

# Ingres 10.0

---

## Object Management Extension User Guide

**INGRES**

ING-10-OME-01

This Documentation is for the end user's informational purposes only and may be subject to change or withdrawal by Ingres Corporation ("Ingres") at any time. This Documentation is the proprietary information of Ingres and is protected by the copyright laws of the United States and international treaties. It is not distributed under a GPL license. You may make printed or electronic copies of this Documentation provided that such copies are for your own internal use and all Ingres copyright notices and legends are affixed to each reproduced copy.

You may publish or distribute this document, in whole or in part, so long as the document remains unchanged and is disseminated with the applicable Ingres software. Any such publication or distribution must be in the same manner and medium as that used by Ingres, e.g., electronic download via website with the software or on a CD-ROM. Any other use, such as any dissemination of printed copies or use of this documentation, in whole or in part, in another publication, requires the prior written consent from an authorized representative of Ingres.

To the extent permitted by applicable law, INGRES PROVIDES THIS DOCUMENTATION "AS IS" WITHOUT WARRANTY OF ANY KIND, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT. IN NO EVENT WILL INGRES BE LIABLE TO THE END USER OR ANY THIRD PARTY FOR ANY LOSS OR DAMAGE, DIRECT OR INDIRECT, FROM THE USER OF THIS DOCUMENTATION, INCLUDING WITHOUT LIMITATION, LOST PROFITS, BUSINESS INTERRUPTION, GOODWILL, OR LOST DATA, EVEN IF INGRES IS EXPRESSLY ADVISED OF SUCH LOSS OR DAMAGE.

The manufacturer of this Documentation is Ingres Corporation.

For government users, the Documentation is delivered with "Restricted Rights" as set forth in 48 C.F.R. Section 12.212, 48 C.F.R. Sections 52.227-19(c)(1) and (2) or DFARS Section 252.227-7013 or applicable successor provisions.

Copyright © 2010 Ingres Corporation. All Rights Reserved.

Ingres, OpenROAD, and EDBC are registered trademarks of Ingres Corporation. All other trademarks, trade names, service marks, and logos referenced herein belong to their respective companies.

# Contents

---

<b>Chapter 1: Introduction</b>	<b>7</b>
Audience .....	7
In This Guide .....	7
System-specific Text in This Guide .....	8
Terminology Use in This Guide .....	8
Syntax Conventions Used in This Guide .....	8
<b>Chapter 2: Introducing Object Management Extension</b>	<b>9</b>
What Is Object Management Extension? .....	9
Structure and Symbol Definitions .....	9
Demonstration Data Types .....	10
How You Add Data Types and Functions .....	10
Required Contents of the Code .....	10
Data Type Definitions .....	11
Function Definitions .....	11
Function Instance Definitions .....	11
Coercion Routines .....	12
Installation and Testing of the New Code .....	13
Object Management Extension Restrictions .....	13
<b>Chapter 3: Understanding DBMS Server Requirements for User-Defined Data Types</b>	<b>15</b>
How the DBMS Server Uses the Code .....	15
Requirements for User-written Code .....	16
Requirements for Data Type Coercion .....	17
IDs for Data Types and Functions .....	18
User-defined Data Types and the Copy Statement .....	18
Large Objects .....	19
<b>Chapter 4: Defining Data Types</b>	<b>21</b>
Data Type Definition .....	21
Structure IIADD_DT_DFN Fields .....	22
Required Routines for Data Type Definition .....	24
Function Parameters .....	25
Structure scb_error .....	26
Structure II_DATA VALUE .....	27

---

The Structure of db_data .....	27
compare Routine—Compare Two Data Elements .....	29
dbtoev Routine—Determine External Data Type .....	30
dhmax Routine—Create Default Maximum Histogram Value .....	32
dhmin Routine—Create Default Minimum Histogram Value .....	33
getempty Routine—Get an Empty Value.....	34
hashprep Routine—Prepare Value for Hash Key.....	35
helem Routine—Create a Histogram Element for Data Value .....	36
hg_dtln Routine—Provide Type and Length for Histogram Value .....	37
hmax Routine—Create Histogram Value for Maximum Value .....	39
hmin Routine—Create Histogram Value for Minimum Value.....	40
keybuild Routine—Build a Key from the Value.....	41
length_check Routine—Check for Valid Length .....	45
minmaxdv Routine—Provide Min/Max Values and Lengths .....	46
seglen Routine—Determine Length of Each Long Segment.....	49
tmcvt Routine—Convert Data Type to Displayable Format .....	50
tmlen Routine—Determine Display Length.....	51
value_check Routine—Check for Valid Values .....	52
xform Routine—Transform Long Types into Segments.....	53

## **Chapter 5: Defining Functions** **55**

Required Definitions.....	55
Structure IIADD_FO_DFN .....	55

## **Chapter 6: Defining Function Instances** **57**

Function Instance Definition.....	57
Structure IIADD_FI_DFN .....	58
Length Definition of Result Data Type.....	61
External Lenspec Routine—Return Result Length of Specified Value .....	62
Complementary Function Instances.....	62
Sorting of the Function Instance Definition Array.....	63
Methods for Defining Function Instances for Large Objects .....	63
Ingres-supplied Filter Functions .....	64
Direct Manipulation of Large Objects.....	66
II_INFORMATION Operation—Return Maximum Length of Peripheral Object Segments .....	68
II_GET Operation—Get Next Segment .....	69
II_PUT Operation—Add a New Segment.....	71
II_COPY Operation—Move a Peripheral Object .....	73

---

## **Chapter 7: Passing Definitions to the DBMS Server** **75**

IIudadt_register Routine .....	75
Structure IIADD_DEFINITION Fields.....	76
Server Routines Provided .....	78
The ii_cb_trace Routine—Output Provided Trace Messages.....	78
The ii_error_fcn Routine—Place Error Information in Status Control Block .....	79
The ii_lo_handler_fcn Routine—Move Through Large Object Segments .....	79
The ii_init_filter_fcn Routine—Set Up Filter Function .....	79
The ii_filter_fcn Routine—Perform Operation by Calling a User Routine .....	80

## **Chapter 8: Installing and Testing Data Types** **81**

How You Install New Data Types in a Windows Environment.....	81
How You Install New Data Types in a VMS Environment .....	81
Template Command File—Create the Shared Image .....	82
II_USERADT Logical—Set Disk Location of the Shared Image .....	83
How You Install New Data Types in a UNIX Environment .....	84
Testing the New Data Type Code .....	85

## **Chapter 9: Using Abstract Spatial Data Types** **87**

Use of Spatial Data Types, Operators, and Functions .....	87
Spatial Data Types in the Spatial Object Library .....	88
Point Data Type .....	89
Box Data Type.....	90
Lseg Data Type .....	91
Line Data Type .....	92
Long Line Data Type.....	93
Polygon Data Type .....	94
Long Polygon Data Type .....	95
Circle Data Type .....	96
Ipoint Data Type.....	97
Ibox Data Type.....	98
Ilsseg Data Type .....	99
Iline Data Type.....	100
Ipolygon Data Type.....	101
Icircle Data Type .....	102
Nbr Data Type.....	103
Spatial Data Types Storage Formats .....	103
Spatial Operators .....	104
Equality Operators .....	105
Binary Spatial Operators .....	105
Overlaps Operator.....	109

---

Nbr Function .....	110
Hilbert Function .....	110
Functions that Support the Spatial Operators.....	110
Spatial Functions .....	111
Spatial Conversion Functions .....	111
Support Routines for Spatial Data Types .....	116
Ordering of Spatial Data Types .....	119
Polygon Length Limits .....	120
How You Install Spatial Data Types in UNIX or Linux Environments.....	121
How You Install Spatial Data Types in a VMS Environment.....	124
How You Install Spatial Data Types in a Windows Environment .....	125
<b>Chapter 10: Writing Aggregate Functions</b> .....	<b>127</b>
Aggregate Function .....	127
Function Definitions for Aggregates.....	128
Code for an Aggregate Function.....	129
<b>Appendix A: Checklist for Creating Data Types</b> .....	<b>131</b>
How You Create Data Types in Windows .....	131
How You Create Data Types in UNIX .....	132
How You Create Data Types in VMS .....	133
<b>Index</b> .....	<b>135</b>

# Chapter 1: Introduction

---

This section contains the following topics:

[Audience](#) (see page 7)

[In This Guide](#) (see page 7)

[System-specific Text in This Guide](#) (see page 8)

[Terminology Use in This Guide](#) (see page 8)

[Syntax Conventions Used in This Guide](#) (see page 8)

This guide provides you with the instructions for using Object Management Extension to add data types and SQL functions to Ingres® and provides the requirements for creating the required source code and procedures for installing your code.

## Audience

This guide assumes that you are an experienced programmer, familiar with the C programming language.

## In This Guide

This guide introduces Object Management Extension and contains information on the following topics:

- Interaction of new data types and SQL functions with the DBMS Server
- Requirements for defining a new data type
- Requirements for defining a new SQL function
- Requirements for defining function instances to support your new data types and functions
- Description of the routine used by the DBMS Server to access the code for new data types and functions
- Instructions for installing and testing your code
- Description of spatial data types

An appendix contains a checklist for defining, installing, and testing a new data type or SQL function.

## System-specific Text in This Guide

This guide provides information that is specific to your operating system, as in these examples:

**UNIX:** This information is specific to the UNIX operating system.

**VMS:** This information is specific to the UNIX operating system.

When necessary for clarity, the symbol ■ is used to indicate the end of the system-specific text.

For sections that pertain to one system only, the system is indicated in the section title.

## Terminology Use in This Guide

The documentation uses the following terminology:

A *command* is an operation that you execute at the operating system level.

A *statement* is an operation that you embed within a program or execute interactively from a terminal monitor.

A statement can be written in Ingres 4GL, a host programming language (such as C), or a database query language (SQL or QUEL).

## Syntax Conventions Used in This Guide

This guide uses the following conventions to describe syntax:

Convention	Usage
Monospace	Indicates keywords, symbols, or punctuation that you must enter as shown
<i>Italics</i>	Represent a variable name for which you must supply an actual value
[ ] (brackets)	Indicate an optional item
{ } (braces)	Indicate an optional item that you can repeat as many times as appropriate
(vertical bar)	Separates items in a list and indicates that you must choose one item

## Chapter 2: Introducing Object Management Extension

---

This section contains the following topics:

- [What Is Object Management Extension?](#) (see page 9)
- [How You Add Data Types and Functions](#) (see page 10)
- [Object Management Extension Restrictions](#) (see page 13)

This chapter introduces Object Management Extension and outlines the general tasks you must perform to add data types and functions to Ingres.

### What Is Object Management Extension?

Object Management Extension is an option that enables you to add data types and SQL functions to the DBMS Server.

You can use a user-defined data type in any context in which you can use a standard Ingres data type, including with Ingres Star. You can use user-defined SQL functions in queries to manipulate both user-defined data types and standard SQL data types. To support your new data types and functions, you can add new capabilities to existing SQL comparison and arithmetic operators.

### Structure and Symbol Definitions

You must write all the code that defines and manipulates your new data types. The header file, `IIADD.H`, which you can include in your C program, contains the structure and symbol definitions that the DBMS Server requires. `IIADD.H` is located in the following directories:

**Windows:** `%II_SYSTEM%\ingres\files\iiadd.h`

**UNIX:** `$II_SYSTEM/ingres/files/iiadd.h`

**VMS:** `II_SYSTEM:[INGRES.FILES]IIADD.H`

## Demonstration Data Types

Object Management Extension includes two demonstration data types:

- `ord_pair`

This demo creates an ordered pair data type, which consists of a simple pair of x and y coordinates. The code is located in the following files:

**Windows:** %II\_SYSTEM%\ingres\demo\udadts\op.c

**UNIX:** \$II\_SYSTEM/ingres/demo/udadts/op.c

**VMS:** II\_SYSTEM:[INGRES.DEMO.UDADTS]OP.C

- `int_list`

This demo creates a data type consisting of a variable-length list of 4-byte integers. The code is located in the following files:

Windows: %II\_SYSTEM%\ingres\demo\udadts\intlist.c

**UNIX:** \$II\_SYSTEM/ingres/demo/udadts/intlist.c

**VMS:** II\_SYSTEM:[INGRES.DEMO.UDADTS]INTLIST.C

## How You Add Data Types and Functions

To add data types and functions you must:

- Write the code that tells the DBMS Server what the new data type or function is and how to manipulate it
- Install the code in a test installation and debug the code
- Install the code in the target installation

## Required Contents of the Code

You must write the code that the DBMS Server uses to manipulate a new data type or SQL function. This code consists of

- Data type or function definition
- Function instance definitions
- Required coercion routines

## Data Type Definitions

A data type definition consists of:

- Name of the data type
- Internal data type identifier (a 2-byte integer)
- Data type status flags
- An underlying data type (if necessary)
- Routines required by the DBMS Server to manipulate the data type

## Function Definitions

A function definition specifies the names of the functions that are used to invoke operations. The definition of a function consists of:

- Function name
- Function identifier (a 2-byte integer)
- Type of operation invoked by the function

## Function Instance Definitions

A function instance definition defines the use of a function or operator in a particular context. Because the same function can be used differently with different data types, each instance of use must be defined. For example, different function instances exist for the + operator: one to add integers, one to concatenate text, one to add money, and so on.

When you are defining a new data type, you must also define a function instance for each function or operator that you use with the new data type. If, for example, you want to add two values of the new data type, you must first define a function instance for the + operator used with the new data types. When you define a new SQL function, you must define a function instance for its use with each data type with which it is used.

The definition of a function instance is composed of:

- Function instance identifier (a 2-byte integer)
- Function instance complement identifier (if this instance is a comparison)
- Operator or function identifier for which this is an instance
- Operator type
- Function instance status flags
- Number of arguments and their data types
- Workspace length
- Data type of the result
- Length of the result
- Address of the routine that performs the instance

**Note:** Object Management Extension does not allow you to redefine the use of the standard operators with existing SQL data types and functions. If you try to do this, you do not receive an error, but the code is ignored.

## Coercion Routines

You must provide the data type coercion routines that support your new data type. For a list and discussion of the required coercion routines, see the chapter "User-Defined Data Types and the DBMS Server Environment."

## Installation and Testing of the New Code

After you have written all of the routines necessary to define and manipulate the new data types and functions, you must create the routine `IIudadt_register`.

**Windows:** After `IIudadt_register` is created, you must rebuild the `IILIBUDT.DLL`. The `IIudadt_register` function serves as the entry point to the `IILIBUDT` Dynamic Linked Library that gives the DBMS Server access to the new data types and functions. ❏

**UNIX:** After `IIudadt_register` is created, you must relink the images to give the DBMS Server access to the new data types and functions. ❏

**VMS:** `IIudadt_register` serves as the entry point to the shared image that the various components of the DBMS Server use to access the new data types and functions. ❏

For information about these steps in the procedure, see the chapter "Passing Definitions to the DBMS Server" and the chapter "Installing and Testing Data Types."

Install your new code in a test installation and test it thoroughly before moving it to the target installation.

## Object Management Extension Restrictions

The following restrictions apply to Object Management Extension:

- You can add a maximum of 128 new data types.
- You cannot export user-defined data types from the DBMS Server.

For this reason, the Ingres user interfaces (such as Query-By-Forms or Report-By-Forms) cannot manipulate user-defined data types in their "natural" state. You must choose a standard SQL data type to represent the new data types for the user interface programs and write the necessary coercion routines.

**Note:** The copy statement allows the byte representation of the data to be exported. However, copy handles this data as char. To avoid errors caused by network character translation, avoid moving the data to a different machine using Ingres Net.



# Chapter 3: Understanding DBMS Server Requirements for User-Defined Data Types

---

This section contains the following topics:

[How the DBMS Server Uses the Code](#) (see page 15)

[Requirements for User-written Code](#) (see page 16)

[Requirements for Data Type Coercion](#) (see page 17)

[IDs for Data Types and Functions](#) (see page 18)

[User-defined Data Types and the Copy Statement](#) (see page 18)

[Large Objects](#) (see page 19)

This chapter describes general requirements for the coding of your user-defined data types.

## How the DBMS Server Uses the Code

The code that supports a user-defined data type or SQL function runs as part of the DBMS Server and is available to the entire installation. The DBMS Server, Recovery, and Archiver processes access this code after the code has been properly installed.

**Important!** A bug in your code—a memory access violation, in particular—can damage the integrity of data or the operating environment of the DBMS Server.

## Requirements for User-written Code

The routines that you write must be written in a language which conforms to the calling and operational conventions of the C language. If you use a language other than C, be sure that the structures you build have the same structure, on a field-by-field basis, as those defined in the header file, IIADD.H. For the location of this header file, see the chapter "Introducing Object Management Extension."

To avoid problems, your code must follow these guidelines:

The code must not make any system calls or perform any other operation, such as generating exceptions or signals, which alter the flow of control within the server.

The code must not perform any operations such as memory allocations or operations such as disk I/O, which cause the process to be suspended.

The code must perform only computations on the data provided by the DBMS Server.

User-defined data types are visible only to the DBMS Server. When a user-defined data type is returned to an Ingres tool such as Query-By-Forms, the DBMS Server must convert your new data type to a standard SQL data type. To specify the external data type, use the dbtoev routine. For more information, see dbtoev in the "Defining Data Types" chapter.

## Requirements for Data Type Coercion

When you define a new data type, you must also define the coercion routines that support the data type. The following coercion routines are required:

- A coercion from any data type to the same data type.

This coercion routine is used by the DBMS Server in variety of instances, for example, during retrieve into and create table as select statements or when changing the length of a data type.

- A coercion from the new data type to the SQL data type that represents this data in user interface applications (such as Query-By-Forms or Report-By-Forms).

User interface applications do not accept user-defined data types directly. Therefore, you must provide a coercion routine to convert your data type to a standard SQL data type for external representation. This coercion routine must be compatible with the input coercion routine as follows:

The output of this coercion routine must be acceptable as input to the input coercion routine. The output of the input coercion routine must be acceptable as input to this coercion routine.

An example of a coercion routine for converting from an internal to an external representation is shown here:

```
sprintf(obuf, "%11.3f, %11.3f", point.x, point.y);
```

- An input coercion.

This coercion must convert data coming from an SQL insert or update query to internal representation. Data represented as character strings in Embedded SQL is sent to the DBMS Server as the varchar data type. At least one input coercion routine, which converts varchar to your data type, is required. The input coercion routine must be compatible with the external coercion routine as follows:

The output of this coercion routine must be acceptable as input to the output coercion routine. The output of the output coercion routine must be acceptable as input to this coercion routine.

An example of a coercion routine for converting from an external to an internal representation is shown here:

```
sscanf(inbuffer, "(%F, %F%1s %1s", &point.x, &point.y,
        end_paren, junk);
```

- Coercions to and from the II\_LONGTEXT data type.

This data type is similar to varchar but is used in different contexts. For example, it is used to display data from various utility programs, such as auditdb.

## IDs for Data Types and Functions

You must assign an identification number to all data types, functions, and function instances when they are created. The ID numbers for each type of object (data types, functions, and function instances) occupy a number space that is unique for that object type. An object's ID must be unique within the same class of objects, but does not have to be unique across object classes. For example, a new data type cannot have an ID that is the same as another data type, but it can have an ID number that is the same as a new function's ID number.

Note: For upward compatibility, make identification numbers for all objects unique across machines.

