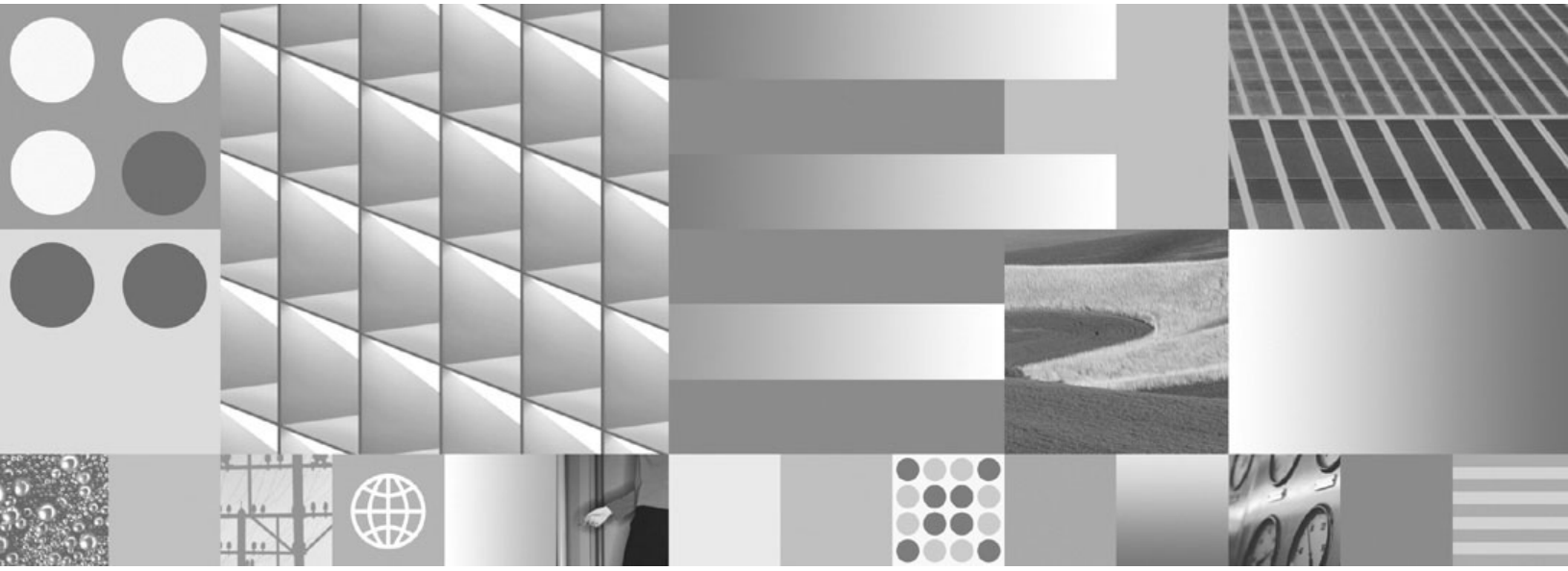
IBM Informix

Version 11.50

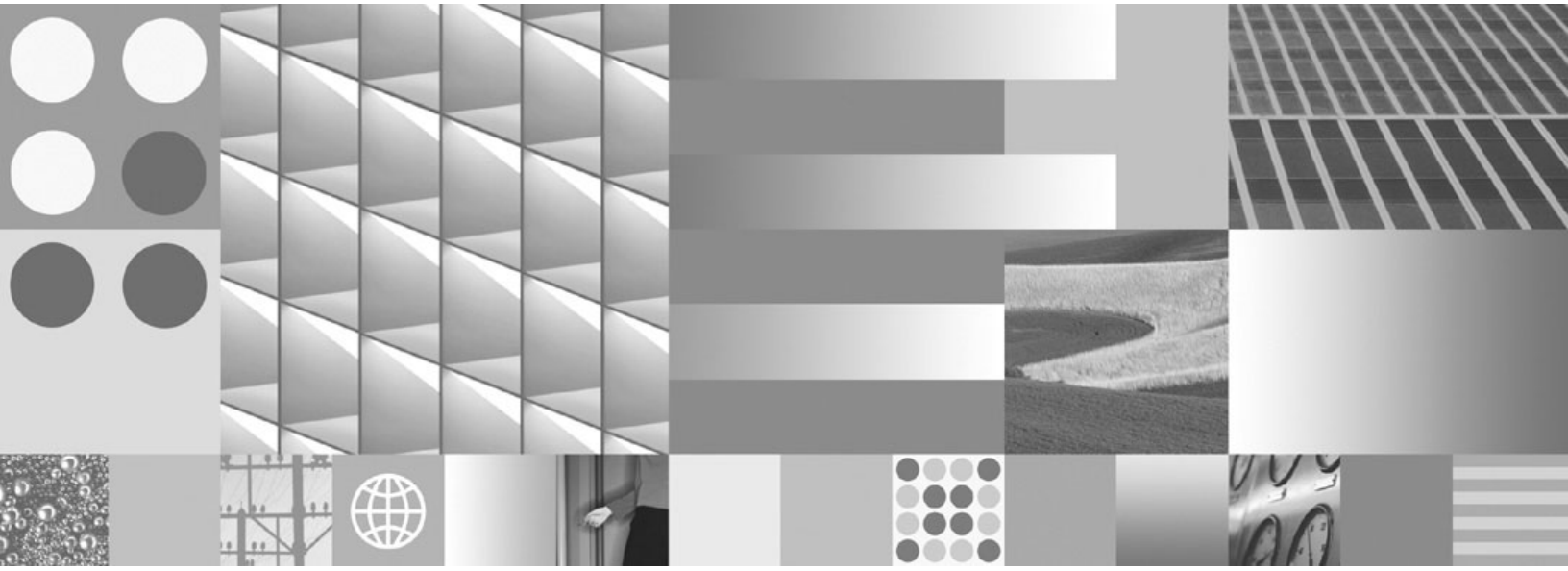


IBM Informix R-Tree Index User's Guide

SC23-9436-00

IBM Informix

Version 11.50



IBM Informix R-Tree Index User's Guide

SC23-9436-00

Note:
Before using this information and the product it supports, read the information in "Notices" on page C-1.

Contents

Introduction

In This Introduction

This introduction provides an overview of the information in this publication and describes the conventions it uses.

About This Publication

This publication 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, how to create a new operator class that uses the R-tree access method to index a user-defined data type, and how to manage databases that use the R-tree secondary access method.

Types of Users

This publication is written for three distinct audiences:

- Application developers
- DataBlade® module developers
- Database administrators

The following table 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.

Chapter	Audience
Chapter 1, "R-Tree Secondary Access Method Concepts," on page 1-1	All users who want in-depth knowledge of how R-tree indexes work
Chapter 2, "Using the R-Tree Secondary Access Method," on page 2-1	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 3, "Developing DataBlade Modules That Use the R-Tree Secondary Access Method," on page 3-1	DataBlade module developers who want to use the R-tree access method to index new data types by creating a new operator class
Chapter 4, "Managing Databases That Use the R-Tree Secondary Access Method," on page 4-1	Database administrators who manage databases that contain R-tree indexes

This publication is written with the assumption that you have the following background:

- A working knowledge of your computer, your operating system, and the utilities that your operating system provides
- Some experience working with relational databases or exposure to database concepts
- Some experience with computer programming
- Some experience with database server administration, operating-system administration, or network administration

If you have limited experience with relational databases, SQL, or your operating system, refer to the *IBM Informix Dynamic Server Getting Started Guide* for your database server for a list of supplementary titles.

