Informix R-Tree Index

User's Guide

Version 9.14
December 1998
Part No. 000-5058

Published by INFORMIX® Press

Informix Corporation
4100 Bohannon Drive
Menlo Park, CA 94025-1032

Documentation Team: Juliet Shackell, Liz Suto, Daniel Howard

Table of Contents

Introduction

In This Introduction

This chapter introduces the *Informix R-Tree Index User's Guide*. Read this chapter for an overview of the information provided in this manual and for an understanding of the conventions used throughout.

About This Manual

This manual describes the Informix R-tree secondary access method and how to access and use its components. It describes how to create an R-tree index on appropriate data types, and how to create a new operator class that uses the R-tree access method to index a user-defined data type.

This section discusses the organization of the manual, the intended audience, and the associated software products you must have to develop and use the access method.

Organization of This Manual

This manual includes the following chapters:

- This introduction provides an overview of the manual, describes the documentation conventions used, and explains the generic style of this documentation.
- Chapter 1, "R-Tree Secondary Access Method Concepts," provides an overview of the Informix R-tree secondary access method and its principal components.
- Chapter 2, "Using the R-Tree Secondary Access Method," describes how to create an R-tree index and the types of queries that use R-tree indexes.

- Chapter 3, "Developing DataBlade Modules That Use the R-Tree Secondary Access Method," describes how to create the required access method objects so that you can create an R-tree index on a column of a user-defined data type. The chapter also discusses user-defined data type design issues, as well as how other DataBlade modules depend on the Informix R-tree secondary access method.
- Chapter 4, "Managing Databases That Use the R-Tree Secondary Access Method," discusses important issues for database administrators that maintain databases that contain R-tree indexes.
- Appendix A, "Sample DataBlade Module," provides a description of the sample DataBlade module used in the guide's examples. The appendix also includes sample C code to create the functions described in Chapter 3, "Developing DataBlade Modules That Use the R-Tree Secondary Access Method."

A glossary of relevant terms follows the chapters and a comprehensive index directs you to areas of particular interest.

Types of Users

This manual is written for three distinct audiences: application developers, DataBlade module developers, and database administrators. This section describes the chapters that are most relevant to each audience type. Although each chapter has a specific audience, all users can benefit from reading the entire guide.

The following table describes the types of users that this manual is written for and the chapters that contain relevant information for each type of user.

Types of Users	Relevant Chapter
All users who want in-depth discussion about how R-tree indexes work	Chapter 1, "R-Tree Secondary Access Method Concepts"
Application developers and schema designers who use R-tree indexes to index existing tables or design schemas that contain tables indexed by R-tree indexes	Chapter 2, "Using the R-Tree Secondary Access Method"
DataBlade module developers who want to use the R-tree access method to index new data types by creating a new operator class	Chapter 3, "Developing DataBlade Modules That Use the R-Tree Secondary Access Method"
Database administrators who manage databases that contain R-tree indexes	Chapter 4, "Managing Databases That Use the R-Tree Secondary Access Method"

Software Dependencies

You must have the following Informix software to use the R-tree secondary access method:

- Informix Dynamic Server with Universal Data Option
- Informix R-Tree Secondary Access Method DataBlade module
- A DataBlade module that uses the Informix R-tree secondary access method, such as the Informix Geodetic DataBlade module

You must use the following sotfware to develop a DataBlade module that uses the R-tree secondary access method:

- DataBlade Developers Kit

You may use the following application development tools with the R-tree secondary access method:

- DB-Access
- INFORMIX-ESQL/C
- DataBlade API

You do not, however, need to install or use these tools to use the R-tree access method.

Documentation Conventions

This section describes the conventions that this manual uses. These conventions make it easier to gather information from this and other volumes in the documentation set:

- Typographical conventions
- Icon conventions

Typographical Conventions

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

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

(1 of 2)

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

(2 of 2)

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

Tip: *The text and many of the examples in this manual show function and data type names in mixed lettercasing (uppercase and lowercase). Because Informix Dynamic Server with Universal Data Option is case insensitive, you do not need to enter function names exactly as shown: you can use uppercase letters, lowercase letters, or any combination of the two.*

Icon Conventions

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

Comment Icons

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

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

Platform Icons

Platform icons identify paragraphs that contain platform-specific information.

Icon	Description
Windows NT	Identifies information that is specific to the Windows environment.
UNIX	Identifies information that is specific to the UNIX environment.

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

Additional Documentation

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

- On-line manuals
- Printed documentation
- Documentation notes and release notes
- Related reading

On-Line Manuals

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

Printed Documentation

The following related Informix documents complement the information in this manual:

- Before you can use the Informix R-tree secondary access method, you must install and configure Informix Dynamic Server with Universal Data Option. The *Administrator's Guide* for your database server provides information about how to configure Universal Data Option and also contains information about how the database server interacts with DataBlade modules.
- Informix Dynamic Server with Universal Data Option includes the Informix R-Tree Secondary Access Method DataBlade module, which contains R-tree error messages. You must use BladeManager to register this DataBlade module into the database in which you create R-tree indexes. See the *DataBlade Module Installation and Registration Guide* for details on registering DataBlade modules.

- If you plan to develop your own DataBlade modules that use the R-tree secondary access method, read the *DataBlade Developers Kit User's Guide*. This manual describes how to develop DataBlade modules using BladeSmith and BladeManager.
- If you have never used Structured Query Language (SQL), read the *Informix Guide to SQL: Tutorial*. It provides a tutorial on SQL as it is implemented by Informix products. It also describes the fundamental ideas and terminology for planning and implementing an object-relational database.
- A companion volume to the *Tutorial*, the *Informix Guide to SQL: Reference*, includes details of the Informix system catalog tables, describes Informix and common environment variables that you should set, and describes the column data types that Informix database servers support.
- The *Guide to GLS Functionality* describes how to use nondefault locales with Informix products. A locale contains language- and culture-specific information that Informix products use when they format and interpret data.
- The *DB-Access User Manual* describes how to access, modify, and retrieve information from Informix database servers with the DB-Access utility.
- *Informix Error Messages* might also be useful if you do not want to look up your error messages on-line.

Documentation Notes and Release Notes

In addition to the set of Informix manuals, the on-line files described in this section supplement the information in this manual.

On-line information about the R-tree secondary access method is found in the on-line files for the following two products:

- Informix Dynamic Server with Universal Data Option

  These on-line files include information about all aspects of the R-tree secondary access method other than the error messages.

- Informix R-Tree Secondary Access Method DataBlade module

  These on-line files include information about the Informix R-Tree Secondary Access Method DataBlade module, which includes the error messages for the access method and the BladeSmith interface object that is installed in dependent DataBlade modules.

Informix Dynamic Server with Universal Data Option On-Line Files

The following table describes the on-line files included with Informix Dynamic Server with Universal Data Option.

On-Line File	Purpose
SERVERS_9.x	Describes feature differences from earlier versions of Informix products, such as the R-tree secondary access method, and how these differences might affect current products. This file also contains information about any known problems and their workarounds.

UNIX

For UNIX installations, on-line files are located in the following directory: **$INFORMIXDIR/release/en_us/0333**. ♦

Windows NT

For Windows NT installations, on-line files are located in the following directory: **%INFORMIXDIR%\release\en_us\04e4**. ♦

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

R-Tree Secondary Access Method DataBlade Module On-Line Files

The following table describes the on-line files included with the Informix R-Tree Secondary Access Method DataBlade module.

On-Line File	Purpose
RLTDOC.TXT	Describes features of the Informix R-Tree Secondary Access Method DataBlade module not covered in the manuals or modified since publication.
RLTREL.TXT	Describes any special actions required to register, configure, and use the Informix R-Tree Secondary Access Method DataBlade module on your computer. This file also describes feature differences from earlier versions of the Informix R-Tree Secondary Access Method DataBlade module and how these differences might affect current products. Additionally, this file contains information about any known problems and their workarounds.

UNIX

For UNIX installations, on-line files are located in the following directory: **$INFORMIXDIR/extend/ifxrltree.*version***, where ***version*** refers to the latest version number of the DataBlade module installed on your computer. ♦

Windows NT

For Windows NT installations, on-line files are located in the following directory: **%INFORMIXDIR%\extend\ifxrltree.*version***, where ***version*** refers to the latest version number of the DataBlade module installed on your computer. ♦

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

Related Reading

The following academic research papers provide additional information about the topics that this manual discusses:

Antonin Guttman. *R-Trees: A Dynamic Index Structure for Spatial Searching*, Proceedings from the ACM SIGMOD Conference on Management of Data, Boston, June 1984.

Marcel Kornacker, Douglas Banks, Micheal Stonebraker. *High-Concurrency Locking in R-trees*, Proceedings from the 21st International Conference on Very Large Databases (VLDB), pages 562-573, September 1995.

Marcel Kornacker, C. Mohan, Joseph Hellerstein. *Concurrency and Recovery in Generalized Search Trees*, Proceedings from the ACM SIGMOD Conference, 1997.

Informix Welcomes Your Comments

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

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

Write to us at the following address:

Informix Software, Inc.
Technical Publications
300 Lakeside Dr., Suite 2700
Oakland, CA 94612

If you prefer to send electronic mail, our address is:

doc@informix.com

We appreciate your feedback.

Chapter

1 R-Tree Secondary Access Method Concepts

In This Chapter

This chapter provides a detailed discussion of the R-tree secondary access method. It includes the following topics:

- "About Access Methods" on page 1-4 describes access methods in general.
- "The R-Tree Secondary Access Method" on page 1-5 describes the structure of an R-tree index and how searches and inserts are performed on the index.
- "About Operator Classes" on page 1-14 introduces the concept of operator classes, how they interact with access methods, and an overview of the operator class requirements of the R-tree access method.
- "R-Tree Functionality Provided by Informix" on page 1-16 discusses the R-tree functionality that is provided by Informix as part of Informix Dynamic Server with Universal Data Option and the Informix R-Tree Secondary Access Method DataBlade module, and the functionality that must be implemented by DataBlade module developers or must be found in other DataBlade modules.
- "Informix Products That Use the R-Tree Access Method" on page 1-18 discusses the Informix products that create user-defined data types that can be indexed by the R-tree secondary access method, and the operator class used to create the R-tree indexes on columns of the data type.

DataBlade module developers must use the DataBlade Developers Kit to develop the objects that form the DataBlade module that uses the R-tree access method. The DataBlade Developers Kit automatically generates most of the SQL commands and C code needed to create the objects. For purposes of clarity, however, this guide gives examples of the SQL commands and C code so that the process of creating the objects is easier to understand.

This guide uses the sample DataBlade module, described in Appendix A, to illustrate how to use the R-tree access method (Chapter 2) and how to create DataBlade modules that implement the R-tree access method (Chapter 3). This sample DataBlade module is also one of the examples provided in the DataBlade Developers Kit.

About Access Methods

An *access method* is a set of server routines that Informix Dynamic Server with Universal Data Option uses to access and manipulate a table or an index. There are two types of access methods: primary and secondary.

Universal Data Option uses a *primary access method* to perform standard table operations, such as inserting, deleting, updating, and retrieving data.

Universal Data Option uses a *secondary access method* to build, use, and manipulate an index structure. Indexes are built on one or more columns of a table to provide a quick way to find rows in a database based on the value in the indexed column or columns.

The routines of a secondary access method encapsulate index operations, such as how to:

- build an index.
- scan the index.
- insert new information into an index as new data is inserted into the indexed table.
- update an index as the indexed table is updated.
- delete data from an index as data is deleted from the indexed table.

These routines are collectively called *purpose functions*.

Secondary access methods are used in combination with *operator classes* that describe when an access method can be used in a query and how to perform the index operations, such as scanning and updating. Operator classes are a way of specifying the routines that play particular roles in access method operations. Operator classes are described in more detail in the section "About Operator Classes" on page 1-14.

Informix Dynamic Server with Universal Data Option provides two secondary access methods:

- B-tree, which stands for *balanced tree*. B-tree is the default access method for ordered data values.
- R-tree, which stands for *range tree*. R-tree is an access method for multidimensional (spatial) and interval data.

The B-tree access method is described in the *Administrator's Guide* for your database server.

Tip: *Indexes that are created and manipulated by a particular secondary access method are referred to by the name of the access method. For example, the R-tree secondary access method is used to create and manipulate R-tree indexes.*

The R-Tree Secondary Access Method

R-tree is a type of secondary access method that is specifically designed to index table columns that contain the following type of data:

- Multidimensional data, such as:
    - spatial data in two or three dimensions. An extra dimenstion representing time could also be included.
    - combinations of numerical values treated as multidimensional values, such as a house configuration that includes number of stories, number of bedrooms, number of baths, age of house, and square feet of floor space.
- Range values, as opposed to single point values, such as the time of a television program (9:00 P.M. to 9:30 P.M.) or the north-south extent of a county on a map

Important: *You can build R-tree indexes only on a single column of a table or on the result of a single function (functional R-tree indexes); you cannot build a single R-tree index on multiple columns.*

To index multiple attributes, incorporate them into a single data type. For more information on creating a new data type, refer to "Designing the User-Defined Data Type" on page 3-6.

The R-tree access method is implemented using the Virtual-Index Interface, a mechanism supported by Informix Dynamic Server with Universal Data Option for creating new secondary access methods.

The purpose of a spatial index, such as R-tree, is to produce during query processing, as fast as possible, a candidate result set that is much smaller than the original set being searched (the table), as opposed to immediately finding the correct result set. The candidate result set that is found by traversing the R-tree index often contains false hits as well as true hits because the index uses enclosing boxes instead of the true shapes of the data objects. The false hits are eliminated by applying a more expensive, exact test to the small candidate set.

Although the index is inexact, it is conservative: it often retrieves too much information, but never too little. The final result of a search that uses the R-tree index is the same as a search that does not use the index or a search that uses an exact test on *every* object in the table.

Another way to look at an R-tree index is that it eliminates large amounts of data that could not possibly qualify in a search, without actually examining the data itself. It does this by eliminating data that falls outside boxes that enclose the area of interest.

R-Tree Index Structure

The hierarchical structure of an R-tree index is similar to that of a B-tree index, although the data stored in the index is quite different.

Bounding Boxes

The R-tree access method organizes data in a tree-shaped structure called an R-tree index. The index uses a *bounding box*, which is a rectangular shape that completely contains the bounded object or objects. Bounding boxes can enclose data objects or other bounding boxes.

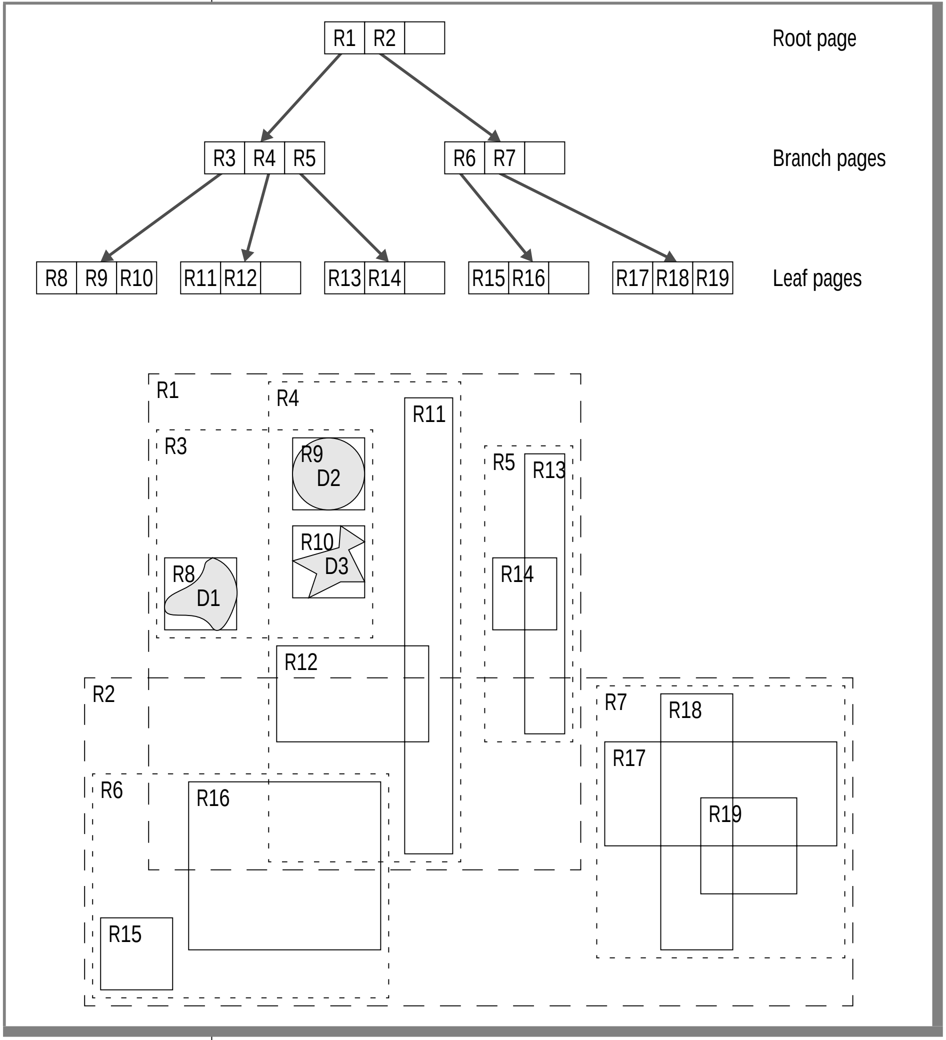
Bounding boxes are usually stored as a set of coordinates of equal dimension as the bounded object. While it is useful for performance reasons to choose the bounding box as small as possible, the R-tree access method does not require it. The minimum bounding box is often, however, the most efficient one. For example, the minimum bounding box for a two-dimensional circle is a square whose side is equal to the diameter of the circle. The minimum bounding box for a three-dimensional sphere is a cube whose edge is equal to the diameter of the sphere.

Tip: *A dimension of a bounding box can be time or some other nonspatial quantity.*

The lower part of Figure 1-1 shows a set of bounding boxes that enclose data objects and other bounding boxes. In the diagram, the data objects are shaded.

Important: *Data objects are only shown for bounding boxes R8, R9, and R10. The other bounding boxes at the leaf level (R11 through R19) also contain data items, but they are omitted from the figure to simplify the graphic.*

Figure 1-1
R-Tree Index Structure



As the figure shows, bounding boxes can enclose a single data object, or one or more bounding boxes. For example, bounding box R8, which is at the leaf level of the tree, contains the data object D1. Bounding box R3, which is at the branch level of the tree, contains the bounding boxes R8, R9, and R10. Bounding box R1, which is at the root level, contains the bounding boxes R3, R4, and R5.

Hierarchical Index Structure

An R-tree index is arranged as a hierarchy of pages. The topmost level of the hierarchy is the *root page*. Intermediate levels, when needed, are *branch pages*. Each branch page contains entries that refer to a subset of pages, or a subtree, in the next level of the index. The bottom level of the index contains a set of *leaf pages*. Each leaf page contains a list of index entries that refer back to rows in the indexed table. Each index entry also includes a copy of the indexed key from the table, or *data object*. The pages of an R-tree index do not usually contain the maximum possible number of index entries.

An R-tree index is height-balanced, which means that all paths down the tree, from the root page to any leaf page, traverse the same number of levels. This also means that all leaf nodes are at the same level.