Identification numbers are stored in system catalogs in databases. You cannot change an ID number without first removing all references to both the ID and the object it represents from the catalogs. All tables, views, database procedures, grants, permits, integrities, and rules that refer to the object must also be destroyed.

Following are the ranges of values that are reserved for IDs and the corresponding data types:

SQL data types: 1 to 8191  
Spatial data types: 8192 to 16383  
User-defined data types: 16384 to 32767

## User-defined Data Types and the Copy Statement

When you use the SQL copy statement with user-defined data type, be aware of the following points:

- The copy statement interprets user-defined data types using the char data type.
- Copying user-defined data types across different machines can produce unexpected results.
- Use bulk copy statement or a field format that specifies the length of the user-defined field. Copying user-defined data types using a format other than fixed or bulk format can produce unexpected results.
- Do not use character delimiters such as colon, newline, or tab with user-defined data types.

## Large Objects

Object Management Extension allows you to create data types that can accommodate objects that exceed the size limits for native data types. These objects are called large objects or peripheral objects. The name peripheral object indicates that these objects are stored outside of the table in which they are declared.

Large objects are stored as a number of segments, each of which is of some simple and small data type. This type is called the underlying data type for the large object. See Structure `IIADD_DT_DFN` Fields. Operations performed on large objects generally operate on a segment at a time.

Large objects can be up to two gigabytes in size. The code does not attempt to materialize the entire object in memory—rather, large objects are represented in memory using the `II_PERIPHERAL` data structure. The `II_PERIPHERAL` structure contains two fields. The first field is the tag and it contains the “style” of the large object. The style can be either real data or coupon. If the tag is seen by the Object Management Extension routines, the value indicates that the large object is a coupon. The coupon status indicates that only the information necessary to find and collect the object is present. At other times during the tag's life, it can contain different values, but these are not seen by the Object Management Extension code. (For example, the tag can have a variety of values that indicate how it is transmitted across the communication link.) The second field contains the length of the object. This field is an eight byte integer, although only four bytes are currently used (`per_length1`).

Large objects cannot be used as table keys, do not have histograms computed, and cannot be sorted. You must specify these restrictions as data type attributes along with the `PERIPHERAL` attribute at the time the data type is declared. See the `dtd_attributes` field in Structure `IIADD_DT_DFN` Fields.

Because of these restrictions, only a few of the required functions are necessary for peripheral objects. Only the `length_check`, `getempty`, `tmlen`, `tmcvt`, `dbtoev`, `seglen`, `value_check`, and `xform` routines must be provided. However, these attributes are independent from the peripheral attribute in the sense that the lack of histogram capability can be specified for non-peripheral objects. If so specified, the histogram routines need not be provided.



# Chapter 4: Defining Data Types

---

This section contains the following topics:

- [Data Type Definition](#) (see page 21)
- [Structure IIADD\\_DT\\_DFN Fields](#) (see page 22)
- [Required Routines for Data Type Definition](#) (see page 24)
- [compare Routine—Compare Two Data Elements](#) (see page 29)
- [dbtoev Routine—Determine External Data Type](#) (see page 30)
- [dhmax Routine—Create Default Maximum Histogram Value](#) (see page 32)
- [dhmin Routine—Create Default Minimum Histogram Value](#) (see page 33)
- [getempty Routine—Get an Empty Value](#) (see page 34)
- [hashprep Routine—Prepare Value for Hash Key](#) (see page 35)
- [helem Routine—Create a Histogram Element for Data Value](#) (see page 36)
- [hg\\_dtlN Routine—Provide Type and Length for Histogram Value](#) (see page 37)
- [hmax Routine—Create Histogram Value for Maximum Value](#) (see page 39)
- [hmin Routine—Create Histogram Value for Minimum Value](#) (see page 40)
- [keybuild Routine—Build a Key from the Value](#) (see page 41)
- [length\\_check Routine—Check for Valid Length](#) (see page 45)
- [minmaxdv Routine—Provide Min/Max Values and Lengths](#) (see page 46)
- [seglen Routine—Determine Length of Each Long Segment](#) (see page 49)
- [tmcvt Routine—Convert Data Type to Displayable Format](#) (see page 50)
- [tmlen Routine—Determine Display Length](#) (see page 51)
- [value\\_check Routine—Check for Valid Values](#) (see page 52)
- [xform Routine—Transform Long Types into Segments](#) (see page 53)

This chapter describes the data type definition and the routines that are required for a data type.

## Data Type Definition

The DBMS Server requires a data type definition for each user-defined abstract data type. This definition specifies the name, length, and ID of the new data type. The definition also points to the routines that manage and manipulate the new data type.

## Structure IIADD\_DT\_DFN Fields

The fields of the structure IIADD\_DT\_DFN compose the data type definition. The first five fields specify the name, length, ID of the new data type, its underlying type, and its attributes. These fields are as follows:

### **dtd\_object\_type**

Contains the value (hex 210) specified by II\_O\_DATATYPE.

### **dtd\_name**

Specifies the name of the new data type. The name must be a character string with a maximum length of 32 bytes. If the string is less than 32 bytes, it must be null terminated. For example, the data type char is specified as 'char\0'.

### **dtd\_id**

Specifies the data type identifier. It is a 2-byte integer field. The ID must be a value between the values represented by ADD\_LOW\_USER and ADD\_HIGH\_USER, 16384 and 16511 respectively. This field cannot be altered once the data type is in use.

### **dtd\_underlying\_id**

Specifies the data type ID, used to store large object segments. This field is used only when the II\_DT\_PERIPHERAL attribute is set.

### **dtd\_attributes**

Specifies the attributes of the data type. If none of the attributes are necessary or appropriate, then set this field to II\_DT\_NOBITS (bits are described in the following table).

Large objects do not have any inherent sort order, cannot be used as keys for tables, and cannot have histograms. To specify the attributes in the `dtd_attributes` field, use the constants listed here:

**II\_DT\_NOBITS**

Indicates that the data type has no specific attributes.

**II\_DT\_NOKEY**

Indicates that the data type cannot be specified as a key column in a `modify` or `create index` statement.

**II\_DT\_NOSORT**

Indicates that the data type cannot be specified as the target of a `sort by` or `order by` clause in a query, nor can the DBMS Server sort on this data type during the execution of a query. Data types tagged with this bit may have no inherent sort order. They are simply marked as “different” by the sort comparison routine.

**II\_DT\_NOHISTOGRAM**

Indicates that histograms cannot be constructed for this data type.

**II\_DT\_PERIPHERAL**

Indicates that the data type is stored outside of the basic table format. This means that the table itself can contain either the full data element or it can contain a “coupon” that can be redeemed later to obtain the actual data type.

The data representing a peripheral data type is always represented by an `II_PERIPHERAL` structure. This structure represents the union of the `II_COUPON` structure and a byte stream (an array of 1 byte/char). This data structure also contains a flag indicating whether this is the real thing or the coupon.

**II\_DT\_VARIABLE\_LEN**

Indicates that the data type can be specified as occurring a maximum number of times. If compressed, only the actual number of occurrences is saved, thereby saving disk storage.

The remaining fields in the `IIADD_DT_DFN` structure are filled with the addresses of the required routines that manipulate the data type. The Required Routines for Data Type Definition section lists these routines and the common characteristics that they share. The remainder of the chapter describes each routine in detail.

## Required Routines for Data Type Definition

For each data type that you add, you must provide the following routines:

- compare
- dbtoev
- dhmax
- dhmin
- getempty
- hashprep
- helem
- hg\_dtl
- hmax
- hmin
- keybuild
- length\_check
- minmaxdv
- seglen
- tmcvt
- tmlen
- value\_check
- xform

These routines are called using the following syntax:

```
status = fid_routine(scb, [arg1 [, arg2]], result)
```

The routines must return a 4-byte integer of type `II_STATUS` whose value represents the overall result of the function. A value of 0 (`II_OK`) means successful completion. A value of 5 (`II_ERROR`) means an error. In the case of an error, the query execution is terminated.

## Function Parameters

Each function has from two to four parameters.

The first parameter must be a Session Control Block. This structure contains information used by the upper layers of the DBMS data type subsystem. Inside the Session Control Block is a structure named `scb_error` of the type `II_ERR_STRUCT`. `Scb_error` (see page 26) must be filled in when the function returns an error.

The last parameter must be a pointer to a result of some type. The result structure can be a single element, such as an integer, containing some sort of indicator, but most often it is an `II_DATA_VALUE` (defined below).

The result can contain valid data that specifies some portion of the work to be done. This is often done when the routine is creating a portion of a value and another portion of the value is created elsewhere. For example, this is done if `getempty()` was providing the data and the actual type and length were being created elsewhere.

The two optional parameters (`arg1` and `arg2`) are pointers to `II_DATA_VALUE` (see page 27) structures that describe the data values manipulated by the routine. Each function uses `II_DATA_VALUE` structures.

## Structure `scb_error`

A structure named `scb_error` of the type `II_ERR_STRUCT` is inside the Session Control Block. `Scb_error` must be filled in whenever the function returns an error.

The fields of the `scb_error` structure are as follows:

**`er_errcode`**

Contains the error code that identifies the error to the calling facility.

**`er_class`**

Contains the value `II_EXTERNAL_ERROR` (3).

**`er_usererr`**

Contains the user error code. This is the same value as that in `er_errcode`.

**`er_sqlstate_err`**

Must contain a valid `SQLSTATE` error code. For details about `SQLSTATE`, see the `SQL Reference Guide`.

**`er_ebuflen`**

Sent by the DBMS Server to specify the size of the buffer pointed to by `er_errmsgp`.

**`er_emsglen`**

Sent back to the DBMS Server to indicate the length of the formatted error message that was placed in the buffer to which `er_errmsgp` points.

**`er_errmsgp`**

Contains a pointer to a buffer where a formatted message can be placed. If this pointer is `NULL` (0), then no message can be provided.

## Structure II\_DATA\_VALUE

The fields in the II\_DATA\_VALUE structure describe the data values manipulated by the routine. The fields are as follows:

### **db\_datatype**

Type identifier of the data.

### **db\_length**

Length of the data.

### **db\_data**

Pointer to the actual data (if appropriate). This data may not be aligned as required by your machine, and you may need to align the data for correct operation.

The structure of db\_data will vary, as described in The Structure of db\_data (see page 27).

### **db\_prec**

Precision of the data value. For most data types, this is not needed, and should be ignored on input and set to 0 for output.

For DECIMAL, the high order byte will represent the value's precision (total number of significant digits), and the low order byte will hold the value's scale (the number of these digits that are to the right of an implied decimal point.)

## The Structure of db\_data

The db\_data field in the II\_DATA\_VALUE structure is a pointer to the data. The exact structure of db\_data will vary according to the data type. For standard SQL data types it will be the equivalent C data structure (see the *Embedded SQL Companion Guide*).

The decimal data type (db\_datatype II\_DECIMAL) structure is described in Internal Structure of a Decimal Value (see page 28).

### Internal Structure of a Decimal Value

A DECIMAL of precision 'P' is an array of unsigned chars of size  $(1 + \text{INT}(\text{Precision}/2))$  where the last nibble is the sign and preceding nibbles are the digits of the number (0x0-0x9) including leading zeros.

The size of the array is defined for the declared size of the variable, not the size actually used (leading and trailing zeros are stored). If there is an even number of digits, the first nibble in the array is unused.

The position of the decimal point cannot be determined from the DECIMAL data itself; it is specified in the metadata ('scale').

The Sign nibble uses values 0xa, 0xc, 0xe, and 0xf for positive numbers (0xc being preferred) and 0xb and 0xd for negative numbers (0xd being preferred).

Here is a sample array:

**Decimal(-9876.5, 5, 1)**

9	8	7	6	5	d
---	---	---	---	---	---

**Decimal(-9876.5, 6, 1)**

	0	9	8	7	6	5	d
--	---	---	---	---	---	---	---

## compare Routine—Compare Two Data Elements

This routine compares values of two user-defined data types. If the `II_DT_NOSORT` attribute is provided in the `dtd_attributes` field, then the compare routine is not necessary.

The input arguments are `II_DATA_VALUE` pointers to the two data elements being compared. The data elements must be of the same type. The final argument, `result`, must be set to be a negative number, 0, or a positive, non-zero number depending on whether the first argument is less than, equal to, or greater than the second argument. That is,

```
if arg1 < arg2,      result is negative
else if arg1 == arg2 result equals 0
else                 result is non-zero, positive
```

The address of this routine must be placed in the `dtd_compare_addr` field of the `IIADD_DT_DFN` structure.

### Inputs

The inputs for this function are:

**scb**

Pointer to Session Control Block

**op1**

First operand. Pointer to a `II_DATA_VALUE` structures which contains the values to be compared.

**op2**

Second operand. Pointer to a `II_DATA_VALUE` structures which contains the values to be compared.

**result**

Pointer to integer to contain the result of the operation

## Outputs

The outputs for this function are:

### **\*result**

Filled with the result of the operations. This routine is set \*result as follows:

< 0 if op1 < op2

> 0 if op1 > op2

= 0 if op1 is equal to op2

## Returns

II\_STATUS

# dbtoev Routine—Determine External Data Type

The dbtoev routine determines the external data type to which a user-defined data type is converted.

This routine returns the external type specification for the input data type. A coercion (function instance) must be defined to convert the input data type to the given output data type and length.

This routine is called by the DBMS Server to determine how to pass a non-exportable data type to an Ingres tool as the result of a select or fetch statement. This routine sets the ev\_value field to the external data type and length for the specified user-defined data type. You must place the address of this routine in the dtd\_dbtoev\_addr field of the IIADD\_DT\_DFN structure.

The output data type (db\_datatype) must be an SQL data type. Valid values are:

- II\_INTEGER
- II\_DECIMAL
- II\_FLOAT
- II\_C
- II\_CHAR
- II\_VARCHAR
- II\_TEXT

The copy SQL statement does not use this interface. User-defined (and other non-exportable) data types are returned in their original state, as char data of the appropriate length.

## Inputs

The inputs for this function are:

**scb**

Pointer to an SCB

**db\_value**

Ptr to II\_DATA\_VALUE for database type

**ev\_value**

Ptr to II\_DATA\_VALUE for export type

## Outputs

The outputs for this function are:

**\*ev\_value**

Filled in as follows:

**db\_datatype**

Type of export value. See the description for a list of valid values for this field.

**db\_length**

Length of export value

**db\_prec**

Must be 0

## Returns

II\_STATUS

## dhmax Routine—Create Default Maximum Histogram Value

This routine creates the default maximum histogram value. The default histogram values are used by the optimizer when no histogram data is present in the system catalogs. For a discussion of creating a default histogram routine, see dhmin Routine. If the II\_DT\_NOHISTOGRAM attribute is set, then this routine is not necessary.

This routine and the hmax routine form a pair, similar to the pair hmin and dhmin, except that hmax and dhmax deal with the maximum and default maximums, respectively, instead of the minimums.

Place the address of this routine in the dtd\_dhmax\_addr field of the IIADD\_DT\_DFN structure.

### Inputs

The inputs for this function are:

**scb**

Pointer to an SCB

**dv\_from**

Pointer to a datavalue containing the type for the value

**dv\_histogram**

Pointer to a datavalue for the histogram

### Outputs

The outputs for this function are:

**\*(dv\_histogram-db\_data)**

Filled with the histogram value

### Returns

II\_STATUS

## dhmin Routine—Create Default Minimum Histogram Value

This routine creates the minimum default histogram value. You must place the address of this routine in the `dtd_dhmin_addr` field of the `IIADD_DT_DFN` structure. If the `II_DT_NOHISTOGRAM` attribute is present, then this routine is not necessary.

The default histogram values are used by the optimizer when no histogram data is present in the system catalogs. (Optimizedb, which creates statistics for use in histograms, cannot be run on user-defined data types.)

This routine differs from the `hmin` routine in that `hmin` provides the histogram for the smallest possible value, whereas `dhmin` provides the histogram for the smallest “usual” value.

No values are provided to this routine—you must determine what the minimum and maximum default values are. For example, if a data type is being used to store temperatures, a valid range is probably absolute 0 to some very high number. However, a reasonable default minimum and maximum, indicating the range used by most queries, is probably -20 degrees (F) to +120 degrees (F) for temperatures in the continental US.

### Inputs

The inputs for this function are:

**scb**

Pointer to a SCB

**dv\_from**

Pointer to a datavalue containing the type and length for the value

**dv\_histogram**

Pointer to a datavalue for the histogram

### Outputs

The outputs for this function are:

**\*(dv\_histogram-db\_data)**

Filled with the histogram value

### Returns

II\_STATUS

## getempty Routine—Get an Empty Value

This routine constructs the given empty value for this data type. 'Empty value' refers to the default value for a data type. (For example, the getempty routine for an integer creates the value 0.) NULLs are handled transparently, outside of this routine.

Place the address of this routine in the `dtd_getempty_addr` field of the `IIADD_DT_DFN` structure.

### Inputs

The inputs for this function are:

**scb**

Pointer to a SCB.

**empty\_dv**

Pointer to `II_DATA_VALUE` in which to place the empty data value:

**db\_datatype**

The data type for the empty data value.

**db\_length**

The length for the empty data value.

**db\_data**

Pointer to location to place the `db_data` field for the empty data value. Note that this is often a pointer into a tuple.

### Outputs

The outputs for this function are:

**\*(empty\_dv-> db\_data)**

The data for the empty data value is entered.

### Returns

`II_STATUS`

## hashprep Routine—Prepare Value for Hash Key

This routine prepares a data value for becoming a hash key. Place the address of this routine in the `dtd_hashprep_addr` field of the `IIADD_DT_DFN` structure. If the `II_DT_NOKEY` attribute is present, then this routine is not necessary.

For most data types, hash key preparation is a simple copy operation, copying the input data to the output. However, some data types may require more processing. For example, character data types may require blank removal or case translation.

The DBMS Server hash algorithm treats the hash key as a simple byte stream. It does not make allowances for the special characteristics of a data type. You must normalize any variable-length data types within this routine. Unused space must be initialized to some known value. For example, character strings are typically padded with blanks. You must also ensure that there are no compiler-generated holes in your data type. Holes can occur when a compiler pads a structure definition for alignment.

This routine must transform any two values of a data type that compare as equal (using the compare routine) into identical byte streams.

### Inputs

The inputs for this function are:

**scb**

Pointer to a SCB

**dv\_from**

Pointer to an `II_DATA_VALUE` for value to be keyed upon.

**dv\_key**

Pointer to an `II_DATA_VALUE` that contains the key.

### Outputs

The outputs for this function are:

**\*dv\_key**

**db\_length**

The length of the key

**db\_data**

The key value

### Returns

`II_STATUS`

## helem Routine—Create a Histogram Element for Data Value

This routine creates a histogram value for a data element. Place the address of this routine in the `dtd_helem_addr` field of the `IIADD_DT_DFN` structure. If the `II_DT_NOHISTOGRAM` attribute is present, then this routine is not necessary.

A histogram value is a representation of the data element. The DBMS query optimizer uses histogram values in the evaluation of query plans. The optimizer restricts the length of the histogram value to 8.

The comparison of the histograms of two values must match the comparison of their respective values, because the histogram value definition (if  $a < b$ , then  $h(a) < h(b)$ ) assumes that 'a < b' uses the same compare routine. Histograms values have a type-for details, see the description `hg_dtIn` Routine.