Software Dependencies

This publication assumes you are using IBM Informix Dynamic Server (IDS), Version 11.50. You must also have the following IBM Informix software to use the R-tree secondary access method:

- The IBM Informix R-Tree Secondary Access Method DataBlade module)
- A DataBlade module that uses the Informix R-tree secondary access method, such as the IBM Informix Geodetic DataBlade Module

You can use the Informix DataBlade Developers Kit to develop a DataBlade module that uses the R-tree secondary access method.

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

- DB–Access
- IBM Informix ESQL/C
- DataBlade API

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

Assumptions About Your Locale

IBM Informix products can support many languages, cultures, and code sets. All the information related to character set, collation, and representation of numeric data, currency, date, and time is brought together in a single environment, called a Global Language Support (GLS) locale.

The examples in this publication are written with the assumption that you are using the default locale, **en_us.8859-1**. This locale supports U.S. English format conventions for date, time, and currency. In addition, this locale supports the ISO 8859-1 code set, which includes the ASCII code set plus many 8-bit characters such as é, è, and ñ.

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

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

Demonstration Database

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

- The **stores_demo** database illustrates a relational schema with information about a fictitious wholesale sporting-goods distributor. Many examples in IBM Informix manuals are based on the **stores_demo** database.
- The **superstores_demo** database illustrates an object-relational schema. The **superstores_demo** database contains examples of extended data types, type and table inheritance, and user-defined routines.

For information about how to create and populate the demonstration databases, see the *IBM Informix DB–Access User's Guide*. For descriptions of the databases and their contents, see the *IBM Informix Guide to SQL: Reference*.

Documentation Conventions

This section describes the following conventions, which are used in the product documentation for IBM® Informix Dynamic Server:

- Typographical conventions
- Feature, product, and platform conventions
- Example code conventions

Typographical Conventions

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

Convention	Meaning
KEYWORD	Keywords of SQL, SPL, and some other programming languages appear in uppercase letters in a serif font.
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	Names of program entities (such as classes, events, and tables), environment variables, file names, path names, and interface elements (such as icons, menu items, and buttons) appear in boldface.
`monospace`	Information that the product displays and information that you enter appear in a monospace typeface.
`KEYSTROKE`	Keys that you are to press appear in uppercase letters in a sans serif font.
>	This symbol indicates a menu item. For example, "Choose **Tools > Options**" means choose the **Options** item from the **Tools** menu.

Feature, Product, and Platform Markup

Feature, product, and platform markup identifies paragraphs that contain feature-specific, product-specific, or platform-specific information. Some examples

of this markup follow:

Dynamic Server

Identifies information that is specific to IBM Informix Dynamic Server

End of Dynamic Server

Windows Only

Identifies information that is specific to the Windows operating system

End of Windows Only

This markup can apply to one or more paragraphs within a section. When an entire section applies to a particular product or platform, this is noted as part of the heading text, for example:

Table Sorting (Windows)

Example Code Conventions

Examples of SQL code occur throughout this publication. Except as noted, the code is not specific to any single IBM Informix application development tool.

If only SQL statements are listed in the example, they are not delimited by semicolons. For instance, you might see the code in the following example:

```
CONNECT TO stores_demo
...

DELETE FROM customer
   WHERE customer_num = 121
...

COMMIT WORK
DISCONNECT CURRENT
```

To use this SQL code for a specific product, you must apply the syntax rules for that product. For example, if you are using an SQL API, you must use EXEC SQL at the start of each statement and a semicolon (or other appropriate delimiter) at the end of the statement. If you are using DB–Access, you must delimit multiple statements with semicolons.

Tip: Ellipsis points in a code example indicate that more code would be added in a full application, but it is not necessary to show it to describe the concept being discussed.

For detailed directions on using SQL statements for a particular application development tool or SQL API, see the documentation for your product.

Additional Documentation

You can view, search, and print all of the product documentation from the IBM Informix Dynamic Server information center on the Web at http://publib.boulder.ibm.com/infocenter/idshelp/v115/index.jsp.

For additional documentation about IBM Informix Dynamic Server and related products, including release notes, machine notes, and documentation notes, go to the online product library page at http://www.ibm.com/software/data/informix/

pubs/library/. Alternatively, you can access or install the product documentation from the Quick Start CD that is shipped with the product.

Compliance with Industry Standards

The American National Standards Institute (ANSI) and the International Organization of Standardization (ISO) have jointly established a set of industry standards for the Structured Query Language (SQL). IBM Informix SQL-based products are fully compliant with SQL-92 Entry Level (published as ANSI X3.135-1992), which is identical to ISO 9075:1992. In addition, many features of IBM Informix database servers comply with the SQL-92 Intermediate and Full Level and X/Open SQL Common Applications Environment (CAE) standards.

How to Provide Documentation Feedback

You are encouraged to send your comments about IBM Informix user documentation by using one of the following methods:

- Send e-mail to docinf@us.ibm.com.
- Go to the Information Center at http://publib.boulder.ibm.com/infocenter/idshelp/v115/index.jsp and open the topic that you want to comment on. Click **Feedback** at the bottom of the page, fill out the form, and submit your feedback.

Feedback from both methods is monitored by those who maintain the user documentation of Dynamic Server. The feedback methods are reserved for reporting errors and omissions in our documentation. For immediate help with a technical problem, contact IBM Technical Support. For instructions, see the IBM Informix Technical Support Web site at http://www.ibm.com/planetwide/.

We appreciate your suggestions.

Chapter 1. R-Tree Secondary Access Method Concepts

In This Chapter

This chapter provides a detailed discussion of the R-tree secondary access method and an in-depth discussion about how R-tree indexes work. It includes the following topics:

- About Access Methods
- The R-Tree Secondary Access Method
- About Operator Classes
- R-Tree Functionality That IBM Provides
- IBM Informix DataBlade Modules That Use the R-Tree Access Method

DataBlade module developers can use the Informix DataBlade Developers Kit to develop the objects that form the DataBlade module that uses the R-tree access method. The Informix DataBlade Developers Kit automatically generates most of the SQL commands and some of the 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 Shapes3 sample DataBlade module, described in Appendix A, to illustrate how to use the R-tree access method and how to create DataBlade modules that implement the R-tree access method.

About Access Methods

An *access method* is a set of database server routines that IBM Informix Dynamic Server (IDS) uses to access and manipulate a table or an index. The two types of access methods are primary and secondary.

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

Dynamic Server 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 1-10.

Dynamic Server provides two secondary access methods:

- B-tree, which stands for *balanced tree*. B-tree is the default secondary 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 your *IBM Informix Administrator's Guide*.

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 types of data:

- Multidimensional data, such as:
  - Spatial data in two or three dimensions

    An extra dimension that represents time could also be included.
  - Combinations of numerical values treated as multidimensional values, such as a configuration for a house that includes the number of stories, the number of bedrooms, the number of baths, the age of the house, and the 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 how to create a new data type, refer to "Designing a User-Defined Data Type" on page 3-3.