Each page in an R-tree index structure is a physical disk page. The R-tree index is designed to minimize the number of pages that need to be fetched from disk during the execution of a query, since disk I/O is often the most costly part.

The upper section of Figure 1-1 shows how the data objects and bounding boxes described in “Bounding Boxes” on page 1-6 might be stored in an R-tree index structure. The root page contains entries for bounding boxes R1 and R2. Together, these two bounding boxes enclose all the objects in the index.

The bounding boxes of an index page can overlap. However, a data object appears only once in the index even if it falls inside more than one bounding box at the branch levels. For example, data object D2 only appears once in the index shown in Figure 1-1 even though it falls inside bounding boxes R9, R3, R4, and R1.

The reason data objects appear only once in an R-tree index is to prevent explosions in the size of the index that would occur if many data objects crowd together in the same area. If each object had to be replicated in several index pages, the size of the R-tree index could reach massive and unwieldy proportions.

An index entry in a leaf page consists of:

- a copy of the key, or data object, from the table.
- a pointer back to the row in the indexed table (also known as a *rowid)*.
- a delete flag that marks the key for deletion. This flag indicates whether a row in the indexed table with this particular key has been deleted.

An index entry in a root or branch page consists of:

- a bounding box that contains all the objects in its child pages.
- a page number that points to a lower-level (branch or leaf) page in the index.

The number of levels needed to support an R-tree index depends on the number of index entries each index page can hold. The number of entries per index page depends, in turn, on the size of the key value. R-tree indexes get wider as the maximum number of index entries per page increases. Almost all the disk space needed by an R-tree index is used by leaf pages, so more entries per page mean fewer levels in the index.

The next sections describe a search and an update of an R-tree index that results from a search or update of the indexed table.

Searching with an R-Tree Index

The simplest kind of search that uses an R-tree access method is for objects that overlap a *search object*. For example, you might want to search for all the polygons stored in the column of a table that overlap a specified polygon. To use the R-tree access method to improve the performance of this type of search, you must have previously created an R-tree index on the table column that contains the polygons, and you must specify a function that checks for overlap (listed in the operator class definition as a *strategy* function) in the WHERE clause of the query statement. Operator classes and strategy functions are described in more detail in the section "About Operator Classes" on page 1-14.

First, the R-tree secondary access method calculates the bounding box of the search object. Then the access method begins a search at the root of the R-tree index structure. The access method compares the bounding box of the search object to the bounding boxes stored in the index entries of the root page. All subtrees whose bounding boxes overlap the search bounding box must be searched, as they might contain qualifying data. Any number of subtrees might need to be searched. The access method then recursively applies the same process to each qualifying subtree. Subtrees whose bounding boxes do not overlap can be skipped; this is where the R-tree access method saves search time and work.

When the search encounters a leaf page, it applies the appropriate strategy function to each key on the leaf page. As usual, the strategy function tests for bounding box overlap between the search object's bounding box and the key's bounding box. If this test passes, the strategy function then applies an *exact* overlap test between the actual search object and the actual key. Keys that qualify according to the strategy function satisfy the query restriction being tested because of this final exact test and result in the set of rows that are returned from the original query.

Inserting into an R-Tree Index

When data is inserted into an R-tree indexed table column, the R-tree index must also be updated with the new information. Insertion into an R-tree index is similar to insertion into a B-tree index in that new index records are added to the leaves, nodes that overflow are split, and splits propagate up the tree.

First, the R-tree secondary access method calculates a bounding box for the new data object. The access method then searches for a leaf page whose existing entries form the tightest group with the new data object. To do this, the R-tree access method searches down the tree, looking for bounding boxes that will be enlarged the least to accomodate the new data object. The access method may also internally use criteria other than the bounding box being enlarged by the smallest amount when the it chooses the best leaf page.

Once the access method finds the best leaf page, and there is space on the corresponding disk page, the access method adds a new index entry that consists of a copy of the new data object. The bounding boxes of the parent index pages all the way up to the root page might also need to be enlarged.

If there is no space left on the leaf page's disk page, the leaf page is split into two pages. Each new leaf page has its own disk page. The access method incorporates the copy of the new data object into one of the new leaf pages and adds an additional bounding box to the parent page. If the parent page is full, it may also need to split, and so on up to the root page. If the root page splits, the tree becomes one level deeper.

When an index page splits, the index entries in the original page must be divided between the two new pages. The division is done in a way that makes it as unlikely as possible that both new pages will need to be examined on subsequent searches. Since the decision to visit a page is based on whether the bounding box of the search object overlaps the bounding boxes of the index entries, the total area of the two new bounding boxes should be as small as possible. Figure 1-2 illustrates this point by comparing efficient and inefficient ways to divide five items into two groups.

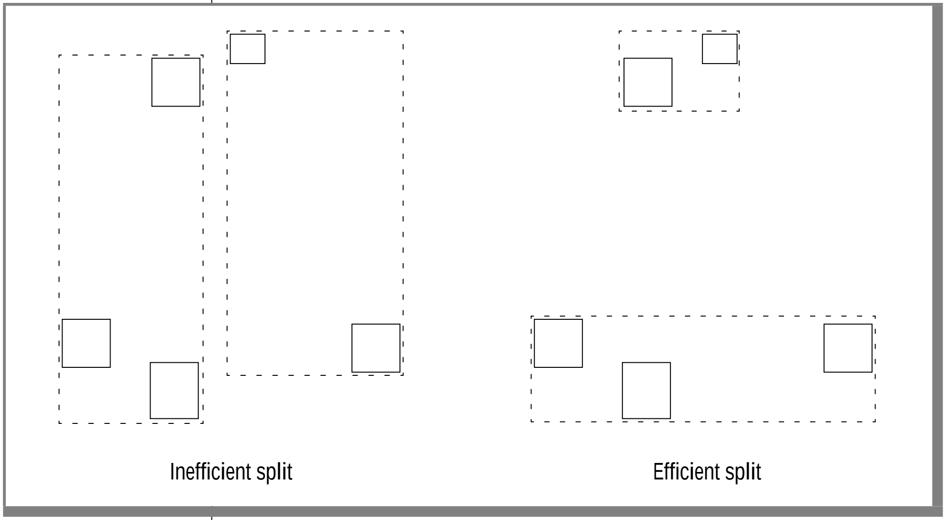


Figure 1-2
Simple Page Splitting Example

Figure 1-3 compares a page split in which the resulting pages overlap each other with a split where the resulting pages do not overlap each other. The split with with overlapping pages is more efficient because the total area of the bounding boxes of the two overlapping pages is smaller than that of the nonoverlapping pages.

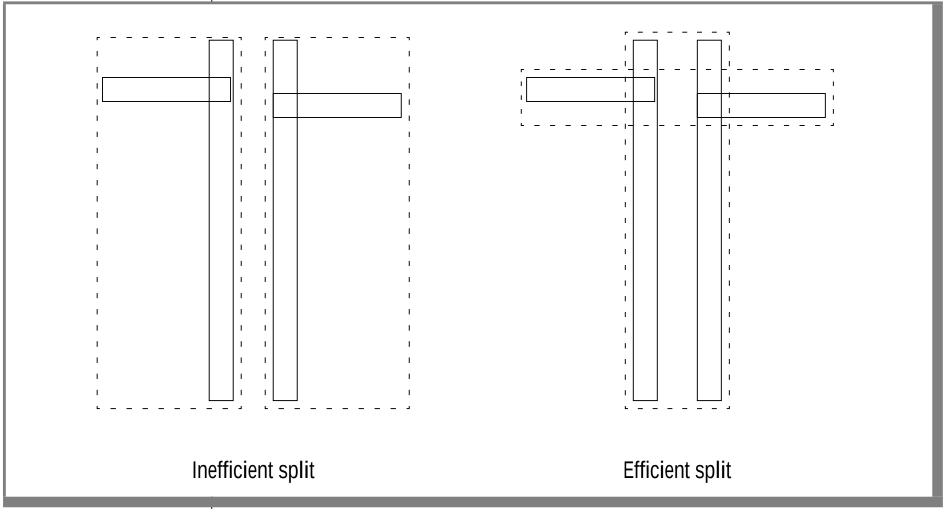


Figure 1-3
Complex Page Splitting Example

The preceding example shows that avoiding overlap is not necessarily the best, and definitely not the only, criterion for dividing index entries between the two resulting pages of a page split.

The R-tree index is initially created by starting with an empty root page and inserting index entries one-by-one.

R-Link Trees and Concurrency

The basic R-tree index structure described in the previous sections works well in a single-user environment but might run into problems if multiple users search and update the index concurrently. R-tree indexes require a particular type of locking during page splits to preserve the integrity of the index structure and ensure correct results from queries. For example, while a page is being split, it is necessary to hold locks on all pages up to and including the root page. This locking behavior is problematic in a concurrent environment. To solve this problem, Informix uses a modified structure called an R-link tree instead of the basic R-tree.

R-link trees are similar to the R-tree structure described in the preceding sections, with the following two key differences:

- All the pages at the same level in the index structure contain a pointer to their right sibling (except for the right-most page, which has a null pointer.) This creates a single list of right-pointing links that includes every page in a particular level.

  When a page splits and a new page is created, the new page is inserted into the list of right-pointing links directly to the right of the old page.

  This sibling relationship between pages has no semantic or spatial meaning and is not used in a search of the index. It is only used to control concurrency.

- Each page in the index is assigned a sequence number that is unique within the tree. Each index entry in a root or branch page includes the *expected* sequence number of its child page, in addition to the information listed in "Hierarchical Index Structure" on page 1-8.

  When a page splits, the new right sibling page is assigned the old page's sequence number, and the old page receives a new sequence number.

The R-link structure allows the R-tree access method to perform index operations without holding locks on pages that might be needed again later. The combination of right-pointing links and sequence numbers lets the R-tree access method detect page splits made by other users and correctly find all the needed pages. Because locks are never held on the index pages while a session is waiting on I/O, multiuser performance is improved.

About Operator Classes

Although an R-tree index may exist on a table column, it may not always be possible for the query optimizer to use it when executing a query, even if the WHERE clause of the query specifies the indexed column.

For example, a query might search for polygons whose area is greater than a specified number. An R-tree index will not be of much use in this type of query because the access method uses the bounding box of the polygons, and not the area, to create the index, as explained in "R-Tree Index Structure" on page 1-6. However, a query that searches for polygons that overlap a specified polygon is a good candidate to use the R-tree index.

An *operator class* helps the query optimizer determine whether a secondary access method can be used in a query. It also defines how to access and modify the index if it *is* used in a query. An operator class specifies a group of functions that work with a new data type and an access method. It links each function to the role it will play in the access method operations.

An operator class consists of a collection of functions that fall into the following two categories:

- Strategy functions

  Strategy functions include all the functions whose evaluation can be assisted by an R-tree index. If a strategy function is specified in the WHERE clause of a query, the R-tree index can be used to evaluate the query. Strategy functions are used both directly by end users in the WHERE clause of SQL queries and internally by the R-tree access method to search the index.

  An example of a strategy function is the **Overlap** function, which determines whether two bounding boxes have any points in common.

- Support functions

  The access method uses the support functions of a secondary access method to build, update, and maintain the index. These functions are not called directly by end users.

  An example of a strategy function is the **Size** function, which calculates the size of a bounding box.

The R-tree access method, similar to all secondary access methods, has specific operator class requirements for the type and number of strategy and support functions that must be defined. By creating a new operator class, DataBlade developers attach names of actual functions to the placeholders for required functions in the operator class structure, which completes the information the database server needs.

A secondary access method usually has a default operator class associated with it. The default operator class for the R-tree access method is called **rtree_ops**.

The **rtree_ops** operator class is generally only used for generic R-tree access method testing and as an example of how to create a new operator class for use with the R-tree access method. It is almost never used directly to create an R-tree index. DataBlade developers should always create a new operator class to use the R-tree access method to index the new data types or to extend the types of queries that use the access method.

Chapter 3, "Developing DataBlade Modules That Use the R-Tree Secondary Access Method," describes in detail how to create an operator class and how to set up the necessary strategy and support functions.

R-Tree Functionality Provided by Informix

R-tree access method functionality is provided in the following two products:

- Informix Dynamic Server with Universal Data Option
- R-Tree Secondary Access Method DataBlade Module

The following sections describe the parts of the R-tree functionality that are provided by each product.

Informix Dynamic Server with Universal Data Option

Informix Dynamic Server with Universal Data Option includes the definition of the R-tree access method and the definition of its default operator class, **rtree_ops**. However, the support and strategy functions that perform the indexing work are *not* included. The **rtree_ops** operator class is intended to be used for generic R-tree testing. While you may reuse it, Informix recommends that you create a new operator class for each new data type that is to be indexed with an R-tree index.

Newly created Informix databases include only standard data types, such as INTEGER, DATETIME, and VARCHAR. Columns of these data types can not be indexed with R-tree indexes. Therefore, to create and use an R-tree index, you must add the following objects to your database:

- One or more user-defined data types that can be indexed with R-tree index.
- A new operator class for the R-tree access method so that you can create R-tree indexes on the user-defined data type.
- The strategy and support functions required by the operator class. You must supply the function code in the form of a shared-object library.

You add new data types to an Informix database by registering a DataBlade module that includes the definition of the data types. The DataBlade module might also include a new operator class so you can index the user-defined data type with an R-tree index. For a list of Informix products that include new data types, support and strategy functions, and operator classes, refer to "Informix Products That Use the R-Tree Access Method" on page 1-18

If you are developing a new DataBlade module, Chapter 3, "Developing DataBlade Modules That Use the R-Tree Secondary Access Method," describes in detail how to create the required strategy and support functions in order to to create a new operator class. The chapter also describes the issues you should be aware of when you design the user-defined data type that will be indexed with the R-tree index.

R-Tree Secondary Access Method DataBlade Module

The Informix R-Tree Secondary Access Method DataBlade module is automatically installed at the time you install Informix Dynamic Server with Universal Data Option.

UNIX

On UNIX, the Informix R-Tree Secondary Access Method DataBlade module is installed in the directory **$INFORMIXDIR/extend/ifxrltree.*version***, where *version* refers to the latest version number of the DataBlade module installed on your computer. ♦

Windows NT

On Windows NT, the Informix R-Tree Secondary Access Method DataBlade module is installed in the directory **%INFORMIXDIR%\extend\ifxrltree.*version***, where *version* refers to the latest version number of the DataBlade module installed on your computer. ♦

Contents of the DataBlade Module

The Informix R-Tree Secondary Access Method DataBlade module consists of:

- a list of error messages used by the R-tree access method.
- a BladeSmith interface object used by DataBlade modules that depend on the Informix R-Tree Secondary Access Method DataBlade module. Refer to Chapter 3, "Developing DataBlade Modules That Use the R-Tree Secondary Access Method," for more information on how to use this interface object.

As described in the preceding section, the R-tree access method itself is built into Informix Dynamic Server with Universal Data Option. The error messages used by the access method, however, are only available if the Informix R-Tree Secondary Access Method DataBlade module is registered in a database.

The R-tree error messages contained in this DataBlade module have error codes of the form `RLTnn`, where:

- `RLT` (R-link tree) is the three-character prefix for all Informix R-Tree Secondary Access Method DataBlade module error codes.
- `nn` is a two-digit number that uniquely identifies each error code.

DataBlade Module Registration

You must register the Informix R-Tree Secondary Access Method DataBlade module in each database in which you plan to use it. You register DataBlade modules using BladeManager.

This registration normally occurs when you register a dependent DataBlade module, that is, one that can only be registered if the Informix R-Tree Secondary Access Method DataBlade module has been previously registered. The dependent DataBlade module first signals to BladeManager that it depends on the Informix R-Tree Secondary Access Method DataBlade module. BladeManager then registers the Informix R-Tree Secondary Access Method DataBlade module before it registers the dependent DataBlade module.

The dependent DataBlade module usually contains the definition of the user-defined data type that can be indexed by the R-tree access method.

For more information about BladeManager, refer to the *DataBlade Module Installation and Registration Guide*.

Informix Products That Use the R-Tree Access Method

Currently, the following two Informix products use the R-tree access method:

- Informix Geodetic DataBlade module

  This DataBlade module is designed to manage spatio-temporal data with global content, such as metadata associated with satellite images.

  The DataBlade module creates a variety of data types, such as **GeoPoint** and **GeoObject**, as well as a variety of functions that operate on the data types, such as **Intersects** and **Outside**. It also provides an operator class, called **GeoObject_ops**, so you can create R-tree indexes on columns of data type **GeoObject**.

- Informix Video Foundation DataBlade module

  This DataBlade module is designed to store, manage, and manipulate video data and its metadata.

  The DataBlade module creates a variety of data types, such as **MedChunk**, as well as a variety of functions that operator on the data types, such as **Within** and **Overlap**. It also provides an operator class, called **MedChunk_ops**, so you can create R-tree indexes on columns of type **MedChunk**.

Appendix A describes a sample DataBlade module that defines four spatial data types and an operator class to allow you to create an R-tree index on columns of these data types. The sample module is not an Informix product but is provided as an example of creating an operator class. The same sample DataBlade module is also provided as an example in the DataBlade Developers Kit.

Chapter

Using the R-Tree Secondary Access Method

2

In This Chapter

This chapter describes how to use the R-tree access method. It includes the following topics:

- "Before You Begin" on page 2-4 describes the steps you must perform before you can create an R-tree index.
- "Creating R-Tree Indexes" on page 2-5 describes the syntax of how to create R-tree indexes on appropriate table columns.
- "When Does the Query Optimizer Use an R-Tree Index?" on page 2-10 describes how to write queries that may use existing R-tree indexes.
- "Functional R-Tree Indexes" on page 2-12 describes the issues of creating functional R-tree indexes.

Before You Begin

Before you can create and use an R-tree index, you must perform the following tasks:

1. Install a DataBlade module on your database server that includes the following objects:
   - A user-defined data type that can be indexed with an R-tree index
   - An operator class that specifies the functions to be used with the R-tree index
   - The support and strategy functions required by the operator class

   Examples of DataBlade modules created by Informix that use the R-tree access method are the Informix Geodetic DataBlade module and the Informix Video Foundation DataBlade module.

   For more information on these modules, refer to "Informix Products That Use the R-Tree Access Method" on page 1-20.
2. Create a database.
3. Register the Informix R-Tree Secondary Access Method DataBlade module into your database using BladeManager.

   If the DataBlade module described in Step 1 defines a dependency on the Informix R-Tree Secondary Access Method DataBlade module, you may skip this step, since BladeManager will automatically prompt you to register the Informix R-Tree Secondary Access Method DataBlade module when you register the DataBlade module described in Step 1.

   For more information on the Informix R-Tree Secondary Access Method DataBlade module, refer to "R-Tree Secondary Access Method DataBlade Module" on page 1-18.
4. Register the DataBlade module described in Step 1 into your database using BladeManager.
5. Create a table that contains a column of the user-defined data type that can be indexed with the R-tree access method.

For information on how to install and register DataBlade modules, refer to the *DataBlade Module Installation and Registration Guide* and to the release notes of your DataBlade module.

Important: *The examples of this chapter use objects defined in the sample DataBlade module described in Appendix A. These objects include the data type MyShape and the operator class* **MyShape_ops**. *Columns of data type MyShape can store points, boxes, and circles.*

This sample DataBlade module is also provided as an example, called Rtree, in the DataBlade Developers Kit.