### Inputs

The inputs for this function are:

**scb**

Pointer to SCB

**dv\_from**

Value for which a histogram is desired

**dv\_histogram**

Pointer to data value into which to place the histogram value

**db\_datatype**

Contains the type of the histogram value

**db\_length**

Contains the length of the histogram value

**db\_data**

Pointer to space of (`db_length`) bytes into which the histogram value is placed

### Outputs

The outputs for this function are:

**\*(dv\_histogram-> db\_data)**

Contains the histogram value

### Returns

II\_STATUS

## hg\_dtln Routine—Provide Type and Length for Histogram Value

The hg\_dtln routine provides the data type and length for a histogram value for a given data type.

This routine builds a datavalue, dv\_histogram, which describes the data type and length of the histogram value for the data type specified in the input dv\_from. Place the address of this routine in the dtd\_hg\_dtln\_addr field of the IIADD\_DT\_DFN structure. If the II\_DT\_NOHISTOGRAM attribute is present, then this routine is not necessary.

### Inputs

The inputs for this function are:

**scb**

Pointer to an SCB

**dv\_from**

Datavalue describing the data type:

**db\_datatype**

Data type name

**db\_length**

Length of the data type

**dv\_histogram**

Pointer to the datavalue provided to describe the histogram value

## Outputs

The outputs for this function are:

### **dv\_histogram**

Filled with the required type and length

### **db\_datatype**

The data type

### **db\_length**

The data length

### **db\_prec**

Must be 0

## Returns

II\_STATUS

## hmax Routine—Create Histogram Value for Maximum Value

This routine is used by the optimizer to obtain the histogram value for the largest value of a type. Place the address of this routine in the `dtd_hmax_addr` field of the `IIADD_DT_DFN` structure. If the `II_DT_NOHISTOGRAM` attribute is present, then this routine is not necessary.

### Inputs

The inputs for this function are:

**scb**

Pointer to an SCB

**dv\_from**

Pointer to a datavalue describing the type and length of the desired histogram value

**dv\_histogram**

Pointer to a datavalue for the histogram

### Outputs

The outputs for this function are:

**\*(dv\_histogram-db\_data)**

Contains the histogram value

### Returns

II\_STATUS

## hmin Routine—Create Histogram Value for Minimum Value

This routine is used by the optimizer to obtain the histogram value for the smallest value of a type. For a discussion of histograms, see helem Routine.

**Note:** The smallest value for the given data type is expected to be known implicitly by the routine.

Place the address of this routine in the `dtd_hmim_addr` field of the `IIADD_DT_DFN` structure. If the `II_DT_NOHISTOGRAM` attribute is present, then this routine is not necessary.

### Inputs

The inputs for this function are:

**scb**

Pointer to SCB

**dv\_from**

Pointer to a datavalue describing the type and length of the user-typed value.

**dv\_histogram**

Pointer to a datavalue for the histogram

### Outputs

The outputs for this function are:

**\*(dv\_histogram-db\_data)**

Contains the histogram value

### Returns

II\_STATUS

## keybuild Routine—Build a Key from the Value

The keybuild routine builds an isam, B-tree, or hash key from the value.

This routine constructs a key pair for use by the system. Place the address of this routine in the `dtd_keybld_addr` field of the `IIADD_DT_DFN` structure. If the `II_DT_NOKEY` attribute is present, then this routine is not necessary.

A key pair consists of a high-key/low-key combination whose values represent the largest and smallest values that match the key, respectively. The key pair that results from this routine is based upon the type of key desired.

The DBMS query optimizer uses this operation for building keys for traversing hash, isam, or B-tree tables. Whenever the DBMS Server must look up a value in a table using an ordered index (either hash, isam, or B-tree), it uses that value to form two other values. These two values represent the 'key', that is, the upper and lower limits of the search space. It is not guaranteed that all values in the relation matching the value are between the upper and lower limits produced by `keybuild()`.

Along with the value being keyed on, the caller of `keybuild` must specify the comparison operator being used (for example, '<'). One of the input parameters for this routine, `.adc_opkey`, represents the type of operation for which this key is being built. The possible values for this parameter and the operators they represent are:

- `II_EQ_OP` ('=')
- `II_NE_OP` ('!=')
- `II_LT_OP` ('<')
- `II_LE_OP` ('<=')
- `II_GT_OP` ('>')
- `II_GE_OP` ('>=')

Keybuild's main purpose is to build the upper and lower values of the search space. These values are called the high key and low key, respectively. In addition, `keybuild` returns the type of key that was formed, which tells what type of search must be performed.

The value returned in `adc_tykey` determines whether or not a key pair is built and, if built, whether the pair is the high or low key. If you are interested only in what type of key is built and not in the actual search space, then set the `db_data` field (in the `II_DATA_VALUE` structure pointed to by `adc_lokey` and/or `adc_hikey`) to point to a zero address.

The following are the values returned in `adc_tykey` and their interpretations:

**II\_KNOMATCH**

No values in the table match, so no scan of the table is done. In this case, the low key is set to maximum value for the data type and length of the column being keyed and the high key is set to the minimum.

**II\_KEXACTKEY**

Only a single value from the table matches. The low and high keys are set to the same value. The execution phase seeks to this point and scans forward until it is sure that it has exhausted all possible matching values.

**II\_KRANGEKEY**

All values in the table that match lay within a range. The low key is set to represent the lowest matching value in the table and the high key is set to represent the highest matching value. The execution phase seeks to the point matching the low key and scans forward until it has exhausted all values that might be less than or equal to the high key.

**II\_KHIGHKEY**

All values in the table that match lie at the low end of the table; they are less than or equal to some value. In this case, the high key is set to that value (the upper bound) and the low key is set to the minimum value for the data type and length of the column being keyed (unbounded). The execution phase starts at the beginning of the table and seeks forward until it has exhausted all values that might be less than or equal to the high key.

**II\_KLOWKEY**

All values in the table that match lie at the high end of the table; they are greater than or equal to some value. In this case, the low key is set to that value (the lower bound) and the high key is set to the maximum value for the data type and length of the column being keyed (unbounded). The execution phase seeks to the point of the low key and scans forward from there.

**II\_KALLMATCH**

All values in the table may match. The low key is set to minimum value for the data type and length of the column being keyed and the high key is set to the maximum value. A full scan of the table must be performed.

The most likely combinations are:

- `II_EQ_OP ('=')` returns `II_KEXACTKEY`, `low_key == value provided`
- `II_NE_OP ('!=')` returns `II_KALLMATCH`, no key provided
- `II_LT_OP ('<')` and `II_LE_OP ('<=')` return `II_KHIGHKEY`, with the `high_key == value provided`
- `II_GT_OP ('>')` and `II_GE_OP ('>=')` return `II_KLOWKEY`, with the `low_key == value provided`

Although there is fairly strong correlation between the key operator and the type of key built, you cannot use the key operator to predict with certainty the type of key built. For example, assume that you are keying on an i2 column with the '<' operator but the supplied key value is an i4 whose value is 50000. The key that is built is II\_KALLMATCH, not II\_KHIGHKEY, as might be expected. Do not rely on the key operator to tell you what type of key is built.

**Note:** The data type of the datavalue in the .adc\_kdv may not be same type as the required key resulting from this routine. If it is not, you must supply a coercion to change it to the required data type.

## Inputs

The inputs for this function are:

### **scb**

Pointer to SCB

### **key\_block**

Pointer to key block data structure:

### **adc\_kdv**

Datavalue for which to build a key.

This datavalue does not need to be of the same type as the required key.

### **adc\_opkey**

Operator type for which key is being built

### **adc\_lokey**

Pointer to area for key. If 0, do not build key.

### **adc\_hikey**

Pointer to area for key. If 0, do not build key.

### **adc\_lokey, adc\_hikey, and adc\_kdv**

Point to II\_DATA\_VALUES

## Outputs

The outputs for this function are:

### **\*key\_block**

Key block filled with following:

#### **adc\_tykey**

Type key provided.

#### **adc\_lokey**

Pointer to area for key. If 0, do not build key. If adc\_tykey is II\_KEXACTKEY or II\_KLOWKEY, this is key built.

#### **adc\_hikey**

Pointer to area for key. If 0, do not build key. If adc\_tykey is II\_KEXACTKEY or II\_KHIGHKEY, this is the key built.

## Returns

II\_STATUS

## length\_check Routine—Check for Valid Length

The length\_check routine checks that the specified length for the data type is valid. If the specified length is user specified, the routine returns the corresponding internal length. If the length is not user specified, it returns a user length corresponding to internal length.

Place the address of this routine in the dtd\_lenchk\_addr field of the IIADD\_DT\_DFN structure.

If the value of user\_specified is not 0, then the length is a value specified by a user or user program—for example, 4 if user typed 'varchar(4)'. If the value of user\_specified is 0, then the length is the internal length, for example, 6 for varchar(4).

If you specify result\_dv, then it must be set to the valid length regardless of the success or failure of the routine. If user\_specified is non-zero, then result\_dv must specify the corresponding internal length. Conversely, if user\_specified is zero, then result\_dv must specify the user length corresponding to the provided internal length.

### Inputs

The inputs for this function are:

**scb**

Pointer to SCB.

**user\_specified**

0 if not user specified, non-zero otherwise.

**dv**

Pointer to datavalue to be checked. If user\_specified is non-zero, then the length field refers to the length specified by the user. Otherwise, it refers to an internal length.

**result\_dv**

Pointer to an II\_DATA\_VALUE into which to place the correct length. This parameter can be NULL (0). When this is the case, simply return success or error status.

### Outputs

The outputs for this function are:

**result\_dv->db\_length**

Contains the valid length. If the user\_specified field is non-zero, this field must be set to the corresponding internal length. If the user\_specified field is 0, then set this to the corresponding user length.

## Returns

II\_STATUS

## minmaxdv Routine—Provide Min/Max Values and Lengths

The minmaxdv routine provides the minimum and maximum values and lengths for a data type.

Place the address of this routine in the dtd\_minmaxdv\_addr field of the IIADD\_DT\_DFN structure. If the II\_DT\_NOHISTOGRAM attribute is present, then this routine is not necessary.

Depending on the input parameters, the routine returns one or both of the following:

- Its minimum and/or maximum value
- Its minimum and/or maximum length

The two input parameters are min\_dv and max\_dv; both are pointers to II\_DATA\_VALUES. The lengths specified (db\_length) for each may be different, but their data types (db\_datatype) must be the same.

The routine uses the following rules to process these inputs:

If an input is NULL, then processing for that input is not performed. This allows the caller who is interested in only the maximum value or only the minimum to use this routine more efficiently.

If the db\_length field of an input is supplied as II\_LEN\_UNKNOWN, no corresponding value is built and placed at the output's db\_data field. Instead, the routine returns the valid internal length to the db\_length field.

If the db\_data field of an input is NULL, then no value is built and placed at the corresponding output's db\_data field.

If none of rules 1-3 apply to an input, then the value for the data type and length is built and placed at db\_data.

## Inputs

The inputs for this function are:

**scb**

Pointer to an SCB.

**min\_dv**

Pointer to II\_DATA\_VALUE for the 'min'. If this is NULL, 'min' processing is skipped:

**db\_datatype**

Its data type. Must be the same as data type for 'max'.

**db\_length**

The length to build the 'min' value for, or II\_LEN\_UNKNOWN, if the 'min' length is requested.

**db\_data**

Pointer to location to place the 'min' non-null value, if requested. If this is NULL no 'min' value is created.

**max\_dv**

Pointer to II\_DATA\_VALUE for the 'max'. If this is NULL, 'max' processing is skipped:

**db\_datatype**

Its data type. Must be the same as data type for 'min'.

**db\_length**

The length to build the 'max' value for, or II\_LEN\_UNKNOWN, if the 'max' length is requested.

**db\_data**

Pointer to location to place the 'max' non-null value, if requested. If this is NULL no 'max' value is created.

## Outputs

The outputs for this function are:

**min\_dv**

If this was supplied as NULL, 'min' processing is skipped.

**db\_length**

If this was supplied as II\_LEN\_UNKNOWN, the 'min' valid internal length for this data type is returned.

**db\_data**

If this was supplied as NULL, or if the db\_length field was supplied as II\_LEN\_UNKNOWN, nothing is returned. Otherwise, the 'min' non-null value for this data type and length is built and placed at the location pointed to by db\_data.

**max\_dv**

If this was supplied as NULL, 'max' processing is skipped.

**db\_length**

If this was supplied as II\_LEN\_UNKNOWN, the 'max' valid internal length for this data type is returned.

**db\_data**

If this was supplied as NULL, or if the db\_length field was supplied as II\_LEN\_UNKNOWN, nothing is returned. Otherwise, the 'max' non-null value for this data type and length is built and placed at the location pointed to by db\_data.

**Returns**

II\_STATUS

## seglen Routine—Determine Length of Each Long Segment

This routine returns the maximum number of bytes that can fit into a segment. For a peripheral like long line, it is the number of points that can fit in the input length times the size of a point plus the size of a line's overhead.

### Inputs

The inputs for this function are:

**scb**

Pointer to a SCB

**dt\_id**

Data type ID of peripheral object

**result\_dv**

Pointer to an II\_DATA\_VALUE with the maximum size of the segment in the length field

### Outputs

The outputs for this function are:

**result\_dv**

Pointer to an II\_DATA\_VALUE that receives the underlying data type, the maximum length of the underlying data type, and the precision of the underlying data type.

### Returns

II\_STATUS

## tmcvt Routine—Convert Data Type to Displayable Format

The tmcvt routine converts data of a user-defined data type from an internal format to a displayable format. (This displayable format is used by a terminal monitor when user-defined data types are sent without conversion to a terminal monitor.) Place the address of this routine in the dtd\_tmcvt\_addr field of the IIADD\_DT\_DFN structure.

This routine is used by the DBMS Server to format various trace statements and error messages.

### Inputs

The inputs for this function are:

**scb**

Pointer to a SCB.

**from\_dv**

Pointer to a datavalue containing the data to be displayed

**to\_dv**

Pointer to a datavalue that provides the output space. The datavalue's db\_data field points to an area of db\_length bytes.

### Outputs

The outputs for this function are:

**to\_dv->db\_data**

Filled with the output

**\*output\_length**

Filled with the number of characters placed in to\_dv->db\_data

### Returns

II\_STATUS

## tmlen Routine—Determine Display Length

The tmlen routine determines the display length of the data type.

This routine returns the default and worst-case lengths for a data type if it were to be printed as text, for example, by a terminal monitor. Although user-defined data types are not returned to a terminal monitor as the user-defined types, this routine is needed by various trace flags and error formatting within the DBMS Server.

Place the address of this routine in the dtd\_tmlen\_addr field of the IIADD\_DT\_DFN structure.

### Inputs

The inputs for this function are:

**scb**

Pointer to a SCB

**dv\_from**

Pointer to the datavalue for which the call is being made

### Outputs

The outputs for this function are:

**def\_width**

Pointer to a 2-byte integer into which the default width was placed

**largest\_width**

Pointer to a 2-byte integer in which the largest (worst case) width was placed

### Returns

II\_STATUS

## value\_check Routine—Check for Valid Values

The `value_check` routine checks for valid values.

For some data types, only certain characters or bit patterns might be valid. This routine checks the patterns for validity. For example, this routine which rejects C data type values which contain null characters.

Place the address of this routine in the `dtd_valchk_addr` field of the `IIADD_DT_DFN` structure.

### Inputs

The inputs for this function are:

**scb**

Pointer to a SCB

**dv**

Pointer to data value in question

### Outputs

This function has no outputs. The return value (`II_OK` or `II_ERROR`) determines correctness.

### Returns

`II_STATUS`

## xform Routine—Transform Long Types into Segments

This xform routine transforms a long data type into its component segments. Place the address of this routine in the `dtd_xform_addr` field of the `IIADD_DT_DFM` structure.

The `shd_exp_action` contains the instructions for this routine. `ADW_START` is the first call. Thereafter, it is examined at the return of this routine to determine the caller's next action. Set `ADV_GET_DATA` to have the caller provide the next section of data. If the caller receives `ADW_GET_DATA` and there is no more data, then it flushes any current data and does not return to this routine. `ADW_FLUSH_SEGMENT` indicates that the caller disposes of the output segment, and supplies a new, empty one on the next call. `ADW_CONTINUE` indicates that the routine is to be called again. `ADW_STOP` indicates that there is a problem and the process has failed; the routine is not called again.

### Inputs

The inputs for this function are:

**scb**

Pointer to a SCB.

**workspace**

Pointer to the workspace for peripheral operations (`II_LO_WKSP`). This workspace contains fields for the: data type being transformed; the action for this call; the pointer to, length of, and amount used of the input area; and the pointer to, length of, and amount used of the output area.

### Outputs

The outputs for this function are:

**workspace****shd\_exp\_action**

Modified to give caller action

**shd\_i\_used, shd\_o\_used**

Modified as used by the routine

### Returns

`II_STATUS`



# Chapter 5: Defining Functions

---

This section contains the following topics:

[Required Definitions](#) (see page 55)

[Structure IIADD\\_FO\\_DFN](#) (see page 55)

This chapter describes how to define functions.

## Required Definitions

To create a new SQL function, you must provide the DBMS Server with the function definition and function instance definitions that describe the use of the function. To specify the function definition information, you must use the structure `IIADD_FO_DFN`, described in this chapter.

## Structure `IIADD_FO_DFN`

To define a new function, use the data structure `IIADD_FO_DFN`, included in the header file, `IIADD.H`. For the location of this file, see the chapter "Introducing Object Management Extension."

The fields in this structure are as follows:

**fod\_object\_type**

Specifies the object type. It must contain the constant `II_O_OPERATION`.

**fod\_name**

Specifies the name of the function. The name can be up to 32 bytes in length and must be a valid database object name. If the name is shorter than 32 bytes, the name must be null terminated. For example, if the name is "op," the field must contain "op\0." If the name is 32 bytes, omit the null terminator.

**fod\_id**

Specifies the function number. Function numbers must be greater than or equal to 16384. Function numbers are symbolically identified in the header file by `II_OPSTART`.

**fod\_type**

Specifies the type of operation invoked by the function or operator-must contain the value `II_NORMAL`.



# Chapter 6: Defining Function Instances

---

This section contains the following topics:

[Function Instance Definition](#) (see page 57)

[Structure IIADD FI DFN](#) (see page 58)

[Length Definition of Result Data Type](#) (see page 61)

[External Lenspec Routine—Return Result Length of Specified Value](#) (see page 62)

[Complementary Function Instances](#) (see page 62)

[Sorting of the Function Instance Definition Array](#) (see page 63)

[Methods for Defining Function Instances for Large Objects](#) (see page 63)

[Ingres-supplied Filter Functions](#) (see page 64)

[II INFORMATION Operation—Return Maximum Length of Peripheral Object Segments](#) (see page 68)

[II GET Operation—Get Next Segment](#) (see page 69)

[II PUT Operation—Add a New Segment](#) (see page 71)

[II COPY Operation—Move a Peripheral Object](#) (see page 73)

This chapter describes how to define function instances.

## Function Instance Definition

Functions and operators (jointly referred to in this chapter as “functions”) can be used in many contexts. For example, you can use the operator “+” to add numbers, concatenate strings, or to add two user-defined data type values. The use of a function within a given context is called a function instance.

The definition of a function instance links a function definition and the function's arguments to a procedure (user-written executable procedure, not a database procedure) and a result (type and length). For every new function that you define, you must provide the DBMS Server with function instance definitions for its use. You must also provide the function instance definitions that define the use of your new data type with the functions with which you intend to use it.