The R-tree access method is implemented internally using the Virtual-Index Interface, a mechanism provided with Dynamic Server so you can create new secondary access methods.

The purpose of a spatial index, such as R-tree, is to produce, during query processing, 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.

An R-tree index is inexact, but it is inclusive. This means that a search that uses the R-tree index 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 indexes are dynamic. This means that an R-tree index maintains itself during updates, inserts, and deletes of the indexed table. In addition, you do not need to know anything about the amount of data or the range of values in the column to be indexed before you create an R-tree index.

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 rectilinear 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 that is 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.

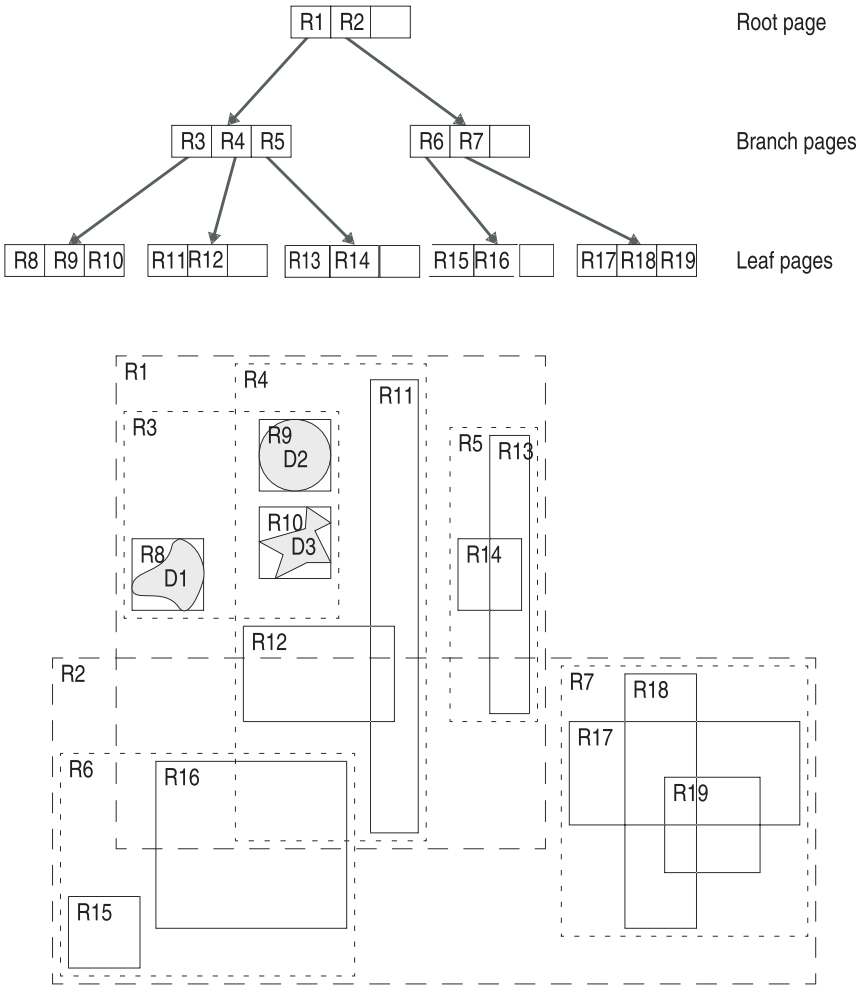


Figure 1-1. R-Tree Index Structure

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.

The R-tree access method evaluates the index entries (data objects and bounding boxes) as opaque objects (strings of bytes). The R-tree access method uses the support and strategy functions to interpret these objects.

Bounding-Box-Only R-Tree Indexes

For R-tree indexes created with Version 9.21 or later of the database server, if the datablade module you are working with is appropriately set up, default R-tree indexes no longer store a copy of the data object in leaf pages. Instead, the leaf pages store bounding box representations of the data object. This type of R-tree index is called a *bounding-box-only* R-tree index.

Important: The Informix Geodetic and Spatial DataBlade modules support bounding-box-only R-tree indexes. To set up your own DataBlade module to support these indexes, you must implement the **RtreeInfo** function with the operation **strat_func_substitutions**. You might also need to redesign your strategy functions that occupy slots 5 and up, if you want them to behave differently at non-leaf pages. This is because you cannot distinguish between leaf and non-leaf items in a bounding-box-only index. For more information, see "The RtreeInfo Function" on page 3-14.

Important: Only R-tree indexes created using Version 9.21 or later of the database server can be bounding-box-only R-tree indexes. R-tree indexes created in Version 9.20 and earlier versions of the database server continue to store copies of data objects in leaf pages.

The advantages of bounding-box-only R-tree indexes are the following:

- The R-tree index is significantly smaller, saving both disk space and the time to build and maintain the index.
- Bottom-up build performance is improved, because memory and temporary dbspace usage are reduced.
- The log space needed to update the index is reduced.

You might want to override this behavior if your table contains other large columns in addition to the column being indexed with the R-tree index. For more information about the BOUNDING_BOX_INDEX index parameter, see "R-Tree Index Parameters" on page 2-3.

Functional R-tree indexes are not bounding-box-only indexes; they store the data objects themselves in leaf pages.

R-tree indexes built in Version 9.20 of the database server continue to work correctly in Version 9.21. If, however, you build a new bounding-box-only R-tree index in Version 9.21 of the database server, this index will not work correctly if you revert to Version 9.20 of the database server.

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 bounding-box 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 the bounding boxes (described in "Bounding Boxes" on page 1-3) stored in an R-tree index

structure are related. The root page contains entries for bounding boxes R1 and R2. Together, these two bounding boxes enclose all the objects in the index.

Tip: Use the **rtreeRootBB()** function to return coordinates of the bounding box that enclose all objects in an R-tree index. For detailed instructions on how to use this function, refer to Chapter 4, "Managing Databases That Use the R-Tree Secondary Access Method," on page 4-1.

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 appears only once in the index that Figure 1-1 on page 1-4 shows, 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 keep the index small. If each object had to be replicated in several index pages, the size of the R-tree index would be larger than it needs to be.

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 *row* ID*)*

The size of an index entry in a leaf page is the size of the data object plus 20 bytes.

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 size of an index entry in a root or branch page is the size of the bounding box plus 12 bytes.

Each type of page in an R-tree index (leaf, branch, or root) also has an overhead of 20 bytes plus the size of the overall bounding box of the page.

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. The number of entries per page determines the branching factor of the tree. More entries per page, or a higher branching factor, means that fewer levels are needed for the same number of leaf pages as well as fewer leaf pages for a given number of base table keys. For any reasonable branching factor, almost all the space that an R-tree index needs is used by leaf pages.

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 create an R-tree index on the table column that contains the polygons, and then 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 "About Operator Classes" on page 1-10.

The R-tree secondary access method uses the bounding box of the search object to guide the search. 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, because 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 are skipped; this is where the R-tree access method saves search time and work. The access method uses the appropriate strategy function to test for overlap of bounding box entries in branch index pages.

When the search encounters a leaf page, it applies the appropriate strategy function to each key on the leaf page. 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.

Nearest-Neighbor Searching

The R-tree access method provides support for *nearest-neighbor* searches, that is, querying for objects in a spatial database that are closest to a specified object or location. Traditionally, without nearest-neighbor support, these kinds of searches are awkward to perform and involve several iterative stages.