Creating R-Tree Indexes

To use the R-tree secondary access method, you must first create an R-tree index on a column whose data type can be indexed by the R-tree access method.

Important: *You can create an R-tree index either before or after inserting data into the table; the quality of the index is uneffected.*

However, if you are loading large amounts of data into the table, you should create the R-tree index after you load the data. When you create an R-tree index on a loaded table, the generation of log records is suppressed, so you will not run out of log space. If, however, you create the index first and then load large amounts of data in a single transaction, you might run out of log space which will cause the transaction to abort.

Syntax

The basic syntax for creating an R-tree index is:

```
CREATE INDEX index_name
ON table_name (column_name op_class)
USING RTREE
index_options;
```

The arguments are described in the following table.

Arguments	Purpose	Restrictions
index_name	The name you want to give your index.	The name must be unique in the database.
table_name	The name of the table that contains the column you want to index.	The table must already exist.
column_name	The name of the column you want to index. For example, you can create an R-tree index on columns of data type MyShape, defined in the sample DataBlade module.	You can create an R-tree index on a single column only; you cannot create a single R-tree index on multiple columns. The data type of this column must support R-tree indexes. Check the DataBlade module user's guide for more information on the data types that support R-tree indexes.

(1 of 2)

Arguments	Purpose	Restrictions
op_class	The name of the operator class. For example, to index columns of data type MyShape, defined in the sample DataBlade module, you must specify the **MyShape_ops** operator class.	If you have registered a DataBlade module in your database that supplies its own operator class, you must specify it when you create an R-tree index. If you do not specify an operator class, or you specify the default **rtree_ops** operator class without knowingly setting up your data type and functions to use it, the R-tree index might appear to work correctly but will function unpredictably. Check the DataBlade module user's guide for more information on which operator class you must specify when you create an R-tree index. NOTE: You must run the UPDATE STATISTICS statement after you create the index or the query optimizer might not choose to use the index at appropriate times.
index_options	The fragmentation and storage options of the index, described in detail in the section "R-Tree Index Options" on page 2-8.	The options available for R-tree indexes are FRAGMENT BY and IN. The options CLUSTER, UNIQUE, DISTINCT, ASC, DESC, and FILLFACTOR are not supported.

(2 of 2)

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

R-Tree Index Options

This section discusses the options to the CREATE INDEX command that are supported by R-tree indexes. Refer to "Example" on page 2-9 for examples of using these options.

FRAGMENT BY

R-tree indexes can be fragmented by expression. You cannot, however, fragment R-tree indexes on the multidimensional column they index.

For example, if you create an R-tree index on a column of type MyShape, you cannot specify this column in the FRAGMENT clause. You must fragment the R-tree index on another column of a standard data type, such as INTEGER or VARCHAR.

If you create an R-tree index on a fragmented table, the R-tree index is also fragmented by default. The index fragments are automatically stored in the same dbspace as the table fragments.

The next section describes where you can store R-tree indexes or fragments of R-tree indexes.

IN

R-tree indexes are stored in dbspaces. If you do not specify an IN clause when you create an R-tree index, the index is stored in the root dbspace.

You cannot store R-tree indexes in sbspaces. If you specify an sbspace in the IN clause of the CREATE INDEX statement, the index is actually stored in the root dbspace.

Example

The following example first creates a table called **circle_tab** that contains a column of data type MyCircle. It then creates an R-tree index called **circle_tab_index** on the **circles** column.

```
CREATE TABLE circle_tab
(
    id        INTEGER,
    circles   MyCircle
);

CREATE INDEX circle_tab_index
ON circle_tab (circles MyShape_ops)
USING RTREE;
```

The following example creates a similar R-tree index, but it will be stored in the **dbsp1** dbspace instead of the root dbspace:

```
CREATE INDEX circle_tab_index2
ON circle_tab ( circles MyShape_ops )
USING RTREE
IN dbsp1;
```

The following example creates a fragmented R-tree index on the **circle_tab** table. All shapes with **id** less than 100 are stored in the **dbsp1** dbspace, and the remainder are stored in the **dbsp2** dbspace.

```
CREATE INDEX circle_tab_index3
ON circle_tab ( circles MyShape_ops )
USING RTREE
FRAGMENT BY EXPRESSION
id < 100 IN dbsp1,
id >= 100 IN dbsp2;
```

The following example first creates a fragmented table called **circle_tab_frag** and then creates an R-tree index on the table called **circle_tab_index4**. Although the R-tree index is not explicitly created with fragmentation, it is fragmented by default because the table it is indexing, **circle_tab_frag**, is fragmented.

```
CREATE TABLE circle_tab_frag
(
    id        INTEGER,
    circles   MyCircle
)
FRAGMENT BY EXPRESSION
id < 100 IN dbsp1,
id >= 100 IN dbsp2;

CREATE INDEX circle_tab_index4
ON circle_tab_frag ( circles MyShape_ops )
USING RTREE;
```

To drop an R-tree index, use the DROP INDEX statement, as shown in the following example:

```
DROP INDEX circle_tab_index;
```

When Does the Query Optimizer Use an R-Tree Index?

The query optimizer can use an R-tree index when evaluating a query if the following statements are true:

- A strategy function of an operator class is used in the WHERE clause of the query.
- One or more arguments of the strategy function are table columns with R-tree indexes associated with the operator class.
- The data type of the arguments of the strategy function specified in the WHERE clause of the query are compatible with the signature of the strategy function. The query optimizer may cast one or both arguments to other data types in an effort to make the arguments match the signature of the strategy function.

For example, the following query can use an R-tree index:

```
SELECT * FROM circle_tab
WHERE Contains (circles, 'circle(-5,-10, 20)'::MyCircle );
```

The query optimizer can use the R-tree index in the preceding example for the following reasons:

- The **Contains** function, specified in the WHERE clause of the query, is a strategy function of the **MyShape_ops** operator class.
- The **circles** column, specified in the **Contains** function in the WHERE clause of the query, is of data type MyCircle and has an R-tree index built on it.
- When the cast from a string data type to the MyCircle data type is applied to the second argument, the cast from MyCircle to MyShape can be internally applied to both arguments. The result of these casts matches the signature of the **Contains** strategy function.

The query optimizer might sometimes decide *not* to use an R-tree index, even when it could be used. Consider the following query:

```
SELECT * FROM circle_tab
WHERE Contains (circles, 'circle(-5,-10, 20)'::MyCircle ) AND
      id = 99;
```

If there is a B-tree index on the **id** column, the query optimizer compares the cost of using the two indexes and might use the B-tree index instead of the R-tree index. It might even decide to perform a sequential scan for a small table to avoid the overhead of using either index.

The query optimizer executes only one R-tree index search per query for a specified column. If you specify a complex qualification (two or more strategy functions joined with the AND keyword) in your query, only one of the strategy functions is used to perform an R-tree index search. The other strategy functions are executed directly on the values returned from the index search to find the values that satisfy *all* the predicates in the query qualification. This is a restriction for R-tree indexes only; other types of secondary indexes that use the Virtual-Index Interface may handle complex qualifications.

***Important:** The query optimizer also uses statistical data on the indexed column to decide whether to use an R-tree index. This statistical data must be kept up-to-date and correct for the query optimizer to make a good decision. Use the UPDATE STATISTICS command to update the statistics for the indexed column.*

For more information on statistics, see Chapter 4, "Managing Databases That Use the R-Tree Secondary Access Method."

Functional R-Tree Indexes

You can also use the R-tree access method to create a functional R-tree index. A *functional index* computes the results of executing a specified function on a table column and stores the results in an index.

To create a functional R-tree index, the return type of the function must be a data type that is compatible with an R-tree index.

You cannot build a functional R-tree index with a function that specifies a smart large object either as a parameter or a return type. This is true for all functional indexes, not just R-tree functional indexes.

For more information on creating functional indexes, refer to the *Informix Guide to SQL: Syntax*.

Chapter

3

Developing DataBlade Modules That Use the R-Tree Secondary Access Method

In This Chapter

This chapter provides information for DataBlade developers who might want to use the R-tree secondary access method to index a new data type. It discusses the following topics:

- "Overview of DataBlade Module Development" on page 3-3 provides a brief overview of developing DataBlade modules and how the R-tree secondary access method fits into the big picture.
- "Deciding Whether to Use the R-Tree Access Method" on page 3-4 discusses when it is appropriate to use the R-tree access method, and when it is best for developers to use some other access method, such as B-tree.
- "Designing the User-Defined Data Type" on page 3-5 discusses the issues that developers should consider when they design the user-defined data type that will be indexed with an R-tree index. For information on creating user-defined data types, see *Extending INFORMIX-Universal Server: Data Types*.
- "Creating a New Operator Class" on page 3-12 discusses how to create a new operator class so that columns of the user-defined data type can be indexed with R-tree indexes.
- "Creating Registration Scripts for Dependent DataBlade Modules" on page 3-33 describes how to update the registration scripts of dependent DataBlade modules to specify that they depend on the Informix R-Tree Secondary Access Method DataBlade module.

Overview of DataBlade Module Development

A DataBlade module is a software package that extends the functionality of Informix Dynamic Server with Universal Data Option. It adds new database objects, such as data types and routines, that extend the SQL syntax and commands you can use with Informix Dynamic Server.

You use the DataBlade Developers Kit (DBDK) to create and package DataBlade modules. With the DBDK, you define the new database objects that will be included in your DataBlade module, import objects from other modules, and generate the source code, SQL scripts, and installation scripts that make up your DataBlade module.

For example, you can use the DBDK to create a DataBlade module that contains spatial data types, such as polygons and circles. The module will probably also include a set of routines that operate on the data types, such as **Area** and **Circumference**.

Your DataBlade module might also include the required routines and operator class to enable users to create R-tree indexes on columns of the user-defined data type. This chapter describes how to add this functionality to your DataBlade module.

The DBDK automatically generates much of the C code and SQL scripts that make up a DataBlade module. This means that most DataBlade module developers do not need to write most of the SQL commands described in this chapter. The commands are provided, however, to better explain the concepts.

For more information on how to design and create DataBlade modules with the DBDK, refer to the *DataBlade Developers Kit User's Guide*.

Important: *The examples in this chapter are taken from the definition of the objects of the sample DataBlade module, described in Appendix A. The appendix provides both a description of the DataBlade module and the C code used to create the functions of the operator class.*

The sample DataBlade module is included as an example in the DataBlade Developers Kit. The example in the DBDK is called Rtree.

Deciding Whether to Use the R-Tree Access Method

The R-tree secondary access method is specifically designed to index multidimensional data and data that represents an interval or range. Examples of these types of data include:

- two-dimensional spatial objects, such as points, lines, and polygons.
- geographic mapping information, such as latitude/longitude points that lie in bounded regions that represent countries, states, or counties.
- grid information, such as building locations within a bounded area.
- video or audio clips, each with a start and stop time. If you create a time range user-defined data type, you could search for overlapping clips more efficiently with an R-tree index than with a B-tree index.
- color information that includes hue, brightness, and saturation.
- multidimensional views of standard relational quantitative data, such as age, salary, sales commission, hire date, and so on.

Unlike other data structures, such as grid-file and quad-tree, the R-tree access method does not require that data values be in a known bounded area.

If you are developing a DataBlade module that includes a user-defined data type of a multidimensional or interval nature, you might want to use the R-tree access method to index columns of this data type.

The type of data most suited to B-tree indexes (the other indexing method included in Informix Dynamic Server with Universal Data Option) is ordered numeric values in one dimension. B-trees indexes should never be used to index range or interval data. The types of data are suited to being indexed with the B-tree access method and not the R-tree access method:

- Numerical data, such as Social Security numbers or employee IDs
- Character data, such as last names and product names

Once you have decided to use the R-tree access method to index a user-defined data type, you must create a new operator class. "Creating a New Operator Class" on page 3-12 describes this process. The next section describes issues you should be aware of when you design the user-defined data type.

Designing the User-Defined Data Type

This section contains the following topics that discuss the issues you should consider when you design a user-defined data type that will be indexed with an R-tree index:

- "Data Objects and Bounding Boxes" on page 3-6
- "Data Type Hierarchies" on page 3-8
- "Maximum Size of the User-Defined Data Type" on page 3-10
- "Loose Bounding Box Calculations" on page 3-11
- "Other User-Defined Data Type Design Considerations" on page 3-12

Important: *This section does not discuss how to create a user-defined data type. For detailed instructions on how to create a new data type, refer to "Extending INFORMIX-Universal Server: Data Types."*

Data Objects and Bounding Boxes

As discussed in Chapter 1, "R-Tree Secondary Access Method Concepts," R-tree indexes store both the bounding boxes of data objects in the indexed table and copies of the data objects in the table. This means that the support and strategy functions that maintain the R-tree index must also operate on both bounding boxes and data objects.

The data type of the parameters to the support and strategy functions is the user-defined data type of the indexed column. Therefore, the user-defined data type of the indexed column must be able to be referred to as both a bounding box and the data object itself. For example, the bounding box information can be hidden inside the object, such as in a header, along with the actual object data.

The R-tree access method code never operates directly on the data inside the objects in the indexed column. Instead, it passes the complete objects to the user-defined support and strategy functions, which may use the bounding box information or the full data object description, as appropriate. It is therefore up to the designer of user-defined support and strategy functions to decide when the bounding box, or data object, should be used in a calculation.

The next two sections describe when the support and strategy functions operate on data objects and when they operate on the bounding boxes of the data objects. Use these descriptions to correctly design your own support and strategy functions.

Operations on Data Objects

When a user creates a table with a user-defined data type column and inserts a new row, the user-defined data type's input functions operate on the actual data object to physically create the new object and insert the row into the table.

If an R-tree index exists on the column, the R-tree access method calls the appropriate support and strategy functions to expand the R-tree index. The functions use the bounding box of the new data object to decide where the copy of the data object, with its bounding box, should be placed in the R-tree index.

Searches can also operate on the actual data object. The search function used in the WHERE clause of a query, such as **Contains**, must be evaluated on the actual data object when a qualifying leaf entry in the R-tree index is found. In other words, true geometry on the actual data object must be used to find a real match. If a user does not create an R-tree index on the column, then the search function is evaluated for *every* data object according to its true geometry. If an R-tree index exists on a column, but the query optimizer decides not to use it, then the search function again operates on all data objects and not on the keys stored in the R-tree index.

Operations on Bounding Boxes

Once a table contains enough rows so that the R-tree index has split into more than one level, the support and strategy functions use a combination of bounding boxes and data objects in their internal calculations when a new row is inserted in the table. The functions generate a new bounding box for the affected pages based on existing key information already stored in the R-tree index and the data object itself, and they calculate where the new key should be placed in the R-tree index. The affected pages are the leaf page on which the new key is stored and the parent pages whose bounding boxes need to be enlarged.

If the query optimizer decides to use an R-tree index in a search, the R-tree index begins its search at the root and branch pages of the index. The R-tree index uses the search function to methodically search the R-tree structure. Since searches of R-tree indexes involve both the bounding boxes and data objects, the support and strategy functions in this case also use both the bounding boxes and data objects in their internal calculations.

Internal C Structure for the User-Defined Data Type

In summary, although the internal C structure for the user-defined data type can be anything the developer wants it to be, the following two rules must be true if columns of this data type are to be indexed with the R-tree access method:

- The data structure must support both the actual data object *and* its bounding box.
- Only *one* C data structure may be defined for the user-defined data type's internal representation. The same data structure must be passed to all functions that accept the user-defined data type as an argument. Examples of such function are the support and strategy functions that maintain the R-tree index.

Data Type Hierarchies

If you are designing two or more similar data types, you should consider implementing your own data type hierarchy to avoid writing strategy and support functions for every possible combination of data type signatures.

To implement your own data type hierarchy

1. Design a single supertype to which the strategy functions will apply.
2. Create implicit casts in SQL from all the subtypes to the supertype.
3. Create implicit casts in SQL from the built-in data types LVARCHAR, SENDRECV, IMPEXP, and IMPEXPBIN to the supertype and all subtypes. This is part of the normal opaque user-defined data type creation. For more information about creating these implicit casts, refer to *Extending INFORMIX-Universal Server: Data Types*.

4. Create the required strategy functions in SQL for *just* the supertype. You do not need to create strategy functions for the subtypes since casts from the subtype to the supertype exist.
5. In SQL, create support functions for the supertype and *all* the subtypes. All of these SQL functions, however, can usually be mapped to the *same* C code; thus only one C function needs to be written.

When the query optimizer is executing a query, if it is unable to find a function for a particular subtype, it implicitly casts the subtype to the supertype and uses the function defined for the supertype.

The support or strategy function that is defined for the supertype must internally determine what actual data type it is operating on, and then it must execute the code that applies for that particular data type. This means that the internal C code for a function defined for the supertype also contains the C code that applies to all subtypes.

Example Data Type Hierarchy

Assume you are designing three data types: MyPoint, MyBox, and MyCircle. Since they are all two-dimensional spatial data types, a supertype called MyShape could also be defined. This type hierarchy is described in Figure 3-1.

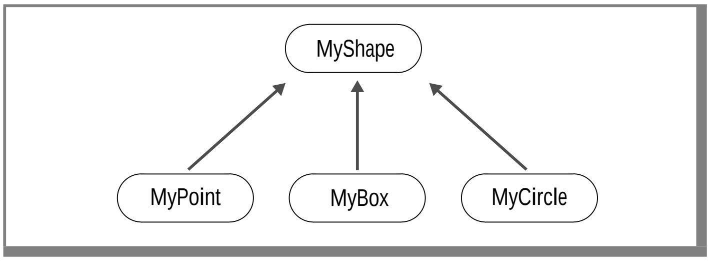


Figure 3-1
Data Type Hierarchy

Using SQL, create casts between the three subtypes (MyPoint, MyBox, and MyCircle) and the supertype, MyShape.

The following two sections describe how to create the strategy and support functions.

Strategy Functions in a Data Type Hierarchy

When you create the strategy functions, such as **Overlaps**, only one function needs to be created in SQL: `Overlaps (MyShape, MyShape)`. The internal C code for this **Overlaps** function first checks to see what actual data type it is operating on (either MyPoint, MyBox, or MyCircle), and then calls the appropriate code for that data type. For example, if the function call in the query was actually `Overlaps (MyCircle, MyCircle)`, the appropriate code for the overlap between two MyCircle data types is executed.

If a query contains the expression `Overlaps (MyCircle, MyCircle)`, the query optimizer first looks for a function with the same signature. It will not find one, since none has been defined. It does, however, find a cast from MyCircle to MyShape, so it searches for an **Overlaps** function that applies to the MyShape data type. Since this function does exist, the query optimizer executes it after implicitly casting MyCircle to MyShape.

By taking advantage of type hierarchies and casting, you avoid having to explicitly create the various combinations of **Overlaps** functions within SQL, such as `Overlaps(MyPoint, MyPoint)`, `Overlaps(MyBox, MyCircle)`, and so on.

The preceding discussion about type hierarchies and strategy functions is true for all strategy functions, not just for the **Overlaps** function.