## Structure IIADD\_FI\_DFN

Function instances are defined by filling in the structure IIADD\_FI\_DFN, once for each instance. IIADD\_FI\_DFN is included in the header file provided with Object Management Extension. The fields in this structure are as follows:

### **fid\_object\_type**

Must contain the constant II\_O\_FUNCTION\_INSTANCE.

### **fid\_id**

Must contain a valid function instance identifier. This is a 2-byte integer, starting at 16384 (II\_FISTART). The DBMS Server uses this number internally to identify the function instance.

### **fid\_cmplmnt**

If you have defined fid\_type as II\_COMPARISON, this field must contain the fid\_opid of this function instance's complement. For more information, see Complementary Function Instances (see page 62).

### **fid\_opid**

Contains the function identifier used to invoke this function instance. The function id can be a constant identifying a standard SQL operator or the identifier of a new function that was defined using the IIADD\_FO\_DFN structure.

If the function instance that you are defining is a data type coercion (that is, fid\_optype = II\_COERCION), this value must be II\_NOOP.

### **fid\_optype**

Contains the operation type invoked by the operator. The legal values for this field are:

II\_COMPARISON

II\_OPERATOR

II\_NORMAL

II\_COERCION

### **fid\_attributes**

Specifies how the function instance behaves.

II\_FID\_F0\_NOFLAGS

II\_FID\_F4\_WORKSPACE

II\_FID\_F8\_INDIRECT

### **fid\_wslength**

Specifies the length of the workspace. Valid only if II\_FID\_F4\_WORKSPACE is set.

**fid\_numargs**

Number of arguments for this function. The `fid_numargs` field is a 4-byte integer. Values in this field are related to the values in the `fid_optype` field. If `fid_optype` is `II_COMPARISON` or `II_OPERATION`, this value must be 2. If it is `II_COERCION`, this value must be 1. If `fid_optype` is `II_NORMAL`, `fid_numargs` can be 0, 1, or 2.

**fid\_args**

Pointer to an array of data types, which are the data types of the arguments for the function instance. Each array element is a 2-byte integer. The first element in the array must point to the first argument, the second element to the second argument, and so on. If the function instance has no arguments, set this pointer to 0.

The proprietary Ingres data types, date and money are not currently supported for user manipulation.

The specified argument data types can be user-defined data types or standard SQL data types. To specify a user-defined data type, use the value in the `dtd_id` field of the `IIADD_DT_DFN` structure for that data type.

To specify a data type, use one of the following values:

`II_INTEGER`

`II_DECIMAL`

`II_FLOAT`

`II_CHAR`

`II_C`

`II_VARCHAR`

`II_TEXT`

`II_LONGTEXT`

To specify a spatial data type, use the following values:

`PNT_TYPE`

`BOX_TYPE`

`LSEG_TYPE`

`LINE_TYPE`

`POLY_TYPE`

`CIRCLE_TYPE`

`LLINE_TYPE`

`LPOLY_TYPE`

`IPNT_TYPE`

`IBOX_TYPE`

ILSEG\_TYPE  
ILINE\_TYPE  
IPOLY\_TYPE  
ICIRCLE\_TYPE  
NBR\_TYPE

**fid\_result**

The data type of the function instance result.

**fid\_rdtype**

Set this field to one of the following values to specify the method used to determine the length of the function instance's result. Note that the arguments referred to are the arguments to the routines that perform the operation defined by the function instance. Valid values:

II\_RES\_FIXED  
II\_RES\_FIRST  
II\_RES\_SECOND  
II\_RES\_LONGER  
II\_RES\_SHORTER  
II\_RES\_KNOWN  
II\_RES\_EXTERN

**fid\_rlength**

Set this field to the length of the function instance's result if the fid\_rdtype field is II\_RES\_FIXED. Otherwise, set this field to II\_LEN\_UNKNOWN (-1).

**fid\_rprec**

Precision of the data value. For most data types, this is not needed and should be set to 0. However, for DECIMAL, the high order byte will represent the value's precision (total number of significant digits), and the low order byte will hold the value's scale (the number of these digits that are to the right of an implied decimal point). Calculate this value and set accordingly.

**fid\_routine**

Address of the routine that performs the function instance.

**lenspec\_routine**

Address of the routine that can be called to compute the result length.

## Length Definition of Result Data Type

The length of a result data type is defined by `fid_rlength` in `II_ADD_FI_DFN` or an external `lenspec` routine. This length is used by the DBMS Server when manipulating values internally. The length component of the actual data value (`II_DATA_VALUE db_length`) should not be changed within the function itself. If the length defined by `fid_rlength` or the `lenspec` routine differs from the data value's `db_length`, then errors may result or data values may be incorrectly interpreted.

The same is true of scale and precision in the case of `DECIMAL`. In this case, the `fid_rprec` or `lenspec` is used and the `db_prec` of the data value should not be changed.

The following macros, defined in `$II_SYSTEM/ingres/files/iiadd.h`, can be used to manipulate `DECIMAL` length values:

### **DB\_PS\_ENCODE\_MACRO(p,s)**

Given a precision `p` and scale `s`, returns the two-byte value combining the two. This macro could be used, for example, when setting `db_prec` value within a user-defined `lenspec` routine.

### **DB\_P\_DECODE\_MACRO(ps)**

Given a two-byte combined value for precision and scale (`db_prec`) of `ps`, returns the precision part.

### **DB\_S\_DECODE\_MACRO(ps)**

Given a two-byte combined value for precision and scale (`db_prec`) of `ps`, returns the scale part.

### **DB\_PREC\_TO\_LEN\_MACRO(prec)**

Given a precision of `prec`, returns the length needed for such a decimal. This macro could be used when setting the `db_length` within a user-defined `lenspec` routine.

## External Lenspec Routine—Return Result Length of Specified Value

The lenspec routine returns the result length for the specified value in the `db_length` field of the `dvr` parameter. This routine is called by the DBMS Server when a function instance's `fid_rtype` field is set to `II_RES_EXTERN`. Lenspec also returns `II_STATUS`.

The parameters for the lenspec routine are as follows:

**scb**

Points to the session control block for error processing.

**opid**

Points to the operator being invoked.

**dv1, dv2**

Pointers to `II_DATA_VALUE`. These pointers specify the inputs to the operator. If the function instance does not use one of the operands, that pointer is zero.

**dvr**

Pointer to an `II_DATA_VALUE` whose length is filled in by this routine.

The `II_DATA_VALUE` parameters are used only to pass type and length information. The `db_data` field in the `II_DATA_VALUE` structure must be ignored.

On return from the routine, `dvr->db_length` must contain a positive value, and the routine should not raise any errors other than `0x22022`.

## Complementary Function Instances

To optimize query performance, the DBMS Server often inverts the sense of a comparison operation. To make this possible, you must define a complementary function instance for every function instance of `fid_optype` `II_COMPARISON` that you define.

For example, if you define a function instance for the equals (=) operator, you must also define a function instance for its complement, not equals (!=). Note that the complement of ">=" is "<", and the complement of "<=" is ">".

Complementing function instance definitions must agree. That is, if A is a function instance whose complement is B (A's `fid_cmplmnt` = B's `fid_opid`), the complement of B must be A (B's `fid_cmplmnt` = A's `fid_opid`).

## Sorting of the Function Instance Definition Array

For the DBMS Server to merge the new function instance definitions with existing function instance definitions, the array of function instance definitions that you create must be specified in sorted order.

The array must be sorted by the number value of the `fid_optype` field. All comparisons (`fid_optype = II_COMPARISON`) must be first, followed by all operators (`fid_optype = II_OPERATOR`), normal functions (`fid_type = II_NORMAL`), and finally, the coercion functions (`fid_optype = II_COERCION`).

In each operator type, sort the function instances by the numeric value of the `fid_opid` field. For example, with the operator type `II_COMPARISON`, the function instances are sorted in the following order:

1. 'not equal' function instances
2. 'less than' function instances
3. 'less than/equal to' function instances
4. 'equal' function instances
5. 'greater than' function instances
6. 'greater than/equal to' function instances

## Methods for Defining Function Instances for Large Objects

Object Management Extension provides two ways to define function instances for large objects. You can use the Ingres-supplied filter functions or you can manipulate the large objects directly.

## Ingres-supplied Filter Functions

Of the callback functions, two have been paired to allow for function instances that must traverse a large object. Ingres provides skeleton code that allows you to pass the object through your filter.

To use a filter function, the calling instance sets up the workspace by calling the function supplied to initialize the workspace. This function does all the work necessary to allow for the simple traversal of the input object. After this, your filter function is called for each segment and is passed an `II_DATA_VALUE`, which provides space for manipulation of the input and output segments, a function to call for each segment, a workspace to be used by this routine, and a continuation indicator.

The continuation indicator states whether this is the first or last call to the filter. The indicator is specified by the use of the `II_C_BEGIN_MASK` and the `II_C_END_MASK`, indicating that this is the beginning or end of the resulting object.

The function to be called is called with pointers to `II_DATA_VALUE`, which describes input and output segments. The routine is expected to convert the input into the output segment. When processing the data, the function indicates the disposition of the current segment by filling in the value of the `adw_shared.shd_exp_action` field of the workspace.

As the routine moves through the object, it is expected to keep the `adw_shared.shd_l1_chk` field filled with the current length of the large object that is the result. The routine that is called back does not necessarily know if it will be called again.

The expected action indicator conveys information between the filter routine and the function instance routine. The indicator is set to `ADW_START` before the first segment routine call. Thereafter, the indicator is examined at the return of the call to determine the next action necessary for the function instance routine to perform.

The possible values and responses are as follows:

### **ADW\_CONTINUE**

The calling routine disposes of the current segment, get a new segment, and recall the routine.

### **ADW\_GET\_DATA**

The function instance routine obtains the next segment of data to be processed by the routine. The output segment remains unchanged.

**ADW\_FLUSH\_SEGMENT**

The function instance routine is expected to dispose of the current (presumably “full”) output segment, and provide a new, empty one for the routine to deal with. The input segment is untouched.

**ADW\_FLUSH\_STOP**

The function flushes the current segment and stop.

**ADW\_STOP**

The routine stops without flushing the current output segment.

If the routine returns ADW\_GET\_DATA and there is no more data to get, the function instance routine disposes of the current segment and assumes that the work of the routine is complete. The routine is not called again.

This calling sequence means that the routine never knows if it will be called again. For this reason, the `adw_i0_chk` and `adw_i1_chk` values must be correct, and the output segment from your filter must always be a valid segment, even when the filter is not complete.

Of the fields in the workspace, only those in the shared section are interpreted by the filter function. The `shd_exp_action`, `shd_i0_chk`, and `shd_i1_chk` fields must be set as described above. The other fields in the `II_SHARED` structure must be left unchanged.

The remaining fields in the workspace are available for each function instance's use.

## Direct Manipulation of Large Objects

In some cases, (for example, two-pass algorithms,) defining a function instance does not work. For these situations, your code can get the peripheral object segments directly through the use of the large object handler. The handler routine performs a variety of operations on large objects.

The handler is called with two arguments. The first is the operation code, which tells the handler which operation to perform. The second argument is a peripheral object control block, `II_POP_CB`. This control block structure is used to pass information to and from the caller of the handler. The first part of this structure is a standard header used in the DBMS Server.

The exact description of this control block structure can be found in the `iiadd.h` header file.

- `pop_next`
- `pop_prev`
- `pop_length`
- `pop_type`
- `pop_s_reserved`
- `pop_l_reserved`
- `pop_ascii_id`

The `pop_type` field must always be set to `II_POP_TYPE`. The `pop_length` field must always be set to the size of the structure. The `pop_s_reserved` and `pop_l_reserved` fields must not be set or altered. These fields are used by the DBMS Server memory management routines. You can set the other fields as required.

The rest of the `II_POP_CB` control block consists of the following fields:

### **pop\_error**

The `err_code` field of this structure contains any error that results from the requested operation. The `II_E_NOMORE` error indicates that there are no more segments to get. Other errors indicate coding errors.

### **pop\_continuation**

Tells the handler routine whether this is the first and/or last invocation of this routine. The use of this field varies by function (refer to the operations that follow for specific use). This field is a mask, so a single operation can be the first (`II_C_BEGIN_MASK`), the last (`II_C_END_MASK`), neither (0), or both (`II_C_BEGIN_MASK | II_C_END_MASK`).

**pop\_temporary**

Indicates the lifetime of the object being created (through an II\_PUT operation). A value of II\_POP\_SHORT\_TEMP indicates that the object must be destroyed at the end of the current query. There are no other valid values.

**pop\_underdv**

Contains information about the underlying data type for the operation. The underlying data type describes the segment being manipulated. The field is partially filled in by the caller (db\_datatype) and partially by the II\_INFORMATION operation (db\_length). Because the length allowed can change in future releases of the software, using the information operation as described aids in upward compatibility. The db\_data field is unused.

**pop\_coupon, pop\_segment**

Pointers to II\_DATA\_VALUES that describe the overall object (the pop\_coupon field) being manipulated, and the individual segment being worked on (pop\_segment).

**pop\_user\_arg**

Provided for the handler's use. This field must not be touched by your code between the first and last call in a series to the handler (that is, between the II\_C\_BEGIN\_MASK instance and the II\_C\_END\_MASK instance). The handler uses this field to maintain a context. This is necessary due to the multi-threaded nature of the DBMS Server.

## II\_INFORMATION Operation—Return Maximum Length of Peripheral Object Segments

This operation returns the maximum length (in bytes) for each segment of a peripheral object.

### Inputs

The input fields for the II\_INFORMATION operation are the parts of the peripheral control block structure that tell the DBMS Server what information to pass and how to pass the information. The inputs are as follows:

**op\_code**

The value of II\_INFORMATION

**pop\_cb**

A pointer to the peripheral operations control block, II\_POP\_CB (for calling).

**pop\_coupon**

The coupon about which the information is required.

### Outputs

The output fields for the II\_INFORMATION operation describe the results the DBMS Server returns. The outputs are as follows:

**pop\_cb**

A pointer to the peripheral operations control block, II\_POP\_CB (for calling).

**pop\_underdv.db\_length**

Maximum length for underlying portions of the data type.

**pop\_error**

The DBMS Server fills this field with the error value, if there is any.

### Returns

II\_STATUS

## II\_GET Operation—Get Next Segment

This operation gets the next segment of some object. The object is represented to II\_GET by a coupon. This operation makes use of the saved information in the peripheral control block, which is passed in the pop\_cb's user argument parameter.

### Inputs

The input fields for the II\_GET operation are the parts of the peripheral control block structure that tell the DBMS Server what and how to pass the information. The inputs are as follows:

**op\_code**

The value of II\_GET.

**pop\_cb**

A pointer to the peripheral operations control block, II\_POP\_CB (for calling).

**pop\_continuation**

Continuation indicator. On the first call II\_C\_BEGIN\_MASK is passed; otherwise, zero (0) is passed.

**pop\_coupon**

Pointer to II\_DATA\_VALUE, contains the coupon used to retrieve the object.

**pop\_segment**

Pointer to II\_DATA\_VALUE to receive the output of II\_GET. An error occurs if the size of the output is insufficient. For size determination, see II\_INFORMATION Operation (see page 68).

**pop\_user\_arg**

Server's internal state. This is set on the first call and must be presented unchanged for subsequent calls.

### Outputs

The output fields for the II\_GET operation describe the results the DBMS Server returns. The outputs are as follows:

**pop\_cb**

A pointer to the peripheral operations control block, II\_POP\_CB (for calling).

**pop\_segment->db\_data**

The DBMS Server fills this field with the next segment.

**pop\_user\_arg**

The DBMS Server fills this field, if appropriate.

**pop\_error**

The DBMS Server fills this field with the error value, if there is any.

**Returns**

II\_STATUS

## II\_PUT Operation—Add a New Segment

This operation adds a new segment to the end of a new peripheral object. This operation takes, as input, the segments of a peripheral object, and returns a completed coupon. When there are multiple segments in an object, the coupon is not complete until the last invocation of the II\_PUT operation.

The underlying data type must be provided by the caller, and describes how the underlying data must be represented. See the `pop_underdv` field in the table below. For example, long varchar objects are stored as varchar segments.

### Inputs

The input fields for the II\_PUT operation are the parts of the peripheral control block structure that tell the DBMS Server what and how to pass the information. The inputs are as follows:

#### **op\_code**

The value of II\_PUT.

#### **pop\_cb**

A pointer to the peripheral operations control block, II\_POP\_CB (for calling).

#### **pop\_continuation**

Indicates the state of the object. On the first call of a multicall operation, II\_C\_BEGIN\_MASK is set. On the last call, II\_C\_END\_MASK is set. A single call can be either the first call, last call, both, or neither.

#### **pop\_coupon**

Pointer to the coupon to be created.

#### **pop\_segment**

A pointer to a data area from which data is taken (to be put into the large object).

#### **pop\_underdv**

A pointer to a II\_DATA\_VALUE that describes the data type to be used for each segment.

#### **pop\_temporary**

Must be II\_POP\_SHORT\_TEMP.

#### **pop\_user\_arg**

Workspace area used by the server. The caller knows nothing of the contents here, but is expected to preserve its contents across multipass calls. Failure to do so results in errors.

## Outputs

The output fields for the II\_PUT operation describe the results returned by the DBMS Server. The outputs are as follows:

### **pop\_cb**

A pointer to the peripheral operations control block, II\_POP\_CB (for calling).

### **pop\_coupon**

For temporaries, are completely filled. This portion cannot be complete until the last call of a multipass operation is completed.

### **pop\_user\_arg**

Is filled in with server specific information to be preserved and returned by the caller if II\_PUT is not a coupon and pop\_continuation is not II\_C\_END\_MASK.

### **pop\_error**

The DBMS Server fills this field with the error value, if there is any.

## Returns

II\_STATUS

## II\_COPY Operation—Move a Peripheral Object

This operation moves a peripheral object by performing gets and puts.

### Inputs

The input fields for the II\_COPY operation are the parts of the peripheral control block structure that tell the DBMS Server what and how to pass the information. The inputs are as follows:

**op\_code**

The value of II\_COPY.

**pop\_cb**

A pointer to the peripheral operations control block, II\_POP\_CB (for calling).

**pop\_segment**

A pointer to the input object. In this case, pop\_segment and the input object are both coupons.

**pop\_coupon**

A pointer to an II\_DATA\_VALUE that describes the area to be filled with the coupon for the copied object.

### Outputs

The output fields for the II\_COPY operation describe the results returned by the DBMS Server. The outputs are as follows:

**pop\_cb**

A pointer to the peripheral operations control block, II\_POP\_CB (for calling).

**pop\_coupon**

The DBMS Server fills this field with the coupon for the copied object.

**pop\_error**

The DBMS Server fills this field with the error value, if there is any.

### Returns

II\_STATUS



# Chapter 7: Passing Definitions to the DBMS Server

---

This section contains the following topics:

[IIudadt\\_register Routine](#) (see page 75)

[Structure IIADD\\_DEFINITION Fields](#) (see page 76)

[Server Routines Provided](#) (see page 78)

This chapter describes how definitions are passed to the DBMS Server.

## IIudadt\_register Routine

The DBMS Server accesses user-written code for new data types, functions, and operator capabilities by calling the routine `IIudadt_register`. This routine, which you must provide, is called at system startup. It fills in a pointer to a data structure that contains the definitions of any user-written data types, functions and function instances. The DBMS Server checks the definitions to ensure that they are correct and merges them with the existing data type manipulation objects.