To perform nearest-neighbor searches, the DataBlade module you are using must be set up for it. For example, the IBM Informix Geodetic DataBlade Module and the IBM Informix Spatial DataBlade Module both provide nearest-neighbor search support.

"Performing Nearest-Neighbor Searches" on page 2-11 explains how to perform nearest-neighbor searches using a DataBlade module that provides this feature.

"Setting Up Nearest-Neighbor Searching" on page 3-33 explains how to add nearest-neighbor support to a DataBlade module.

In this release, nearest-neighbor search is not supported with fragmented indexes.

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. The access method searches down the tree from the root page, looking for data objects whose bounding box best fits the new data object. Then it descends into that subtree, repeating the selection process at each internal page until it reaches a leaf page.

As the R-tree access method searches down the tree, it looks for bounding boxes that will be enlarged the least to accommodate the new data object. The access method might also use internal criteria other than the bounding box being enlarged by the smallest amount when 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 no space is left on the leaf page for the new data object, the leaf page is split into two pages. This means that a new page is allocated and the contents of the old page, plus the new data object, are divided between the old and the new pages. If the parent page is full, it might 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. Because 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. Notice that the total area of the new bounding boxes in the efficient split example is smaller than the bounding boxes in the inefficient example.

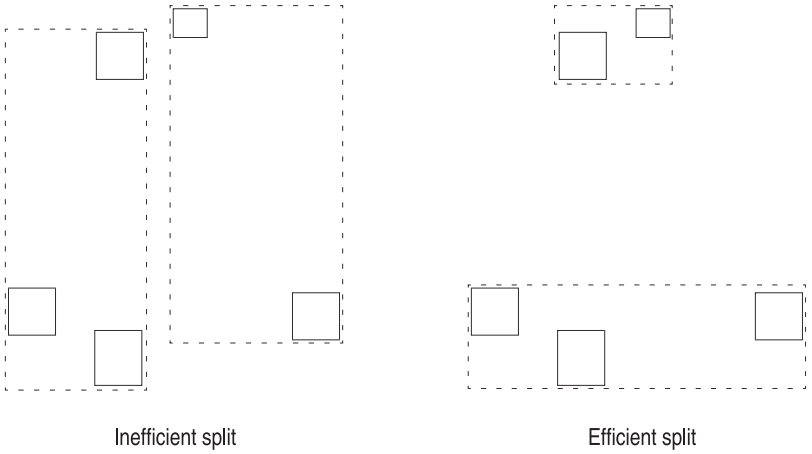


Figure 1-2. Comparison of Efficient and Inefficient Splits of Five Items Into Two Groups

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 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.

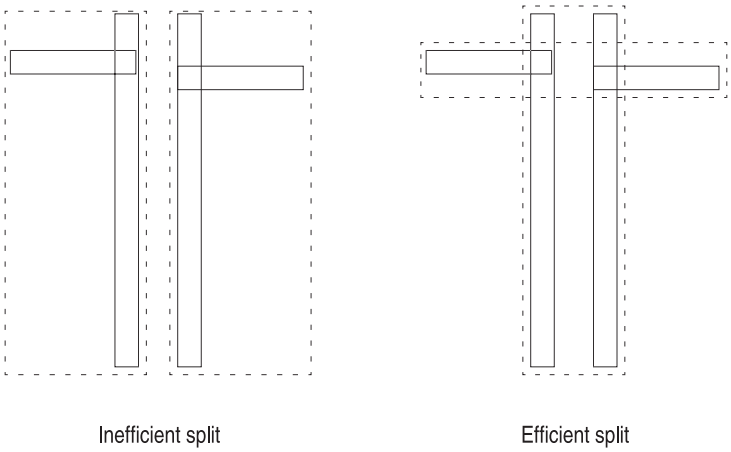


Figure 1-3. Comparison of a Split in Which the Resulting Pages Overlap (an Efficient Split) and Do Not Overlap (an Inefficient Split)

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 rightmost 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 keep the index structure consistent and to maintain the correct functioning of the index while it allows concurrent access and updates.
- 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 1-5.

  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.

About Operator Classes

Although an R-tree index might exist on a table column, it might not always be possible for the query optimizer to use it when you execute 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 likely be of 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. However, a query that searches for polygons that overlap a specified polygon will likely 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 defines a way to organize the functions that are implemented in a DataBlade module and defines how to make them known to the query optimizer and the access method. It identifies the functions that fill particular roles 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 support 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. The **rtree_ops** operator class has a fixed set of four strategy functions, and it cannot be extended. For this reason, and others described in Chapter 3, "Developing DataBlade Modules That Use the R-Tree Secondary Access Method," on page 3-1, 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 describes in detail how to create an operator class and how to set up the necessary strategy and support functions.

R-Tree Functionality That IBM Provides

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

- IBM Informix Dynamic Server (IDS)
- R-Tree Secondary Access Method DataBlade Module

The following sections describe the parts of the R-tree functionality that each product provides.

R-Tree Functionality in IBM Informix Dynamic Server

IBM Informix Dynamic Server (IDS) 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; they must be implemented outside the database server, usually as part of a DataBlade module. The **rtree_ops** operator class is intended to be used for generic R-tree testing. While you can reuse it, it is recommended 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 cannot 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 an 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.

To add new data types to an Informix database, you register 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 IBM Informix DataBlade modules that include new data types, support and strategy functions, and operator classes, refer to "IBM Informix DataBlade Modules That Use the R-Tree Access Method" on page 1-13.

If you are developing a new DataBlade module, read Chapter 3, "Developing DataBlade Modules That Use the R-Tree Secondary Access Method," on page 3-1. It describes in detail how to create the required strategy and support functions in order 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 IBM Informix R-Tree Secondary Access Method DataBlade module is automatically installed at the time you install Dynamic Server.

UNIX Only

On UNIX®, the IBM 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.

End of UNIX Only

Windows Only

On Windows, the IBM 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.

End of Windows Only

Contents of the DataBlade Module

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

- A list of error messages that the R-tree access method uses
- A BladeSmith interface object **ifxrltree1** that the DataBlade modules that depend on the IBM Informix R-Tree Secondary Access Method DataBlade module use

  For more information on how to use this interface object, refer to Chapter 3, "Developing DataBlade Modules That Use the R-Tree Secondary Access Method," on page 3-1.

As the preceding section describes, the R-tree access method itself is built into Dynamic Server. The error messages that the access method uses, however, are only available if the IBM 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 `RTRnn`, where:

- `RTR` is the three-character prefix for all IBM Informix R-Tree Secondary Access Method DataBlade module error codes.
- `nn` are two characters (0 to 9 or A to Z) that uniquely identify each error code.

DataBlade Module Registration

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

This registration normally occurs when you register a dependent DataBlade module, that is, one that can only be registered if the IBM 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 IBM Informix R-Tree Secondary Access Method DataBlade module. BladeManager then registers the IBM 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 the R-tree access method can index.

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

IBM Informix DataBlade Modules That Use the R-Tree Access Method

The following IBM Informix DataBlade modules use the R-tree access method:

- IBM 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, and the **Nearest** function to allow you to perform nearest-neighbor searches.