Support Functions in a Data Type Hierarchy

When you create the support functions such as **Union**, you must create separate SQL functions for each indexable column type. For example, you must create the following SQL **Union** functions:

```
Union (MyPoint, MyPoint, MyPoint)
Union (MyBox, MyBox, MyBox)
Union (MyCircle, MyCircle, MyCircle)
Union (MyShape, MyShape, MyShape)
```

All these **Union** support functions, however, can be mapped to the *same* C code. Similar to strategy functions, the internal C code that the **Union** functions map to first checks to see what actual data type it is operating on (either MyPoint, MyBox, or MyCircle), and then calls the appropriate code for that data type. For example, if the function call is `Union (MyCircle, MyCircle, MyCircle)`, it executes the appropriate code for the union of two MyCircle data types.

The preceding discussion about type hierarchies and support functions is true for all support functions, not just for the **Union** function.

Maximum Size of the User-Defined Data Type

A copy of the data object is stored as part of the key in the leaf pages of an R-tree index. Each index page is a database disk page. R-tree index entries, however, cannot span disk pages as table rows can.

Therefore, the maximum size of a data object that is stored in a table, and thus the maximum size of its user-defined data type, is governed by the R-tree disk page size of 2 KB. After allowing for R-tree index overhead, about 1960 bytes are available.

Furthermore, R-tree indexes should always fit at least two keys on a single leaf page. This means that the maximum size of a user-defined data type is 980 bytes, or half of 1960.

Although 960 bytes might be sufficient to store simple boxes and circles, it is probably not sufficient to store very large polygons. DataBlade modules that create user-defined data types that store very large values must implement them as either smart large objects or multirepresentational data types. Multi-representational user-defined data types store a value in the table if it is smaller than 960 bytes, or in a smart large object otherwise. There is no size limitation on smart large objects or multirepresentational data types.

Loose Bounding Box Calculations

In an R-tree index, bounding boxes are used to conservatively identify data that might qualify during a search. A more accurate check is always applied as a second step. For this reason, one might think that the bounding box of an object could be *loose*, or not an exact fit, without causing anything worse than a few initial false hits. It is often difficult to calculate an exact bounding box for some objects, such as great circle arcs on the surface of the earth, so there is a compelling reason to use an approximation.

However, there is a subtle danger when using loose bounding boxes. For example, assume the bounding box for data object `A` is looser than the bounding box for data object `B`. Even if data object `A` is within data object `B`, `A`'s bounding box might stick out beyond `B`'s, due to its looseness, and the R-tree index may conclude prematurely that the function `Within(A,B)` is FALSE and not apply a more careful test.

There are two solutions to this problem:

- Calculate exact bounding boxes for all data objects.
- Add a compensating factor, the maximum looseness, to the size of one of the arguments before comparing bounding boxes. You program this compensating factor in the bounding box portion of the strategy function code.

  In the example in the preceding paragraph, add `X` to the size of `B`'s bounding box, where `X` is the maximum looseness of `A`'s bounding box, before comparing `A` and `B`'s bounding boxes.

Other User-Defined Data Type Design Considerations

When you design a new user-defined data type to store multidimensional data, include all the dimensions likely to be used in a query. For example, suppose you are designing a user-defined data type to store information on beach resorts for a travel application. Since queries for resorts often include a time element, such as when are the high and low season rates for a particular resort, you might want to include a time dimension in the resort data type, as well as the usual location. When you create an R-tree index on a column of this data type, the time dimension will be built into the index, and queries that specify time might execute faster.

Include dimensions that are also selective. This means that the values in a particular dimension effectively separate desired data from undesired data. For example, latitude and longitude spans are probably selective in a database of satellite photos because they can separate out just the few pictures in an area of interest from many other pictures scattered over the earth.

Creating a New Operator Class

DataBlade modules usually supply their own operator class when implementing the R-tree access method. For example, the Informix Geodetic DataBlade module adds the **GeoObject_ops** operator class. This section describes how to create a new operator class.

Although the R-tree access method includes a default operator class called **rtree_ops**, Informix recommends you always create a new operator class if you are developing a DataBlade module that uses the R-tree access method.

The **rtree_ops** operator class is provided primarily for generic R-tree testing and as an example of how to create a new operator class. The **rtree_ops** operator class restricts the number of strategy functions to the four required ones: **Overlap**, **Equal**, **Contains**, and **Within**. If you want to create more than these four strategy functions, you *must* create your own operator class.

Steps to Create a New Operator Class

The following steps describe how to create a new operator class:

1. Create the required support functions. This step includes writing the C code using the DataBlade API to implement the required support functions and defining in BladeSmith the SQL statements to register the function with the database server. BladeSmith is a tool included in the DataBlade Developers Kit (DBDK).

   This step is described in "Support Functions" on page 3-14.

2. Create the required strategy functions. Similar to support functions, this step includes writing the C code using the DataBlade API to implement the required strategy functions and defining in BladeSmith the SQL statements to register the function with the database server. BladeSmith is a tool included in the DataBlade Developers Kit (DBDK).

   This step is described in "Strategy Functions" on page 3-19.

3. Create the operator class by creating custom SQL in BladeSmith to register the operator class with the database server.

   This step is described in "Syntax for Creating a New Operator Class" on page 3-31.

Each access method has different requirements for the support and strategy functions. The following sections describe the support and strategy functions required by the R-tree access method and examples on how to create them.

When you use the DataBlade Developers Kit (DBDK) to create an operator class, you do not have to create the SQL statements to register the support and strategy functions with the database server since the DBDK will automatically generate the necessary scripts. You will, however, need to write the C code that actually implements the support and strategy functions.

The DBDK does not automatically generate the SQL statement to create an operator class. Instead, you must create custom SQL files from BladeSmith by choosing **Edit→Insert→SQL Files**.

For more information on the DataBlade Developers Kit and BladeSmith, refer to *DataBlade Developers Kit User's Guide*.

For more information on the DataBlade API, refer to the *DataBlade API Programmer's Manual*.

Important: *The R-tree access method requires that all support and strategy functions be nonvariant, or that they always return the same results when invoked with the same arguments. To define a nonvariant function, specify* NOT VARIANT *in the WITH clause of the CREATE FUNCTION statement.*

If you use the DBDK to create the data type that is to be indexed by an R-tree index, and specify that the R-tree support and strategy functions be automatically generated, the NOT VARIANT clause will be included automatically in the CREATE FUNCTION statement. If, however, you create the support and strategy functions yourself, the VARIANT clause is included by default.

Support Functions

Support functions are user-defined functions used internally by the Informix database server to construct and maintain an R-tree index. They should never be explicitly executed by end users.

The support functions are used by the R-tree access method to determine the leaf page on which an index key belongs, and to create the special bounding-box only keys used internally by the R-tree index. For more information on bounding boxes, refer to "Bounding Boxes" on page 1-6.

The R-tree access method requires that you create the following three support functions:

- **Union**
- **Size**
- **Inter**

You must list these functions in the order shown when you execute the CREATE OPCLASS statement to register the operator class with the database server. This SQL statement is described in "Syntax for Creating a New Operator Class" on page 3-31.

The following sections describe how the R-tree access method uses the support functions, describe how you should write each function, and provide an example of an SQL statement used to create the **Union** function. Examples of the SQL statement to create the **Size** and **Inter** functions are not provided because they are very similar to the **Union** example.

Tip: *You can name support functions anything you want. It is useful, however, to name support functions in a way that describes what they do. For example, it makes sense to name a function that calculates the size of a bounding box* ***Size****.*

For convenience, this guide uses the names ***Union****,* ***Size****, and* ***Inter*** *when describing the three required support functions. These are also the names that the default operator class* ***rtree_ops*** *uses for its support functions.*

Internal Uses of the Support Functions

The R-tree access method uses the three required support functions in combination when it maintains the R-tree index. For example, when the access method is deciding into which subtree to place a new entry, it uses the **Union** and **Size** functions to determine how much each bounding box needs to expand if the new entry were added to that subtree. After a page splits, the access method uses the **Union** function to calculate a new bounding box for all entries on a page.

Important: *Support functions may be executed many times during the creation of an R-tree index. For this reason, Informix recommends that the corresponding C code for the support function be as fast and efficient as possible. Examples of increasing speed and efficiency in the C code is to not allocate memory, not open and close database connections, and so on.*

Union

The R-tree access method uses the **Union** function to find a new all-inclusive bounding box for the index entries on an index page when a new entry is added. The union of the old bounding box and the bounding box of the new entry is the new, possibly enlarged, bounding box for the entire index page.

The R-tree access method also uses the **Union** function when it calculates onto which index page it should put a new index entry. In conjunction with the **Size** function, the **Union** function shows how much the old bounding box must be enlarged to include the new index entry. The access method also uses the **Union** function after a page split to calculate the bounding box for the new page and to evaluate the new groupings between the old and new pages.

The signature of the **Union** support function must be:

```
Union (UDT, UDT, UDT) RETURNS INTEGER
```

UDT refers to *user-defined type*, or the data type you want to index with the R-tree access method.

Write the **Union** function to calculate the overall bounding box of the bounding boxes of the objects in the first two parameters and to store the result in the third parameter.

Although the function's signature indicates it returns an INTEGER value, the R-tree access method code ignores this return value. The result is returned as the third parameter of the function.

For variable UDTs, the third parameter of the **Union** function is not initialized; it contains a valid **mi_lvarchar** data type, with a conservatively large amount of memory allocated to it. The size is set in the function to the size, in bytes, of the largest possible result.

The result returned in the third paramater of the **Union** function must be fixed size and not a large object. Set its side large enough for any return value.

The R-tree access method implementation assumes that the size returned from the first call to the **Union** function is the size of all internal index keys. Therefore, when you write the code for the **Union** function, pick a maximum size for any internal index keys of an R-tree index and set the size of the union to that value.

See "Union Support Function" in Appendix A for sample C code of the **Union** function. The C code uses the DataBlade API to interact with the database server.

Size

The R-tree access method uses the **Size** function to compare bounding boxes to determine which is larger. Although the typical interpretation of size can be the area or volume of a bounding box, you can use any interpretation as long as the result can be used to compare the relative sizes of bounding boxes.

The signature of the **Size** support function must be:

```
Size (UDT, double precision ) RETURNS INTEGER
```

UDT refers to *user-defined type*, or the data type you want to index with the R-tree access method.

Write the **Size** function to calculate the relative size of the bounding box of the object in the first parameter and to store the result in the second parameter as a double-precision value. The R-tree access method uses this function when determining how best to split the disk page associated with an index page of the R-tree index. Ideally, a disk page is divided into two pages whose overall bounding boxes are as compact and small as possible.

Return a value of `0` for objects that do not exist in space, such as the intersection of two widely separated boxes, and return a small non-zero value for degenerate objects such as lines and points in two-dimensional space.

Although the function's signature indicates it returns an INTEGER value, the R-tree access method code ignores this return value. The result is returned as the second DOUBLE PRECISION parameter of the function.

See "Size Support Function" in Appendix A for sample C code of the **Size** function. The C code uses the DataBlade API to interact with the database server.

Inter

The signature of the **Inter** support function must be:

```
Inter (UDT, UDT, UDT) RETURNS INTEGER
```

UDT refers to *user-defined type*, or the data type you want to index with the R-tree access method.

Write the **Inter** function to calculate the intersection of the bounding boxes of the objects in the first two parameters and to store the result in the third parameter.

Although the function's signature indicates it returns an INTEGER value, the R-tree access method code ignores this return value. The result is returned as the third parameter of the function.

For variable length UDTs, the third argument of the **Inter** function is not initialized; it contains a valid **mi_lvarchar** data type, but the size must be set to the real length in the function.

See "Inter Support Function" in Appendix A for sample C code of the **Inter** function. The C code uses the DataBlade API to interact with the database server.

Implicit Casts

The database server automatically resolves internal function signatures for a subtype that inherits a function from a supertype in the following two cases:

- **Distinct types.** The database server automatically creates casts between the distinct type and source type.
- **Opaque types.** You must create the casts to support a type hierarchy.

You must first create a cast with the CREATE IMPLICIT CAST statement for it to be used implicitly during the execution of a query. The query optimizer tries to find implicit casts when it tries to make arguments fit support and strategy function signatures.

Example of Creating a Support Function

This example describes the SQL statement that registers the **Union** support function with the database server and a cross-reference to the sample C code in Appendix A to create the function. The example is based on the objects of the sample DataBlade module, described in Appendix A.

The SQL statements to register the **Size** and **Inter** support functions with the database server are very similar to the SQL statement to register the **Union** function. Sample C code to implement the **Size** and **Inter** functions is provided in Appendix A.

Tip: *The DataBlade Developers Kit automatically generates the SQL statement to create the function.*

The following SQL statement registers the **Union** support function with the database server. The three parameters of the function are all of data type MyShape. The C function **MyShapeUnion**, found in the shared object file **$INFORMIXDIR/extend/Shapes.3.6/Shapes.bld**, contains the actual C code that calculates the union of two objects of type MyShape.

```
CREATE FUNCTION Union (MyShape, MyShape, MyShape)
RETURNS INTEGER
WITH
(
    NOT VARIANT
)
EXTERNAL NAME "$INFORMIXDIR/extend/Shapes.3.6/Shapes.bld (MyShapeUnion)"
LANGUAGE C;
```

See "Union Support Function" in Appendix A for the sample C code of the **MyShapeUnion** function. The C code uses the DataBlade API to interact with the database server.

For more information on the DataBlade API, refer to the *DataBlade API Programmer's Manual*.

For more detailed information on and examples for creating user-defined functions, refer to *Extending INFORMIX-Universal Server: User-Defined Routines*.

Strategy Functions

Strategy functions are user-defined functions that can be used in queries to select data. Registering them as strategy functions via the CREATE OPCLASS statement lets the optimizer know that an associated R-tree index can be used to execute a query that contains one of those functions.

For example, assume there is an R-tree index on a column called **boxes**, and **Overlap** is defined as a strategy function. If a query contains the qualification `WHERE Overlap (a, region)`, the query optimizer will consider using the R-tree index to evaluate the query.

You can include up to eight strategy functions when you create a new operator class for the R-tree access method. You must, however, include the following four strategy functions:

- **Overlap**
- **Equal**
- **Contains**
- **Within**

You must list these functions first, in the order shown, when you execute the CREATE OPCLASS statement to register the operator class with the database server. This SQL statement is described in "Syntax for Creating a New Operator Class" on page 3-31.

The four required strategy functions are defined in more detail in later sections of this chapter, along with an example of creating the **Contains** strategy function.

Tip: *You can name strategy functions anything you want. It is useful, however, to name strategy functions in a way that describes what they do. For example, it makes sense to name a function that calculates whether one object overlaps another* ***Overlap****.*

For convenience, this guide uses the names ***Overlap****,* ***Equal****,* ***Contains****, and* ***Within*** *when describing the four required strategy functions. These are also the names that the default operator class* ***rtree_ops*** *uses for its strategy functions.*

Internal Uses of the Strategy Functions

The main purpose of the strategy functions is to tell the query optimizer when it should consider using an R-tree index, as described in the preceding section. However, the R-tree access method also uses the strategy functions internally to search in the R-tree index, to delete entries from the index, and to optimize the performance of updates to the index.

Searches

The R-tree access method uses the four required strategy functions in a variety of combinations when searching in an R-tree index, as described by the following table.

Slot Number	Strategy Function	Commutator Function	Function Called on an Index Key in a Nonleaf Page
1	Overlap	Overlap	Overlap
2	Equal	Equal	Contains
3	Contains	Within	Contains
4	Within	Contains	Overlap
5	Available for use	Same function	Same function
...	...	...	...
8	Avialable for use	Same function	Same function

The first column of the table (**Slot Number)** refers to the position in the CREATE OPCLASS statement of the strategy function. The four required strategy functions must be listed first, in the order shown in the second column.

The third column (**Commutator Function**) specifies the function that the R-tree access method uses as the commutator of a particular strategy function. The **Within** function is considered the commutator of the **Contains** function. For slots 5 through 8, which are available for any use and are not required, the strategy function is its own commutator function.

Commutator functions can be used as substitute functions by the query optimizer. For example, suppose a query has the predicate `Within(A, B)` in its WHERE clause, where `A` is a constant search object and `B` is a table column with an R-tree index defined on it. Predicate functions in WHERE clauses are written to work with an index on the *first* argument, so the **Within** function cannot be used in this case, since the R-tree index is on the *second* argument. The commutator information allows the optimizer to substitute `Contains(B, A)`, which allows the R-tree index on `B` to be used in the execution of the query.

The strategy functions in slots 5 through 8 can have commutator functions specified by the COMMUTATOR = <*FUNCTION*> modifier of the CREATE FUNCTION statements used to register the functions in SQL. If no commutator function is specified, the query optimizer will not attempt to change the order of the arguments in order to get an indexed column as the first argument. The following example registers the **Contains** strategy function and specifies that the **Within** function is its commutator:

```
CREATE FUNCTION Contains (MyShape, MyShape)
RETURNS BOOLEAN
WITH
(
    COMMUTATOR = Within,
    NOT VARIANT
)
EXTERNAL NAME "$INFORMIXDIR/extend/Shapes.3.6/Shapes.bld
(MyShapeContains)"
LANGUAGE C;
```

The fourth column (**Function Called on an Index Key in a Nonleaf Page**) specifies the function that the R-tree access method uses when searching for an index key in a nonleaf page. The following paragraph explains why the entry for **Within** is **Overlap**, and the entry for **Equal** is **Contains**.

Suppose a query has the predicate `Within(A, B)` in its WHERE clause, where `B` is a constant search object and `A` is a table column with an R-tree index defined on it. When a leaf page of the index is searched, the index entries are true candidates to match the query, so the **Within** function is used directly for each index entry. The search of a branch page, however, is actually a search to see if there exists an entry in the subtree below the branch page that is within the search object `B`. In this case, the search does not care whether the bounding box of the subtree is *within* `B`, but whether the bounding box of the subtree *overlaps* `B`. This is because a small entry within the subtree, in the overlapping portion of the bounding box, could be completely within `B`. Therefore, an index search that uses the **Within** function must substitute the **Overlap** function for nonleaf (branch) index pages.

Deletes and Updates

The access method uses the **Contains** function for index scans that search for leaf objects that must be deleted from the R-tree index after their associated row in the table is deleted.

The access method uses the **Equal** function to optimize the performance of updates to the R-tree index. When a row in a table is updated, any R-tree index on the table may also need to be updated. Updates usually translate into deleting the old entry and inserting the new entry. First, however, the access method uses the **Equal** strategy function to check whether the new entry is truly different from the old entry. If they are both equal, the access method does not actually perform the update.

Overlap

The **Overlap** function returns a Boolean value that indicates whether two objects overlap, or intersect, or have at least one point in common.

Figure 3-2 shows a circle that overlaps a triangle. The circle, however, does not overlap the box, since the circle does not have any points in common with the box.

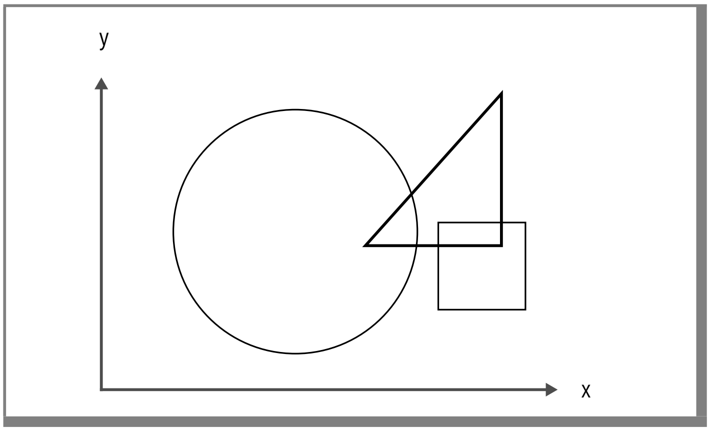


Figure 3-2
Example of a Circle That Overlaps a Triangle

The signature of the **Overlaps** functions must be:

```
Overlaps (UDT, UDT) RETURNS BOOLEAN
```

UDT refers to *user-defined type*, or the data type you want to index with the R-tree access method.