This routine is called using the following syntax:

```
status=IIudadt_register(add_block);
```

Status has the data type `II_STATUS`. `Add_block` is a pointer to a field that must be set to point to a data structure of type `IIADD_DEFINITION` if you are adding new data types, functions, or function instances. If you are not adding new data types (or functions or function instances), set this pointer to zero. This is the default.

**Note:** If provided, `IIADD_DEFINITION` exists as a compiler-generated data area. Do not dynamically allocate this data area or place it on the stack.

## Structure IIADD\_DEFINITION Fields

The fields in the IIADD\_DEFINITION data structure are as follows:

### **add\_risk\_consistency**

This field, if set to IIADD\_INCONSISTENT, tells the DBMS Server to start even if the values of the current add\_major\_id and add\_minor\_id fields do not agree with those in use by the Ingres recovery system. Setting this field is very dangerous and must only be done in test installations.

If the add\_risk\_consistency field is set, the DBMS Server ignores any disagreements between the major and minor ID fields of the support processes and the system. This is very dangerous in a production environment.

### **add\_major\_id and add\_minor\_id**

The DBMS Server uses these values to ensure that all support processes know about all the data types in the system, ensuring recoverability across the installation. When a support process starts up, it checks the major and minor ID values in this block against the major and minor ID values known to the system. If the major\_id values are not equal and the process' minor\_id is not equal to or greater than the system's, the process not does start.

The add\_major\_id field represents the high order 4 bytes and the add\_minor\_id, the low order 4 bytes of an 8-byte value. The initial values of the major and minor IDs, representing Ingres in the original state, are 0x80000000 and 0, respectively.

To display the current values for these fields, run the lockstat utility. Look for a lock with the key SYS\_CONTROL II\_DATATYPE\_LEVEL. This key has an 8-byte value associated with it. The high order 4 bytes of this value represent the major\_id and the low order 4 bytes, the minor\_id. For example, the value associated with this key for the default data type level is '8000000000000000', representing the major\_id of 0x80000000 and the minor\_id of 0 which is Ingres in the original state.

Increment the add\_major\_id field each time you add a new user-defined data type. User values for this field must be greater than 0. Minor\_id must be greater than or equal to 0. Reset the add\_minor\_id field to 0 whenever the add\_major\_id is changed.

### **add\_l\_user\_string**

Specifies the length of the character string pointed to by add\_user\_string.

### **add\_user\_string**

Pointer to a character string that is included in the system log.

**add\_trace**

4-byte integer that can be filled with a bitmask to set trace functions to be provided during system startup. There are two bits used by trace functions:

IIADD\_T\_LOG\_MASK (1) directs the DBMS Server to place a number of trace messages in the various trace logs which exist in the system. While operating the DBMS Server, defining II\_DBMS\_LOG to a file causes the DBMS Server to place copies of the system error log messages in that file. If the IIADD\_T\_LOG\_MASK bit is set, user-defined abstract data type initialization places status and error messages in this file.

IIADD\_T\_FAIL\_MASK (2) directs the system to shut down if there are failures while initializing the user-defined data types. This is useful in test situations.

By default, the system ignores any errors encountered in processing new data type information and continues startup as if no user information was provided.

**add\_count**

Must be set to the sum of the add\_dt\_cnt, add\_fo\_cnt, and add\_fi\_cnt fields.

**add\_dt\_cnt**

A 4-byte integer field that must be set to the count of data types to be added.

**add\_dt\_dfn**

This field must point to an array of IIADD\_DT\_DFN structures that define new data types. (For instructions on filling in the IIADD\_DT\_DFN structures, see the chapter "Defining Data Types.") If the add\_dt\_cnt field is zero, set add\_dt\_dfn to zero.

**add\_fo\_cnt**

This field is a 4-byte integer that must be set to the count of functions to be added.

**add\_fo\_dfn**

This field must point to an array of IIADD\_FO\_DFN structures that define new functions. For details, see the chapter "Defining Functions." If the add\_fo\_cnt field is zero, set add\_fo\_dfn to zero.

**add\_fi\_cnt**

This field is a 4-byte integer that must be set to the count of function instances to be added.

**add\_fi\_dfn**

This field must point to an array of IIADD\_FI\_DFN structures that define new function instances. For details, see the chapter “Defining Function Instances.” If the add\_fi\_cnt field is zero, set add\_fi\_dfn to zero.

## Server Routines Provided

A number of routines are provided for you to use through the callback block. When you use one of these functions, save the addresses of the routines used in your code before IIudadt\_register() returns. After IIudadt\_register() completes, the call\_back\_block address is no longer valid.

These routines all return II\_STATUS values that specify their completion status.

### The ii\_cb\_trace Routine—Output Provided Trace Messages

The ii\_cb\_trace routine outputs trace messages provided to it. The parameters for this routine are as follows:

**dispose\_mask**

Specifies how to dispose of the message:

**II\_TRACE\_FE\_MASK**

Indicates that the string must be sent to the front end

**II\_TRACE\_ELOG\_MASK**

Indicates that the string must be sent to the error log

**II\_TRACE\_TLOG\_MASK**

Indicates that the string must go to the trace log only (II\_DBMS\_LOG)

**length**

Specifies the length of the string

**string**

Pointer to the string to be traced

**Note:** If II\_DBMS\_LOG is defined, messages going to the error log also appear in the trace log.

## The `ii_error_fcn` Routine—Place Error Information in Status Control Block

This routine places error information in the status control block (scb). As a result, you do not need to know the internal scb format. The parameters for this routine are as follows:

**scb**

A pointer to the scb into which the error must be placed.

**err\_num**

The error number to include in the message that goes to the scb.

**text**

A pointer to the character string to include in the message that goes to the scb. The message should be in the format "*datetime stringE\_XXNNNNmessage text*".

An alternate method of placing error information in the scb is to use the routine `us_error`, as defined in the demo UDTs. See `$II_SYSTEM/ingres/demo/udadts/common.c`. No special format for the error message is required for this routine.

## The `ii_io_handler_fcn` Routine—Move Through Large Object Segments

You can use this routine to move through large object segments. It is called with two arguments: an operation code, and a pointer to a `pop_cb` (peripheral operations control block). For details, see *Large Objects and the Methods for Defining Function Instances for Large Objects*.

## The `ii_init_filter_fcn` Routine—Set Up Filter Function

This routine sets up the filter function for large objects. It is called with three arguments: a pointer to an scb, a pointer to an `II_DATA_VALUE` that describes the large object being input, and a pointer to an `II_DATA_VALUE` that describes the workspace provided. For details, see *Large Objects and Methods for Defining Function Instances for Large Objects*.

## The `ii_filter_fcn` Routine—Perform Operation by Calling a User Routine

This routine performs an operation by calling back a user routine for each segment in the input object. It is called with the following arguments:

- A pointer to an `scb`
- Pointers to the `II_DATA_VALUE`'s for the input and output segments
- Pointer to a function to be called
- A pointer to a workspace to be used (previously initialized by a call to the initialization function)
- A continuation indicator indicating whether this is the first time this has been used.

For details, see [Large Objects](#) (see page 19) and [Methods for Defining Function Instances for Large Objects](#) (see page 63).

# Chapter 8: Installing and Testing Data Types

---

This section contains the following topics:

[How You Install New Data Types in a Windows Environment](#) (see page 81)

[How You Install New Data Types in a VMS Environment](#) (see page 81)

[How You Install New Data Types in a UNIX Environment](#) (see page 84)

[Testing the New Data Type Code](#) (see page 85)

This chapter contains instructions for installing and testing spatial and user-defined data types. Install and debug your code in a test environment before you install in the production environment. The installation process builds both the local server and the Star Server with user-defined data types (because both these servers are defined as the same image).

## How You Install New Data Types in a Windows Environment

Ingres processes access the new data types and functions by means of a dynamic linked library (DLL), whose entry point is the routine `IIudadt_register()`.

Installing a new data type or function consists of building the `IILIBUDT.DLL` using the User Defined Data Type Linker (`iilink.exe`) and copying the DLL into the `%II_SYSTEM%\ingres\bin` directory. The User Defined Data Type Linker prompts for the location of the source and library files to be used and backs up the existing DLL before replacing it.

Optionally, spatial objects or demo user defined data types can be installed using the User Defined Data Type Linker.

## How You Install New Data Types in a VMS Environment

Ingres processes access the new data types and functions by means of a shared image, whose entry point is the routine `IIudadt_register()`. Installing a new data type or function consists of building the shared image and defining `II_USERADT` to point to the location of the shared image.

## Template Command File—Create the Shared Image

Ingres provides a template command file for building the shared image for use in a test installation. The template is located in the following file:

II\_SYSTEM:[INGRES.LIBRARY]II\_USERADT\_BUILD.COM

This template, with comments, follows:

```
$ inst := 'f$trnlm("II_INSTALLATION")
$ if inst .eqs. ""
  then
$   inst_code := <production>
$ else
$   inst_code = inst
$ endif
$ Write sys$output "Building Sharable Image for User Defined Datatypes for
'inst_code' installation"
$ if f$getsyi("hw_model") .lt. 1024
$ then
$!
$!       VAX version
$!
$ macro/object=ii_system:[ingres.library]ii_useradt_xfer.obj -
  ii_system:[ingres.library]ii_useradt_xfer.mar
$ link/share=ii_system:[ingres.library]iiuseradt'inst'/sysshr -
  sys$input/opt /nodebug 'p1'
!
! This CLUSTER statement forces the transfer vector to the beginning of
! the shared image. It should not be removed.
!
Cluster = TRANSFER_VECTOR,,,ii_system:[ingres.library]ii_useradt_xfer.obj
ii_system:[ingres.library]iiclsadt.obj
!
! Replace the object module below with the object modules
! defining Installation Datatypes
!
  ii_system:[ingres.library]iiuseradt.obj
!
! End of object modules defining datatypes.
!
NAME = IIUSERADT
!
! Note that the shared image id should not be changed. INGRES expects this
! level. The shared image ID can be changed ONLY by the product vendor.
!
IDENTIFICATION = "v2-000"
GSMATCH=LEQUAL, 2, 0
$ exit
$!
$ else
$!
$!       Alpha version
$!
$ link/share/notrace/nodebug/ -
exe=ii_system:[ingres.library]iiuseradt'inst' SYS$INPUT/OPTION
ii_system:[ingres.library]iiclsadt.obj -
```

```

,ii_system:[ingres.library]iiuseradt.obj
NAME = IIUSERADT
IDENTIFICATION = "V2-000"
GSMATCH=LEQUAL, 2, 0
SYMBOL_VECTOR = (iiudadt_register = PROCEDURE)
  SYMBOL_VECTOR = (iiclsadt_register = PROCEDURE)
$ exit
$ endif

```

The transfer vector for the shared image is built by the following file:

```
ii_system:[ingres.library]ii_useradt_xfer.mar
```

If it is necessary to add additional entry points to the transfer vector, add them after the IIudadt\_register entry point defined in this file. If you do not, the resulting shared image does not operate correctly with Ingres.

In a production environment, remove the /debug flag and link the shared image NOTRACEBACK. Install it using the VMS Install Utility. For additional information about creating and maintaining shareable images, see the VMS documentation (in particular, the Guide to Creating Modular Procedures).

## II\_USERADT Logical—Set Disk Location of the Shared Image

II\_USERADT points to the location of the shared image. By default, II\_USERADT points to the following executable:

```
ii_system:[ingres.library]iiuseradt.exe
```

This location is overwritten during installations of new releases of Ingres. To avoid conflicts, place your shared image in another location and redefine II\_USERADT.

### System Level Installation

```
$define/system/exec II_USERADT II_SYSTEM.[ingres.library]iiuseradt.exe
```

### Group Level Installation

```
$define/group/exec II_USERADT II_SYSTEM.[ingres.library]iiuseradtxx.exe
```

where xx = the two-character installation code

By default, if the logical II\_USERADT is not defined, Ingres looks for the following shared image:

```
SYS$SHARE:II_USERADT.EXE
```

## Definition of II\_USERADT in a Test Installation

Because various Ingres programs are installed with privilege, II\_USERADT must be defined at EXECUTIVE mode to be visible to these installed images. This requirement presents a problem in Ingres test installations, because logical names are not defined at a SYSTEM level in test installations and the current release of VMS only sees SYSTEM-level EXECUTIVE mode logical names when invoking privileged images.

To change this behavior, you must redefine the logical name LNM\$FILE\_DEV at EXECUTIVE mode to include the logical name tables in which to look.

To run the test installation, issue the following command:

```
$ define/exec/table = lnm$process_directory- lnm$file_dev lnm$process,lnm$job,-  
lnm$group,lnm$system
```

Consult your system manager and the VMS documentation for more details.

## How You Install New Data Types in a UNIX Environment

To install your user-defined data types, the DBMS, RCP, and JSP executables must be relinked (loaded) using the iilink command. The iilink command prompts for the object modules to be linked into these executables. At the prompt, specify a list of object modules, an archive library, or, if your system supports it, a shared object.

On systems which support shared object access, avoid the -l form of shared object access. The executables being linked are setuid programs, and the combination of shared objects with setuid programs leads to a number of complications in dealing with multiple installations.

For the purpose of testing, you can link only the DBMS by specifying the -dbms flag when you issue the iilink command. When you link to create your production executable, omit the -dbms flag.

The iilink utility prompts for an extension for the executable filename. The default is to create standard executables. For test purposes you can add an extension to the file names. For example, if you specify "test" as the extension, iilink creates the executable iibms.test instead of iibms.

The iilink command can only be run by the ingres user.

Optionally, spatial objects or demo user-defined data types can be installed using the iilink utility.

## Testing the New Data Type Code

Because your code runs as part of the DBMS Server, any bugs in your code can have severe consequences to the system. To avoid damaging production data, run your code in a test installation before you install it in the production installation.

Symbols within the DBMS Server are not available for customer use.

Because it is inconvenient during testing to have to shut down and restart an entire installation to check out the system, Ingres has provided the field `add_risk_consistency` in the `IIADD_DEFINITION` structure. If set to the value `IIADD_INCONSISTENT`, this flag allows the system to startup when a newly started process does not agree with the remainder of the Ingres installation.

**Caution!** Running with the `add_risk_consistency` set to `IIADD_INCONSISTENT` risks the data integrity of all databases in that installation. If failures occur, it is possible that the DBMS Server cannot back out transactions from any database. This flag must be used with extreme care and must never be used in a production environment.



# Chapter 9: Using Abstract Spatial Data Types

---

This section contains the following topics:

[Use of Spatial Data Types, Operators, and Functions](#) (see page 87)

[Spatial Data Types in the Spatial Object Library](#) (see page 88)

[Spatial Operators](#) (see page 104)

[Functions that Support the Spatial Operators](#) (see page 110)

[Support Routines for Spatial Data Types](#) (see page 116)

[Ordering of Spatial Data Types](#) (see page 119)

[Polygon Length Limits](#) (see page 120)

[How You Install Spatial Data Types in UNIX or Linux Environments](#) (see page 121)

[How You Install Spatial Data Types in a VMS Environment](#) (see page 124)

[How You Install Spatial Data Types in a Windows Environment](#) (see page 125)

This chapter describes the use of spatial data types in SQL statements. It also describes spatial operators and functions.

## Use of Spatial Data Types, Operators, and Functions

The Ingres Spatial Object Library provides spatial data types that enable you to manipulate spatial data using Ingres. Although spatial data types represent data that is more complex than the basic SQL data types, you can use them in any appropriate context within an SQL statement.

Spatial operators and functions enable you to perform complex operations on spatial data stored in tables.

## Spatial Data Types in the Spatial Object Library

The Spatial Object Library contains the following data types:

- point
- box
- lseg
- line and long line
- polygon and long polygon
- circle

In addition, there are integer variations of these data types:

- ipoint
- ibox
- ilseg
- iline
- ipolygon
- icircle

Finally, there is another data type that is used in rtree index processing: nbr (normalized bounding region).

## Point Data Type

The point data type consists of an x and a y coordinate value. Each coordinate value is specified using a floating point number (float8). The point data type is the major component of the other spatial data types. The format of the float literal in the point data type is as follows:

```
<x:b1>d.dddddddddddddE+eee
```

with 16-decimal digit precision and a 3-digit exponent. This enables the point data type to support Ingres float representation (-1.0e-38 to +1.0e+38 with 16-digit precision). If your hardware supports the IEEE standard for floating point numbers, the float type is accurate to 15-decimal precision and ranges from  $-10^{308}$  to  $+10^{308}$ .

The point data type is specified using two floats:

```
(x_value, y_value)
```

For example:

```
(3,4)
```

The following example creates and populates a table with point values. Points can be real or integer values.

```
create table point_table (id char(2), obj point);
insert into point_table values ('AA', '(1,1)');
insert into point_table values ('BA', '(0.0, 12.237)');
insert into point_table values ('GH', '(1603452, -20321)');
```

## Box Data Type

A box is defined as a rectangle whose sides are parallel to the coordinate system axis. Boxes do not have to be square. The box data type is specified as two points:

- Lower-left corner point
- Upper-right corner point

The string representation of the box data type is:

*(lower\_left\_point, upper\_right\_point)*

For example:

*((1,2.25), (5,6))*

All boxes must contain two different points. Identical points generate a box with both a width and height of zero.

The following example creates and populates a table with boxes.

```
create table box_table (id char(2), obj box);
insert into box_table values ('A', '((0,0), (2, 2)) ');
insert into box_table values ('B', '((-40.345, -40.123), (4.0, 4.0))');
insert into box_table values ('I', '((-160, -660), (60,60))');
```

## Lseg Data Type

The lseg (line segment) data type contains two parts:

- Begin point
- End point

The string representation of the lseg data type is:

*(begin\_point, end\_point)*

For example:

*((1,2), (3.46,-4.0))*

The begin and end points of the line segment cannot be identical. Identical points generate a zero length for the line segment.

The following example creates and populates a table with lseg attributes.

```
create table lseg_table (id char(2), obj lseg);
insert into lseg_table values ('A', '((0,0), (1,2))');
insert into lseg_table values ('B', '((120,160), (60,160))');
insert into lseg_table values ('I', '((-160,-660), (60,60))');
```

## Line Data Type

The line data type contains two parts:

- Number of points (*npoints*)
- An array of points

The *npoints* field is a 4-byte integer containing the number of points in the points array. The points array is an array of the point data type. Each array entry contains 16 bytes (an x and a y float value).

Line literals are specified as list of points. For example:

`((1,2), (3,4), (5,6), (6,6), (7,9))`

Each (x,y) pair must conform to the literal representation of the point data type. All lines must contain at least two points.

Tables are divided into 2048 byte pages. Of this total, 2000 bytes are available for data storages. This limits the total number of points available for the line (and polygon) data structure. Each point in the line requires 16 bytes.

**Windows and UNIX:** The *npoints* field is padded to 8 bytes.

Because the maximum row size is 2000, and  $8 + 124 * 16 = 1992$ , the maximum user-specified line size is 124 points. If the line size is 125, the row size is 2008, which is too big. The row cannot exceed the page boundary.

When you create line columns in tables, you specify the maximum number of points in the line (for example, `line(10)`). If shorter lines are inserted into the column, the same amount of storage space is consumed as for lines of the maximum (declared) size.