- IBM Informix Spatial DataBlade Module

  This DataBlade module also manages spatio-temporal data. It treats the earth as a flat map and uses planimetric (flat-plane) geometry. The Spatial DataBlade module is best used for regional data sets and applications.

  The DataBlade module creates a variety of data types, such as ST_LineString and ST_Polygon, as well as a variety of functions that operate on the data types, such as **ST_Distance** and **ST_Overlaps**. It also provides an operator class, called **ST_Geometry_Ops**, so you can create R-tree indexes on columns of spatial data types, and the **SE_Nearest** and **SE_NearestBBox** functions to allow you to perform *nearest-neighbor* searches.

- IBM 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 operate 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 the Shapes3 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 IBM Informix product, but is provided as an example of creating an operator class.

Chapter 2. Using the R-Tree Secondary Access Method

In This Chapter

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

- Before You Begin
- Creating R-Tree Indexes
- When Does the Query Optimizer Use an R-Tree Index?
- R-Tree Indexes and Null Values
- Performing Nearest-Neighbor Searches
- Database Isolation Levels and R-Tree Indexes
- Functional R-Tree Indexes

Before You Begin

You can create an R-tree index on a table after you complete 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 that use the R-tree access method are the IBM Informix Geodetic DataBlade Module and the IBM Informix Video Foundation DataBlade Module.

For more information on these modules, refer to "IBM Informix DataBlade Modules That Use the R-Tree Access Method" on page 1-13.

2. Create a database.
3. Register the IBM 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 IBM Informix R-Tree Secondary Access Method DataBlade module, you can skip this step, because BladeManager automatically prompts you to register the IBM Informix R-Tree Secondary Access Method DataBlade module when you register the DataBlade module described in step 1.

   For more information on the IBM Informix R-Tree Secondary Access Method DataBlade module, refer to "R-Tree Secondary Access Method DataBlade Module" on page 1-12.
4. Register the DataBlade module described in step 1 into your database using BladeManager.
5. Create a table that contains one or more columns 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 *IBM Informix 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 Shapes3 DataBlade module that Appendix A describes. These objects include the data type MyShape and the operator class **MyShape_ops**. Columns of data type MyShape can store points, boxes, and circles.

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: R-tree indexes must be created in dbspaces with the default page size.

You can create an R-tree index either before or after you insert data into the table. 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 do 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 causes the transaction to abort.

In addition, if you use the bottom-up build method, described later in this chapter, to create the R-tree index after you have loaded the data, the size of the index is typically about two-thirds the size of the index built with a slower method. The R-tree access method uses bottom-up building when creating an R-tree index only when data currently exists in the table.

Syntax

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

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

The *parameters* and *index_options* arguments are optional.

Important: The ONLINE keyword of the CREATE INDEX and DROP INDEX statements is not supported for R-Tree indexes.

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. For more information on the data types that support R-tree indexes, check the DataBlade module user's guide.
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 in your database a DataBlade module 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 if 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. 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.
parameters	The parameters that specify how an R-tree index is built These parameters only affect the building of the index, not the subsequent use of the index.	You can specify the following index parameters: BOTTOM_UP_BUILD, BOUNDING_BOX_INDEX, NO_SORT, SORT_MEMORY, FILLFACTOR. For detailed information about each index parameter and when you should use it, refer to "R-Tree Index Parameters" on page 2-3.
index_options	The fragmentation and storage options of the index, described in detail in the section "R-Tree Index Options" on page 2-6	The options available for R-tree indexes are FRAGMENT BY and IN. The options CLUSTER, UNIQUE, DISTINCT, ASC, DESC, and FILLFACTOR are not supported.

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

R-Tree Index Parameters

You use index parameters to specify how the R-tree access method builds an R-tree index on a table column. The index parameters only affect the creation of the index; they do not affect subsequent use of the index.

Each index parameter is set to a value in single quotes. For example, if you want to specify a fill factor of 80, you specify the index parameter as `FILLFACTOR='80'`. For detailed examples of using index parameters, refer to "Examples of Creating R-Tree Indexes" on page 2-7.

The following table describes each R-tree index parameter in detail.

Index Parameter	Description	Default Value
BOTTOM_UP_BUILD	Specifies whether to use bottom-up building when creating an R-tree index By default, the R-tree secondary access method builds an R-tree index by using an algorithm that bulk loads data very quickly into the index. This is also called bottom-up building. To use bottom-up building, you must create a temporary dbspace. You can set this index parameter to NO (do not use bottom-up building to build the R-tree index) or YES. For detailed information on bottom-up building of R-tree indexes, refer to "Bottom-Up Building of R-Tree Indexes" on page 2-5.	Yes
SORT_MEMORY	Specifies the amount of shared memory in kilobytes (per index fragment) that the R-tree secondary access method uses for sorting when it creates an R-tree index with the bottom-up building method. This index parameter only applies if BOTTOM_UP_BUILD is also specified. Increase the value of SORT_MEMORY to speed up the R-tree index creation. The minimum value you can set this index parameter to is 8. The maximum value is determined by the amount of shared memory available on your computer. You can also specify the shared memory the R-tree access method uses for sorting by setting the ONCONFIG parameters DS_TOTAL_MEMORY and DS_MAX_QUERIES, as described in the **Default Value** column.	The value of the ONCONFIG parameter DS_TOTAL_MEMORY divided by the value of the ONCONFIG parameter DS_MAX_QUERIES If the two ONCONFIG parameters are not specified in the ONCONFIG file, then the default values of the two ONCONFIG parameters are used. The default value for DS_TOTAL_MEMORY is 256 KB and the default value for DS_MAX_QUERIES is 2.
FILLFACTOR	Specifies what percentage of an index page should be filled with entries as the R-tree access method creates the R-tree index The unfilled part of an index page is then available for future growth of the index. This index parameter only applies if BOTTOM_UP_BUILD is also specified. If you specify a low value, the index will be larger, but there will be more space on each index page to accommodate future entries in the index. Although it is not necessary to leave space for future entries, if the pages are too full, the first few new entries will cause many page splits and thus slow performance. If you specify a high value, the R-tree index will be smaller, but new additions to the index might cause more page splits. This index parameter is similar to that for B-tree indexes. You can set this index parameter to an integer between 1 and 100.	`100` This means that all index pages will be completely filled.
NO_SORT	Speeds up the creation of R-tree indexes on already-sorted tables The NO_SORT index parameter is only valid with R-tree indexes that support bottom-up build. The DataBlade module you are using must provide a function that returns a numeric spatial key given an object of the data type that is being indexed. The procedure shown in"Using the NO_SORT Index Parameter" on page 2-6 explains how to first sort a table and then create an R-tree index using the NO_SORT index parameter.	`NO`

Index Parameter	Description	Default Value

BOUNDING_BOX_ INDEX	When set to NO, creates an R-tree index that stores copies of the data objects themselves in the leaf pages (instead of just their bounding boxes) During an R-tree index scan, if the index is a bounding-box-only index (the default), the table is accessed for the final exact geometry check. For this reason, many more additional page reads might occur during a scan if the row size of the table is large due to large columns. In this case, to improve performance, you might want to create your R-tree index so that copies of the data objects are stored in the leaf pages. Specify BOUNDING_BOX_INDEX='NO' in the CREATE INDEX statement, as the following example shows:
	`CREATE INDEX circle_tab_index5`
	`ON circle_tab ( circles MyShape_ops )`
	`USING RTREE`
	`(BOUNDING_BOX_INDEX='NO');`
		YES

Bottom-Up Building of R-Tree Indexes

When you create an R-tree index, by default, the access method builds the index using a fast bulk-loading algorithm, called bottom-up building. You can set BOTTOM_UP_BUILD='NO' to not use bottom-up building to build the R-tree index.

