutorial*.

Granting Privileges for an Opaque Data Type

Once you have created the opaque type and registered it with the database, use the GRANT statement to define the following privileges on this data type:

- Privileges on the use of the opaque data type
- Privileges on the support functions of the opaque data type

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

Privileges on the Opaque Data Type

To create a new opaque type within a database, you must have the Resource privilege on the database. The CREATE OPAQUE TYPE statement creates a new opaque type with the Usage privilege granted to the owner of the opaque type and the DBA. To use the opaque data type in an SQL statement, you must have the Usage privilege. The owner can grant the Usage privilege to other users with the USAGE ON TYPE clause of the GRANT statement.

The database server checks for the Usage privilege whenever the opaque-type name appears in an SQL statement (such as a column type in CREATE TABLE or a cast type in CREATE CAST). The database server does *not* check for the Usage privilege when an SQL statement:

- accesses columns of the opaque type.

  The Select, Insert, Update, and Delete table-level privileges determine access to a column.
- invokes a user-defined routine with the opaque type as an argument.

  The Execute routine-level privilege determines access to a user-defined routine.

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

```
GRANT USAGE ON TYPE circle TO dexter
```

The **sysxtdtypeauth** system catalog table stores type-level privileges. This table contains privileges for each opaque and distinct data type that is defined in the database. The table contains one row for each set of privileges granted.

Privileges on the Support Functions

To register a support function within a database, you must have the Resource privilege on the database. The CREATE FUNCTION statement registers the new support function with the Execute privilege granted to the owner of the support function and the DBA. Such a function is called an owner-privileged function. To execute a support function in an SQL statement, you must have the Execute privilege. Usually, this default privilege is adequate for support functions that are implicit casts because they should not generally be called within SQL statements. Support functions that are explicit casts might have the Execute privilege granted so that users can call them explicitly. The owner grants the Execute privilege to other users with the EXECUTE ON clause of the GRANT statement.

The **sysprocauth** system catalog table stores routine-level privileges. This table contains privileges for each user-defined routine and therefore for all support functions that are defined in the database. The table contains one row for each set of privileges granted.

Creating SQL-Invoked Functions

An *SQL-invoked function* is a user-defined function that an end user can explicitly call in an SQL statement. You might write SQL-invoked functions to extend the functionality of an opaque data type in the following ways:

- Writing new versions of arithmetic or built-in functions to provide arithmetic operations and built-in functions on the opaque type
- Writing new versions of relational-operator functions to provide comparison operations on the opaque type

- Writing new end-user routines to provide additional functionality for the opaque data type
- Writing new casting functions to provide additional data conversions to and from the opaque data type

The versions of these functions that Universal Server defines handle the built-in data types. For your opaque data type to use any of these functions, you can write a version of the function that handles the opaque data type. (For more information on the details of writing user-defined functions, see *Extending INFORMIX-Universal Server: User-Defined Routines*.)

Operating on Data

Universal Server supports the following types of SQL-invoked functions that allow you to operate on data in expressions of SQL statements:

- Arithmetic and text operator functions
- Built-in functions
- Aggregate functions

Operator Functions for Opaque Data Types

Universal Server provides the following types of operators for expressions in SQL statements:

- Arithmetic operators usually operate on numeric values.
- Text operators operate on character strings.

Tip: *Universal Server also provides relational operators. For more information on the relational operators and their operator functions, see "Comparing Data" on page 5-22.*

Universal Server provides operator functions for the arithmetic operators (see Figure 2-4 on page 2-5) and text operators (Figure 2-5 on page 2-6). The versions of the operator functions that Universal Server provides handle the built-in data types. You can write a new version of one of these operator function to provide the associated operation on your new opaque data type.

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

1. The name of the operator function must match the name that Figure 2-4 on page 2-5 or Figure 2-5 on page 2-6 lists. However, the name is not case sensitive; the **plus()** function is the same as the **Plus()** function.
2. The operator function must handle the correct number of parameters.
3. The operator function must return the correct data type, where appropriate.

Built-In Functions for Opaque Data Types

Universal Server provides special SQL-invoked functions, called *built-in functions*, that provide some basic mathematical operations. Figure 2-7 on page 2-8 shows the built-in functions that Universal Server defines. The versions of the built-in functions that Universal Server provides handle the built-in data types. You can write a new version of a built-in function to provide the associated operation on your new opaque data type. If you write a new version of a built-in function, make sure you follow these rules:

1. The name of the built-in function must match the name that Figure 2-7 lists. However, the name is not case sensitive; the **abs()** function is the same as the **Abs()** function.
2. The built-in function must be one that can be overridden.
3. The built-in function must handle the correct number of parameters, and these parameters must be of the correct data type. Figure 2-7 lists the number and data types of the parameters.
4. The built-in function must return the correct data type, where appropriate.

For more information on built-in functions, see page 2-8.

Aggregate Functions for Opaque Data Types

Universal Server supports only the following aggregate functions on an opaque data type:

- COUNT
- MIN
- MAX

For any other aggregate function to work with your opaque type, you must create your own version of the aggregate function. For more information on aggregate functions, see the Expression segment in the *Informix Guide to SQL: Syntax*.

Comparing Data

Universal Server supports the following types of functions that allow you to compare data in expressions of SQL statements:

- SQL operators in a conditional clause
- Relational operator functions

Conditions for Opaque Data Types

Universal Server supports the following conditions on an opaque type in the conditional clause of SQL statements:

- The IS and IS NOT operators
- The IN operator if the **equal()** function has been defined
- The BETWEEN operator if the **compare()** function has been defined

Tip: *Universal Server also uses the* ***compare()*** *function as the support function for the default B-tree operator class. For more information, see "Extending the btree_ops Operator Class" on page 4-12.*

For more information on the conditional clause, see the Condition segment in the *Informix Guide to SQL: Syntax*. For more information on the **equal()** function, see "Relational Operators for Opaque Data Types" on page 5-23. For more information on the **compare()** function, see "Opaque-Type Sorting" on page 5-24.

Relational Operators for Opaque Data Types

Universal Server provides operator functions for the relational operators that Figure 2-6 on page 2-6 shows. The versions of the relational-operator functions that Universal Server provides handle the built-in data types. You can write a new version of a relational-operator function to provide the associated operation on your new opaque data type. If you write a new version of a relational-operator function, make sure you follow these rules:

1. The name of the relational-operator function must match the name that Figure 2-6 lists. However, the name is not case sensitive; the **equal()** function is the same as the **Equal()** function.
2. The relational-operator function must take two parameters, both of the opaque data type.
3. The relational-operator function must be a Boolean function; that is, it must return a BOOLEAN value.

You must define an **equal()** function to handle your opaque data type if you want to allow columns of this type to be:

- constrained as UNIQUE or PRIMARY KEY.

  For more information on constraints, see the CREATE TABLE statement in the *Informix Guide to SQL: Syntax*.
- compared with the equal (=) operator in an expression.
- used with the IN operator in a condition.

The **equal()** function can only compare *bit-hashable* data types; that is, equality can be determined with a bit-wise compare. This comparison means that two values are equal if they have the same internal representation. The database server uses a built-in hash function to perform this comparison.

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

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

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

Opaque-Type Sorting

The **compare()** function is an SQL-invoked function that sorts the target data type. Universal Server uses the **compare()** function to execute the following clauses and keywords of the SELECT statement:

- The ORDER BY clause
- The UNIQUE and DISTINCT keywords
- The UNION keyword

The database server also uses the **compare()** function to evaluate the BETWEEN operator in the condition of an SQL statement. For more information on conditional clauses, see the Condition segment in the *Informix Guide to SQL: Syntax*.

Universal Server provides versions of the **compare()** function that handle the built-in data types. For the database server be able to sort an opaque type, you must define a **compare()** function that handles this opaque type. If you write a new version of a **compare()** function, make sure you follow these rules:

1. The name of the function must be **compare()**. The name, however, is not case sensitive; the **compare()** function is the same as the **Compare()** function.
2. The function must accept two arguments, each of the data type to be compared.
3. The function must return an integer value to indicate the result of the comparison, as follows:
   - <0 to indicate that the first argument is less than (<) the second argument
   - 0 to indicate that the two arguments are equal (=)
   - >0 to indicate that the first argument is greater than (>) the second argument

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

The **compare()** function is also the support function for the default operator class of the B-tree secondary access method. For more information, see "The Generic B-Tree Index" on page 4-4.