The following example creates and populates a table with line values.

```
create table line_table (id char(2), obj line(3));
insert into line_table values ('A', '((0,0), (1,1), (1,2)) ');
insert into line_table values ('B', '((60, 180), (120, 180), (90,130))');
insert into line_table values('C', '((120,160), (60,160))');
```

## Long Line Data Type

The long line data type has the same characteristics as the line data type, but can accommodate up to 2G (gigabytes) of data (approximately 100 million points). Long line data is stored in one or more 2048-byte segments. Each segment contains an integer indicating the number of points in the segment, followed by an array of points specified as pairs of 8-byte floating point values.

The long line data type can be inserted into and selected from tables using the varchar or long varchar data type—for details, see the *SQL Reference Guide*.

The long line data type is subject to the following restrictions:

- Long line columns cannot be part of a table key
- Long line columns cannot be part of a secondary index
- Long line columns cannot be used in the order by clause of a select statement
- Long line columns cannot have query optimization statistics
- Long line columns cannot be directly compared (must be coerced)

The following example creates a table with a long line column and inserts rows into it. Note that the long line column specification in the create table statement omits a length specifier.

```
create table long_line_table (id char(2), obj long line);
insert into long_line_table values ('A', '((0,0), (1,1),
(1,2), (60, 180), (120, 180), (90,130), (120,160), (60,160))');
```

## Polygon Data Type

The polygon data type contains two parts:

- Number of points (*npoints*)
- An array of points

The *npoints* part is a long integer containing the number of points in the points array. The polygon data type is identical to the line data type, except that polygon data type implies closure. The string representation of the polygon data type is a list of points:

$((x_1,y_1), (x_2,y_2), (x_3,y_3), \dots, (x_N,y_N))$

For example:

$((1,2), (3,4), (5,6), (6,6), (8,3))$

Each  $(x,y)$  pair conforms to the string representation of the point data type. All polygons must have a non-zero area, must not be self-intersecting, and must not contain duplicate points. A polygon data type can be either concave or convex.

**Windows and UNIX:** The *npoints* field is padded to 8 bytes.

The polygon data type contains a 4-byte integer *npoints* field and an array of a variable number of points. Because the maximum row size is 2000, and  $8 + 124 * 16 = 1992$ , the maximum number of vertices is 124 and the minimum number of vertices is 3 (at least 3 points are required to form a polygon).

When you create polygon columns, you specify the maximum number of points in the polygon—for example, "mypoly polygon(11)". Every polygon you insert into the table consumes the maximum space required for a polygon of the declared length, even if the polygon has fewer points than the column can accommodate.

The following example creates and populates a table with polygon attributes.

```
create table polygon_table (id char(2), obj polygon(3));
insert into polygon_table values ('Z', '((0,0), (2,1), (1,2)) ');
insert into poly_table values ('A', '((-20, -20), (-20, 20), (20,20))');
insert into poly_table values ('B', '((-40, -40), (-40, 40), (40,40))');
insert into poly_table values ('C', '((-60, -60), (-60, 60), (60,60))');
```

## Long Polygon Data Type

The long polygon data type has the same characteristics as the polygon data type, but can accommodate up to 2G (gigabytes) of data (approximately 100 million points). Long polygon data is stored in one or more 2048-byte segments. Each segment contains an integer indicating the number of points in the segment, followed by an array of points specified as pairs of 8-byte floating point values.

The long polygon data type can be inserted into and selected from tables using the varchar or long varchar data type—for details, see the *SQL Reference Guide*.

The long polygon data type is subject to the following restrictions:

- Long polygon columns cannot be part of a table key
- Long polygon columns cannot be part of a secondary index
- Long polygon columns cannot be used in the order by clause of a select statement
- Long polygon columns cannot have query optimization statistics
- Long polygon columns cannot be directly compared (must be coerced)

The following example creates a table with a long polygon column and inserts rows into it. Note that the long polygon column specification in the create table statement omits a length specifier.

```
create table long_poly_table (id char(2), obj long polygon);
insert into long_poly_table values ('Z', '((0,1), (1,1),
(1,2), (160, 280), (220, 340), (190,930), (120,260), (60,160))');
```

## Circle Data Type

The circle data type is composed of two parts:

- Center point
- Radius

The center point is of type point. The radius is a floating point number (double). The literal representation of the circle data type is:

*(center\_point, radius)*

For example:

*((1.25,4.32), 5.1)*

The following example creates a table with a circle column and inserts rows.

```
create table circle_table (id char(2), obj circle);
insert into circle_table values ('S', '((1,1),1)');
insert into circle_table values ('A', '((0, 0),20)');
insert into circle_table values ('B', '((0, 0),40)');
insert into circle_table values ('C', '((0,0),60)');
```

## Ipoint Data Type

The ipoint data type consists of an x and a y coordinate value. Each coordinate value is specified using an integer number (integer4). The ipoint data type is the major component of the other spatial data types. The format of the integer literal in the ipoint data type is as follows:

```
<x:b1>ddddddddd
```

with 10-digit precision. This enables the ipoint data type to support Ingres integer representation (-2,147,483,648 to +2,147,483,647).

The ipoint data type is specified using two integers:

```
(x_value, y_value)
```

For example:

```
(3, 4)
```

The following example creates and populates a table with ipoint values. Ipoints can only be integer values.

```
create table ipoint_table (id char(2), obj ipoint);
insert into ipoint_table values ('AA','(1,1)');
insert into ipoint_table values ('BB','(4233, 133333)');
insert into ipoint_table values ('CC','(1603452, -20321)');
```

## Ibox Data Type

An ibox is a box of ipoints.

An ibox is defined as a rectangle whose sides are parallel to the coordinate system axis. Iboxes do not have to be square. The ibox data type is specified as two points:

- Lower-left corner point
- Upper-right corner point

The string representation of the ibox data type is:

*(lower\_left\_point, upper\_right\_point)*

For example:

*((1,2), (5,6))*

All iboxes must contain two different points, because identical points generate a box with both a width and height of zero.

The following example creates and populates a table with iboxes.

```
create table ibox_table (id char(2), obj ibox);
insert into ibox_table values ('A', '((0,0), (2, 2)) ');
insert into ibox_table values ('B', '(( -40, -40), (4, 4))');
insert into ibox_table values ('I', '(( -160, -160), (60,60))');
```

## Ilseg Data Type

An ilseg is an lseg of ipoints.

The ilseg (integer line segment) data type contains two parts:

- Begin point
- End point

The string representation of the ilseg data type is:

*(begin\_point, end\_point)*

For example:

*((1,2), (3,4))*

The begin and end points of the line segment cannot be identical, because identical ipoints generate a zero length for the integer line segment.

The following example creates and populates a table with ilseg values.

```
create table ilseg_table (id char(2), obj ilseg);
insert into ilseg_table values ('A', '((0,0), (1,2))');
insert into ilseg_table values ('B', '((120,160), (60,160))');
insert into ilseg_table values ('I', '(((-160,-660), (60,60))');
```

## Iline Data Type

An iline is a line of ipoints.

The iline data type contains two parts:

- Number of points (npoints)
- An array of ipoints

The *npoints* field is a 4-byte integer containing the number of ipoints in the ipoints array. The ipoints array is an array of the ipoint data type. Each array entry is 8 bytes (an x and a y integer value).

Iline literals are specified as list of ipoints, for example:

((1, 2), (3, 4), (5, 6), (6, 6), (7, 9))

Each (x,y) pair must conform to the literal representation of the ipoint data type. All ilines must contain at least two points, must not be self-intersecting, and must not contain duplicate points.

Columns are limited to 2000 bytes in Ingres. This limits the total number of ipoints available for the iline (and ipolygon) data structure. Each ipoint in the iline requires 8 bytes.

For example, a table with an iline column defined as "iline(10)" requires 84 bytes:

$\text{iline}(10) = 4 \text{ bytes (npoints)} + 10 * 8 \text{ bytes (ipoints array)} = 84 \text{ bytes}$

Because the maximum column size is 2000, and  $4 + 249 * 8 = 1992$ , the maximum user-specified iline size is 249 ipoints. If the iline size is 250, the row size is 2004, which is too big. The row cannot exceed the column boundary.

When you create iline columns in tables, you specify the maximum number of ipoints in the iline (for example, iline(10)). If shorter lines are inserted into the column, the same amount of storage space is consumed as for lines of the maximum (declared) size. If the table is compressed, the actual storage consumed by iline(10) with 3 points is less than with 4 or more points.

The following example creates and populates a table with iline values.

```
create table iline_table (id char(2), obj iline(3));
insert into iline_table values ('A', '((0,0), (1,1), (1,2))');
insert into iline_table values ('B', '((60,180), (120,180),
(90,130))');
insert into iline_table values ('C', '((120,160), (60,160))');
```

## Ipolygon Data Type

An ipolygon is a polygon of ipoints.

The ipolygon data type contains two parts:

- Number of ipoints (*npoints*)
- An array of ipoints

The *npoints* part is a 4-byte integer containing the number of ipoints in the ipoints array. The ipolygon data type is identical to the *iline* data type, except that ipolygon data type implies closure. The string representation of the ipolygon data type is a list of points:

`((x1,y1), (x2,y2), (x3,y3),..., (xN,yN))`

For example:

`((1,2), (3,4), (5,6), (6,6), (8,3))`

Each (x,y) pair conforms to the string representation of the ipoint data type. All ipolygons must have a non-zero area, must not be self-intersecting, and must not contain duplicate points. An ipolygon data type can be either concave or convex.

Each ipoint in the ipolygon requires 8 bytes. For example, a table created with an ipolygon defined as "ipolygon(11)" requires 92 bytes:

`ipolygon(11) = 4 bytes (for npoints) + 11*8 bytes (points array) = 92 bytes`

The ipolygon data type contains a 4-byte integer *npoints* field and an array of a variable number of ipoints. Because the maximum row size is 2000, and  $4 + 249*8 = 1996$ , the maximum number of vertices is 249 and the minimum number of vertices is three (at least three points are required to form a polygon).

When you create ipolygon columns, you specify the maximum number of points in the ipolygon—for example, "myipoly ipolygon(11)". Every ipolygon you insert into the table consumes the maximum space required for a ipolygon of the declared length, even if the ipolygon has fewer points than the column can accommodate. If the table is compressed, the actual storage consumed by ipolygon(11) with 3 points is less than with 4 or more points.

The following example creates and populates a table with ipolygon attributes.

```
create table ipolygon_table (id char(2), obj ipolygon(3));
insert into ipolygon_table values ('Z', '((0,0), (2,1), (1,2))');
insert into ipoly_table values ('A', '((-20, -20), (-20, 20,
(20,20))');
insert into ipoly_table values ('B', '((-40, -40), (-40, 40)
(40,40))');
insert into ipoly_table values ('C', '((-60, -60), (-60, 60),
(60,60))');
```

## Icircle Data Type

An icircle is a circle with an ipoint center and an integer radius.

The icircle data type is composed of two parts:

- Center point
- Radius

The center point is of type ipoint. The radius is a 4-byte number. The literal representation of the icircle data type is:

*(center\_point, radius)*

For example:

*((1,2), 5)*

The following example creates a table with an icircle column and inserts rows.

```
create table icircle_table (id char(2), obj icircle);
insert into icircle_table values ('S', '((1,1), 1)');
insert into icircle_table values ('A', '((0,0), 20)');
insert into icircle_table values ('B', '((0,0), 40)');
insert into icircle_table values ('C', '((0,0), 60)');
```

## Nbr Data Type

An nbr is similar to an ibox. It is comprised of four 3-byte integers in a lower-left and upper-right arrangement like a box or an ibox. However, an nbr can have the same lower-left and upper-right coordinates, like a point has, or the same x or y coordinates, like a line has.

An nbr is defined as a rectangle whose sides are parallel to the coordinate system axis. Further, the nbr is normalized within a coordinate system. The nbr coordinate values are normalized to values from (0,0). An nbr cannot be specified; it must be derived from a spatial object and a range, for example, `nbr(object,range)`. The range is either a box or an ibox. The range describes the entire region that contains all objects. In other words, the range is the smallest x, y values as the lower-left corner, and the largest x, y as the upper-right corner. For instance, the range of latitude and longitude is `((-180,-90),(180,90))`.

## Spatial Data Types Storage Formats

Spatial data is stored as shown in the following table:

Notation	Storage Format	Range
point	2 8-byte floats	-1.0e+38 to +1.0e+38 (16 digit precision)
lseg	2 points (begin_point, end_point)	-1.0e+38 to +1.0e+38 (16 digit precision)
line	1 integer	(2-124)
	1 array of points	-1.0e+38 to +1.0e+38 (16 digit precision)
box	2 points (lower_left, upper_right)	-1.0e+38 to +1.0e+38 (16 digit precision)
polygon	1 integer	(3-124)
	1 array of points	-1.0e+38 to +1.0e+38 (16 digit precision)
circle	1 point (center_point)	-1.0e+38 to +1.0e+38 (16 digit precision)
	1 float (radius)	same as float (must have a value > 0.0)
ipoint	2 4-byte integers	-2,147,483,648 to +2,147,483,647

Notation	Storage Format	Range
ilseg	2 ipoints (begin_point, end_point)	-2,147,483,648 to +2,147,483,647
iline	1 4-byte integer	2 to 249
	1 array of ipoints	-2,147,483,648 to +2,147,483,647
ibox	2 ipoints (lower_left, upper_right)	-2,147,483,648 to +2,147,483,647
ipolygon	1 4-byte integer	3 to 249
	1 array of points	-2,147,483,648 to +2,147,483,647
icircle	1 ipoint (center_point)	-2,147,483,648 to +2,147,483,647
	1 4-byte integer (radius)	1 to +2,147,483,647
nbr	4 3-byte integers	1 to +16,777,215

**Note:** If your hardware supports the IEEE standard for floating point numbers, the float type is accurate to 15-decimal precision and ranges from  $-10^{308}$  to  $+10^{308}$ . This also applies to the spatial data types.

## Spatial Operators

Spatial operators let you perform complex operations on spatial data stored in tables. The spatial operators that can be used with spatial data types are as follows:

- Equality operators
- Binary spatial operators
- Overlaps operators

## Equality Operators

The equality operator (=) compares the internal representation of two operands and determines whether they represent identical values. This operator is valid only when comparing operands of the same type. You cannot compare a box to a polygon, for example.

You can also compare spatial data types for inequality, using the "<>" (not equal to) comparison operator. All comparisons are performed on a point-by-point basis (except for the circle data type), and similar but unequal operands fail this test. For example, when testing line segments:

```
((1,2), (3,4)) <> ((3,4), (1,2))
```

## Binary Spatial Operators

There are three binary spatial operators:

**Inside**—Determines whether an object of a particular data type is inside another spatial object

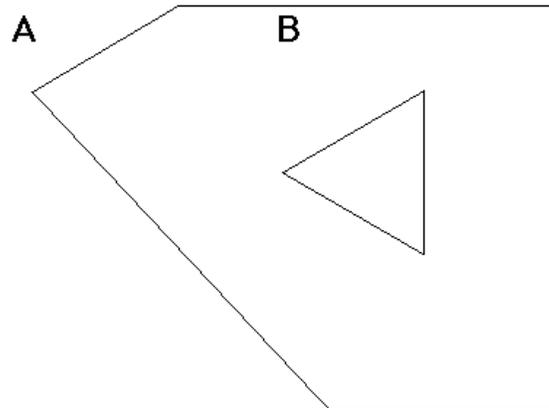
**Intersects**—Determines whether an object of a particular data type intersects another spatial object

**Overlaps**—Determines whether two spatial data objects have any points in common

Prefix and postfix notation is supported for these operators. The prefix notation for these operators is "operator(operand1, operand2)". By contrast, the comparison (=, <>, >, <, <=, >=) and logical (and, or, not) operators support infix notation, which is not available for binary spatial operators.

## Inside Operators

The inside spatial operators determine whether one operand is contained within the boundary of another operand, as shown in the following example, where B is inside A:



An operand that shares a line segment, a portion of a line segment, or even a single point with another operand (congruency) and meets the other qualifications for inside, satisfies the definition of inside. Boundaries, therefore, are considered to be inside.

If the result of inside is true, the inside operator returns a value of 1, otherwise it returns a value of 0. Prefix and postfix notation is supported, so the inside operator is defined as:

`inside (spat_type1, spat_type2) = 1`

or:

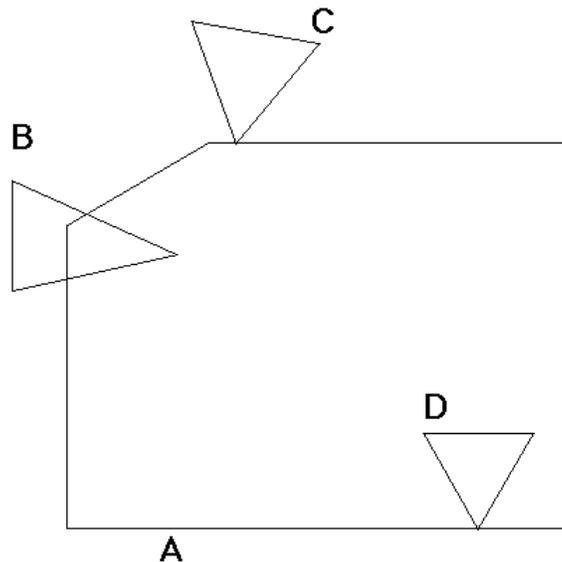
`spat_type1 inside spat_type2`

The following table shows the combinations of spatial data types for which inside spatial operators are supported:

	<b>point</b>	<b>box</b>	<b>lseg</b>	<b>line and long line</b>	<b>polygon and long polygon</b>	<b>circle</b>
point	N	Y	N	N	Y	Y
box	N	Y	N	N	Y	Y
lseg	N	Y	N	N	Y	Y
line and long line	N	Y	N	N	Y	Y
polygon and long polygon	N	Y	N	N	Y	Y
circle	N	Y	N	N	Y	Y

## Intersects Operator

The intersects spatial operator determines whether one operand intersects one or more points or edges (boundaries) of another operand. In the following illustration, B intersects A, C intersects A, and D intersects A:



If the result of intersects is true, the intersects operator returns a value of 1; otherwise the value returned is zero.

Prefix and postfix notation is supported-the intersects operator is specified as follows:

`intersects (spat_type1, spat_type2) = 1`

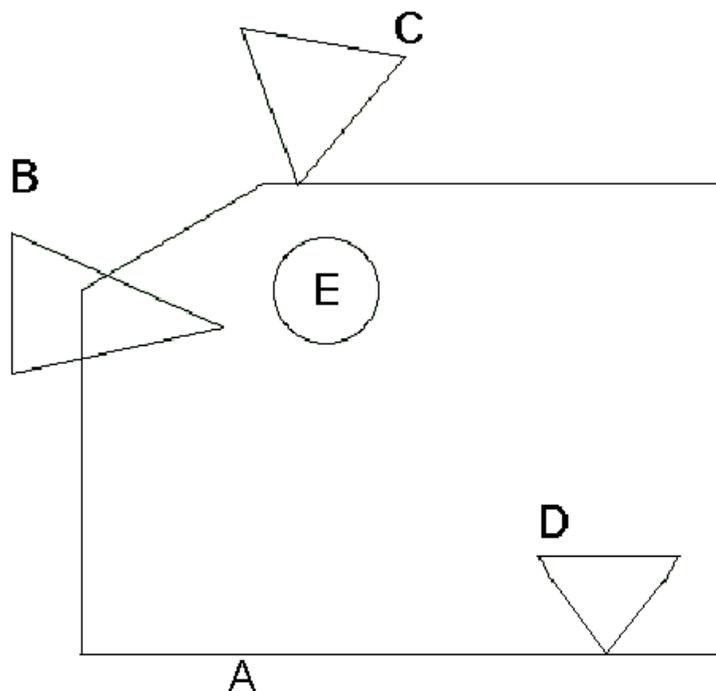
or:

`spat_type1 intersects spat_type2`

You can use the intersects operator with all of the spatial data types.