The algorithm assumes that the four bulk-loading support functions (**SFCbits**, **ObjectLength**, **SFCvalue**, and **SetUnion**) exist and are defined by the operator class specified in the CREATE INDEX statement. The section Support Functions in "Creating a New Operator Class" on page 3-8 explains what these functions do, if they are supplied by the DataBlade module you are using. For example, the **SFCvalue** function returns a spatial key, which you can use to sort input data. If the four bulk-loading functions do not exist, the access method builds the R-tree index using a slower algorithm.

You must also have previously created a temporary dbspace for the access method to use bottom-up building when you create an R-tree index. If a temporary dbspace does not exist, or it is too small, then the access method builds the R-tree index using a slower algorithm.

Use the following expression to calculate the minimum size, in bytes, of the temporary dbspace you need to create an R-tree index with a 4-byte spatial key:

numrows * (24 + *L*)

The *numrows* variable is the number of rows in the table, and *L* is the maximum size of the data objects being indexed.

Use the following expression to calculate the minimum size, in bytes, of the temporary dbspace you need to create an R-tree index with an 8-byte spatial key:

numrows * (30 + *L*)

The default value of the SORT_MEMORY index parameter, specified in "R-Tree Index Parameters" on page 2-3, is too small for most R-tree indexes. For this reason, you should specify a larger value when you create the index.

Using the NO_SORT Index Parameter

If the DataBlade module that you are using provides a function that returns a numeric spatial key given an object of the data type that is being indexed, you can use this function to create a statically clustered table according to a functional B-tree index. Then, when you create an R-tree index on the resulting clustered table, the R-tree secondary access method does not need to sort the data as it builds an index from the bottom up, because the table is already sorted according to the same criterion that the R-tree bottom-up build would use.

To first sort a table and then create an R-tree index using the NO_SORT index parameter:

1. Check your DataBlade module documentation for a function that returns a spatial key given an object of the data type that is being indexed.

   For this procedure, assume this function is called **SpatKey()**.
2. Create a clustered functional B-tree index on your table using the **SpatKey()** function, as in:

   ```
   CREATE CLUSTER INDEX btree_func_index on
       table1 (SpatKey(column1));
   ```

   btree_func_index is the name of the clustered functional B-tree index, **table1** is the name of the table, and **column1** is the name of the column that contains the spatial data.
3. Create the R-tree index on the *spatial_column_name* column, specifying the NO_SORT='YES' index parameter:

   ```
   CREATE INDEX rtree_index ON table1 (column1 my_ops)
       USING RTREE (NO_SORT = 'YES');
   ```

   In the example, **rtree_index** is the name of the R-tree index and **my_ops** is the name of the operator class associated with the data type of column **column1**.
4. Because the R-tree index does not use the clustered functional B-tree index, you can drop the B-tree index if you want:

   ```
   DROP INDEX btree_func_index;
   ```

R-Tree Index Options

This section discusses the options to the CREATE INDEX command that R-tree indexes support.

Using the FRAGMENT Clause

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 in a dbspace with the default page size, the R-tree index is also fragmented by default. The index fragments are automatically stored in the same dbspace as the table fragments. You cannot create an R-tree index on a fragmented table in a dbspace with a non-default page size.

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

Using the IN Clause

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 same dbspace or dbspaces as the table on which it is built and inherits the distribution scheme of the table.

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 same dbspace or dbspaces as the table.

Examples of Creating R-Tree Indexes

The following example shows how to create a table called **circle_tab** that contains a column of data type MyCircle and 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 shows how to create a similar R-tree index that is stored in the **dbsp1** dbspace instead of the dbspace in which the **circle_tab** table is stored:

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

The following example shows how to create a fragmented R-tree index on the **circle_tab** table:

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

All shapes with **id** less than 100 are stored in the **dbsp1** dbspace, and the remainder are stored in the **dbsp2** dbspace.

The following example shows how to create a fragmented table called **circle_tab_frag** and then an R-tree index on the table called **circle_tab_index4**:

```
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;
```

All shapes with **id** less than 100 are stored in the **dbsp1** dbspace, and the remainder are stored in the **dbsp2** dbspace.

The following example shows how to create a fragmented table called **circle_tab_frag** and then an R-tree index on the table called **circle_tab_index4**:

```
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;
```

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.

The following example shows how to specify index parameters when you create an R-tree index:

```
CREATE INDEX circle_tab_index5
ON circle_tab ( circles MyShape_ops )
USING RTREE (BOTTOM_UP_BUILD='YES', FILLFACTOR='80', SORT_MEMORY='320');
```

The parameters specify that the R-tree index should be built using fast bulk loading, that the fillfactor is 80, and that the R-tree access method has 320 KB of shared memory available for sorting.

The following example shows how to drop an R-tree index:

```
DROP INDEX circle_tab_index;
```

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

The query optimizer can choose to use an R-tree index when it evaluates a query if the following statements are true:

- A strategy function of the 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 might 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 a B-tree index is on the **id** column, the query optimizer 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 optimizer chooses which index to use, or whether to use an index at all, by comparing the cost of each option. Cost is an estimate of the number of pages that need to be accessed. The cost of using an R-tree index is calculated by using the selectivity and per-row cost functions provided by the DataBlade module (for example, the IBM Informix Geodetic DataBlade Module and the IBM Informix Spatial DataBlade Module both provide these support functions). See "Selectivity and Cost Functions" on page 3-30 for information about how to include selectivity and per-row cost functions in a DataBlade module.

The following query retrieves cities with names that start with *San* that are located within the specified polygon. The optimizer can choose either a B-tree index (on **name**), an R-tree index (on **obj**) or a sequential table scan:

```
SELECT location FROM cities WHERE
name LIKE "San%" AND
Intersect(obj, 'GeoPolygon((((-49,45), (34, 48),
(3, -45), (0, -48))), ANY, ANY)');
```

To determine which index was actually used, use SET EXPLAIN ON.

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," on page 4-1.

Complex Qualifications

A complex qualification is a WHERE clause in which two or more logical operators are used on the *same* column on which the R-tree index is defined.

If you specify a complex qualification in a query that includes only AND or only OR logical operators, then the entire qualification is evaluated using one R-tree index scan. An example of a complex qualification that uses only AND logical operators is `WHERE Overlap(A,B) AND Contains(A,C) AND Contains(A,D)`, where **A** is the indexed column. An example of a complex qualification that uses only OR logical operators is `WHERE Overlap(A,B) OR Contains(A,C) OR Contains(A,D)`.

If, however, you mix AND and OR logical operators in the same complex qualification, the R-tree index is not used at all for any of the predicates. This means that arbitrary logical expressions cannot necessarily use an R-tree index. An

example of a complex qualification that mixes AND and OR logical operators is `WHERE Overlap(A,B) AND Contains(A,C) OR Contains(A,D)`, where **A** is the indexed column.