Creating End-User Routines

Universal Server allows you to define SQL-invoked functions and procedures that the end user can use in expressions of SQL statements. These end-user routines provide additional functionality that an end user might need to work with the opaque data type. To create an end-user routine, follow these steps:

- Write the end-user routine in either SPL or C.
- Register an end-user function in the database with the CREATE FUNCTION statement or an end-user procedure with the CREATE PROCEDURE statement.
- Grant the Usage privilege to **public** (or to any other users who are to have access).

You might also write end-user functions to serve as additional casting functions for the opaque type. To create a casting function, you must use the CREATE FUNCTION statement to register the end-user function and the CREATE CAST function to register this function as a casting function.

For more information about how to write end-user routines, see "End-User Routines" on page 2-10.

Customizing Use of Access Methods

Universal Server provides the full implementation of the generic B-tree secondary access method, and it provides definitions for the R-tree secondary access method. By default, the CREATE INDEX statement (without the USING clause) builds a generic B-tree index for the column or user-defined function.

When you create an opaque data type, you must ensure that secondary access methods exist that support the new data type. Consider the following factors about the secondary access methods and their support for the opaque data type:

- Does the generic B-tree support the opaque data type?
- If the opaque-type data is spatial, can you use the R-tree index?
- Do other secondary access methods exist that might better index your opaque-type data?

To create an index of a particular secondary access method on a column of an opaque data type, the database server must find an operator class that is associated with the secondary access method. This operator class must specify operations (strategy functions) on the opaque type as well as the functions that the secondary access method uses (support functions).

For more information about an operator class and operator-class functions, see "What Is an Operator Class?" on page 4-6.

Using the Generic B-Tree

The generic B-tree secondary access method has a default operator class, **btree_ops**, whose operator-class functions handle indexing for the built-in data types. These operator-class functions have the following functionality for built-in data types:

- They order the data in lexicographical sequence.

  If this sequence is not logical for your opaque data type, you can define operator-class functions for the opaque data type that provide the sequence you need.
- They expect to compare two single, one-dimensional values for a given data type.

  If the opaque data type holds more than one value but you can define a single value for it, you can define operator-class functions for the opaque data type that compare two of these one-dimensional values. If you cannot define a one-dimensional value for the opaque data type, you cannot use a B-tree index as its secondary access method. For more information, see "A Fixed-Length Opaque Type: circle" on page 5-30.

To provide support for columns and user-defined functions of your opaque type, you can extend the **btree_ops** operator-class functions so that they handle the new opaque data type. The generic B-tree secondary access method uses the new operator-class functions to store values of your opaque data type in a B-tree index.

For more information about how to extend the default B-tree operator class, see "Extending the btree_ops Operator Class" on page 4-12. For an example of how to extend the **btree_ops** operator class for a fixed-length opaque type, see "A Fixed-Length Opaque Type: circle" on page 5-30.

Indexing Spatial Data

The way that the generic B-tree secondary access method orders data is useful for one-dimensional data. The operator-class functions (strategy and support functions) for the default B-tree operator class, **btree_ops**, order one-dimensional values within the B-tree index. If the data in your opaque data type is multidimensional, you might need to use a secondary access method that can order the multidimensional data.

The R-tree secondary access method is useful for spatial or multidimensional data such as maps and diagrams. If your database implements an R-tree, you can create an R-tree index on the spatial-type columns that are likely to be used in a search condition.

Important: *To use an R-tree index, you must install a spatial DataBlade module such as the Spatial DataBlade module, Geodetic DataBlade, or any other third-party DataBlade module that implements an R-tree index.*

For information about how to create an R-tree index, see "The R-Tree Index" on page 4-5. For information on how to extend the default R-tree operator class, see "Extending the rtree_ops Operator Class" on page 4-18.

Other Kinds of Data

Your opaque data type might have data that is not optimally indexed by a generic B-tree or an R-tree. Often, DataBlade modules that define new opaque data types provide their own secondary access methods for these data types. For more information on the secondary access methods, check the user guide for your DataBlade module. For information, see "Other User-Defined Secondary Access Methods" on page 4-6.

Other Operations on Opaque Data Types

This section describes the following operations that you can perform on opaque data types:

- How to access an opaque data type from a client application
- How to drop an opaque data type from a database

Accessing an Opaque Data Type

Once you have created the opaque data type, the following client programs can use it once they connect to the database in which it is registered:

- An INFORMIX-ESQL/C application uses SQL statements and an **lvarchar**, **fixed binary**, or **var binary** host variable.

  For more information, see the chapter on opaque types in the *INFORMIX-ESQL/C Programmer's Manual*.
- An external routine (a user-defined routine written in the C language) uses the Informix DataBlade API.

  For more information, see the *DataBlade API Programmer's Manual*.
- An SPL routine (a user-defined routine written in Stored Procedure Language, SPL) uses SQL statements.

  For more information, see the chapter on SPL in the *Informix Guide to SQL: Tutorial*.

An opaque data type can be used in any way that other data types of the database can be used.

Dropping an Opaque Data Type

You cannot drop an opaque type if any dependencies on it still exist in the database. Therefore, to drop an opaque data type from a database, you reverse the process of registering the database, as follows:

1. Remove or change the data type of any columns in the database that have the opaque data type as their data type.

   Use the ALTER TABLE statement to change the data type of database columns. Use the DROP TABLE statement to remove the entire table.
2. The REVOKE statement with the USAGE ON TYPE clause removes one set of privileges assigned to the opaque data type.

   This statement removes the row of the **sysxtdtypeauth** system catalog table that defines the privileges of the opaque type. Use the statement to drop each set of privileges that have been assigned to the opaque type.

3. The REVOKE statement with the EXECUTE ON FUNCTION or EXECUTE ON ROUTINE clause removes the privileges assigned to a support function of the opaque data type.

   This statement removes the row of the **sysprocauth** system catalog table that defines the privileges of the opaque type. Use the statement to drop each set of privileges that have been assigned to a support function. You must drop the privileges for each support function. If you assigned a specific name to the support function, use the SPECIFIC keyword to identify the specific name.

4. The DROP CAST statement drops a casting function for a support function of an opaque data type.

   This statement removes the row of the **syscasts** system catalog table that defines the casting function for a support function. Use the statement to drop each of the casts you defined. For more information, see "Creating the Casts" on page 5-17.

5. The DROP FUNCTION or DROP ROUTINE statement removes a support function of the opaque data type from the current database.

   This statement removes the row of the **sysprocedures** system catalog table that registers a support function. Use the statement to drop each of the support functions that you registered. For more information, see "Registering Support Functions" on page 5-15.

6. The DROP TYPE statement removes the opaque data type from the current database.

   This statement removes the row of the **sysxtdtypes** system catalog table that describes the opaque data type. Once you drop an opaque type from a database, no users of that database can access the type. You must be the owner of the opaque type or have DBA privileges to remove the type.

To use these SQL statements, you must be either the owner of the object you drop or have DBA privileges. For more information on the syntax of the REVOKE, DROP FUNCTION, DROP ROUTINE, DROP CAST, and DROP TYPE statements, see their descriptions in the *Informix Guide to SQL: Syntax*.

A Fixed-Length Opaque Type: circle

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

Creating the Fixed-Length Internal Structures

Figure 5-2 shows the internal data structures for the **circle** data type.

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

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

Figure 5-2
Internal Data Structures for the circle Opaque Data Type

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

Writing the circle Support Functions

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

Support Function	Function Signature
input	`circle_t * circle_input(extrnl_format) mi_lvarchar *extrnl_format;`
output	`mi_lvarchar * circle_output(intrnl_format) circle_t *intrnl_format;`
receive	`circle_t * circle_receive(client_intrnl_format) mi_sendrecv *client_intrnl_format;`
send	`mi_sendrecv * circle_send(srvr_intrnl_format) circle_t *srvr_intrnl_format;`

The actual code for the **circle** data type would be written in the C language, compiled, and put in the **circle.so** shared library.