The **Overlaps** function returns `TRUE` if the object in the first parameter overlaps or intersects the object in the second parameter and `FALSE` otherwise.

When you design the **Overlaps** function, you might want to first test if the bounding boxes of the two objects overlap; and if they do, then test if the actual objects overlap. The first test is a relatively quick and easy calculation and might eliminate many candidates before the second, more complicated test.

For example, Figure 3-3 shows that the first bounding box test eliminates the box-circle overlap immediately, but the second actual object test is required to find out if the triangle and circle overlap. In this case, they do not.

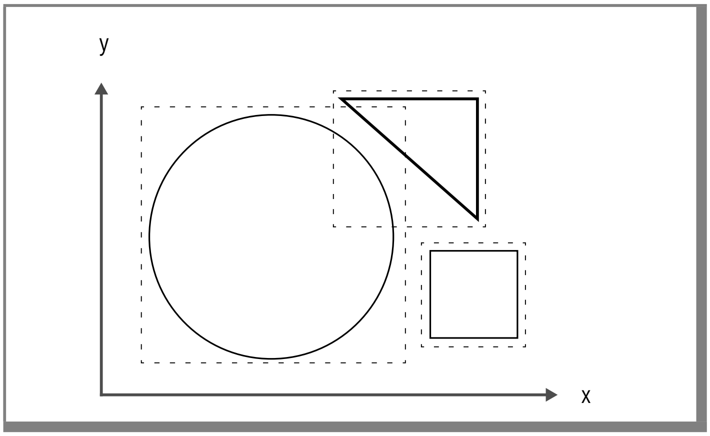


Figure 3-3
Bounding Box Example of the Overlap Function

Appendix A contains sample C code to create an **Overlap** function that takes the MyShape data type as its two parameters.

Equal

The **Equal** function returns a Boolean value that indicates whether two objects are equal. For example, in two-dimensional space, two points that have the same coordinates might be equal, as are two circles that have the same center and radius.

Important: *The meaning of "equality" between two spatial objects is often unclear, especially when floating point numbers are used. Bit-wise equality might be useful for eliminating duplicate data, but not much else. Application and data type designers need to define carefully what they mean when they say two spatial objects are equal.*

*SQL requires that you define an **Equal** function for your data type so that SELECT UNIQUE queries can execute successfully.*

The signature of the **Equal** functions must be:

```
Equal (UDT, UDT) RETURNS BOOLEAN
```

UDT refers to *user-defined type*, or the data type you want to index with the R-tree access method.

The **Equal** function returns TRUE if the two objects contained in the two parameters are equal and FALSE otherwise. It is up to the application or data type designer to define what "equal" means for the user-defined data type.

Appendix A contains sample C code to create an **Equal** function that takes the MyShape data type as its two parameters.

Contains

The **Contains** function returns a Boolean value that indicates whether an object entirely contains another object.

Figure 3-4 shows a circle that contains a box. The circle, however, does not contain the triangle, since part of the triangle lies outside the circle.

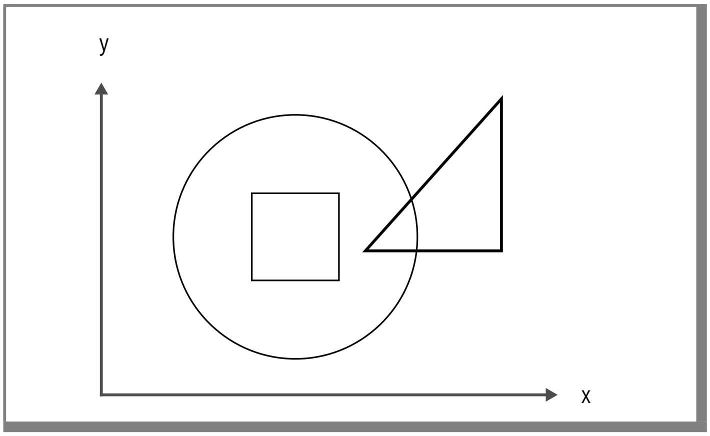


Figure 3-4
Example of a Circle That Contains a Box

The signature of the **Contains** functions must be:

```
Contains (UDT, UDT) RETURNS BOOLEAN
```

UDT refers to *user-defined type*, or the data type you want to index with the R-tree access method.

The **Contains** function returns TRUE if the object in the first parameter completely contains the object in the second parameter and FALSE otherwise.

When you design the **Contains** function, you might want to first test if the bounding box of the first object contains the bounding object of the second object; and if it does, then test if the actual first object contains the actual second object. The first test is a relatively quick and easy calculation and might eliminate many candidates before the second, more complicated test.

For example, Figure 3-5 shows that the first bounding box test eliminates the box-circle containment immediately, but the second actual object test is required to find out if the circle actually contains the triangle. In this case, it does not.

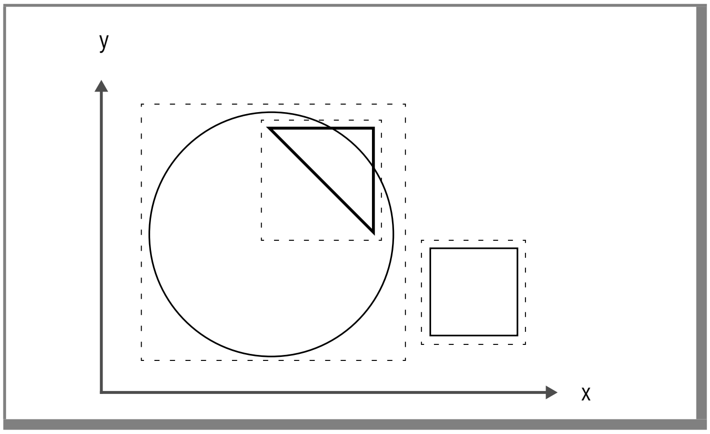


Figure 3-5
Bounding Box Example of the Contains Function

If you allow loose, or inexact, bounding boxes, be careful when you calculate the containment of bounding boxes. For example, Figure 3-6 shows that although the exact bounding box of the rectangle does not contain the loose bounding box of the circle, the rectangle still contains the circle.

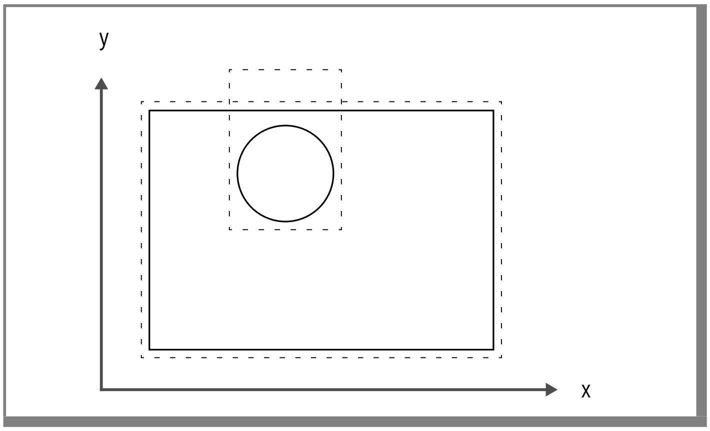


Figure 3-6
Containment and Loose Bounding Boxes

In this case, a preliminary test for bounding box containment is useless unless you use a compensating factor to account for the circle's loose bounding box. For more information on loose bounding boxes, refer to "Loose Bounding Box Calculations" on page 3-11.

Tip: *The **Within** strategy function is the commutator of the **Contains** strategy function. Remember to specify the **Within** function in the COMMUTATOR clause in the CREATE FUNCTION command when you create the **Contains** function. See "Example of Creating a Strategy Function" on page 3-30 for an example of specifying a commutator when creating a function.*

Appendix A contains sample C code to create a **Contains** function that takes the MyShape data type as its two parameters.

Within

The **Within** function returns a Boolean value that indicates whether an object is contained by another object. It is similar to the **Contains** function, but the order of the two parameters is switched.

Figure 3-7 shows a box that is within, or contained by, a circle. The triangle, however, is not within either the circle or the box, since all or part of the triangle lies outside both the circle and the box.

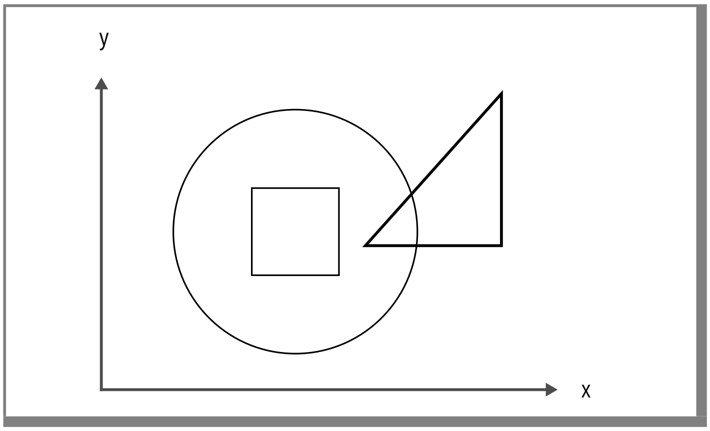


Figure 3-7
Example of a Box That is Within a Circle

The signature of the **Within** functions must be:

```
Within (UDT, UDT) RETURNS BOOLEAN
```

UDT refers to *user-defined type*, or the data type you want to index with the R-tree access method.

The **Within** function returns TRUE if the object in the first parameter is within, or completely contained in, the object in the second parameter and FALSE otherwise.

When you design the **Within** function, you might want to first test if the bounding box of the first object is contained in the bounding object of the second object; and if it does, then test if the actual first object is contained in the actual second object. The first test is a relatively quick and easy calculation and might eliminate many candidates before the second, more complicated test.

For example, Figure 3-8 shows that the first bounding box test eliminates the box-circle containment immediately, but the second actual object test is required to find out if the triangle is actually within the circle. In this case, it is not.

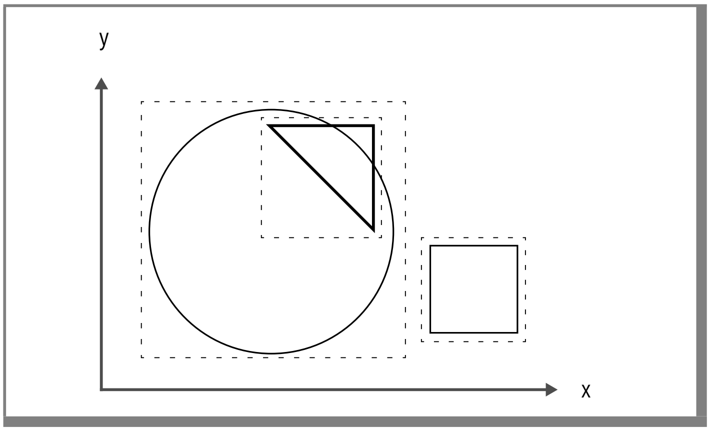


Figure 3-8
Bounding Box Example of the Within Function

If you allow loose, or inexact, bounding boxes, be careful when you calculate the containment of bounding boxes. For example, Figure 3-9 shows that although the loose bounding box of the circle is not within the exact bounding box of the rectangle, the circle is still within the rectangle.

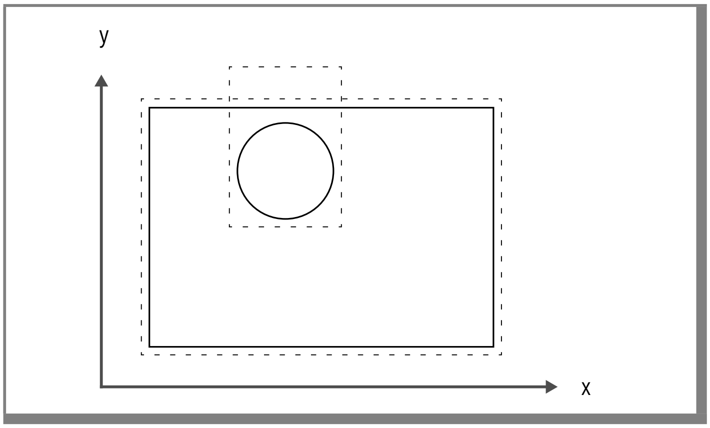


Figure 3-9
Containment and Loose Bounding Boxes

For more information on loose bounding boxes, refer to "Loose Bounding Box Calculations" on page 3-11.

Tip: *The **Contains** function is the commutator of the **Within** function. Remember to specify the **Contains** function in the COMMUTATOR clause in the CREATE FUNCTION command when you create the **Within** function. See "Example of Creating a Strategy Function" on page 3-30 for an example of specifying a commutator when creating a function.*

Appendix A contains sample C code to create a **Within** function that takes the MyShape data type as its two parameters.

Other Strategy Functions

You can create up to four nonrequired strategy functions for an operator class. This means that together with the four required functions, you can have a total of eight strategy functions defined for a particular operator class.

For example, you might want to create a function that calculates whether one object is outside a second object. You create the **Outside** function in the same way you create the other required functions, except that the C code to implement the function is quite different. When you create the operator class with the CREATE OPCLASS statement, you list the **Outside** function as the fifth strategy function, right after the four required strategy functions.

Other types of strategy functions you might want to create include specialized **Overlap** and **Within** functions. For example, these functions could implement whether two objects "overlap a lot," "overlap a little," or "interlock but do not touch."

The CREATE OPCLASS statement is described in "Syntax for Creating a New Operator Class" on page 3-31.

Example of Creating a Strategy Function

This example describes the SQL statement that registers the **Contains** strategy function with the database server . The sample C code to create the function is provided in Appendix A. The example is based on the objects of the sample DataBlade module, described in Appendix A.

The SQL statements to register the **Overlap, Equal,** and **Within** strategy functions with the database server are very similar to the SQL statement to register the **Contains** function. Sample C code to implement the **Overlap**, **Equal**, and **Within** functions is provided in Appendix A.

Tip: *The DataBlade Developers Kit automatically generates the SQL statement to create the function.*

The following SQL statement registers the **Contains** strategy function with the database server. The two parameters of the function are both of data type MyShape. The C function **MyShapeContains**, found in the shared object file **$INFORMIXDIR/extend/Shapes.3.6/Shapes.bld**, contains the actual C code that calculates whether the first object contains the second object. The statement specifies that the commutator of the **Contains** function is the **Within** function.

```
CREATE FUNCTION Contains (MyShape, MyShape)
RETURNS BOOLEAN
WITH
(
    COMMUTATOR = Within,
    NOT VARIANT
)
EXTERNAL NAME "$INFORMIXDIR/extend/Shapes.3.6/Shapes.bld (MyShapeContains)"
LANGUAGE C;
```

See "Contains Strategy Function" in Appendix A for the sample C code of the **MyShapeContains** function. The C code uses the DataBlade API to interact with the database server.

For more information on the DataBlade API, refer to the *DataBlade API Programmer's Manual*.

For more detailed information on and examples for creating user-defined functions, refer to *Extending INFORMIX-Universal Server: User-Defined Routines*.

Syntax for Creating a New Operator Class

Once you have created all the required support and strategy functions, you are ready to put everything together by creating the operator class.

The following syntax creates an operator class for use with the R-tree access method:

```
CREATE OPCLASS opclass
FOR RTREE
STRATEGIES (strategy, strategy, strategy, strategy [, strategy])
SUPPORT (support, support, support);
```

The FOR RTREE clause indicates to the database server that the operator class is for use with the R-tree access method.

The parameters are described in the following table.

Arguments	Purpose	Restrictions
opclass	The name you want to give your operator class.	The name must be unique in the database.
strategy	The names of the strategy functions you have previously created. Four strategy functions are required; any others are optional.	You can list a maximum of eight functions. You must include the following four strategy functions: **Overlap**, **Equal**, **Contains**, and **Within**. You can name them whatever you chose, but they must be listed as the first, second, third, and fourth functions, respectively.
support	The names of the three required support functions you have previously created.	You must include the following three support functions: **Union**, **Size**, and **Inter**. You can name them whatever you chose, but they must be listed as the first, second, and third functions, respectively.

If you are using the DataBlade Developers Kit (DBDK) to create an operator class, you do not have to create the SQL statements to register the support and strategy functions with the database server since the DBDK will automatically generate the necessary scripts. However, the DBDK does not automatically generate the SQL statement to create an operator class. Instead, you must create custom SQL files from BladeSmith by choosing **Edit→Insert→SQL Files**.

The following example is taken from the sample DataBlade module, described in Appendix A. It shows how to create the **MyShape_ops** operator class. The strategy functions are called **Overlap**, **Equal**, **Contains**, and **Within**. The support functions are called **Union**, **Size**, and **Inter**.

```
CREATE OPCLASS MyShape_ops
FOR RTREE
STRATEGIES (Overlap, Equal, Contains, Within)
SUPPORT (Union, Size, Inter);
```

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

For more information on the DataBlade Developers Kit (DBDK) and BladeSmith, refer to *DataBlade Developers Kit User's Guide*.

Creating Registration Scripts for Dependent DataBlade Modules

Once you have created one or more user-defined data types, an operator class, and other objects, use the DataBlade Developers Kit (DBDK) to package all the objects into an installable module.

All R-tree error messages are contained in the Informix R-Tree Secondary Access Method DataBlade module. Therefore, you must always register the Informix R-Tree Secondary Access Method DataBlade module into your database when using the R-tree access method so that the correct error message is returned if you encounter an R-tree error.

When you are developing a DataBlade module that uses the R-tree secondary access method, you can create a dependency on the Informix R-Tree Secondary Access Method DataBlade module so that BladeManager will automatically register both DataBlade modules in the correct order. BladeManager is the Informix product you use to register DataBlade modules in a database. You also use the DBDK to create a registration script that signals this dependency.

The dependency is signaled by importing an interface object provided by the Informix R-Tree Secondary Access Method DataBlade module. During registration of the dependent DataBlade module, BladeManager checks interface dependencies and warns the user registering the DataBlade modules if the Informix R-Tree Secondary Access Method DataBlade module is not already registered in the database.

Importing the ifxrltree Interface Object

In the BladeSmith tool, which is part of the DataBlade Developers Kit, an interface object represents a set of functionality provided by the DataBlade module that defines the interface. Each interface object has a unique name. The interface for the Informix R-Tree Secondary Access Method DataBlade module is named **ifxrltree1**. The functionality that it represents is the set of error objects defined in the module.

To complete the BladeSmith project for a DataBlade module dependent on the R-Tree access method, you must import the **ifxrltree1** interface object into the dependent DataBlade module's project file.

The **ifxrltree1** interface object is located in the Informix R-Tree Secondary Access Method DataBlade module project file, **ifxrltree.ibs**. This project file is located in the **$INFORMIXDIR/extend/ifxrltree.*version*** directory, where *version* refers to the version of the Informix R-Tree Secondary Access Method DataBlade module installed on your computer.