## Overlaps Operator

The overlaps spatial operator determines whether one operand is wholly contained within the boundary of another operand or if one operand intersects one or more points or edges of another spatial object. Hence, object A overlaps object B if there are any common points between A and B. In the following illustration, B overlaps A, C overlaps A, D overlaps A, and E overlaps A:



Note that B, C, and D both overlap and intersect A, however, E, which is wholly contained in A, only overlaps A.

If the result of overlaps is true, the overlaps operator returns a value of 1; otherwise the value returned is zero.

Prefix and postfix notation is supported-the overlaps function can be specified as follows:

```
overlaps (spat_type1, spat_type2) = 1
```

or:

```
spat_type1 overlaps spat_type2
```

You can use the overlaps function with all of the spatial data types.

## Nbr Function

The `nbr` spatial function produces an `nbr` data type from a spatial object and a box or an `ibox`. The box or `ibox` specifies the range of values that the object can take on. The `nbr` is a normalized bounding region for the object. The `nbr` can be thought of as a concise bounding box. Note that `bbox(point)` is not supported, but an `nbr(point,box)` is supported.

The `nbr` result is a pair of two 3-byte coordinates, for a total of 12 bytes. It is stored in lower-left `x, y` and upper-right `x, y` sequence, similar to a box.

The `nbr` is used by the `hilbert` function:

```
nbr_type1 = nbr (spat_type, box_of_range_type1)
```

You can use the `nbr` function with all the spatial data types. The range must be specified as a box or an `ibox`.

## Hilbert Function

The Hilbert spatial function produces a 6-byte field from an `nbr`. The Hilbert value has the property that objects with close Hilbert values are spatially close. However, it is possible to find two spatially close objects that have very different Hilbert numbers. With the Hilbert value you can spatially cluster tables for better performing spatial queries and spatial joins by storing the rows in Hilbert sequence.

Hilbert values are not unique; it is possible for two close spatial objects to generate the same Hilbert value.

The Hilbert result is a 6-byte binary value:

```
hilbert_value = hilbert (nbr_type1)
```

This can be used with the `nbr` and `box` functions:

```
hilbert_value = hilbert ( nbr(spat_type1, box(range_type1)) )
```

## Functions that Support the Spatial Operators

Spatial functions let you perform complex operations on spatial data stored in tables. There are three types of functions that support the spatial operators:

- Spatial functions
- Spatial conversion functions
- Type conversion functions

## Spatial Functions

You can use the spatial functions to perform calculations on the spatial data types. These functions all return floating point (double) values as a result. There are four spatial functions:

**Area**—calculates the area of a box, circle, polygon, long polygon, ibox, icircle, or ipolygon

**Length**—calculates the length of a line, long line, lseg, iline, or ilseg

**Perimeter**—calculates the perimeter of a box, circle, polygon, long polygon, ibox, icircle, or ipolygon

**Distance**—calculates the distance between two points or two ipoints

The distance function differs from the other functions because it requires two operands (points) as arguments.

## Spatial Conversion Functions

The four spatial conversion functions accept a spatial data type and return either a point value or a floating point (double) value corresponding to either the x or y value of the point. These functions are:

- Point\_x
- Point\_y
- Box\_ll
- Box\_ur

The box\_ll and box\_ur functions each accept a box data type as input and return a point value corresponding to either the lower-left or upper-right corner point. Similarly, these functions also accept an ibox and return an ipoint value.

The following table lists the explicit type conversion functions available for the spatial data types:

<b>Name</b>	<b>First Operand</b>	<b>Second Operand</b>	<b>Result Type</b>	<b>Description</b>
bbox	lseg line long line polygon long polygon circle		box	Converts any of the defined operands to internal box representation. Bounding box limits are defined as the minimum and maximum values found in any x,y data pair in the data type being converted.
bbox	iline ilseg ipolygon icircle		ibox	Converts any of the defined operands to internal ibox representation.
box	char varchar		box	Converts a char or varchar string to internal box representation.
box	ibox		box	Converts an ibox to a box.
box	point	point	box	Converts two points to an internal box representation.
box_ll	ibox		ipoint	Returns the lower-left coordinate of an ibox.
box_ur	ibox		ipoint	Returns the upper-right coordinate of an ibox.
box_ll	box		point	Returns the lower-left coordinate of a box.
box_ur	box		point	Returns the upper-right coordinate of a box.
char	point lseg box line polygon circle ipoint ilseg ibox iline ipolygon icircle		char	Converts the spatial data type to its character representation.

<b>Name</b>	<b>First Operand</b>	<b>Second Operand</b>	<b>Result Type</b>	<b>Description</b>
	nbr			
circle	char varchar		circle	Converts a char or varchar string to internal circle representation.
circle	icircle		circle	Converts an icircle to a circle.
circle	point	float	circle	Converts a point and a float to an internal circle representation.
hilbert	nbr		varbyte	Calculates the hilbert value for an nbr. The hilbert is half as long as the nbr.
ibox	char varchar		ibox	Converts a char or varchar string to internal ibox representation.
ibox	ipoint	ipoint	ibox	Converts two ipoints to an internal ibox representation.
icircle	char varchar		icircle	Converts a char or varchar string to internal icircle representation.
icircle	ipoint	integer	icircle	Converts an ipoint and an integer to an internal icircle representation.
iline	char varchar		iline	Converts a char or varchar string to an internal iline representation.
ilseg	char varchar		ilseg	Converts a char or varchar string to an internal ilseg representation.
ilseg	ipoint	ipoint	ilseg	Converts two ipoints to an internal ilseg representation.
ipoint	char varchar		ipoint	Converts a char or varchar string to internal ipoint representation.
ipoint	integer	integer	ipoint	Converts two integers to an internal ipoint representation.

<b>Name</b>	<b>First Operand</b>	<b>Second Operand</b>	<b>Result Type</b>	<b>Description</b>
ipolygon	char varchar		ipolygon	Converts a char or varchar string to internal ipolygon representation.
line	char varchar		line	Converts a char or varchar string to internal line representation.
long_line	char varchar		long line	Converts a char or varchar string to internal long line representation.
long_polygon	char varchar		line	Converts a char or varchar string to internal long polygon representation.
lseg	char varchar		lseg	Converts a char or varchar string to internal lseg representation.
lseg	ilseg		lseg	Converts an ilseg to an lseg.
lseg	point	point	lseg	Converts two points to an internal lseg representation.
nbr	point box lseg line polygon circle ipoint ibox ilseg iline ipolygon icircle	box ibox	nbr	Converts a spatial data type with a range specified by a box or an ibox to an internal nbr representation.  The nbr function supports the integer spatial types only when the second parameter is ibox and float spatial types only when the second parameter is box.
point	char varchar		point	Converts a char or varchar string to internal point representation.
point	float	float	point	Converts two floats to an internal point representation.
point_x	ipoint		integer	Returns the x coordinate of an ipoint.
point_y	ipoint		integer	Returns the y coordinate of an ipoint.

<b>Name</b>	<b>First Operand</b>	<b>Second Operand</b>	<b>Result Type</b>	<b>Description</b>
polygon	char varchar		polygon	Converts a char or varchar string to internal polygon representation.
varbyte	point box lseg line polygon circle ipoint ibox ilseg iline ipolygon icircle nbr		varbyte	Copies a spatial data type in its binary form to a varbyte representation.
varchar	point box lseg line polygon circle ipoint ibox ilseg iline ipolygon icircle nbr		varchar	Converts a spatial data type to its character representation in varchar form.

## Support Routines for Spatial Data Types

The routines in the following tables can be used with the spatial data types to perform an explicit type conversion between SQL data types and spatial data types. Each routine returns a zero for success and a non-zero value for failure.

To call these routines you must include the following files in your source code:

- spatialc.h
- spcirf.h
- spplyf.h
- splinf.h
- spsegf.h
- spboxf.h
- sppntf.h

The following table describes the support routines for all the spatial data types:

<b>Routine Name and Description</b>	<b>Inputs</b>	<b>Outputs</b>	<b>Implementation</b>
POINT_TO_CHAR Converts point data types to character representation.	Pointer to a point Pointer to a 2000 char buffer	Buffer filled with null-terminated character representation of a point	int POINT_TO_CHAR (POINT *pnt, char*buf)
CHAR_TO_POINT Converts character string to point data type.	Pointer to null-terminated character string Pointer to point to be filled	Point variable filled with point value	int CHAR_TO_POINT (char*str, POINT*pnt)
BOX_TO_CHAR Converts box data type to character representation.	Pointer to a box Pointer to a 2000 char buffer	Buffer filled with null-terminated character representation of box	int BOX_TO_CHAR (BOX *the_box, char*buf)
CHAR_TO_BOX Converts character string to box data type.	Pointer to null-terminated character string Pointer to box to be filled	Box variable filled with box value	int CHAR_TO_BOX (char *str, BOX* the_box)
LSEG_TO_CHAR Converts lseg data type to character	Pointer to an lseg Pointer to a 2000 char buffer	Buffer filled with null-terminated character representation of	int LSEG_TO_CHAR (LSEG *the_lseg, char *buf)

<b>Routine Name and Description</b>	<b>Inputs</b>	<b>Outputs</b>	<b>Implementation</b>
representation.		lseg	
CHAR_TO_LSEG Converts character string to lseg data type.	Pointer to null-terminated character string  Pointer to lseg to be filled	Lseg variable filled with lseg value	int CHAR_TO_LSEG (char *str, LSEG *the_lseg)
LINE_TO_CHAR Converts line data type to character representation.	Pointer to a line  Pointer to a 2000 char buffer	Buffer filled with null-terminated character representation of line	int LINE_TO_CHAR (LINE *lin, char *buf)
CHAR_TO_LINE Converts character string to line data type.	Pointer to null-terminated character string  Pointer to line to be filled maximum number of points in line	Line variable filled with line points	int CHAR_TO_LINE (char *str, LINE *lin, long max_points)
POLYGON_TO_CHAR Converts polygon data type to character representation.	Pointer to a polygon  Pointer to a 2000 char buffer	Buffer filled with null-terminated character representation of polygon	int POLYGON_TO_CHAR (POLYGON *poly, char *buf)
CHAR_TO_POLYGON Converts character string to polygon data type.	Pointer to null-terminated character string  Pointer to polygon to be filled  Maximum number of points in polygon	Polygon variable filled with polygon vertices	int CHAR_TO_POLYGON (char *str, POLYGON *poly, long max_points)
CIRCLE_TO_CHAR Converts circle data type to character representation.	Pointer to a circle  Pointer to a 2000 char buffer	Buffer filled with null-terminated character representation of circle	int CIRCLE_TO_CHAR (CIRCLE *the_circle, char *buf)
CHAR_TO_CIRCLE Converts character string to circle data type.	Pointer to null-terminated character string  Pointer to circle to be filled	Circle variable filled with circle value	int CHAR_TO_CIRCLE (char *str, CIRCLE *the_circle)
IPOINT_TO_CHAR	Pointer to a ipoint	Buffer filled with null-	int IPOINT_TO_CHAR

<b>Routine Name and Description</b>	<b>Inputs</b>	<b>Outputs</b>	<b>Implementation</b>
Converts ipoint data types to character representation.	Pointer to a 2000 char buffer	terminated character representation of an ipoint	(IPOINT* ipnt, char* buf)
<b>CHAR_TO_IPOINT</b> Converts character string to ipoint data type.	Pointer to null-terminated character string  Pointer to ipoint to be filled	Point variable filled with ipoint value	int CHAR_TO_IPOINT (char* str, IPOINT* ipnt)
<b>IBOX_TO_CHAR</b> Converts ibox data type to character representation.	Pointer to an ibox  Pointer to a 2000 char buffer	Buffer filled with null-terminated character representation of ibox	int IBOX_TO_CHAR (IBOX* ibox, char* buf)
<b>CHAR_TO_IBOX</b> Converts a character string to ibox data type.	Pointer to null-terminated character string  Pointer to ibox to be filled	Box variable filled with ibox value	int CHAR_TO_IBOX (char* str, IBOX ibox)
<b>ILSEG_TO_CHAR</b> Converts lseg data type to character representation.	Pointer to an ilseg  Pointer to a 2000 char buffer	Buffer filled with null-terminated character representation of ilseg	int ILSEG_TO_CHAR (ILSEG* ilseg, char* buf)
<b>CHAR_TO_ILSEG</b> Converts a character string to ilseg data type.	Pointer to null-terminated character string  Pointer to ilseg to be filled	Ilseg variable filled with lseg value	int CHAR_TO_ILSEG (char* str, ILSEG ilseg)
<b>ILINE_TO_CHAR</b> Converts iline data type to character representation.	Pointer to an iline  Pointer to a 2000 char buffer	Buffer filled with null-terminated character representation of iline	int ILINE_TO_CHAR (ILINE* iline, char* buf)
<b>CHAR_TO_ILINE</b> Converts a character string to iline data type.	Pointer to null-terminated character string  Pointer to iline to be filled maximum number of points in iline	Line variable filled with iline points	int CHAR_TO_ILINE (char* str, ILINE iline)

<b>Routine Name and Description</b>	<b>Inputs</b>	<b>Outputs</b>	<b>Implementation</b>
<b>IPOLYGON_TO_CHAR</b> Converts ipolygon data type to character representation.	Pointer to an ipolygon Pointer to a 2000 char buffer	Buffer filled with null-terminated character representation of ipolygon	int IPOLYGON_TO_CHAR (IPOLYGON* ip, char* buf)
<b>CHAR_TO_IPOLYGON</b> Converts character string to ipolygon data type.	Pointer to null-terminated character string Pointer to ipolygon to be filled Maximum number of points in ipolygon	Ipolygon variable filled with ipolygon vertices	int CHAR_TO_IPOLYGON (char* str, IPOLYGON* ip)
<b>ICIRCLE_TO_CHAR</b> Converts icircle data type to character representation.	Pointer to a icircle Pointer to a 2000 char buffer	Buffer filled with null-terminated character representation of icircle	int ICIRCLE_TO_CHAR (ICIRCLE* icirc, char* buf)
<b>CHAR_TO_ICIRCLE</b> Converts character string to icircle data type.	Pointer to null-terminated character string Pointer to icircle to be filled	Circle variable filled with icircle value	int CHAR_TO_ICIRCLE (char* str, ICIRCLE icirc)

## Ordering of Spatial Data Types

You can specify spatial data type columns in the order by clause of select statements, and as key columns in create index statements. However, the order in which spatial data is returned (as the result of a query) is not guaranteed to have any particular geometric meaning.

## Polygon Length Limits

Polygon, line, ipolygon, and iline columns can contain a variable number of points or ipoints. Ipoints are stored as two 4-byte integer values, whereas points are stored as two 8-byte floating point values.

The number of points that can be stored in a column is limited to 2000.

The maximum number of ipoints that can be stored in an ipolygon or an iline is 249. The maximum number of points that can be stored in a polygon or line is 124.

## How You Install Spatial Data Types in UNIX or Linux Environments

The spatial object package is included in the standard distribution on UNIX and Linux. The spatial object files are installed by default during an RPM install on Linux or a full ingbuild install on UNIX.

Custom ingbuild installs allow you to install spatial objects at a later time than the initial install. Before you can use spatial data types in your applications, you must install the spatial object package and relink the server.

Follow these steps to install the package and relink the server:

1. Shut down the existing Ingres installation by issuing the following command:

```
ingstop
```

2. Install the spatial data types library from the distribution.

For UNIX environments, the package can be installed using ingbuild.

### To install the package using ingbuild

- a. Change to \$II\_SYSTEM/ingres/install directory, and start the forms-based Ingres installation utility:

```
ingbuild
```

- b. Respond to the installation utility dialogs. Select and install the Spatial Objects package from the custom install screen.

The spatial data types library is installed on your system.

3. Invoke the ilink utility by issuing the following command:

```
ilink
```

The ilink utility enables you to link in spatial data types or other user-defined data types. You can specify an extension for the file name of the server created by ilink so that the existing server file (iimerge) is not overwritten.

4. Restart the DBMS Server using the ingstart command.

You can now use spatial data types.

The following example illustrates the process of linking the DBMS Server to include spatial data types:

```
$ iilink
```

```
Loading INGRES merged server program ...
Using Shared Libraries ...
```

```
-----
|
| INGRES Spatial Objects consist of six spatial datatypes: POINT, BOX
| LINE, LINE SEGMENT, CIRCLE and POLYGON, as well as a number of
| spatial operators that operate on these spatial datatypes.
| INGRES Spatial Objects have been installed and configured and
| will automatically be included in the INGRES installation.
| To prevent the inclusion of the INGRES Spatial Object
| library re-run iilink with the -nosol option.
|
|-----
```

```
-----
|
| These INGRES binaries are loaded to allow you to add User Defined
| Data Types (UDTs) to this INGRES installation.
|
| You should now enter the modules where your User Defined Data Types
| are defined. You can either enter the name of an object file(s)
| or the name of a library.
| Examples are:
|   /project1/obj/*.o
|   /project1/obj/filename.o
|   /project1/lib/myuadt.a
|   $II_SYSTEM/ingres/demo/udadts/libdemoudt.1.so
|
| If you don't have any User Defined Data Types created, press RETURN,
| and the default object file will be used to load the INGRES binaries.
|
|-----
```

```
Enter the full pathname of the object file or library to be loaded, or press
RETURN for the default object file:
```

```
-----
|
| An extension may be supplied at this time to differentiate your test
| binaries from the existing ones.
|
| For example, if you enter the extension "test", the DBMS binary is
| created at:
|
|   $II_SYSTEM/ingres/bin/iimerge.test
|
| Otherwise, the DBMS binary is created at:
|
|   $II_SYSTEM/ingres/bin/iimerge
|
| where it overrides the existing DBMS binary.
|
|-----
```

-----

```
Enter the file extension for the test binaries:
Loading iimage ...

Done loading iimage:
-rwsr-xr-x 1 ingres users 18911 2007-07-10 12:06
/install/ingres/bin/iimage
Creating links to iimage...
/install/ingres/bin/cacheutil linked to iimage
/install/ingres/bin/dmfacp linked to iimage
/install/ingres/bin/dmfjsp linked to iimage
/install/ingres/bin/dmfrcp linked to iimage
/install/ingres/bin/iidbms linked to iimage
/install/ingres/bin/iishowres linked to iimage
/install/ingres/bin/iistar linked to iimage
/install/ingres/bin/lartool linked to iimage
/install/ingres/bin/lockstat linked to iimage
/install/ingres/bin/logdump linked to iimage
/install/ingres/bin/logstat linked to iimage
/install/ingres/bin/rcpconfig linked to iimage
/install/ingres/bin/rcpstat linked to iimage
/install/ingres/bin/repstat linked to iimage
Links to iimage have been created.
```

## How You Install Spatial Data Types in a VMS Environment

Ingres processes access the spatial objects and user-defined data types by means of shared images. The entry point is the routine `IIudadt_register()` for user-defined data types and `IIclsadt_register()` for Ingres spatial objects. Installing user-defined data types or Ingres spatial objects requires:

1. Building the shared image. This shared image is by default placed in `II_SYSTEM:[INGRES.LIBRARY]` and given the name `iiuseradt xx.exe` where `xx` is the two-character installation code.
2. Defining `II_USERADT` to point to the location of the shared image. This step is only necessary if you chose to place your spatial objects and user-defined data types in a location other than the default of `II_SYSTEM:[INGRES.LIBRARY]`.
3. Ensuring that the proper version of `II_USERADT` is installed.
4. Starting up Ingres server processes.