Suppose the input and output functions of the **circle** data type define the external format that Figure 5-3 shows.

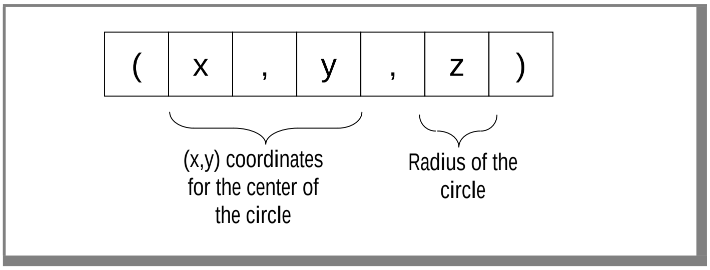


Figure 5-3
External Format of the circle Opaque Data Type

The input support function, **circle_input()**, would accept a string of the form in Figure 5-3, parse the string and store each value in the **circle_t** structure. The output support function, **circle_output()**, would perform the inverse operation: it would take the values from the **circle_t** structure and build a string of the form in Figure 5-3.

Registering the circle Opaque Type

Figure 5-4 shows the SQL statements that register the **circle** data type and its input, output, send, and receive support functions in the database.

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

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

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

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

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

Figure 5-4
Registering the circle Opaque Data Type

The CREATE OPAQUE TYPE statement in Figure 5-4 registers the **circle** data type, which has the following characteristics:

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

The CREATE FUNCTION statements in Figure 5-4 register the following support functions:

- The input support function is registered in the database as **circle_in()**. Its source code is the **circle_input()** function in the **/usr/lib/circle.so** shared library.
- The output support function is registered in the database as **circle_out()**. Its source code is the **circle_output()** function in the **/usr/lib/circle.so** shared library.

- The receive support function is registered in the database as **circle_rcv()**. Its source code is the **circle_receive()** function in the **/usr/lib/circle.so** shared library.
- The send support function is registered in the database as **circle_snd()**. Its source code is the **circle_send()** function in the **/usr/lib/circle.so** shared library.

The CREATE CAST statements create the appropriate casts for the support functions. For more information on these casts, see page 5-17.

Granting Privileges on the circle Data Type

The CREATE OPAQUE TYPE statement (see Figure 5-4 on page 5-32) creates the **circle** data type with the Usage privilege granted to the owner (the person who ran the CREATE OPAQUE TYPE statement). The following GRANT statement allows all users of the database to use the **circle** data type in SQL statements:

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

The CREATE FUNCTION statements (also in Figure 5-4) register the support functions for the **circle** data type with the Execute privilege granted to the owner (the person who ran the CREATE FUNCTION statements). The following GRANT statements grant the Execute privilege to all users of the database on the support statements that are defined as explicit casts:

```
GRANT EXECUTE ON FUNCTION circle_out TO PUBLIC;
GRANT EXECUTE ON FUNCTION circle_snd TO PUBLIC;
```

Creating SQL-Invoked Functions for the circle Data Type

The **circle** data type has two end-user functions:

- The **radius()** function obtains the **radius** value from a **circle_t** structure.

  The actual code for the **radius()** function is written in the C language, compiled, and put in the **circle.so** shared library.
- The **area()** function calculates the area of the circle that a **circle_t** structure describes.

  The actual code for the **area()** function is written in the SPL language, so it is provided directly in the CREATE FUNCTION statement.

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

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

CREATE FUNCTION area(crcl_val circle) RETURNING FLOAT
    DEFINE pi FLOAT;

    LET pi = 3.1415926;
    RETURN (pi * POW(RADIUS(circle), 2);
END FUNCTION;
```

Providing Indexes for the circle Data Type

Suppose that you decide that end users can define generic B-tree indexes on the **circle** data type. To generate a lexicographical sequence, the B-tree index compares the area of two **circle** values to determines the order of the values. Because the operator-class functions of the B-tree index do not know about the **circle** data type, you must write the following functions so that the B-tree index can order the values correctly:

- The strategy functions of the default operator class for the generic B-tree

  For more information about these functions, see "The B-Tree Strategy Functions" on page 4-8.

- The support function of the default operator class for the generic B-tree

  For more information about these functions, see "The B-Tree Support Function" on page 4-8.

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

B-Tree Strategy Function	Meaning for circle Data Type	Function Signature
lessthan()	**area**(circle1) < **area**(circle2)	`boolean circ le_lessthan(crc1, crc2)` `circle *crc1, *crc2;`
lessthanorequal()	**area**(circle1) <= **area**(circle2)	`boolean circle_lessthanorequal(crc1, crc2)` `circle *crc1, *crc2;`
equal()	**area**(circle1) = **area**(circle2)	`boolean circle_equal(crc1, crc2)` `circle *crc1, *crc2;`
greaterthan()	**area**(circle1) > **area**(circle2)	`boolean circle_greaterthan(crc1, crc2)` `circle *crc1, *crc2;`
greaterthanorequal()	**area**(circle1) >= **area**(circle2)	`boolean circle_greaterthanorequal(crc1, crc2)` `circle *crc1, *crc2;`

These functions might already have been defined when you provided support for the relational operators on your opaque type. (For more information, see "Comparing Data" on page 5-22.) In this case, the **lessthan()**, **lessthanorequal()**, **equal()**, **greaterthan()**, and **greaterthanorequal()** functions have already been written, compiled, stored in a shared library, and registered in the database. Therefore, the generic B-tree index can already use them as its strategy functions.

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

Figure 5-5 shows the CREATE FUNCTION statements that register the B-tree strategy functions for the **circle** data type with the database.

```
CREATE FUNCTION lessthan(cir1 circle, cir2 circle)
    RETURNS BOOLEAN
    EXTERNAL NAME '/usr/lib/circle.so(circle_lessthan)'
    LANGUAGE C NOT VARIANT;

CREATE FUNCTION lessthanorequal(cir1 circle, cir2 circle)
    RETURNS BOOLEAN
    EXTERNAL NAME
        '/usr/lib/circle.so(circle_lessthanorequal)'
    LANGUAGE C NOT VARIANT;

CREATE FUNCTION equal(cir1 circle, cir2 circle)
    RETURNS BOOLEAN
    EXTERNAL NAME '/usr/lib/circle.so(circle_equal)'
    LANGUAGE C NOT VARIANT;

CREATE FUNCTION greaterthan(cir1 circle, cir2 circle)
    RETURNS BOOLEAN
    EXTERNAL NAME '/usr/lib/circle.so(circle_greaterthan)'
    LANGUAGE C NOT VARIANT;

CREATE FUNCTION greaterthanorequal(cir1 circle, cir2 circle)
    RETURNS BOOLEAN
    EXTERNAL NAME
        '/usr/lib/circle.so(circle_greaterthanorequal)'
    LANGUAGE C NOT VARIANT;
```

Figure 5-5
Registering the B-Tree Strategy Functions for the circle Opaque Data Type

The **compare()** function is the support function for the generic B-tree index. You might have already defined this function to provide support for the comparison operations in a SELECT statement (such as the ORDER BY clause or the BETWEEN operator) on your opaque type. (For more information, see "Comparing Data" on page 5-22.) In this case, the **compare()** function has already been written, compiled, stored in a shared library, and registered in the database. Therefore, the generic B-tree index can use it as its support function for the **circle** data type.

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

```
CREATE FUNCTION compare(cir1 circle, cir2 circle)
    RETURNS integer
    EXTERNAL NAME '/usr/lib/circle.so(circle_compare)'
    LANGUAGE C NOT VARIANT
```

With the strategy and support functions registered, you are ready to define generic B-tree indexes on the **circle** data type. The code fragment in Figure 5-6 creates a generic B-tree index called **circle_ix** on the **circle_col** column.

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

Using the circle Opaque Data Type

Figure 5-6 shows the SQL statements that create a table called **circle_tab** that has a column of type **circle** and insert several rows into this table.

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

Figure 5-6
Creating a Column of the circle Opaque Data Type

For information about the CREATE INDEX statement in Figure 5-6, see "Providing Indexes for the circle Data Type" on page 5-35.

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

A Varying-Length Opaque Type: image

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

This section briefly outlines how to create the **image** data type.

Creating the Varying-Length Internal Structure

Figure 5-7 shows the internal structure for the **image** data type in the database.