To import the ifxrltree1 interface object

1. If necessary, copy the **ifxrltree.ibs** project file from its location under the **$INFORMIXDIR/extend/ifxrltree.*version*** directory to a directory accessible from the Windows environment in which you run BladeSmith.
2. In BladeSmith, open **ifxrltree.ibs** in addition to opening the project of the dependent DataBlade module.
3. In the **ifxrltree** project (the project name for the Informix R-Tree Secondary Access Method DataBlade module), select the **ifxrltree1** interface object and copy it to the clipboard.
4. In the project of the dependent DataBlade module, choose **Edit→Import→From Clipboard** to import the **ifxrltree1** interface.

See the *DataBlade Developers Kit User’s Guide* for details on using BladeSmith. Refer to the *DataBlade Module Installation and Registration Guide* for more information on BladeManager.

Chapter

4

Managing Databases That Use the R-Tree Secondary Access Method

In This Chapter

This chapter discusses the following administrative issues related to the R-tree secondary access method:

- "Performance Tips" on page 4-3
- "Estimating the Size of an R-Tree Index" on page 4-6
- "System Catalogs" on page 4-9

Performance Tips

This section discusses tips on improving the performance of using R-tree indexes. It includes topics on how to maintain accurate statistics and how to improve the performance of queries that use R-tree indexes.

You might also want to refer to "Designing the User-Defined Data Type" on page 3-6, which describes performance considerations when designing the user-defined data type of the column that will be indexed with an R-tree index.

The *Performance Guide* for your database server covers other performance issues that are also relevant to R-tree indexes. Refer to that guide for more information on performance issues.

Updating Statistics

The operator class that is specified when you create an R-tree index defines the strategy functions that tell the query optimizer when to *consider* using an R-tree index when the strategy function appears in the WHERE clause of a query.

The query optimizer, however, might decide not to use an R-tree index when it calculates how to execute a query, even if a strategy function is specified in the WHERE clause. The query optimizer uses available statistics to calculate the cost of using or not using the index. If not using an R-tree index is less costly that using it, the query optimizer might decide to execute a table scan instead of an index scan.

Use the SQL statement UPDATE STATISTICS to ensure that the statistics on an R-tree indexed column are always correct and up to date. Incorrect statistics can cause a query to execute slower than if there are no statistics on the indexed column at all.

You should run UPDATE STATISTICS whenever you make extensive modifications to the table or the distribution of the data in the indexed column changes significantly.

Important: *Be sure to always run UPDATE STATISTICS after loading data into a table that has an R-tree index. Without the new statistics, the query optimizer may think the table is small and never even consider using the R-tree index.*

The following example shows how to update the statistics of the **boxes** column of the **box_tab** table:

```
UPDATE STATISTICS FOR TABLE box_tab (boxes);
```

When the UPDATE STATISTICS command is executed on a column with an R-tree index, the MEDIUM and HIGH keywords are ignored.

The following statistics are generated when the UPDATE STATISTICS command is executed on a colum that has an R-tree index:

- The number of levels in the R-tree index
- An estimated number of entries in a branch page
- An estimated number of entries in a leaf page
- An estimated number of leaf pages
- The number of unique values in the index
- The number of clusters in the index

For more detailed information on the UPDATE STATISTICS statement, refer to the *Informix Guide to SQL: Syntax*.

Deletions

Deletions from tables that have an R-tree index might be slow if the WHERE clause of the DELETE statement does not specify the R-tree indexed column.

When deletions from tables are done with a DELETE statement that uses an R-tree index to find the rows to be deleted, the entries in the R-tree index can also can be deleted or marked as deleted at the same time. This is relatively efficient. However, when rows are deleted by a query that does *not* use an R-tree index, a separate index search is needed for *each* deleted row to find the corresponding index entry. This might slow the overall performance of the delete operation.

Therefore, if a large fraction of the rows are to be deleted this way, it might be faster to first drop the R-tree index, delete all the rows, and then re-create the index.

For example, assume you have an **employees** table that includes the following two columns: **id**, the employee's unique ID, and **location**, a map that shows the location of the employee's office. A B-tree index exists on the **id** column, and an R-tree index exists on the **location** column.

Further assume that all current employees have IDs greater than 2000, and you want to clean up the table by deleting all the rows whose **id** is less than 2000, or nonexistent employees. The DELETE statement might look like the following example:

```
DELETE FROM employees
WHERE id < 2000;
```

Because a B-tree index exists on the **id** column, the database server will quickly find and delete all the relevant rows in the *table*. However, because an R-tree index exists on the **location** column, each corresponding entry in the R-tree index must also be flagged for deletion. Since the database server has no quick way of finding the deleted rows in the R-tree index, it must perform an index search for *each* row that is deleted. The performance of this deletion might improve if the R-tree index on the location column is dropped first and then re-created after the deletion is complete.

Important: *Although a delete affecting many rows might execute slowly due to the presence of an R-tree index, the deletion of data and the update of the index will still execute correctly.*

Effectiveness of Bounding Box Representation

The characteristics of the data stored in an R-tree indexed column can affect the performance of queries that search the data. The higher the *selectivity* of the data, the faster the queries will execute. Although you might not have any control over what your data looks like, it is useful to know how it can affect queries.

The selectivity of data indexed with the R-tree access method is affected by two characteristics of the data: how much overlap occurs and the relative sizes of close objects. The more overlap that occurs between the bounding boxes of the objects, the lower the selectivity of the data. Grouping many small bounding boxes close to one very large bounding box lowers the selectivity of the small bounding boxes as it increases the selectivity of the large bounding box.

An example of data that may have good selectivity is the set of lakes on a map. Although the lakes might be oddly shaped, they are compact and well represented by bounding boxes. In a small area, the bounding boxes of far away lakes do not appear.

An example of data that may have bad selectivity is satellite ground tracks. Over time, the tracks cover most of the earth, so the bounding boxes of a particular satellite greatly overlap the bounding boxes of other satellites. Looking at a particular place on earth does not eliminate many satellites, unless time can also be used for finer resolution. Airline routes also behave similarly.

Estimating the Size of an R-Tree Index

There are two ways to estimate the size of an R-tree index:

- "Calculating Index Size Based on Number of Rows" on page 4-7 shows how to estimate index size by performing a series of calculations.
- "Using the oncheck Utility to Calculate Index Size" on page 4-8 shows how to use the **oncheck** utility to estimate index size.

Calculating Index Size Based on Number of Rows

You can estimate the size of an R-tree index in pages by performing a series of calculations based on the number of rows in the table.

The following procedure only estimates the number of leaf pages in the R-tree index; it does not calculate the number of branch pages. This is because almost all of the space in an R-tree index is usually taken up by leaf pages, due to the wide shape of the tree. Therefore, calculating the number of leaf pages is usually adequate for a rough estimate of the *total* number of disk pages that make up the R-tree index.

To estimate the size of an R-tree index in disk pages

1. Determine the size, in bytes, of the key value for the data type being indexed. This value is referred to in this section as *colsize*.

   Entries of this size will appear in index leaf pages.

   If you are indexing a user-defined data type, the size of the key value will be value of the INTERNALLENGTH variable of the CREATE OPAQUE TYPE statement.

2. Determine the size, in bytes, of the bounding box for the data type. This value is referred to in this section as *boundingbox*.

   The size of the bounding box is usually a variant of the user-defined key object, which is understood by the user code but opaque to the R-tree code.

   Keys of this type are used in index branch pages.

3. Determine the size, in bytes, of each index entry in the leaf page with the following formula that incorporates the overhead:

   ```
   leafentrysize = colsize + 16 bytes
   ```

4. Determine the *pagesize* in bytes of the database server that you use. To obtain the page size, run the following command and look for the value next to `Page Size`:

   ```
   oncheck -pr
   ```

5. Estimate the number of entries per index-leaf page with the following formula:

   ```
   leafpagents = trunc ( pagefree / leafentrysize ) * 60%
   ```

   where

   ```
   pagefree = pagesize - 88
   ```

   The value *leafpagents* is multiplied by 60% because index leaf pages are usually just over half full.

   The **trunc()** function notation indicates you should round down to the nearest integer value.

6. Estimate the number of leaf pages with the following formula:

   ```
   leaves = rows / pagents
   ```

   Use the SQL statement SELECT COUNT(*) FROM <*table*> to calculate the number of rows in the table.

The number of leaf pages that make up the R-tree index is very close to the *total* number of disk pages that make up the index.

Important: *As rows are deleted from the table, and new ones are inserted, the number of index entries can vary within a page. The calculation described in this section yields an estimate for an R-tree index whose leaf pages are 60% full. Your R-tree index may be smaller or larger depending on the activity within the table and the data that you store.*

Using the oncheck Utility to Calculate Index Size

You can also use the **-pT** option of the **oncheck** utility to estimate the size of an R-tree index. The syntax is as follows:

```
oncheck -pT <dbname>:<tablename>
```

The **-pT** option of the **oncheck** utility prints out space allocation information for the specified table and all the indexes that exist on the table, including R-tree indexes. For example, to display space allocation information for the **circle_tab** table in the **rtree** database, run the following command as user **informix** at the UNIX shell or Windows command prompt:

```
oncheck -pT rtree:circle_tab
```

For more information on the **oncheck** utility, refer to the *Administrator's Guide* for your database server.

System Catalogs

The R-tree access method is table driven. This means that information about the R-tree access method is stored in system catalogs, which the database server queries when it uses the R-tree access method.

The principal system catalogs that contain access method information are **sysams**, **sysopclasses**, and **sysindices**.

sysams

When the R-tree access method is initially created, information about the access method is stored in the **sysams** system catalog. The database server uses this information to dynamically load support for the access method and call the correct user-defined function for a given task. These tasks include creating an R-tree index, scanning the index, inserting into the index, and updating the index.

Some of the columns of the **sysams** table include:

- **am_name**, the internal name of the access method. For the R-tree access method, the value of this column is `rtree`.
- **am_type**, the type of the index: primary (`P`) or secondary (`S`). R-tree is a secondary (`S`) index.
- **am_sptype,** the storage type of the index: dbspace (`D`), external to the database (`X`), sbspace (`S`), or any (`A`). R-tree indexes are stored in dbspaces (`D`).
- **am_defopclass**, the unique identifier of the default operator class. The unique identifier for the R-tree access method is `2`, which corresponds to the row for **rtree_ops** in the **sysopclasses** system catalog.

The following query returns values for the **am_name**, **am_owner**, **am_id**, **am_sptype**, and **am_defopclass** columns of the **sysams** system catalog for the `rtree` entry:

```
SELECT am_name, am_owner, am_id, am_type, am_sptype, am_defopclass
FROM sysams
WHERE am_name = 'rtree';


am_name            am_owner       am_id am_type am_sptype am_defopclass

rtree              informix           2 S       D                     2
```

The query shows that the internal name of the R-tree access method is `rtree`, which is the name you specify in the USING clause of the CREATE INDEX statement when you create an R-tree index. The **am_sptype** column shows that R-tree indexes are stored in dbspaces, often in the same dbspace the indexed table is stored. The identifier for the default operator class, shown by the **am_defopclass** column, is `2`. A query of the **sysopclasses** system catalog would show that **rtree_ops** has a unique identifier of `2` and is thus the default operator class for the R-tree access method.

For a complete description of the columns of the **sysams** system table, refer to the *Informix Guide to SQL: Reference*.

sysopclasses

The **sysopclasses** system catalog stores information about operator classes. Each time a new operator class is created with the CREATE OPCLASS statement, a row is added to this table.

Some of the columns of the **sysopclasses** table include:

- **opclassname**, the internal name of the operator class.
- **amid**, the unique identifier of the access method that uses the operator class.
- **ops**, the list of strategy functions defined for the operator class. Information about the strategy function is stored in the **sysprocedures** system table.

- **support**, the list of support functions defined for the operato class. Information about the support function is stored in the **sysprocedures** system table.

The following query returns all columns of the **sysopclasses** system catalog for the **MyShape_ops** operator class:

```
SELECT *
FROM sysopclasses
WHERE opclassname = 'myshape_ops';



opclassname  myshape_ops
owner        informix
amid         2
opclassid    100
ops          overlap;equal;contains;within;
support      union;size;inter;
```

Tip: *Because Informix always converts object names to lowercase when updating system catalogs, the preceding query searches for the **myshape_ops** operator class instead of the **MyShape_ops** operator class.*

The query shows that the strategy functions for the **myshape_ops** operator class are **Overlap**, **Equal**, **Contains**, and **Within**. The support functions are **Union**, **Size**, and **Inter**, as required.

The following query of the **sysprocedures** table returns information about the available **Within** strategy functions, such as their signatures and connections to the shared library:

```
SELECT paramtypes, externalname
FROM sysprocedures
WHERE procname = 'within';



paramtypes    myshape,myshape
externalname
$INFORMIXDIR/extend/Shapes.3.6/Shapes.bld(MyShapeWithin)

paramtypes    mypoint,mypoint
externalname
$INFORMIXDIR/extend/Shapes.3.6/Shapes.bld(MyPointWithin)

paramtypes    mycircle,mycircle
externalname
$INFORMIXDIR/extend/Shapes.3.6/Shapes.bld(MyCircleWithin)

paramtypes    mybox,mybox
externalname
$INFORMIXDIR/extend/Shapes.3.6/Shapes.bld(MyBoxWithin)
```

The result shows that four **Within** functions exist in the database for the four data types MyShape, MyPoint, MyCircle, and MyBox.

To determine the operator classes that are already available in your database for the R-tree access method, execute the following query:

```
SELECT opclassname, opclassid
FROM sysopclasses, sysams
WHERE sysopclasses.amid = sysams.am_id AND
      sysams.am_name = 'rtree';


opclassname          opclassid

rtree_ops                    2
myshape_ops                100
```

The result shows that the database contains two operator classes that can be used with the R-tree access method: **rtree_ops** and **myshape_ops**.

Important: *If you have registered a DataBlade module that supplies its own operator class, you must specify it when you create an R-tree index. Do not specify the default* ***rtree_ops*** *operator class.*

For a complete description of the columns of the **sysopclasses** system table, refer to *Informix Guide to SQL: Reference*.

sysindices

The **sysindices** system catalog stores information about indexes, including R-tree indexes.

Some of the columns of the **sysindices** table include:

- **idxname**, the name of the index.
- **tabid**, the unique identifier of the indexed table.
- **amid**, the unique identifier of the access method used to create the index. This is a join column with the **sysams** table.

Since DB-Access provides information about the indexes that exist for a particular table, you do not have to query the **sysindices** table directly.

Sample DataBlade Module

This appendix describes the sample DataBlade module used in the examples in this guide.

The sample DataBlade module is not sold by Informix. It is provided as one of the many examples in the DataBlade Developers Kit, an Informix product that helps developers design, create, and package DataBlade modules in a Windows environment.

The R-tree example provided in the DataBlade Developers Kit is located in the following directory:

```
%INFORMIXDIR%\dbdk\examples\indexes\dapi\Rtree
```

%INFORMIXDIR% refers to the root Informix directory in which the DataBlade Developers Kit is installed on your Windows computer. The example includes the C code to create the data types and functions, BladeManager registration scripts, functional tests to show how the DataBlade module works, and project files to package the module with BladePack.

The first section of this appendix, "Description of the Sample DataBlade Module" on page A-2, describes the data types and operators provided by the sample DataBlade module. The second section, "Sample C Code" on page A-5, provides the C code to create the strategy and support functions defined in the operator class. The header files **Shapes.h** and **ShapeTypes.h** that describe common elements are also included at the end of this appendix.

The sample DataBlade module in the DataBlade Developers Kit does not create a new operator class, but instead it uses the default **rtree_ops** operator class to create R-tree indexes. Since Informix recommends you always create a new operator class when you develop a DataBlade module, this appendix describes how to create the **MyShape_ops** operator class.

For more information on the DataBlade Developers Kit, refer to the *DataBlade Developers Kit User's Guide*. For more information on BladeManager, refer to the *DataBlade Module Installation and Registration Guide*.

Description of the Sample DataBlade Module

This section describes the data types and operators that make up the sample DataBlade module.

Data Types

The sample DataBlade module defines four spatial data types that allow you to create table columns that contain two-dimensional objects such as points, circles, and boxes. The four new data types are called MyShape, MyPoint, MyCircle, and MyBox.

The MyShape data type implements the behavior of all four types. The MyPoint, MyCircle, and MyBox data types delegate to the MyShape data type for their functionality. This means that the C code that implements the functions of MyPoint, for example, calls the C code that implements the corresponding MyShape function and passes it points as arguments.

The following example creates a table called **box_tab**. It contains a column called **boxes** of data type MyBox.

```
CREATE TABLE box_tab
(
    id       INTEGER,
    boxes    MyBox
);
```

The following INSERT statements show how to insert three different boxes into the **box_tab** table:

```
INSERT INTO box_tab
VALUES (1, 'box(10,10,40,40)' );

INSERT INTO box_tab
VALUES (2, 'box(-10,-20,5,9)' );
```

A box is described by its lower-left and upper-right coordinates. For example, the first INSERT statement inserts a box whose lower-left coordinate is `(10,10)` and upper-right coordinate is `(40,40)`.

Similarly, the following examples show how to create and insert into tables that have MyCircle and MyPoint columns:

```
CREATE TABLE circle_point_tab
(
    id         INTEGER,
    circles    MyCircle,
    points     MyPoint
);

INSERT INTO circle_point_tab
VALUES (1, 'circle(20,30,15)', 'point(10,15)' );

INSERT INTO circle_point_tab
VALUES (2, 'circle(-30,-10,25)', 'point(-20,-5)' );
```

Operators

The sample DataBlade module defines the following four operators that can be used on columns of data type MyShape, MyBox, MyCircle, and MyPoint in the WHERE clause of a query:

- **Overlap** returns a Boolean value to indicate whether two shapes intersect or overlap.
- **Equal** returns a Boolean value to indicate whether two shapes are the same or occupy the same space.
- **Contains** returns a Boolean value to indicate whether the first shape contains the second shape.

- **Within** returns a Boolean value to indicate whether the first shape is within or is contained by the second shape.

These operators, of course, are also the strategy functions defined by the operator class.

The following example uses the **Overlap** operator to return all the boxes in the **box_tab** table that overlap a box whose lower-left coordinate is `(30,20)` and upper-right coordinate is `(60,50)`:

```
SELECT * FROM box_tab
WHERE Overlap (boxes, 'box(30,20,60,50)' );



id     1
boxes  box(10.000,10.000,40.000,40.000)
```

This example uses the **Contains** operator to return all the boxes in the **box_tab** table that contain a box whose lower-left coordinate is `(-5,-10)` and upper-right coordinate is `(2,5)`:

```
SELECT * FROM box_tab
WHERE Contains (boxes, 'box(-5,-10,2,5)' );



id     2
boxes  box(-10.000,-20.000,5.000,9.000)
```

Operator Class

The sample DataBlade module uses the default **rtree_ops** operator class when it creates R-tree indexes on its sample tables. However, Informix recommends that you always create a new operator class if you are using the R-tree access method to index a new data type.