Ingres provides the template command files for building the shared image for use in a test and production installation. The templates are located in:

```
II_SYSTEM:[INGRES.LIBRARY]II_USERADT_BUILD.COM
```

```
II_SYSTEM:[INGRES.LIBRARY]II_CLSADT_BUILD.COM
```

```
II_SYSTEM:[INGRES.LIBRARY]II_ALLADT_BUILD.COM
```

**Note:** `II_CLSADT_BUILD.COM` and `II_ALLADT_BUILD.COM` are only installed if the spatial objects package is installed during the `VMSINSTAL` process.

The `II_USERADT_BUILD.COM` creates a shared image for user defined data types only; `II_CLSADT_BUILD.COM` creates a shared image for Ingres spatial objects only; and `II_ALLADT_BUILD.COM` creates a shared image for both Ingres spatial objects and user-defined data types. These scripts build "skeleton" versions of the respective shared images. For example code which can be used in an `II_USERADT` image, see the files in `II_SYSTEM:[INGRES.DEMO.UDADTS]`.

Follow the regular Ingres installation procedures to bring up the server processes.

If the installation was not completely shut down while the user built the `II_USERADTxx` image (that is, they only shut down the servers), the new image must be installed before bringing the servers up. The command to do this (from a suitably privileged account) is:

```
$ INSTALL replace II_USERADT
```

When you build the server after adding a spatial data type, you can see multiple occurrences of the following linker message:

```
%LINK-I-UNALIGNRELO, unaligned longword relocation generated at location  
%XXXXXXXXXX
```

This message is informational and does not require any action on your part.

## How You Install Spatial Data Types in a Windows Environment

Before you can use spatial data types in your applications, you must relink the IILIBUDT.DLL, as described in this section. To link the server, use the `illink` utility.

1. Stop Ingres by using Ingres Visual Manager, Ingres Service Manager, or the `ingstop` command.
2. Invoke the User Defined Data Type Linker Wizard.

The User Defined Data Type Linker utility enables you to link in spatial data types or other user-defined data types. You can specify an extension for the file name of the server created by the wizard so that the existing dynamic link library (IILIBUDT.DLL) is not overwritten.

To install Spatial Data Types, choose the Include Spatial Objects checkbox in the User Defined Data Type Linker dialog.

3. Restart Ingres through Ingres Visual Manager, Ingres Service Manager, or the `ingstart` command. You can now use spatial data types.



# Chapter 10: Writing Aggregate Functions

---

This section contains the following topics:

[Aggregate Function](#) (see page 127)

[Function Definitions for Aggregates](#) (see page 128)

[Code for an Aggregate Function](#) (see page 129)

This chapter gives a brief overview of writing aggregate functions for user-defined data types.

## Aggregate Function

Aggregate functions take a collection of values as input (contrasted with scalar functions, which take a single input). Aggregate functions are used to perform a summary operation on the set of input values.

The GROUP BY clause in SQL provides the basis for identifying the sets of parameter values. The set of rows with the same values for the GROUP BY columns produces the input parameters for each execution of the aggregate function, or in the absence of a GROUP BY clause, all rows from the query produce a single set of input parameters to the aggregate.

SQL supports the following aggregate functions for its intrinsic data types:

- count
- max
- min
- sum
- avg
- variance
- standard deviation
- correlation and regression analysis

## Function Definitions for Aggregates

Functions for user-defined types require the following:

- A function definition (using the IIADD\_FO\_DFN structure) to define the function name
- A function instance definition (using the IIADD\_FI\_DFN structure) to define the functions that will perform the operations

Function definitions for count(), max(), min() and sum() are already included in Ingres, in the same sense that arithmetic and comparison operations are pre-defined for user-defined data types; however, avg() is NOT pre-defined. Users can also code their own aggregate functions with distinct function names. The IIADD\_FO\_DFN structure instance must define "fod\_type" to be II\_AGGREGATE in this case.

Like all functions on user-defined data types, aggregate functions must also include definitions of each function instance for the function on a specific user-defined type. These are defined by IIADD\_FI\_DFN structure instances, and for aggregate functions they must include a "fid\_optype" of II\_AGGREGATE.

The instances of the function instance structure must be in a particular sorted sequence. Function instances with fid\_optype of II\_AGGREGATE must be placed between those for II\_OPERATOR and II\_NORMAL. If the function instance is for one of the standard Ingres aggregates, the "fid\_opid" field must be set to the appropriate code (II\_COUNT\_OP for COUNT, II\_MAX\_OP for MAX, II\_MIN\_OP for MIN, II\_SUM\_OP for SUM). If the function instance is for a user-defined aggregate function, the fid\_opid field must contain the value from the "fod\_id" field of the corresponding IIADD\_FO\_DFN structure instance.

**Note:** The function instance definitions must **not** use the II\_RES\_EXTERN value for the "fid\_rtype" field.

## Code for an Aggregate Function

The code generated into a query plan by Ingres for the evaluation of an aggregate function consists of three parts.

The first part is executed for each new set of GROUP BY column values. In the current implementation, Ingres builds a work field that contains either the “empty” value (as generated by the “getempty” method of the type definition) or if the function is max or min, the minimum or maximum value for the type as generated by the “minmaxdv” method. Each successive set of GROUP BY column values calls this code again to reset the work field to the initialization value.

The second part of code is executed for each row of a particular set of GROUP BY column values and invokes the function variable defined in the aggregate function instance definition. This function variable is passed the parameters defined for the function instance. The first parameter defines the result of the function execution and the second-through-nth-parameters describe the parameters of the aggregate function invocation syntax. Each parameter to the function variable is a pointer to a II\_DATA\_VALUE structure instance that describes and addresses the corresponding value. Since the function variable is called with each row to be aggregated, it is assumed that it will perform the aggregation into the result parameter. The result parameter is the same work field whose initialization is described in the preceding paragraph.

So, for example, an implementation of the function variable for the “max” aggregate might simply compare the current value of the aggregate parameter with the value in the work field, replacing it if the new value is “larger” (remember, that the work field will be initialized to the minimum value for the data type for each new group of rows). Likewise, an implementation of the function variable for the “sum” aggregate might add the current parameter value to the value in the work field, accumulating the sum in the work field.

The last part of code is executed after each group of rows (defined by a distinct set of GROUP BY column values) is processed. Ingres simply copies the current contents of the work field to the result location (based on the assumption that the aggregate is accumulated in the work field).

### Example—Function to perform the “sum” operation on the ORD\_PAIR type

```

/*
** Name: usop_sum() - sum a set of ord_pair's (just sums each element).
**
** Description:
**
** Inputs:
**   scb          Pointer to a session control block.
**   rdv          Pointer to II_DATA_VALUE to hold
**               resulting summed result.
**   dv1         Pointer to II_DATA_VALUE of the first

```

```

**                                     operand, which is a ORD_PAIR datatype.
**
** Outputs:
**   rdv
**   .db_data           Pointer to resulting currency value.
**
** Returns:
**   II_STATUS
**
## History:
##   19-oct-05 (inkdo01)
##   Written as proof of concept for UDT aggregation.
*/
II_STATUS
usop_sum(
II_SCB           *scb,
II_DATA_VALUE   *rdv,
II_DATA_VALUE   *dv1)
{
    ORD_PAIR     *ival, *rval;

    ival = (ORD_PAIR *)dv1->db_data;
    rval = (ORD_PAIR *)rdv->db_data;

    /* Simply accumulate the sums of the x & y values
       in the result work field. */
    rval->op_x += ival->op_x;
    rval->op_y += ival->op_y;

    return( II_OK );
}

```

**Note:** Ingres currently does not support the AVG operator for user-defined types because Ingres assumes a division operator is not generally available for user-defined types. (AVG is compiled as a SUM divided by a COUNT.) Users, however, can implement a SUM operator and explicitly code "sum(abc) / count(abc) as "avg(abc)" in a query. Also, users can code type-specific functions to perform AVG (for example, avg\_op, to compute the average of a set of ordered pairs) using an algorithm appropriate to the type.

# Appendix A: Checklist for Creating Data Types

---

This section contains the following topics:

[How You Create Data Types in Windows](#) (see page 131)

[How You Create Data Types in UNIX](#) (see page 132)

[How You Create Data Types in VMS](#) (see page 133)

This appendix summarizes the process of defining, installing, and testing a new data type or SQL function.

## How You Create Data Types in Windows

The general procedure for defining and installing user-defined data types and functions in a Windows environment is as follows:

1. Design and code the routines needed to perform the functions necessary to manipulate the new or existing data types.
2. Establish a test installation in which to test the new data types and functions.
3. Create the `IIudadt_register( )` routine.
4. Relink the `IILIBUDT.DLL` using the User Defined Data Type Linker Utility.
5. Build module tests that call all routines in the prescribed manner to test the basic functionality outside of the complexity of the Ingres system. It is advisable to verify that all sections of code are reached.
6. Start a new DBMS server or Star server and test the new functionality. Testing must involve a large variety of query types to ensure full coverage of the code.
7. Once you are confident that the server works, shut down and restart your entire test installation to verify that the support processes (the Recovery, Archiver, Star, and Remote Command) operate correctly. You can verify this through Ingres Visual Manager.

8. When the code has been fully tested, make sure that the `add_risk_consistency` field of the `IIADD_DEFINITION` data structure is set to `IIADD_CONSISTENT`. For more information about this field, see `Structure IIADD_DEFINITION Fields` (see page 76).
9. Install the new code in the target system.

Move all of your code to the desired location for your target system. Once your code is in place, you must shut down and restart your installation so that the support processes recognize the new code.

## How You Create Data Types in UNIX

The general procedure for defining and installing user-defined data types and functions in a UNIX environment is as follows:

1. Design and code the routines necessary to perform the functions necessary to manipulate the new or existing data types.
2. Establish a test installation in which to test the new data types and functions.
3. Create the `IIudadt_register( )` routine.
4. Relink the images by running the `ilink` script.
5. Build module tests which call all routines in the prescribed manner to test the basic functionality outside of the complexity of the Ingres system. It is advisable to verify that all sections of code are reached.
6. Start a new DBMS server or Star server and test the new functionality. Testing must involve a large variety of query types to ensure full coverage of the code.
7. Once you are confident that the server works, shut down and restart your entire test installation to verify that the support processes (the Recovery, Archiver, and cluster server) operate correctly. You can verify this by checking the following error logs:

```
$II_SYSTEM/ingres/files/errlog.log
```

```
$II_SYSTEM/ingres/files/ii_rcp.log,
```

```
$II_SYSTEM/ingres/files/ii_acp.log
```

**Note:** If your system is configured for Ingres Cluster Solution, each node in the Ingres cluster maintains a separate Archiver and Recovery error log. Each log is distinguished by having `_nodename` appended to the base log name, where `nodename` is the Ingres node name for the host machine as returned by `iipmhost`.

8. When the code has been fully tested, make sure that the `add_risk_consistency` field of the `IIADD_DEFINITION` data structure is set to `IIADD_CONSISTENT`. For more information about this field, see `Structure IIADD_DEFINITION Fields` (see page 76).
9. Install the new code in the target system.

Move all of your code to the desired location for your target system. Once your code is in place, you must shutdown and restart your installation so that the support processes recognize the new code.

## How You Create Data Types in VMS

The general procedure for defining and installing user-defined data types and functions in the VMS environment is as follows:

1. Design and code the routines necessary to perform the functions necessary to manipulate the new or existing data types.
2. Establish a test installation in which to test the new data types and functions.
3. Create the `IIudadt_register( )` routine.
4. Create the shared image by adding object module descriptions to the template command files found in `II_SYSTEM:[INGRES.LIBRARY]II_USERADT_BUILD.COM`.
5. Define the logical name `II_USERADT` to point to this shared image.
6. Build module tests which call all routines in the prescribed manner to test the basic functionality outside of the complexity of the Ingres system. It is advisable to verify that all sections of code are reached.
7. Start a new DBMS server or Star server and test the new functionality. Testing must involve a large variety of query types to ensure full coverage of the code.
8. Once you are confident that the server works, shutdown and restart your entire test installation to verify that the support processes (the Recovery, Archiver, and cluster server) operate correctly. You can verify this by checking the error logs:

`ii_config:errlog.log`

`ii_config:ii_rcp.log`

`ii_config:ii_acp.log`

**Note:** If your system is configured for Ingres Cluster Solution, each node in the Ingres cluster maintains a separate Archiver and Recovery error log. Each log is distinguished by having `_nodename` appended to the base log name, where `nodename` is the Ingres node name for the host machine as returned by `iipmhost`.

9. When the code has been fully tested, make sure that the `add_risk_consistency` field of the `IIADD_DEFINITION` data structure is set to `IIADD_CONSISTENT`. For more information about this field, see [Structure IIADD\\_DEFINITION Fields](#) (see page 76).
10. Install the new code in the target system.

Move all of your code to the desired location for your target system and repointing the `II_USERADT` logical if necessary. Once your code is in place, you must shutdown and restart your installation so that the support processes recognize the new code.

# Index

---

## (

(braces) • 8

## [

[ ] (brackets) • 8

## |

| (vertical bar) • 8

## A

abstract spatial data type

- box • 90
- circle • 96
- functions • 87
- hilbert function • 110
- ibox • 98
- icircle • 102
- iline • 100
- ilseg (line segment) • 99
- intersects spatial operator • 108
- ipoint • 97
- ipolygon • 101
- key columns • 119
- line • 92
- long line • 93
- long polygon • 95
- lseg (line segment) • 91
- nbr function • 110
- operators • 87
- overlaps spatial operator • 109
- point • 89
- polygon • 94
- spatial conversion functions • 111
- spatial operators • 104
- storage formats • 103
- support routines • 116

area function • 111

## B

binary spatial operators • 105

bold typeface • 8

box (spatial data type) • 90

## C

circle (spatial data type) • 96

coercion

- defining a new data type • 17
- routines • 17

compare routine • 29

conventions

- syntax • 8

## D

data type coercion • 17

data type definition • 11

- adding abstract data types • 21

IIADD\_DT\_DFN • 22

- required routines • 24

dbtoev routine • 19, 30

demonstration data types • 10

dhmax routine • 32

dhmin routine • 33

dispose mask

- ii\_cb\_trace routine • 78

distance function • 111

## E

err\_num

- ii\_error\_fcn routine • 79

## F

filter functions

- using • 64

function definition • 11

IIADD\_FO\_DFN • 55

- requirements • 55

function instance definition • 11, 57

IIADD\_FI\_DFN • 58

function instances

- defining for large objects • 63

functions

- spatial operators • 110

---

## G

getempty routine • 19, 34

## H

handler routine • 66  
hashprep routine • 35  
header file  
    IIADD.H • 9  
helem routine • 36  
hg\_dtIn routine • 37  
hilbert spatial function • 110  
hmax routine • 39  
hmin routine • 40

## I

ibox (spatial data type) • 98  
icircle (spatial data type) • 102  
identifiers  
    data type • 18  
    function • 18  
    function instance • 18  
ii\_cb\_trace routine • 78  
II\_COPY operation • 73  
II\_COUPON • 22  
II\_DATA\_VALUE • 24  
II\_DT\_PERIPHERAL • 22  
ii\_error\_fcn routine • 79  
ii\_filter\_fcn routine • 80  
II\_GET operation • 69  
II\_INFORMATION operation • 68  
ii\_init\_filter\_fcn routine • 79  
ii\_lo\_handler\_fcn routine • 79  
II\_PERIPHERAL • 22  
    description • 19  
II\_PUT operation • 71  
II\_USERADT  
    shared image • 81, 83  
IIADD.H  
    location • 9  
IIADD\_DEFINITION • 76  
IIADD\_DT\_DFN  
    description of fields • 22  
IIADD\_FI\_DFN • 58  
IIADD\_FO\_DFN • 55  
iilink (command) • 84  
Iiudadt\_register • 75  
    calling syntax • 75

iline (spatial data type) • 100  
ilseg (spatial data type) • 99  
Ingres Cluster Solution • 132, 133  
input coercion routine • 17  
inside operator • 106  
installation  
    II\_USERADT • 81  
    shared image template • 82  
    shared image transfer vector • 82  
    testing • 84  
installing data types and functions  
    unix • 84  
    VMS • 81  
    Windows • 81  
int\_list data type • 10  
intersects operator • 108  
ipoint (spatial data type) • 97  
ipolygon (spatial data type) • 101  
italics • 8

## K

keybuild routine • 41

## L

large objects  
    defining function instances • 63  
    description • 19  
    manipulating directly • 66  
    using filter functions • 64  
length  
    ii\_cb\_trace routine • 78  
length function • 111  
length\_check routine • 19, 45  
line (spatial data type) • 92  
long line (spatial data type) • 93  
long polygon (spatial data type) • 95  
lseg (spatial data type) • 91

## M

minmaxdv routine • 46

## N

nbr spatial function • 110

## O

Object Management Extension

---

- description • 9
- restrictions • 13

operators

- spatial data type • 104

ord\_pair data type • 10

output coercion routine • 17

overlaps operator • 109

## P

perimeter function • 111

Peripheral Object Control Block (POP\_CB)

- structure • 66

peripheral objects

- description • 19

point (spatial data type) • 89

polygon (spatial data type) • 94

- maximum number of points • 120

procedure checklist

- UNIX • 132
- VMS • 133
- Windows • 131

## R

required routines

- calling syntax • 24
- common characteristics • 24
- compare • 29
- dbtoev • 30
- dhmax • 32
- dhmin • 33
- getempty • 34
- hashprep • 35
- helem • 36
- hg\_dtlN • 37
- hmax • 39
- hmin • 40
- IIudadt\_register • 75
- keybuild • 41
- length\_check • 45
- minmaxdv • 46
- tmcvt • 50
- tmlen • 51
- value\_check • 52

restrictions, OME • 13

routines

- list of server • 78

## S

scb

- ii\_error\_fcn routine • 79

scb\_error • 24

seglen routine • 49

server routines • 78

shared image

- disk location • 83
- template • 82
- transfer vector • 82

spatial functions • 111

spatial operators

- functions • 110

Star Server

- installation process • 81

statement syntax • 8

storage formats

- spatial data types • 103

string

- ii\_cb\_trace routine • 78

support routines

- spatial data type • 116

syntax

- conventions • 8

syntax descriptions • 8

## T

testing

- data type code • 85

tmcvt routine • 19, 50

tmlen routine • 19, 51

transfer vector for shared image • 82

## U

underlying data type

- description • 19

UNIX

- installing and testing new code • 13
- installing data types • 84
- procedure checklist • 132

user-defined data types

- copying • 18
- DBMS Server and • 15
- guidelines • 16
- language requirements • 16

---

## V

value\_check routine • 19, 52

### VMS

- installing and testing new code • 13
- installing data types and functions • 81
- procedure checklist • 133

## W

### Windows

- installing and testing new code • 13
- installing data types and functions • 81
- procedure checklist • 131

## X

xform routine • 53