```
typedef struct
    {
    int     img_len;
    int     img_thresh;
    int     img_flags;
    union
        {
        ifx_lop_timg_lobhandle;
        char    img_data[4];

    } image_t;
```

Figure 5-7
Internal Structure for the image Opaque Data Type

Writing the image Support Functions

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

Support Function	Function Signature

input	`image_t * image_input(extrnl_format)`
	`    mi_lvarchar *extrnl_format;`

output	`mi_lvarchar * image_output(intrnl_format)`
	`    image_t *intrnl_format;`

receive	`image_t * image_receive(client_intrnl_format)`
	`    mi_sendrecv *client_intrnl_format;`

send	`mi_sendrecv * image_send(srvr_intrnl_format)`
	`    image_t *srvr_intrnl_format;`

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

Support Function	Function Signature

import	`image_t * image_import(extrnl_bcopy_format)`
	`    mi_impexp *extrnl_bcopy_format;`

export	`mi_impexp * image_export(intrnl_bcopy_format)`
	`    image_t *intrnl_bcopy_format;`

importbinary	`image_t * image_impbin(client_intrnl_bcopy_format)`
	`    mi_impexpbin *client_intrnl_bcopy_format;`

exportbinary	`mi_impexpbin * image_expbin(srvr_intrnl_bcopy_format)`
	`    image_t *srvr_intrnl_bcopy_format;`

lohandles()	`mi_sendrecv * image_lohandles(intrnl_format)`
	`    image_t *intrnl_format;`

assign()	`image_t * image_assign(intrnl_format)`
	`    image_t *intrnl_format;`

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

Registering the image Opaque Type

Figure 5-8 shows the SQL statements that register the **image** data type and its basic support functions in the database.

```
CREATE OPAQUE TYPE image (INTERNALLENGTH = VARIABLE);

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

CREATE FUNCTION image_out(im image) RETURNS LVARCHAR
    EXTERNAL NAME '/usr/lib/image.so(image_output)'
    LANGUAGE C NOT VARIANT;
CREATE EXPLICIT CAST (image AS LVARCHAR WITH image_out);

CREATE FUNCTION image_rcv(cl_im SENDRECV) RETURNS image
    EXTERNAL NAME '/usr/lib/image.so(image_receive)'
    LANGUAGE C NOT VARIANT;
CREATE IMPLICIT CAST (SENDRECV AS image) WITH image_rcv);

CREATE FUNCTION image_snd(srv_im image) RETURNS SENDRECV
    EXTERNAL NAME '/usr/lib/image.so(image_send)'
    LANGUAGE C NOT VARIANT;
CREATE EXPLICIT CAST (image AS SENDRECV WITH image_snd);
```

Figure 5-8
Registering the image Opaque Data Type and Its Basic Support Functions

The CREATE OPAQUE TYPE statement in Figure 5-8 registers the **image** data type that has the following characteristics:

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

The CREATE FUNCTION statements in Figure 5-8 register the following basic support functions:

- The input support function is registered in the database as **image_in()**. Its source code is the **image_input()** function in the **/usr/lib/image.so** shared library.
- The output support function is registered in the database as **image_out()**. Its source code is the **image_output()** function in the **/usr/lib/image.so** shared library.

- The receive support function is registered in the database as **image_rcv()**. Its source code is the **image_receive()** function in the **/usr/lib/image.so** shared library.
- The send support function is registered in the database as **image_snd()**. Its source code is the **image_send()** function in the **/usr/lib/image.so** shared library.

Figure 5-8 on page 5-40 also shows the CREATE CAST statements to create the appropriate cast definitions for the input, output, receive, and send support functions. For more information on the type of casts to create, see “Creating the Casts” on page 5-17.

Figure 5-9 shows the SQL statements that register the other support functions for the **image** data type in the database.

Figure 5-9
Registering the Other Support Functions for the image Opaque Data Type

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

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

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

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

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

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

The CREATE FUNCTION statements in Figure 5-9 on page 5-41 register the following support functions:

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

Figure 5-8 on page 5-40 also shows the CREATE CAST statements to create the cast definitions for the import, export, importbinary, and exportbinary support functions. For more information on these casts, see page 5-17.

Granting Privileges on the image Data Type

The CREATE OPAQUE TYPE statement (see Figure 5-8 on page 5-40) creates the **image** data type with the Usage privilege granted to the owner (the person who ran the CREATE OPAQUE TYPE statement). The following GRANT statement allows all users of the database to use the **image** data type in SQL statements:

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

The CREATE FUNCTION statements (see Figure 5-8 on page 5-40 and Figure 5-9 on page 5-41) register the support functions for the **image** data type with the Execute privilege granted to the owner (the person who ran the CREATE FUNCTION statements). The following GRANT statements grant the Execute privilege to all users of the database on the support statements that are defined as explicit casts:

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

Creating SQL-Invoked Functions for the image Data Type

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

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

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

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

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

Providing Indexes for the image Data Type

Suppose you decide that the default secondary access method, a generic B-tree index, does not adequately handle data in columns of type **image**. Because the image data type holds spatial data, it is a good candidate for an R-tree index. To be able to use an R-tree index, you must first install a spatial DataBlade module that implement the R-tree index. For more information, see "Extending the rtree_ops Operator Class" on page 4-18.

When the strategy and support functions for the R-tree secondary access method are registered, you are ready to define R-tree indexes on the **image** data type. The code fragment in Figure 5-10 creates an R-tree index called **image_ix** on the **image_col** column. This CREATE INDEX statement assumes that the DataBlade module enhances the existing default operator class for the R-tree access secondary method.

Using the image Opaque Type

Figure 5-10 shows the SQL statements that create a table called **image_tab** that has a column of type **image** and an image identifier.