R-Tree Indexes and Null Values

The R-tree secondary access method uses bounding boxes of data objects to calculate how to insert the object into an R-tree index and how to subsequently search for the object. An R-tree index, however, cannot create a bounding box for a null value. For this reason, an R-tree index treats null values differently from non-null values, as the following sections describe.

How an R-Tree Index Internally Handles Null Values

If you insert a null value into a spatial column on which you created an R-tree index, the index ignores the insertion and does not create a reference back to the row in the table. Similarly, if you delete a null value from the table, the R-tree index ignores the deletion and the index is not changed, because no reference back to the table row ever existed.

If you update a null value in the table to a non-null value, the R-tree index ignores the deletion of the null value and inserts the non-null value into the R-tree index. Similarly, if you update a non-null value to a null value, the R-tree index deletes the non-null value from the R-tree index and ignores the insertion of the null value.

How Strategy Functions Handle Null Values

If you specify a null value for any of the arguments of a strategy function in the WHERE clause of a query, the query *always* returns 0 rows. This is true even if you specified that the strategy function handles nulls when you created the strategy function with the CREATE FUNCTION statement.

For example, assume you previously inserted a null value into the **circle_tab** table with the following INSERT statement:

```
INSERT INTO circle_tab VALUES (1, NULL);
```

The following query that uses the **Equal** strategy function to search for null values always returns 0 rows, even though a null value does exist in the table:

```
SELECT * FROM circle_tab WHERE Equal (circles, NULL);
```

Zero rows are always returned because null values are never part of an R-tree index; they are stored only in the table. To search for null values in a column on which you created an R-tree index, use the IS NULL condition in the WHERE clause of the query, as the following example shows:

```
SELECT * FROM circle_tab WHERE circles IS NULL;
```

The preceding query does not use the R-tree index, and thus the database server must perform a full table scan. However, because the query is searching the table, the query returns what you expect: those rows whose **circles** column is null.

Performing Nearest-Neighbor Searches

If the DataBlade module you are using is set up for it, you can perform nearest-neighbor searches. For example, the IBM Informix Geodetic DataBlade Module and the IBM Informix Spatial DataBlade Module both provide nearest-neighbor search support.

After you create an R-tree index on the column on which you want to perform nearest-neighbor queries, you can use the functions that your DataBlade documentation identifies as nearest-neighbor functions to perform nearest-neighbor queries.

Nearest-neighbor searches return results in order of increasing distance from the specified object or location. Without any other restriction, a nearest-neighbor query returns a result for all rows returned by the query. Often, you will want to restrict the results, for example, using the FIRST *n* syntax to obtain just the first few results (as shown in the example below).

The WHERE clause of a nearest-neighbor query can include other qualifications, provided the clause is connected by AND.

A DataBlade module might provide more than one nearest-neighbor function. For example, the IBM Informix Spatial DataBlade Module provides the **SE_Nearest** and **SE_NearestBBox** functions. The **SE_Nearest** function calculates precise distances between objects. **SE_NearestBBox** calculates distances as measured between objects' bounding boxes (envelopes). Because this calculation is simpler, **SE_NearestBBox** executes more quickly but might return objects in a different order depending on the actual shape of the objects.

Limitations

The WHERE clause of a nearest-neighbor query cannot include:

- Clauses connected by OR
- Clauses connected by AND NOT

Only one nearest-neighbor function can be used per query.

Using a fragmented R-tree index for nearest-neighbor queries raises an error. The results are not returned in nearest-distance order because the query is executed on each separate index fragment, and results from each fragment are combined in an unspecified order.

Example

The following example shows how to perform a nearest-neighbor search using the IBM Informix Spatial DataBlade Module. The **SE_Nearest** function allows you to perform the search.

The **cities** table was created with the following statement. It contains the names and locations of approximately 300 world cities.

```
CREATE TABLE cities (name varchar(255),
                     locn ST_Point);
```

An R-tree index is created on the **locn** column:

```
CREATE INDEX cities_idx ON cities (locn ST_Geometry_ops)
   USING RTREE;

UPDATE STATISTICS FOR TABLE cities (locn);
```

Now search for the five cities nearest London:

```
SELECT FIRST 5 name FROM cities
   WHERE SE_Nearest(locn, '0 point(0 51)');
```

The query returns the following results:

```
name  London

name  Birmingham

name  Paris

name  Nantes

name  Amsterdam
```

Database Isolation Levels and R-Tree Indexes

Database isolation levels affect the degree of concurrency among processes that attempt to access the same rows at the same time. There are four levels of isolation:

- DIRTY READ
- REPEATABLE READ
- COMMITTED READ
- CURSOR STABILITY

Use the SQL statement SET ISOLATION to set the isolation level for your session.

If you specify the COMMITTED READ isolation level and use an R-tree index to select rows in a table, all the rows returned by the search are guaranteed to be committed. The same query, however, might *not* read some of the rows that have been deleted, but not yet committed, by another concurrent transaction.

While R-tree indexes can use the COMMITTED READ isolation level as described in the previous paragraph, R-Tree indexes cannot use the COMMITTED READ LAST COMMITTED isolation level feature.

This behavior differs slightly from that of Informix B-tree indexes. *IBM Informix Guide to SQL: Syntax* provides detailed information on the type of concurrency that each isolation level enforces for B-tree indexes.

Functional R-Tree Indexes

You can also use the R-tree access method to create a functional R-tree index. A *functional index* supports retrieval of table rows according to the value of some computation done on the columns of the rows. The value is not actually stored in the table, but it is precomputed and used to build 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 an opaque data type that contains a reference to a smart large object as a return type. This is true for all functional indexes, not just R-tree functional indexes.

Functional R-tree indexes are not bounding-box-only indexes; they store the data objects themselves in leaf pages.

The examples in the rest of this section show how to create and use a functional R-tree index on a table that stores point coordinates. Although the table does not contain any columns of a data type that can be indexed by an R-tree index, the functional R-tree index allows you to use the R-tree access method to search for specific points in the table.

The following example shows how to create and populate a **coordinates** table that has two columns that store the point coordinates; the **x** column stores *x*-coordinates and the **y** column stores *y*-coordinates:

```
CREATE TABLE coordinates
(
    id INTEGER,
    x  FLOAT,
    y  FLOAT
);

INSERT INTO coordinates VALUES (1, 2.0, 3.0 );
INSERT INTO coordinates VALUES (2, 4.0, 5.0 );
```

The following example shows how to create a functional R-tree index called **coordinates_idx** on the two coordinate columns of the **coordinates** table using the **MyPoint()** function:

```
CREATE INDEX coordinates_idx
ON coordinates (MyPoint (x,y) MyShape_ops)
USING RTREE;
```

The following example shows a query that could use the **coordinates_idx** functional R-tree index:

```
SELECT id FROM coordinates
WHERE MyPoint(x,y) = 'point(2.0, 3.0)';
```

The query searches for all points in the coordinates table that have the coordinates `(2.0, 3.0)`.