The following example creates the operator class **MyShape_ops** for use with the sample DataBlade module:

```
CREATE OPCLASS MyShape_ops
FOR RTREE
STRATEGIES (Overlap, Equal, Contains, Within)
SUPPORT (Union, Size, Inter);
```

After following the directions in the examples directory of the DataBlade Developers Kit to package the sample R-tree DataBlade module and register it in your database, you create the **MyShape_ops** operator class by executing the preceding example SQL statement.

The **MyShape_ops** operator class can be used to create R-tree indexes on columns of type MyShape, MyPoint, MyCircle, and MyBox, as shown in the following example:

```
CREATE INDEX box_tab_index
ON box_tab ( boxes MyShape_ops )
USING RTREE;
```

Sample C Code

The sample DataBlade module includes four data types: MyShape, MyBox, MyCircle, and MyPoint. The MyShape data type implements the behavior of all four data types.

The C functions that implement the strategy and support functions for the MyBox, MyCircle, and MyPoint data types simply call the corresponding C function for the MyShape data type with the appropriate parameters, as shown in the following example:

```
/*****************************************************************
**
** Function name:
**
**  MyBoxContains
**
** Description:
**
** Special Comments:
**
**  Entrypoint for the SQL routine Contains (MyBox,MyBox) returns boolean.
**
** Parameters:
**
** Return value:
**
**  mi_integer
**
** History:
**
**  06/26/1998 - Generated by BladeSmith Version 3.60.626  .
**
** Identification:
**
** NOTE:
**
**  BladeSmith will add and remove parameters from the function
**  prototype, and will generate tracing calls.  ONLY EDIT code
**  in blocks marked  Your_<section>.  Any other  modifications
**  will require manual merging.
**
*****************************************************************
*/

UDREXPORT
mi_integer MyBoxContains
(

MyBox *                 Argument1,

MyBox *                 Argument2,
MI_FPARAM *             Gen_fparam        /* Standard info - see DBDK docs.*/
)

{
    MI_CONNECTION * Gen_Con;          /* The connection handle.             */
    mi_integer      Gen_RetVal;       /* The return value.                  */
    /* ------ {{Your_Declarations (PreserveSection) BEGIN ------ */
    /* ------ }}Your_Declarations (#0000) END ------ */

    /* Use the NULL connection. */
    Gen_Con = NULL;

    /* ------ {{Your_Code (PreserveSection) BEGIN ------ */
    Gen_RetVal = MyShapeContains( (MyShape *) Argument1, (MyShape *)Argument2 ,
Gen_fparam );
    /* ------ }}Your_Code (#BD8C) END ------ */


    /* Return the function's return value. */
    return Gen_RetVal;
}
```

As you can see, the **MyBoxContains** C function calls the **MyShapeContains** C function.

For this reason, this appendix supplies the C code for the strategy and support functions of just the MyShape data type and not for the other data types.

Overlap Strategy Function

```
/****************************************************************
**
** Function name:
**
**  MyShapeOverlap
**
** Description:
**
** Special Comments:
**
**  Entrypoint for the SQL routine Overlap (MyShape,MyShape) returns boolean.
**
** Parameters:
**
** Return value:
**
**  mi_integer
**
** History:
**
**  06/26/1998 - Generated by BladeSmith Version 3.60.626  .
**
** Identification:
**
** NOTE:
**
**  BladeSmith will add and remove parameters from the function
**  prototype, and will generate tracing calls.  ONLY EDIT code
**  in blocks marked  Your_<section>.  Any other  modifications
**  will require manual merging.
**
****************************************************************
*/

UDREXPORT
mi_integer MyShapeOverlap
(

MyShape *               Argument1,

MyShape *               Argument2,
MI_FPARAM *             Gen_fparam        /* Standard info - see DBDK docs.*/
)

{
    MI_CONNECTION * Gen_Con;          /* The connection handle.             */
    mi_integer      Gen_RetVal;       /* The return value.                  */
    /* ------ {{Your_Declarations (PreserveSection) BEGIN ------ */
    mi_boolean  boxOverlap;
    ShapeHdr* h1 = (ShapeHdr*) mi_get_vardata ( (mi_lvarchar*) Argument1 );
    ShapeHdr* h2 = (ShapeHdr*) mi_get_vardata ( (mi_lvarchar*) Argument2 );
    /* ------ }}Your_Declarations (#0000) END ------ */

    /* Use the NULL connection. */
    Gen_Con = NULL;

    /* ------ {{Your_Code (PreserveSection) BEGIN ------ */
    /*
     * Bounding box check.
     */
    boxOverlap = ( h1->xmin <= h2->xmax && h2->xmin <= h1->xmax &&
              h1->ymin <= h2->ymax && h2->ymin <= h1->ymax );

    if (!boxOverlap)
    {
        Gen_RetVal =  MI_FALSE;
    }
    else
    {
        if (h1->tag == HeaderT || h2->tag == HeaderT)
       Gen_RetVal = MI_TRUE;
        else
       Gen_RetVal = Dispatch(intersectTable, MI_TRUE,
                    (Shape*) h1, (Shape*) h2, Gen_fparam);
    }
    /* ------ }}Your_Code (#CAD2) END ------ */


    /* Return the function's return value. */
    return Gen_RetVal;
}
```

Equal Strategy Function

```
/*****************************************************************
**
** Function name:
**
**  MyShapeEqual
**
** Description:
**
**  Determine if one UDT value is equal to another.
**
** Special Comments:
**
**  Compares two variable-length opaque types for equality
**
** Parameters:
**
**  mi_bitvarying * Gen_param1;       The first UDT value to compare.
**  mi_bitvarying * Gen_param2;       The second UDT value to compare.
**  MI_FPARAM *     Gen_fparam;       Standard info - see DBDK docs.
**
** Return value:
**
**  mi_boolean                        The comparison result.
**
** History:
**
**  06/26/1998 - Generated by BladeSmith Version 3.60.626  .
**
** Identification:
**
**  Warning: Do not remove or modify this comment:
**      MyShapeEqual FunctionId: 9a830881-133b-11d1-80af-0800095a455b
**
*****************************************************************
*/

mi_boolean
MyShapeEqual
(
mi_bitvarying *        Gen_param1,      /* The first UDT value to compare.    */
mi_bitvarying *        Gen_param2,      /* The second UDT value to compare.   */
MI_FPARAM *            Gen_fparam       /* Standard info - see DBDK docs.     */
)
{
    /* Call Compare to perform the comparison. */
    return (mi_boolean)(0 == MyShapeCompare( Gen_param1,
            Gen_param2, Gen_fparam ));
}
```

Contains Strategy Function

```
/***************************************************************
**
** Function name:
**
**  MyShapeContains
**
** Description:
**
** Special Comments:
**
**  Entrypoint for the SQL routine Contains (MyShape,MyShape) returns boolean.
**
** Parameters:
**
** Return value:
**
**  mi_integer
**
** History:
**
**  06/26/1998 - Generated by BladeSmith Version 3.60.626  .
**
** Identification:
**
** NOTE:
**
**  BladeSmith will add and remove parameters from the function
**  prototype, and will generate tracing calls.  ONLY EDIT code
**  in blocks marked  Your_<section>.  Any other  modifications
**  will require manual merging.
**
****************************************************************
*/

UDREXPORT
mi_integer MyShapeContains
(

MyShape *               Argument1,

MyShape *               Argument2,
MI_FPARAM *             Gen_fparam        /* Standard info - see DBDK docs.*/
)

{
    MI_CONNECTION * Gen_Con;          /* The connection handle.             */
    mi_integer      Gen_RetVal;       /* The return value.                  */
    /* ------ {{Your_Declarations (PreserveSection) BEGIN ------ */
    ShapeHdr* h1;
    ShapeHdr* h2;
    mi_boolean boxOverlap;
```

```
/* ------ }}Your_Declarations (#0000) END ------ */

/* Use the NULL connection. */
Gen_Con = NULL;

/* ------ {{Your_Code (PreserveSection) BEGIN ------ */
h1 = (ShapeHdr*) mi_get_vardata ( (mi_lvarchar*) Argument1 );
h2 = (ShapeHdr*) mi_get_vardata ( (mi_lvarchar*) Argument2 );

/* PROGRAMMER's NOTE:
   This function determines if shape 1( argument1 ) contains shape 2
   (argument2). If the first region is a point, then NULL value is
   returned. Else we return the true only if the second region is
   completely contained in the first.
*/

/*
 * Return NULL for non-region first argument.
 */
switch (h1->tag)
  {
  case HeaderT:
  case BoxT:
  case CircleT:
    boxOverlap = MI_TRUE; /* This is set just to enter the next if loop */
        break;

  case PointT:
  default:
        mi_fp_setreturnisnull((Gen_fparam), 0, MI_TRUE);
        Gen_RetVal = MI_FALSE;
  }

if( boxOverlap )
  {

    boxOverlap = ( h1->xmin <= h2->xmax && h2->xmin <= h1->xmax &&
          h1->ymin <= h2->ymax && h2->ymin <= h1->ymax );

    if (!boxOverlap)
      {
    Gen_RetVal =  MI_FALSE;
      }
    else
      {
    /*
     * Internal index node case.
     */
    if (h1->tag == HeaderT || h2->tag == HeaderT)
      {
        Gen_RetVal =  MI_TRUE;
      }
    else
      {
            /*
             * Geometric test.
             */
            Gen_RetVal =  Dispatch(insideTable, MI_FALSE,
                       (Shape*) h2, (Shape*) h1, Gen_fparam);
          }
          }
      }
    /* ------ }}Your_Code (#BD8C) END ------ */


    /* Return the function's return value. */
    return Gen_RetVal;
}
```

Within Strategy Function

```
/*************************************************************
**
** Function name:
**
**  MyShapeWithin
**
** Description:
**
** Special Comments:
**
**  Entrypoint for the SQL routine Within (MyShape,MyShape) returns boolean.
**
** Parameters:
**
** Return value:
**
**  mi_integer
**
** History:
**
**  06/26/1998 - Generated by BladeSmith Version 3.60.626  .
**
** Identification:
**
** NOTE:
**
**  BladeSmith will add and remove parameters from the function
**  prototype, and will generate tracing calls.  ONLY EDIT code
**  in blocks marked  Your_<section>.  Any other  modifications
**  will require manual merging.
**
**************************************************************
*/

UDREXPORT
mi_integer MyShapeWithin
(

MyShape *               Argument1,

MyShape *               Argument2,
MI_FPARAM *             Gen_fparam        /* Standard info - see DBDK docs.*/
)

{
    MI_CONNECTION * Gen_Con;          /* The connection handle.             */
    mi_integer      Gen_RetVal;       /* The return value.                  */
    /* ------ {{Your_Declarations (PreserveSection) BEGIN ------ */
    ShapeHdr * h1;
    ShapeHdr * h2;
    mi_boolean boxOverlap = MI_FALSE;
    /* ------ }}Your_Declarations (#0000) END ------ */

    /* Use the NULL connection. */
    Gen_Con = NULL;

    /* ------ {{Your_Code (PreserveSection) BEGIN ------ */
    h1 = (ShapeHdr*) mi_get_vardata ( (mi_lvarchar*) Argument1 );
    h2 = (ShapeHdr*) mi_get_vardata ( (mi_lvarchar*) Argument2 );

    /*
     * Return NULL for non-region second argument.
     */
    switch (h2->tag)
      {
      case HeaderT:
      case BoxT:
      case CircleT:
        boxOverlap = MI_TRUE; /* Used only to enter the next if */
            break;

      case PointT:
      default:
            mi_fp_setreturnisnull((Gen_fparam), 0, MI_TRUE);
            Gen_RetVal =  MI_FALSE;
      }
    if( boxOverlap )
      {
        boxOverlap = ( h1->xmin <= h2->xmax && h2->xmin <= h1->xmax &&
              h1->ymin <= h2->ymax && h2->ymin <= h1->ymax );

        if (!boxOverlap)
          {
        Gen_RetVal = MI_FALSE;
          }
        else
          {
        /*
         * Internal index node case.
         */
        if (h1->tag == HeaderT || h2->tag == HeaderT)
          {
            Gen_RetVal = MI_TRUE;
          }
        else
          {
            /*
             * Geometric test
             */
            Gen_RetVal = Dispatch(insideTable, MI_FALSE,
                     (Shape*) h1, (Shape*) h2, Gen_fparam);
          }
          }
      }
    /* ------ }}Your_Code (#0L2V) END ------ */
```

```
    /* Return the function's return value. */
    return Gen_RetVal;
}
```

Union Support Function

```
/*************************************************************
**
** Function name:
**
**  MyShapeUnion
**
** Description:
**
** Special Comments:
**
**  Entrypoint for the SQL routine Union (MyShape,MyShape,MyShape) returns
integer.
**
** Parameters:
**
** Return value:
**
**  mi_integer
**
** History:
**
**  06/26/1998 - Generated by BladeSmith Version 3.60.626  .
**
** Identification:
**
** NOTE:
**
**  BladeSmith will add and remove parameters from the function
**  prototype, and will generate tracing calls.  ONLY EDIT code
**  in blocks marked  Your_<section>.  Any other  modifications
**  will require manual merging.
**
*************************************************************
*/

UDREXPORT
mi_integer MyShapeUnion
(

MyShape *              Argument1,

MyShape *              Argument2,

MyShape *              Argument3,
MI_FPARAM *            Gen_fparam        /* Standard info - see DBDK docs.*/
)

{
    MI_CONNECTION * Gen_Con;          /* The connection handle.             */
    mi_integer      Gen_RetVal;       /* The return value.                  */
    /* ------ {{Your_Declarations (PreserveSection) BEGIN ------ */
    ShapeHdr* h1 = (ShapeHdr*) mi_get_vardata ( (mi_lvarchar*) Argument1 );
```

```
ShapeHdr* h2 = (ShapeHdr*) mi_get_vardata ( (mi_lvarchar*) Argument2 );
ShapeHdr* h3 = (ShapeHdr*) mi_get_vardata ( (mi_lvarchar*) Argument3 );

/* ------ }}Your_Declarations (#0000) END ------ */

/* Use the NULL connection. */
Gen_Con = NULL;

/* ------ {{Your_Code (PreserveSection) BEGIN ------ */
if ( h1 == h2 )
{
    /* This is a 'self-union', which is how the rtree determines how
     * big your header structure is.
     */
    h3->tag   = HeaderT;
    h3->flags = 0;
    h3->xmin  = h1->xmin;
    h3->ymin  = h1->ymin;
    h3->xmax  = h1->xmax;
    h3->ymax  = h1->ymax;
}
else
{
    /*
     * Caution!  h1 and h3 may both reference the same structure!
     * Likewise, h2 and h3 may both reference the same structure!
     * This is because the Rtree reuses variables to save memory.
     * This means we have to be careful to save the union of h1 and h2
     * in a temporary structure as we compute it, otherwise we will
     * prematurely wipe out h1 or h2 as we save the union in h3.
     */
    ShapeHdr htemp;
    htemp.xmin = (h1->xmin < h2->xmin) ? h1->xmin : h2->xmin;
    htemp.ymin = (h1->ymin < h2->ymin) ? h1->ymin : h2->ymin;
    htemp.xmax = (h1->xmax > h2->xmax) ? h1->xmax : h2->xmax;
    htemp.ymax = (h1->ymax > h2->ymax) ? h1->ymax : h2->ymax;
    h3->tag   = HeaderT;
    h3->flags = 0;
    h3->xmin  = htemp.xmin;
    h3->ymin  = htemp.ymin;
    h3->xmax  = htemp.xmax;
    h3->ymax  = htemp.ymax;
}

/*
 * Theoretically you should only have to do this for the self-union
 * case, since after that the Rtree should know how big each
 * element to be stored on internal node pages will be.  However,
 * early versions of the Rtree code did not always 'remember' the
 * size correctly, so we do it for every union call.  It's not a
 * very expensive function: it just sets a field in the mi_lvarchar
 * opaque data structure.
 */
mi_set_varlen ( (mi_lvarchar*) Argument3, sizeof(ShapeHdr) );
```

```
    Gen_RetVal = 1;
    /* ------ }}Your_Code (#NJLJ) END ------ */


    /* Return the function's return value. */
    return Gen_RetVal;
}
```

Size Support Function

```
/*****************************************************************
**
** Function name:
**
**  MyShapeSize
**
** Description:
**
** Special Comments:
**
**  Entrypoint for the SQL routine Size (MyShape,double precision) returns integer.
**
** Parameters:
**
** Return value:
**
**  mi_integer
**
** History:
**
**  06/26/1998 - Generated by BladeSmith Version 3.60.626  .
**
** Identification:
**
** NOTE:
**
**  BladeSmith will add and remove parameters from the function
**  prototype, and will generate tracing calls.  ONLY EDIT code
**  in blocks marked  Your_<section>.  Any other  modifications
**  will require manual merging.
**
*****************************************************************
*/

UDREXPORT
mi_integer MyShapeSize
(

MyShape *               Argument1,

mi_double_precision *   Argument2,
MI_FPARAM *             Gen_fparam        /* Standard info - see DBDK docs.*/
)

{
    MI_CONNECTION * Gen_Con;          /* The connection handle.             */
    mi_integer      Gen_RetVal;       /* The return value.                  */
    /* ------ {{Your_Declarations (PreserveSection) BEGIN ------ */
    ShapeHdr* hdr = (ShapeHdr*) mi_get_vardata ((mi_lvarchar*) Argument1);

    /* ------ }}Your_Declarations (#0000) END ------ */
    /* Use the NULL connection. */
    Gen_Con = NULL;

    /* ------ {{Your_Code (PreserveSection) BEGIN ------ */
    if ( hdr->flags == 1 )
    {
        /*
         * Since flag is set, this Size() call follows an Inter() call
         * in which the two objects did not intersect.  Return zero
         * to tell the Rtree that this is the case.
         */
        *Argument2 = 0.0;
    }
    else
    {
        *Argument2 = (mi_double) (hdr->xmax - hdr->xmin) *
                          (hdr->ymax - hdr->ymin);
        /*
         * Guard against returning zero.  Since we are using an area method
         * to compute size, this could happen for a point, or a horizontal
         * or vertical line segment (box with zero height/width).
         */
        if (*Argument2 < 0.000001)
        *Argument2 = 0.000001;
    }

    Gen_RetVal = 1;
    /* ------ }}Your_Code (#ED32) END ------ */


    /* Return the function's return value. */
    return Gen_RetVal;
}
```