```
CREATE TABLE image_tab
(
    image_id   integer not null primary key),
    image_colimage
);
CREATE INDEX image_ix ON image_tab (image_col)
    USING rtree;
```

Figure 5-10
Creating a Column of the image Opaque Data Type

For information about the CREATE INDEX statement in Figure 5-10, see "Providing Indexes for the image Data Type" on page 5-43.

In the *INFORMIX-ESQL/C Programmer's Manual*, the chapter on opaque types shows examples of how to access the **image** opaque type with a **var binary** host variable.

Chapter

Writing Support Functions

6

This chapter describes the support functions for the following objects:

- An opaque data type
- An operator class

For an introduction to an opaque data type, see Chapter 5, "Creating an Opaque Data Type." For an introduction to an operator class, see Chapter 4, "Extending an Operator Class."

Support Functions for Opaque Data Types

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

Function	Purpose	For More Information
input	Converts the opaque-type data from its external to its internal representation. Supports insertion of textual data into a column of the opaque type. Is an implicit cast from the LVARCHAR to opaque data type.	page 6-6
output	Converts the opaque-type data from its internal to external representation. Supports selection of textual data from a column of the opaque type. Is an explicit cast from the opaque to LVARCHAR opaque data type.	page 6-8

(1 of 3)

Function	Purpose	For More Information
receive	Converts the opaque-type data from its internal representation on the client computer to its internal representation on the server computer. Supports insertion of binary data in a column of the opaque type. Is an implicit cast from the SENDRECV to the opaque data type.	page 6-11
send	Converts the opaque-type data from its internal representation on the server computer to its internal representation on the client computer. Supports selection of binary data from a column of the opaque type. Is an explicit cast from the opaque to the SENDRECV data type.	page 6-13
import	Performs processing of opaque type for bulk load of textual data in a column of the opaque type. Is an implicit cast from the IMPEXP to the opaque data type.	page 6-16
export	Performs processing of opaque type for bulk unload of textual data from a column of the opaque type. Is an explicit cast from the opaque to the IMPEXP data type.	page 6-16
importbinary	Performs processing of opaque type for bulk load of binary data in a column of the opaque type. Is an implicit cast from the IMPEXPBIN to the opaque data type.	page 6-18
exportbinary	Performs processing of opaque type for bulk unload of binary data from a column of the opaque type. Is an explicit cast from the opaque to the IMPEXPBIN data type.	page 6-18
assign()	Performs any processing required before the database server stores the opaque-type data to disk. Supports storage of opaque type for INSERT, UPDATE, and LOAD statements.	page 6-19
destroy()	Performs any processing necessary before the database server removes a row that contains opaque-type data. Supports deletion of opaque type for DELETE and DROP TABLE statements.	page 6-20

(2 of 3)

Function	Purpose	For More Information
lohandles()	Returns a list of the embedded large-object handles in the opaque data type.	page 6-21
compare()	Supports sort of opaque-type data during ORDER BY, UNIQUE, DISTINCT, and UNION clauses and CREATE INDEX operations.	page 6-22

(3 of 3)

Handling the External Representation

Every opaque type has its internal and external representation. The internal representation is the internal structure that you define for the opaque type. (For more information, see "The Internal Structure" on page 5-4.) The external representation is a character string that is a printable version of the opaque value.

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

- The input function converts from external to internal representation.
- The output function converts from internal to external representation.

These support functions do not have to be named *input* and *output*, but they do have to perform the specified conversions. They should be reciprocal functions; that is, the input function should produce a value that the output function accepts as an argument and vice versa.

GLS

For your opaque data type to accept an external representation on non-default locales, you must use the Informix GLS API in the input and output functions to access Informix locales from within these functions. For more information, see "Handling Locale-Sensitive Data" on page 6-23. ♦

The LVARCHAR Data Type

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

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

E/C

INFORMIX-ESQL/C applications use **lvarchar** or other character-based host variables in SQL statements to transfer the external representation of an opaque type. The database server implicitly invokes the input and output support functions when it receives an SQL statement that contains an **lvarchar** host variable. ♦

For more information on how to use SQL statements to register support functions, see page 5-15.

Input Support Function

The database server calls the input function when it receives the external representation of an opaque type from a client application. For example, when a client application issues an INSERT or UPDATE statement, it can send the character representation of an opaque type to the database server to be stored in an opaque-type column. The database server calls the input function to convert this external representation to an internal representation that it stores on disk.

Figure 6-1 shows when Universal Server executes the input support function.

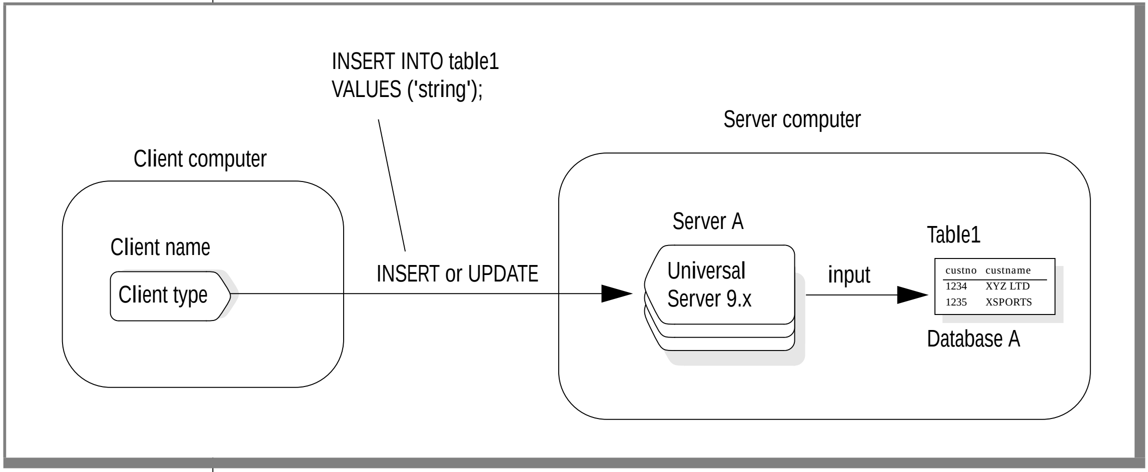


Figure 6-1
Execution of the input Support Function

If the opaque data type is pass-by-reference, the input support function should perform the following tasks:

- Allocate enough space to hold the internal representation

  It can use the **mi_alloc()** DataBlade API function to allocate the space for the internal structure.
- Parse the input string

  It must obtain the individual members from the input string and store them into the appropriate fields of the internal structure
- Return a pointer to the internal structure

If the opaque data type is pass-by-value, the input support function should perform these same basic tasks but return the actual value in the internal structure instead of a pointer to this structure. You can use pass-by-value only for opaque types that are less than 4 bytes in length.

The input function takes an **mi_lvarchar** value as an argument and returns the internal structure for the opaque type. The following function signature is an input support function for an opaque data type whose internal structure is **ll_longlong_t**:

```
ll_longlong_t * ll_longlong_input(extrnl_format)
    mi_lvarchar *extrnl_format;
```

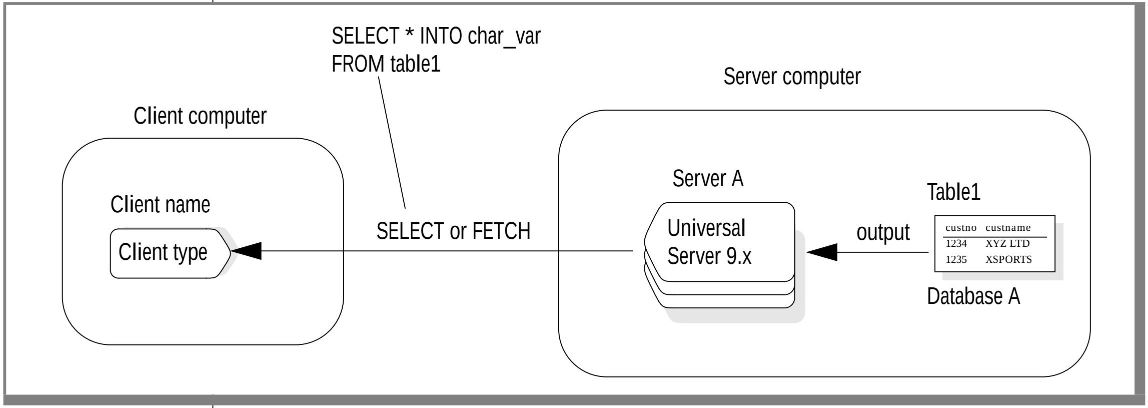
The **ll_longlong_input()** function is a casting function from the LVARCHAR data type to the **ll_longlong_t** internal structure. It must be registered as an implicit casting function with the CREATE IMPLICIT CAST statement. For more information on casting functions, see "Creating the Casts" on page 5-17.

Output Support Function

The database server calls the output function when it sends the external representation of an opaque type to a client application. For example, when a client application issues a SELECT or FETCH statement, it can save the data of an opaque type that it receives from the database server in a character host variable. The database server calls the output function to convert the internal representation that is stored on disk to the external representation that the character host variable requires.

Figure 6-2 shows when Universal Server executes the output support function.

Figure 6-2
Execution of the output Support Function



If the opaque data type is pass-by-reference, the output support function should perform the following tasks:

- Accept a pointer to the internal representation as an argument
- Allocate enough space to hold the external representation

  It can use the **mi_alloc()** DataBlade API function to allocate the space for the character string. For more information on memory management and the **mi_alloc()** function, refer to the *DataBlade API Programmer's Manual*.
- Create the output string from the individual members of the internal structure

  It must build the external representation with the values from the appropriate fields of the internal structure.
- Return a pointer to the character string

If the opaque data type is pass-by-value, the output support function should perform the same basic tasks but accept the actual value in the internal structure. You can use pass-by-value only for opaque types that are less than 4 bytes in length.

The output function takes the internal structure for the opaque type as an argument and returns an **mi_lvarchar** value. The following function signature is for an output support function of an opaque data type whose internal structure is **ll_longlong_t**:

```
mi_lvarchar * ll_longlong_output(intrnl_format)
    ll_longlong_t *intrnl_format;
```

The **ll_longlong_output()** function is a casting function from the **ll_longlong_t** internal structure to the LVARCHAR data type. It must be registered as an explicit casting function with the CREATE EXPLICIT CAST statement. For more information on casting functions, see "Creating the Casts" on page 5-17.

Handling Platform-Specific Internal Representations

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

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

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

GLS

For your opaque data type to accept an internal representation on non-default locales, you must use the Informix GLS API in the receive and send functions to access Informix locales from within these functions. For more information, see "Handling Locale-Sensitive Data" on page 6-23. ♦

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

- The receive function converts from the client internal representation to the server internal representation.
- The output function converts from server internal representation to the client internal representation.

These support functions do not have to be named *receive* and *send*, but they do have to perform the specified conversions. They should be reciprocal functions; that is, the receive function should produce a value that the send function accepts as an argument and vice versa.

The SENDRECV Data Type

SQL statements support an internal data type called SENDRECV to hold the internal representation of an opaque data type when it is transferred between the client computer and the server computer. The SENDRECV data type allows for any possible change in the size of the data when it is converted between the two representations. The receive and send support functions serve as casting functions between the SENDRECV and opaque data type.

Tip: *The DataBlade API provides the **mi_sendrecv** data type to hold the internal representation of opaque-type data. For more information, see the "DataBlade API Programmer's Manual."*

For more information on how to use SQL statements to register support functions, see page 5-15.

E/C

The SENDRECV data type is not surfaced in INFORMIX-ESQL/C applications. Instead, these applications use **fixed binary** and **var binary** host variables in SQL statements to transfer the internal representation of an opaque type on the client computer. The database server implicitly invokes the receive and send support functions when it receives an SQL statement that contains a **fixed binary** or **var binary** host variable. ♦

Receive Support Function

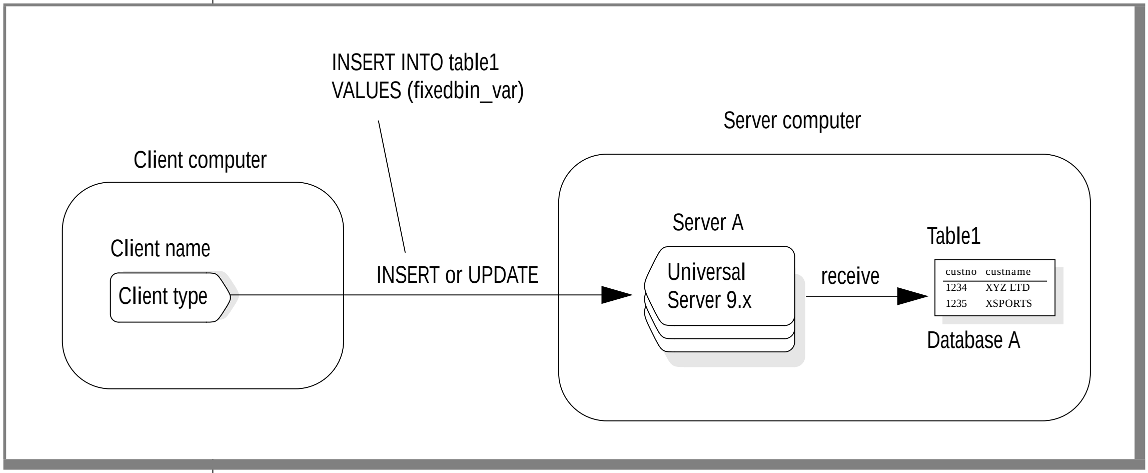
The database server calls the **receive** function when it receives the internal representation of an opaque type from a client application. For example, when a client application issues an INSERT or UPDATE statement, it can send the internal representation of an opaque type to the database server to be stored in a column.

E/C

INFORMIX-ESQL/C applications use the **fixed binary** and **var binary** host variables to send the internal representation of an opaque data type. ♦

Figure 6-3 shows when Universal Server executes the receive support function.

Figure 6-3
Execution of the receive Support Function



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

The receive function takes as an argument an **mi_sendrecv** structure (that holds the internal structure on the client computer) and returns the internal structure for the opaque type (the internal representation on the server computer). The following function signature is for a receive support function of an opaque data type whose internal structure is **ll_longlong_t**:

```
ll_longlong_t * ll_longlong_receive(client_intrnl_format)
    mi_sendrecv *client_intrnl_format;
```

The **ll_longlong_receive()** function is a casting function from the SENDRECV data type to the **ll_longlong_t** internal structure. It must be registered as an implicit casting function with the CREATE IMPLICIT CAST statement. For more information on casting functions, see "Creating the Casts" on page 5-17.

Send Support Function

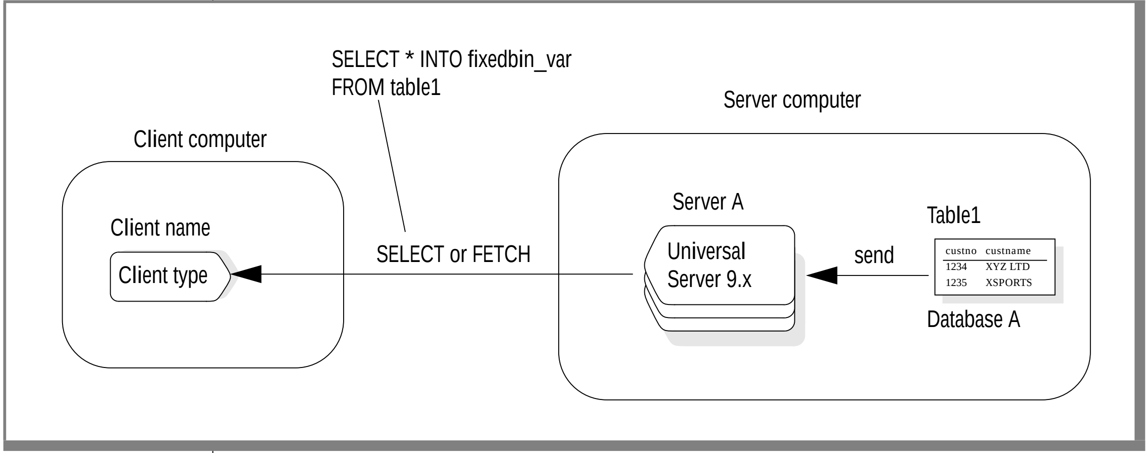
The database server calls the send function when it sends the internal representation of an opaque type to a client application. For example, when a client application issues a SELECT or FETCH statement, it can save the data of an opaque type that it receives from the database server in a host variable that conforms to the internal representation of the opaque type.

E/C

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

Figure 6-4 shows when Universal Server executes the **send** support function.

Figure 6-4
Execution of the send Support Function



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

The send function takes as an argument the internal structure for the opaque type on the server computer and returns an **mi_sendrecv** structure that holds the internal structure on the client computer. The following function signature is for a send support function of an opaque data type whose internal structure is **ll_longlong_t**:

```
mi_sendrecv * ll_longlong_send(srvr_intrnl_format)
     ll_longlong_t *srvr_intrnl_format;
```

The **ll_longlong_send()** function is a casting function from the **ll_longlong_t** internal structure to the SENDRECV data type. It must be registered as an explicit casting function with the CREATE EXPLICIT CAST statement. For more information on casting functions, see "Creating the Casts" on page 5-17.

Performing Bulk Copies

Universal Server can copy data in and out of a database with a bulk copy operation. In a bulk copy, the database server sends large numbers of column values in a copy file, rather than copying each column value individually. For large amounts of data, bulk copying is far more efficient than moving values individually.

The following Informix utilities can perform bulk copies:

- DB-Access performs bulk copies with the LOAD and UNLOAD statements.
- The **dbimport** and **dbexport** utilities perform bulk copies.
- The High Performance Loader (HPL) performs bulk copies.
- The PLOAD utility loads and unloads a database from external files.

Universal Server can perform bulk copies on binary (internal) or character (external) representations of opaque-type data.

Import and Export Support Functions

The import and export support functions perform any tasks needed to process the external representation of an opaque type for a bulk copy. When Universal Server copies data into or out of a database in external format, it calls the following support functions for every value copied to or from the copy file:

- The import function imports textual data by converting from external to internal representation.
- The export function exports textual data by converting from internal to external representation.

These support functions do not have to be named *import* and *export*, but they do have to perform the specified conversions. They should be reciprocal functions; that is, the import function should produce a value that the export function accepts as an argument and vice versa.

The import and export functions can take special actions on the values before they are copied. Typically, only opaque data types that contain smart large objects have import and export functions defined for them. For example, the export function for such a data type would create a file on the client computer, write the smart-large-object data from the database into this file, and send the name of the client file as the data to store in the copy file. Similarly, the import function for such a data type would take the client file name from the copy file, open the client file, and load the large-object data from the copy file into the database. The advantage of this design is that the smart-large-object data does not appear in the copy file; therefore, the copy file grows more slowly and is easier for people to read.

For small opaque data types, you do not usually need to defined the import and export support functions. If they are not defined, the database server uses the input and output functions, respectively, when it performs bulk copies.

The IMPEXP Data Type

SQL statements support an internal data type called IMPEXP to hold the external representation of an opaque data type for a bulk copy. The IMPEXP data type allows for any possible change in the size of the data when it is converted between the two representations. The import and export support functions serve as cast functions between the IMPEXP and opaque data type.

Tip: *The DataBlade API provides the* ***mi_impexp*** *data type to transfer the external representation of opaque-type data for a bulk copy. For more information, see the "DataBlade API Programmer's Manual."*

For more information on how to use SQL statements to register support functions, see page 5-15.

Import

The import function takes as an argument an **mi_impexp** structure (that holds the bulk-copy format of the external representation of the user-defined type) and returns the internal structure for the user-defined type. The following function signature is for an import support function of an opaque data type whose internal structure is **ll_longlong_t**:

```
ll_longlong_t * ll_longlong_import(extrnl_bcopy_format)
    mi_impexp *extrnl_bcopy_format;
```

The **ll_longlong_import()** function is a casting function from the IMPEXP data type to the **ll_longlong_t** data structure. It must be registered as an implicit casting function with the CREATE IMPLICIT CAST statement. For more information on casting functions, see "Creating the Casts" on page 5-17.

Any files that the import function reads must reside on the server computer. If you do not provide an import support function, Universal Server uses the input support function to import textual data.

Export

The export function takes as an argument the internal structure for the opaque type and returns an **mi_impexp** structure that holds the bulk-copy format of the external representation of the opaque type. The following function signature is for an export support function of an opaque data type whose internal structure is **ll_longlong_t**:

```
mi_impexp * ll_longlong_export(intrnl_bcopy_format)
    ll_longlong_t *intrnl_bcopy_format;
```

The **ll_longlong_export()** function is a casting function from the **ll_longlong_t** internal structure to the IMPEXP data type. It must be registered as an explicit casting function with the CREATE EXPLICIT CAST statement. For more information on casting functions, see "Creating the Casts" on page 5-17.

If you do not provide an export support function, Universal Server uses the output support function to export textual data.

Importbinary and Exportbinary Support Functions

The importbinary and exportbinary support functions perform any tasks needed to process the internal (binary) representation of an opaque type for a bulk copy, as follows:

- The importbinary function imports binary data by converting from some binary representation to the internal representation.
- The exportbinary function exports binary data by converting from internal representation to some binary representation.

These support functions do not have to be named *importbinary* and *exportbinary*, but they do have to perform the specified conversions. They should be reciprocal functions; that is, the importbinary function should produce a value that the exportbinary function accepts as an argument and vice versa. The Informix DataBlade API provides functions that support conversion between different internal representations of opaque types.

For opaque data types that have identical external and internal representations, the import and importbinary support functions can be the same function. Similarly, the export and exportbinary support functions can be the same function.

The IMPEXPBIN Data Type

SQL statements support an internal data type called IMPEXPBIN to hold the internal representation of an opaque data type for a bulk copy. The IMPEXPBIN data type allows for any possible change in the size of the data when it is converted between the two representations. The importbinary and exportbinary support functions serve as cast functions between the IMPEXPBIN and opaque data type.

Tip: *The DataBlade API provides the* ***mi_impexpbin*** *data type to transfer the internal representation of opaque-type data for a bulk copy. For more information, see the "DataBlade API Programmer's Manual."*

For more information on how to use SQL statements to register support functions, see page 5-15.

Importbinary

The importbinary function takes as an argument an **mi_impexpbin** structure (that holds the bulk-copy format of the internal format of the opaque type) and returns the internal structure for the opaque type. The following function signature is for an importbinary support function of an opaque data type whose internal structure is **ll_longlong_t**:

```
ll_longlong_t * ll_longlong_importbin(client_intrnl_bcopy_format)
    mi_impexpbin *client_intrnl_bcopy_format;
```

The **ll_longlong_importbin()** function is a casting function from the IMPEXPBIN data type to the **ll_longlong_t** internal structure. It must be registered as an implicit casting function with the CREATE IMPLICIT CAST statement. For more information on casting functions, see "Creating the Casts" on page 5-17.

Any files that the import function reads must reside on the server computer. If you do not provide an importbinary support function, Universal Server imports the binary data in the server internal representation of the opaque data type.

Exportbinary

The exportbinary function takes as an argument the internal structure for the opaque type and returns an **mi_impexpbin** structure that holds the bulk-copy format of the internal representation of the opaque type. The following function signature is for an exportbinary support function of an opaque data type whose internal structure is **ll_longlong_t**:

```
mi_impexpbin * ll_longlong_exportbin(srvr_intrnl_bopy_format)
    ll_longlong_t *srvr_intrnl_bcopy_format;
```

The **ll_longlong_exportbin()** function is a casting function from the **ll_longlong_t** internal structure to the IMPEXPBIN data type. It must be registered as an explicit casting function with the CREATE EXPLICIT CAST statement. For more information on casting functions, see "Creating the Casts" on page 5-17.

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

Inserting and Deleting Data

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

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

A common use of the **assign()** and **destroy()** support functions is for an opaque data type that contains spatial or multirepresentational data. Such a data type might provide a choice of how to store the data: inside the internal structure or, for very large objects, in a smart large object. If the data is stored in a smart large object, the internal structure of the opaque data type contains the LO-pointer structure to identify the location of the data; it does not contain the data itself. The **assign()** support function can make the decision of how to store the data, and the **destroy()** support function can decide how remove the data, regardless of where it is stored. For an example of such an opaque type, see "A Varying-Length Opaque Type: image" on page 5-38.

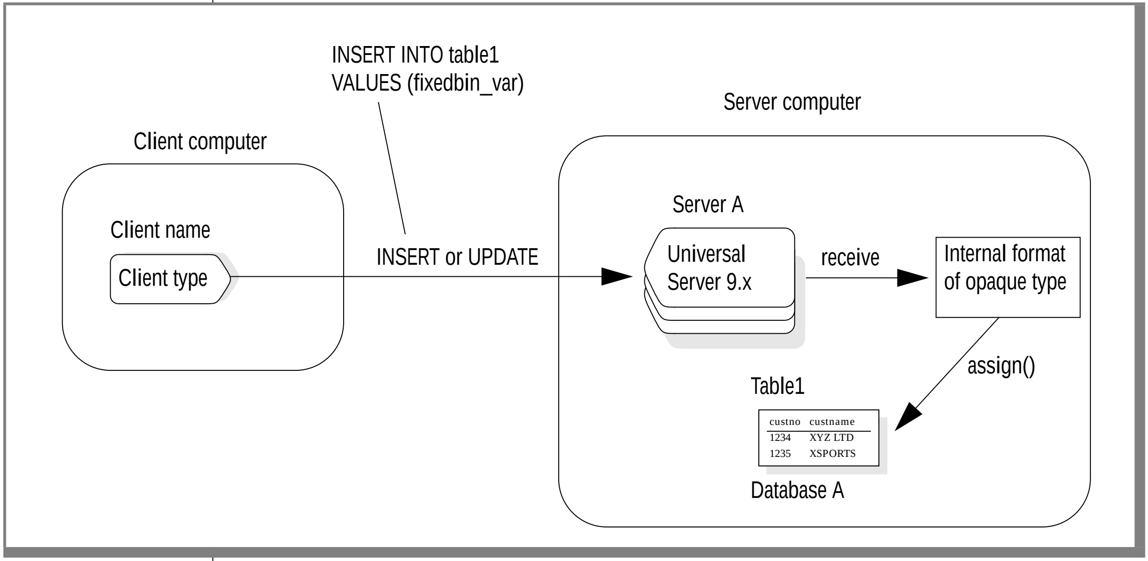
These support functions must be named **assign** and **destroy**, but the names are case insensitive. They must perform the tasks that should occur before the opaque data type is stored to or removed from disk. The **assign()** and **destroy()** functions are required for opaque types that have smart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

The assign() Support Function

Universal Server calls the **assign()** support function just before it stores the internal representation of an opaque type on disk. For example, when a client application issues an INSERT, UPDATE or LOAD statement, the database server calls the **assign()** function before it saves the internal representation of an opaque type in a column.

Figure 6-5 shows when Universal Server executes the **assign()** function.

Figure 6-5
Execution of the assign() Support Function



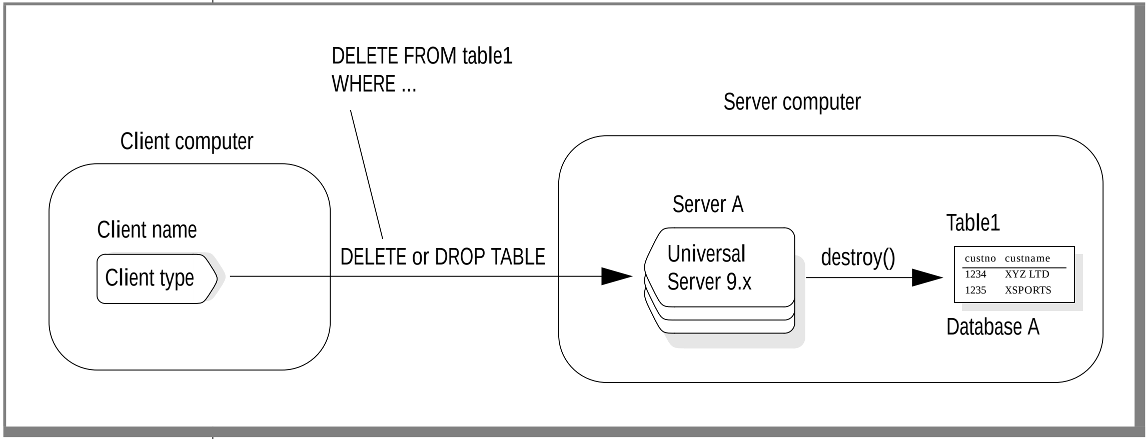
When you store a value of an opaque data type, the **assign()** function takes as an argument the internal structure for the opaque data type and returns the data that is actually stored in the table.

The destroy() Support Function

Universal Server calls the **destroy()** support function just before it removes the internal representation of an opaque type from disk. For example, when a client application issues a DELETE or DROP TABLE statement, the database server calls the **destroy()** function before it deletes an opaque-type value from a column.

Figure 6-6 shows when Universal Server executes the **destroy()** function.

Figure 6-6
Execution of the destroy() Support Function



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

Handling Smart Large Objects

If an opaque data type contains an embedded smart large object, you must define an **lohandles()** support function for the opaque type. The **lohandles()** support function takes an instance of the opaque type and returns a list of the -pointer structures for the smart large objects that are embedded in the data type.

Universal Server uses the **lohandles()** support function whenever it must search opaque-type values for references to smart large objects. Examples of this search include the following:

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

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

- **assign()**
- **destroy()**
- An import function
- An export function
- An importbinary function
- An exportbinary function

For more information on **assign()** and **destroy()** support functions, see "Inserting and Deleting Data" on page 6-19. For information on the import, export, importbinary, and exportbinary support functions, see "Performing Bulk Copies" on page 6-14.

Comparing Data

The **compare()** function is an SQL-invoked function that sorts the target data type. Universal Server uses the **compare()** function to execute the following clauses and keywords of the SELECT statement:

- The ORDER BY clause
- The UNIQUE and DISTINCT keywords
- The UNION keyword

The database server also uses the **compare()** function to evaluate the BETWEEN operator in the condition of an SQL statement. For more information on conditional clauses, see the Condition segment in the *Informix Guide to SQL: Syntax*.

For Universal Server to be able to sort an opaque type, you must define a **compare()** function that handles the opaque type. This **compare()** function must follow these rules:

1. The name of the function must be **compare()**. However, the name is not case sensitive; the **compare()** function is the same as the **Compare()** function.
2. The function must accept two arguments, each of the data type to be compared.

3. The function must return an integer value to indicate the result of the comparison, as follows:
   - <0 to indicate that the first argument is less than (<) the second argument
   - 0 to indicate that the two arguments are equal (=)
   - >0 to indicate that the first argument is greater than (>) the second argument

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

The **compare()** function is the support function for the built-in secondary access method, B-tree. For more information on the built-in secondary access method, see "The Generic B-Tree Index" on page 4-4. For more information on how to customize a secondary access method for an opaque data type, see "Using an Operator Class" on page 4-3.

Handling Locale-Sensitive Data

GLS

An Informix database has a fixed locale per database. This locale, the *database locale*, is attached to the database at the time that the database is created. In any given database, all character data types (such as CHAR, NCHAR, VARCHAR, NVARCHAR, and TEXT) contain data in the code set that the database locale supports.

An opaque data type can hold character data. The following support functions provide the ability to transfer opaque-type data between a client application and the database server:

- The input and output support functions provide the ability to transfer the external representation of the opaque type.
- The receive and send support functions provide the ability to transfer the internal representation of the opaque type.

However, the ability to transfer the data between client application and database server is not sufficient to support locale-sensitive data. It does not ensure that the data is correctly manipulated at each end. You must ensure that both sides of the connection handle the locale-sensitive data, as follows:

- At the client side of the connection, the client application must handle the locale-sensitive data for opaque-type columns correctly.

  It must also have the **CLIENT_LOCALE** environment variable set correctly.
- At the server side of the connection, you must ensure that the appropriate support functions handle the locale-sensitive data.

  In addition, the **DB_LOCALE** and **SERVER_LOCALE** environment variables must be set correctly.

For more information on the **CLIENT_LOCALE**, **DB_LOCALE**, and **SERVER_LOCALE** environment variables, see the *Guide to GLS Functionality*.

To help you write support functions to handle locale-sensitive data, Informix provides the Informix GLS API. The GLS API is a thread-safe library. This library contains C functions that allow your support functions to obtain locale-specific information from Informix GLS locales, including:

- functions to manipulate locale-sensitive data in a portable fashion.
- functions to handle single-byte and multibyte character access.
- functions to manipulate other locale-sensitive data, such as the end-user formats of date, time, or monetary data.

For an overview of the GLS API, see the *Guide to GLS Functionality*. For a description of the GLS API functions, see the *Guide to GLS Functionality*. ♦

In Input and Output Support Functions

The LVARCHAR (and **mi_lvarchar**) data type can hold data in the code set of the client or database locale. This data includes single-byte (ASCII and non-ASCII) and multibyte character data. The LVARCHAR data type holds opaque-type data as it is transferred to and from the database server in its external representation. Therefore, the external representation of an opaque data type can hold single-byte or multibyte data.

However, you must write the input and output support functions to interpret the LVARCHAR data in the correct locale. These support functions might need to perform code-set conversion if the client locale and database locale support different code sets. For more information on code-set conversion, see the *Guide to GLS Functionality*.

In Receive and Send Support Functions

The SENDRECV (and **mi_sendrecv**) data type holds the internal structure of an opaque type. This internal structure can contain the following types of locale-sensitive data:

- Character fields that can hold data in the code set of the client or database locale

  This data includes single-byte (ASCII and non-ASCII) and multibyte character data.
- Monetary, date, or time fields that hold a locale-specific representation of the data

The client application has no way of interpreting the fields of the internal structure because an opaque type is encapsulated.

The SENDRECV data type holds opaque-type data as it is transferred to and from the database server in this internal representation. You must write the receive and send support functions to interpret the locale-specific data within the SENDRECV structure.

Index