Inter Support Function

```
/*****************************************************************
**
** Function name:
**
**  MyShapeInter
**
** Description:
**
** Special Comments:
**
**  Entrypoint for the SQL routine Inter (MyShape,MyShape,MyShape) returns
integer.
**
** Parameters:
**
** Return value:
**
**  mi_integer
**
** History:
**
**  06/26/1998 - Generated by BladeSmith Version 3.60.626  .
**
** Identification:
**
** NOTE:
**
**  BladeSmith will add and remove parameters from the function
**  prototype, and will generate tracing calls.  ONLY EDIT code
**  in blocks marked  Your_<section>.  Any other  modifications
**  will require manual merging.
**
*****************************************************************
*/

UDREXPORT
mi_integer MyShapeInter
(

MyShape *               Argument1,

MyShape *               Argument2,

MyShape *               Argument3,
MI_FPARAM *             Gen_fparam        /* Standard info - see DBDK docs.*/
)

{
    MI_CONNECTION * Gen_Con;          /* The connection handle.             */
    mi_integer      Gen_RetVal;       /* The return value.                  */
    /* ------ {{Your_Declarations (PreserveSection) BEGIN ------ */
    ShapeHdr* h1 = (ShapeHdr*) mi_get_vardata ( (mi_lvarchar*) Argument1 );
    ShapeHdr* h2 = (ShapeHdr*) mi_get_vardata ( (mi_lvarchar*) Argument2 );
    ShapeHdr* h3 = (ShapeHdr*) mi_get_vardata ( (mi_lvarchar*) Argument3 );

    /* ------ }}Your_Declarations (#0000) END ------ */

    /* Use the NULL connection. */
    Gen_Con = NULL;

    /* ------ {{Your_Code (PreserveSection) BEGIN ------ */
    if (!((h1->xmin <= h2->xmax) &&
          (h1->xmax >= h2->xmin) &&
          (h1->ymin <= h2->ymax) &&
          (h1->ymax >= h2->ymin)))
    {
        /*
         * Bounding boxes of the two shapes do not intersect.
         * Set the flag word to indicate this.
         */
            h3->tag   = HeaderT;
        h3->flags = 1;
        h3->xmin = h3->ymin = h3->xmax  = h3->ymax = 0;
        Gen_RetVal = 0;
    }
    else
    {
        /*
         * Bounding boxes of the two shapes do intersect.
         * Clear the flag word, and compute the bbox intersection.
         * Note that even if h3==h1, we aren't prematurely wiping
         * out any data.  (Compare with Union() function.)
         */
            h3->tag   = HeaderT;
        h3->flags = 0;
        h3->xmin = (h1->xmin > h2->xmin) ? h1->xmin : h2->xmin;
        h3->ymin = (h1->ymin > h2->ymin) ? h1->ymin : h2->ymin;
        h3->xmax = (h1->xmax < h2->xmax) ? h1->xmax : h2->xmax;
        h3->ymax = (h1->ymax < h2->ymax) ? h1->ymax : h2->ymax;
        Gen_RetVal = 1;
    }
    mi_set_varlen ( (mi_lvarchar*) Argument3, sizeof(ShapeHdr) );

    /* ------ }}Your_Code (#PJLJ) END ------ */


    /* Return the function's return value. */
    return Gen_RetVal;
}
```

Shapes.h Header File

```
/*
** Title:          Shapes
** SCCSid:         %W% %E% %U%
** CCid:           %W% %E% %U%
** Author:         Informix Software Inc.
** Created:        06/26/1998 13:27
** Description:    This is the generated 'C' file for the Shapes DataBlade.
** Comments:       Generated for project Shapes.3.6
*/


/*
**  Special Note: This file should not be  modified.
**  No merging is performed on this header file.  It
**  is regenerated each time the project is written.
*/

#ifndef HDR_Shapes_H
#define HDR_Shapes_H

/*
**  Configure tracing by setting TRACE_DEBUG_Shapes
**  to 0 to completely disable tracing or 1 to  enable
**  tracing.  This define can be set from the compiler
**  command line by using the -DTRACE_DEBUG_Shapes=0 flag.
*/
#ifndef TRACE_DEBUG_Shapes
#define TRACE_DEBUG_Shapes 1
#endif

#ifndef DBDK_LOHSIZE
#define DBDK_LOHSIZEsizeof(MI_LO_HANDLE)
#define DBDK_LOBINFNSIZEDBDK_LOHSIZE
#endif

/*
**  Large object file name mask.  '?' is a wild-card that
**  is filled in when a large object is written to disk.
*/
#define LO_FN_MASK"????????.lo"

/*  Error messages
**
**  English  versions  of  these error messages  are  automatically
**  added to the  syserrors  table as part of your DataBlade module
**  registration.  If you do  not  like the  default  messages, you
**  can create new errors  and  change  these  defines to use  your
**  new codes. You can not, however, change the text of the default
**  messages because they are shared by other DataBlade modules.
*/
#define ERRORMESG1"UGEN1"
#define ERRORMESG2"UGEN2"
```

```
#define ERRORMESG3"UGEN3"
#define ERRORMESG4"UGEN4"
#define ERRORMESG5"UGEN5"
#define ERRORMESG6"UGEN6"
#define ERRORMESG7"UGEN7"
#define ERRORMESG8"UGEN8"
#define ERRORMESG9"UGEN9"
#define ERRORMESG10"UGENA"
#define ERRORMESG11"UGENB"
#define ERRORMESG12"UGENC"
#define ERRORMESG13"UGEND"
#define ERRORMESG14"UGENE"
#define ERRORMESG15"UGENF"
#define ERRORMESG16"UGENG"
#define ERRORMESG17"UGENH"
#define ERRORMESG18"UGENI"
#define ERRORMESG19"UGENJ"

/* Use DBDK_TRACE to direct trace messages to the trace file. */
#if TRACE_DEBUG_Shapes
#define DBDK_TRACE(1 << 16)
#else
#define DBDK_TRACE0
#endif

/*
** Print a message to the trace file and for  the user.
** N.B.: This macro uses Gen_Con.  Your  function  must
**       declare Gen_Con as MI_CONNECTION * and  either
**       open the connection or set it to NULL.
*/
#define DBDK_TRACE_ERROR( Caller, ErrNo, ErrLevel )             \
                           Gen_Trace                            \
                           (                                    \
                               Gen_Con,                         \
                               Caller,                          \
                               __FILE__,                        \
                               __LINE__,                        \
                               ErrNo,                           \
                               "Shapes",                        \
                               ErrLevel,                        \
                               MI_SQL | DBDK_TRACE              \
                           );

/* Print a message to the trace file. */
#if TRACE_DEBUG_Shapes

/*
** Print a message to the trace file.
** N.B.: This macro uses Gen_Con.  Your  function  must
**       declare Gen_Con as MI_CONNECTION * and  either
**       open the connection or set it to NULL.
*/
#define DBDK_TRACE_MSG( Caller, ErrNo, ErrLevel )               \
                                Gen_Trace                        \
                                (                                \
                                    Gen_Con,                     \
                                    Caller,                      \
                                    __FILE__,                    \
                                    __LINE__,                    \
                                    ErrNo,                       \
                                    "Shapes",                    \
                                    ErrLevel,                    \
                                    DBDK_TRACE                   \
                                );

#else

#define DBDK_TRACE_MSG( Caller, ErrNo, ErrLevel )

#endif

/* These macros are used on entry to, and on exit from, a function. */
#define DBDK_TRACE_ENTER( Caller )  DBDK_TRACE_MSG( Caller, ERRORMESG13, 20 )
#define DBDK_TRACE_EXIT( Caller )   DBDK_TRACE_MSG( Caller, ERRORMESG14, 20 )

/*
**  Interval types.
*/
#define YEAR_TO_MONTH   1
#define DAY_TO_SECOND   2

/*
**  UDREXPORT is normally used to export a function from the DataBlade when
**  linking on NT.  UNIX source files should maintain this define in source
**  for use when porting back to NT.
*/
#ifndef UDREXPORT
#define UDREXPORT
#endif

/* Function prototypes. */
mi_integer Gen_nstrwords( gl_mchar_t *, mi_integer );
gl_mchar_t * Gen_sscanf
(
    MI_CONNECTION *      Gen_Con,
    char *               Gen_Caller,
    gl_mchar_t *         Gen_InData,
    mi_integer           Gen_InDataLen,
    mi_integer           Gen_Width,
    char *               Gen_Format,
    char *               Gen_Result
);
void       Gen_LoadLOFromFile
(
    MI_CONNECTION *      Gen_Con,
    char *               Gen_Caller,
    char *               Gen_LOFile,
    MI_LO_HANDLE *      Gen_pLOh
);
void Gen_StoreLOToFile
(
    MI_CONNECTION *     Gen_Con,
    char *              Gen_Caller,
    char *              Gen_LOFile,
    MI_LO_HANDLE *      Gen_pLOh
);
void Gen_Trace
(
    MI_CONNECTION *     Gen_Con,
    char *              Gen_Caller,
    char *              Gen_FileName,
    mi_integer          Gen_LineNo,
    char *              Gen_MsgNo,
    char *              Gen_Class,
    mi_integer          Gen_Threshold,
    mi_integer          Gen_MsgType
);


/* {{FUNCTION(11190130-cf56-11d1-9ce6-080070e6b366) (CopySection) */

/* Do not modify. */


/*
** BladeSmith 3.60.630   typedef MyShape
** This UDT is implemented in this DataBlade in the 'C' language.
*/
typedef struct
{
mi_char                    data[1];
}
MyShape;
/* Warning: Do not modify. MyShape checksum: 0 */


/* Do not modify. */


/*
** BladeSmith 3.60.630   typedef MyPoint
** This UDT is implemented in this DataBlade in the 'C' language.
*/
typedef struct
{
mi_char                    data[1];
}
MyPoint;
/* Warning: Do not modify. MyPoint checksum: 0 */
/* Do not modify. */


/*
** BladeSmith 3.60.630   typedef MyCircle
** This UDT is implemented in this DataBlade in the 'C' language.
*/
typedef struct
{
mi_char                    data[1];
}
MyCircle;
/* Warning: Do not modify. MyCircle checksum: 0 */


/* Do not modify. */


/*
** BladeSmith 3.60.630   typedef MyBox
** This UDT is implemented in this DataBlade in the 'C' language.
*/
typedef struct
{
mi_char                    data[1];
}
MyBox;
/* Warning: Do not modify. MyBox checksum: 0 */

/* }}FUNCTION (#8F9U) */

#endif
```

ShapeTypes.h Header File

```
#include <mi.h>

typedef mi_double_precision mi_double;

typedef enum
{
    PointT   = 1,
    CircleT  = 2,
    BoxT     = 3,
    HeaderT  = 4
}
ShapeTypeEnum;

typedef struct
{
    ShapeTypeEnum tag;          /* type of this object           (4 bytes) */
    mi_integer    flags;        /*                               (4 bytes) */
    mi_double     xmin, ymin;
    mi_double     xmax, ymax;
}
ShapeHdr;

/*
 *  MyShape is the structure which describes that data; it is
 *  accessible via an mi_get_vardata() call.
 */
typedef struct
{
    ShapeHdr hdr;
    mi_char    data[8];      /* start of multirep data       (8+ bytes) */
}
Shape;

typedef struct
{
    mi_double  x;
    mi_double  y;
}
PointData;


typedef struct
{
    PointData  ll;           /* coordinates of lower left corner */
    PointData  ur;           /* coordinates of upper right corner */
}
BoxData;

typedef struct
{
    PointData  c;            /* center */
    mi_double    r;            /* radius */
```

```
}
CircleData;

#ifdef __cplusplus
extern "C"
{
#endif

typedef mi_boolean (*operatorFunction)   (Shape*, Shape*);

typedef operatorFunction* functionTable;


/* Function prototypes */
mi_boolean PointXPoint   (Shape*  obj1, Shape*  obj2);
mi_boolean PointXCircle  (Shape*  obj1, Shape*  obj2);
mi_boolean PointXBox     (Shape*  obj1, Shape*  obj2);
mi_boolean CircleXCircle (Shape*  obj1, Shape*  obj2);
mi_boolean CircleICircle (Shape*  obj1, Shape*  obj2);
mi_boolean CircleIBox    (Shape*  obj1, Shape*  obj2);
mi_boolean CircleXBox    (Shape*  obj1, Shape*  obj2);
mi_boolean BoxXBox       (Shape*  obj1, Shape*  obj2);
mi_boolean BoxICircle    (Shape*  obj1, Shape*  obj2);
mi_boolean BoxIBox       (Shape*  obj1, Shape*  obj2);

mi_boolean PointWPoint   (Shape*  obj1,
             Shape*  obj2,
                          mi_double dist);

mi_boolean PointWBox     (Shape*  obj1,
             Shape*  obj2,
                          mi_double dist);

mi_boolean Dispatch      (functionTable tab,
                          mi_boolean    commutative,
                          Shape*      obj1,
                          Shape*      obj2,
                          MI_FPARAM*    fp);

#ifdef __cplusplus
}
#endif

static operatorFunction intersectTable[] = {
                    /* PointT   = 1                */
    PointXPoint,    /*               PointT   = 1 */
    PointXCircle,   /*               CircleT  = 2 */
    PointXBox,      /*               BoxT     = 3 */

    NULL,           /* CircleT  = 2                */
    CircleXCircle,  /*               CircleT  = 2 */
    CircleXBox,     /*               BoxT     = 3 */

    NULL,NULL,      /* BoxT     = 3                */
    BoxXBox         /*               BoxT     = 3 */

};


static operatorFunction insideTable[] = {
                    /* PointT   = 1               */
    NULL,           /*               PointT   = 1 */
    PointXCircle,   /*               CircleT  = 2 */
    PointXBox,      /*               BoxT     = 3 */
                    /* CircleT  = 2               */
    NULL,           /*               PointT   = 1 */
    CircleICircle,  /*               CircleT  = 2 */
    CircleIBox,     /*               BoxT     = 3 */
                    /* BoxT     = 3               */
    NULL,           /*               PointT   = 1 */
    BoxICircle,     /*               CircleT  = 2 */
    BoxIBox         /*               BoxT     = 3 */
};
```

Glossary

access method	A set of server routines that Informix Dynamic Server with Universal Data Option uses to store and access the data in an index or a table. B-tree is the default secondary access method. Some DataBlade modules have their own access methods, with routines defined by the module. See also *primary access method, secondary access method.*
bounding box	A rectangular shape that completely contains the bounded object or objects. Bounding boxes are usually stored as a set of coordinates of the same dimensionality as the bounded object or objects.
branch page	A location on a tree structure that has at least one page below and one page above it. In an R-tree index, branch pages are located in the intermediate levels, between the root page and leaf pages.
B-tree index	A type of index that uses a balanced tree structure for efficient record retrieval. B-tree indexes store key data in ascending or descending order. A B-tree index is balanced when the leaf pages are all at the same level from the root page. The goal is to keep equal numbers of items on each side of each page to minimize the path from the root to any leaf page. As items are inserted and deleted, the B-tree is restructured to keep the pages balanced and the search paths uniform.
commutator function	A Boolean function **g()** is a commutator of a Boolean function **f()** if **g(a,b) = f(b,a)** for all possible values of **a** and **b**.

	For example, **Inside(A,B)** is the same as **Contains(B,A)**, which means the **Inside** function is the commutator of the **Contains** function, and vice versa.
concurrency	The ability of two or more processes to access the same database simultaneously.
data object	The data that is stored in an R-tree indexed column of a table and in the R-tree index itself.
dbspace	A logical collection of one or more chunks of contiguous disk space within which you store databases and tables. Because chunks represent specific regions of disk space, the creators of databases and tables can control where their data is physically located by placing databases or tables in specific dbspaces. Large objects are stored in sbspaces.
	See also *sbspace*.
expression based fragmentation	A method of splitting a table or index into fragments in which the result of an expression determines in which fragment a row will reside. For example, suppose you have a table with an **id** column. You can fragment an index on this table such that all entries for rows with **id** less than 5000 are stored in one fragment and the remainder in another fragment.
functional index	An index that stores the result of executing a specified function on a table column.
INFORMIXDIR	The UNIX or Windows NT environment variable that specifies the directory in which the database server is installed.
interface	In the DataBlade Developers Kit, an object that represents a set of data types and routines in one DataBlade module that can be used by other modules. BladeManager ensures that the originating module is registered before a module that requires the interface.
key	A unique identifier. A key is a column or combination of columns whose value is unique for each row. Among the various keys available are primary keys and foreign keys.
leaf page	A location on a tree structure that has at least one page above it and no pages below it. In an R-tree index, leaf pages are located in the final levels and contain data objects and rowids.

locking	The process of temporarily limiting access to an object (database, table, page, or row) to prevent conflicting interactions among concurrent processes. Locking helps ensure data integrity. The database server guarantees that, as long as the data is locked, no other program can modify it.
multi-representational data type	A data type whose storage location depends on the size of the column value. If the column value is smaller than a predetermined size, the value is stored in-row. If the column value is larger, it is stored in a smart large object.
operator class	The set of operators and functions that Informix Dynamic Server with Universal Data Option associates with a secondary access method. When an index is created, it is associated with a particular operator class.
primary access method	A set of routines that perform table operations such as inserting, deleting, updating, and searching data. Informix Dynamic Server with Universal Data Option provides a virtual table interface (VTI), with which advanced users can create primary access methods for virtual tables.
project file	A file created and used by BladeSmith, part of the DataBlade Developers Kit, to manage a DataBlade module project.
purpose function	One of a set of functions that an access method uses to create an index, search an index, drop an index, as well as insert entries into an index, delete from an index, and so on.
query optimizer	A server facility that estimates the most efficient plan for executing a query in the database server. The optimizer considers the CPU cost and the I/O cost of executing a plan.
registration	Loading a DataBlade module's objects into a database. Registration makes a DataBlade module available for use by client applications that open that database.
root page	The top-most level in a tree structure. In an R-tree index, the root page may have zero or more branch pages or leaf pages below it, depending on the size of the R-tree index.
root dbspace	The dbspace containing databases, tables, logical logs, reserved pages, and internal tables that describe all storage spaces on the database server. The root dbspace is the default location for temporary tables. See also *dbspace*.

round-robin fragmentation	A method of splitting a table or index into fragments in which the database server balances the number of rows in each fragment. As more rows are inserted, the database server decides in which fragment they should reside.
routine signature	The information that the database server uses to identify a routine. The signature of a routine includes the type of the routine (function or procedure), the routine name, the number of parameters, the data types of the parameters, and the order of the parameters. In an ANSI-compliant database, the name of the routine is specified as **owner.name**.
rowid	An integer that defines the physical location of a row. The database server assigns a unique rowid to each row in a nonfragmented table. If you want to access data in a fragmented table by rowid, you must create a rowid column.
R-tree index	A type of index that uses a tree structure based on overlapping bounding rectangles to speed access to spatial and multidimensional data types.
sbspace	A logical storage area that contains one or more chunks that store only smart large object data.
search object	The object specified in the WHERE clause of a query that is used to search for objects in the database. For example, if you specify `Contains(points, 'pnt(10,20)')` in the WHERE clause of a query, `pnt(10,20)` is the search object.
secondary access method	A set of server functions that build, access, and manipulate an index structure, as opposed to a base table: for example, a B-tree, an R-tree, or any other index structure provided by a DataBlade module. Typically, a secondary access method speeds up the retrieval of data. When an SQL query uses an index created using a secondary access method, it accesses the index using the set of functions in an operator class belonging to that access method. See also *operator class.*
selectivity	Of the total number of rows in a table, the fraction that is returned by a query. The more selective the query, the smaller the fraction.
signature	See *routine signature.*
strategy functions	The functions for which the optimizer can use an index scan in a query. These functions are specified (by name only or by their full signature) by the operator class with which the index is created.

support functions	The functions used internally by an access method to build and maintain an index structure. They are not available for use in SQL queries.
system catalog	A group of database tables that contain information about the database itself, such as the names of tables or columns in the database, the number of rows in a table, the information about indexes and database privileges, and so on.
user-defined data type	A data type written in one of the languages that Informix Dynamic Server with Universal Data Option supports. The data type is not provided as part of the Informix Dynamic Server with Universal Data Option but must be registered separately, usually via a DataBlade module.
user-defined routine	A routine, written in one of the languages that Informix Dynamic Server with Universal Data Option supports, that provides added functionality for data types or encapsulates application logic.

Index