For more information on how to create functional indexes, refer to the *IBM 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 by creating a new operator class. It discusses the following topics:

- Overview of DataBlade Module Development

- Deciding Whether to Use the R-Tree Access Method
- Designing a User-Defined Data Type
- Creating a New Operator Class
- Setting Up Nearest-Neighbor Searching
- Creating Registration Scripts for Dependent DataBlade Modules
- "Repairing R-tree Indexes After Migrating to a Different Version of a DataBlade Module" on page 3-37

Overview of DataBlade Module Development

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

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 some 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 *IBM Informix DataBlade Developers Kit User's Guide*.

Important: The examples in this chapter are taken from the definition of the objects of the Shapes3 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.

Deciding Whether to Use the R-Tree Access Method

The R-tree secondary access method is specifically designed to index data with the following two special properties:

- The data is multidimensional.
- On a given dimension, a data object spans some width. That is, it corresponds to an interval or range, not a point.

Examples of these types of data include:

- Two-dimensional spatial objects, such as points, lines, and polygons

- Geographic mapping information, defined in terms of latitude and longitude, that includes pointlike objects, such as cities; linelike objects, such as roads and rivers; and regionlike objects, such as counties, states, and land masses
- Video or audio clips, each with a start and stop time

  If you create a time range user-defined data type, you can 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

An R-tree index works on data with only one of these properties (multi-dimensional points or ranges along a single dimension) but data corresponding to points on a single dimension is better indexed with a B-tree index.

Unlike other data structures, such as a grid-file and a 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 Dynamic Server) is ordered numeric values in one dimension. Do not use B-tree indexes to index range or interval data. The following 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 employee IDs
- Character data, such as last names and product names

After you decide 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 3-8 describes this process. The next section describes issues you should be aware of when you design the user-defined data type.

Designing a User-Defined Data Type

This section contains the topics you should consider when you design a user-defined data type.

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 *IBM Informix User-Defined Routines and Data Types Developer's Guide* .

Data Objects and Bounding Boxes

As discussed in Chapter 1, "R-Tree Secondary Access Method Concepts," on page 1-1, 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, 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 can 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 to use the bounding box and when to use the data object 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 searches the tree as described in "Searching with an R-Tree Index" on page 1-6. Because 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 can be defined for the internal representation of the user-defined data.

  The same data structure must be passed to all functions that accept the user-defined data type as an argument. Examples of such functions 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 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 how to create these implicit casts, refer to *IBM Informix User-Defined Routines and Data Types Developer's Guide*.
4. Create the required strategy functions in SQL for *just* the supertype.

   You do not need to create strategy functions for the subtypes because 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.

If the query optimizer is unable to find a function for a particular subtype when it is executing a query, the query optimizer 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. Because 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.

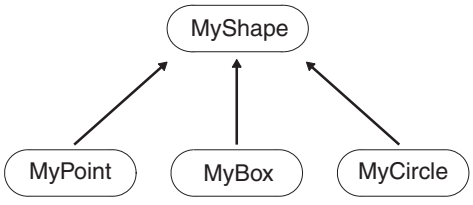


Figure 3-1. Data Type Hierarchy

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, because 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. Because 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.

Union Support Function in a Data Type Hierarchy

When you create the **Union** support function, 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 is true only for the **Union** support function and not for the other support functions.

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, minus the size in bytes of the bounding box of the data object, are available.

Furthermore, R-tree indexes should always fit at least two keys on a single leaf page. Although the R-tree index works correctly with just one key per leaf page, the index performs better when two or more keys fit on single page. This means that the maximum size, in bytes, of a user-defined data type that is to be indexed with an R-tree index should optimally be:

```
(2000 - B - (K * 20)) / K
```

In the formula, *B* refers to the size, in bytes, of the bounding box of the data object, and *K* refers to the number of keys you want to fit on a page. For example, if you want to fit three keys on a single page, then the maximum size of the data type is:

```
(1940 - B) / 3
```

Although this maximum size 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. Multirepresentational user-defined data types store a value in the table if it is smaller than the maximum size of the user-defined data type, 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 identify all 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 possibility you might get inaccurate results when you use 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 extend beyond `B`'s, due to its looseness. The **Within** strategy function, if written to rely on a preliminary bounding box check, might return FALSE when it should return TRUE. As a result, the R-tree access method code that called the **Within** function might miss some qualifying data.

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.

The R-tree access method code might call a different strategy function when it processes internal pages. For example, the access method uses the **Contains** strategy function for internal pages when it processes a query that specifies the **Equal** function. The bounding box logic must be correct in all cases.

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. Because 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 is 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 IBM 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**, it is recommended that 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 includes only the four required strategy functions: **Overlap**, **Equal**, **Contains**, and **Within**. If you want to create more than these four strategy functions, you *must* create your own operator class.

The **rtree_ops** operator class also restricts the number of support functions to the three required ones: **Union**, **Size**, and **Inter**. Because bottom-up building of R-tree indexes requires that you also create the **SFCbits**, **ObjectLength**, **SFCvalue**, and **SetUnion** functions, the **rtree_ops** operator class does not support bottom-up building.

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.

   This step is described in "Support Functions" on page 3-9.
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.

   This step is described in "Strategy Functions" on page 3-21.
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 that the R-tree access method requires and examples of how to create them.

When you use the 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 because the DBDK automatically generates the necessary scripts. You do, 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 about DBDK and BladeSmith, refer to the *IBM Informix DataBlade Developers Kit User's Guide*.

For more information on the DataBlade API, refer to the *IBM Informix DataBlade API Programmer's Guide*.

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 is included automatically in the CREATE FUNCTION statement. If, however, you create the support and strategy functions yourself, the function is VARIANT by default.

Support Functions

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

The R-tree access method uses support functions 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 1-3.

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

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

If you plan to support bounding-box-only R-tree indexes (described in "Bounding-Box-Only R-Tree Indexes" on page 1-4), which are the default R-tree indexes created by Version 9.21.UC1 or later of the database server, or you plan to support nearest-neighbor searches, you must also implement the **RtreeInfo** support function with the operation **strat_func_substitutions**.

Important: To support bounding-box-only indexes or nearest-neighbor searches, you might also need to redesign your strategy functions that occupy slots 5 and up, if you want them to behave differently at nonleaf pages. This is because you cannot distinguish between leaf and nonleaf items in a bounding-box-only index. For more information, see "The RtreeInfo Function" on page 3-14.

You must list the **Union**, **Size**, and **Inter** support functions in the order shown when you execute the CREATE OPCLASS statement to register the operator class with the database server. In other words, you must list the **Union**, **Size**, and **Inter** support functions as the first, second, and third support functions, respectively, in the CREATE OPCLASS statement. This SQL statement is described in "Syntax for Creating a New Operator Class" on page 3-31.

In addition to the required support functions, the R-tree access method also recognizes the following four optional support functions that it uses to enhance the performance of the statement that creates the R-tree index:

- **SFCbits**
- **ObjectLength**
- **SFCvalue**
- **SetUnion**

You are not required to include these support functions in your operator class. However, since these functions are specifically designed to improve the performance of the creation of R-tree indexes, it is highly recommended that you include them in your operator class.

If you decide to include these optional support functions in your operator class, you must list them *after* the required support functions, in the order shown, when you execute the CREATE OPCLASS statement to register the operator class with the database server. In other words, you must list the **SFCbits**, **ObjectLength**, **SFCvalue**, and **SetUnion** support functions as the fourth, fifth, sixth, and seventh support functions, respectively, in the CREATE OPCLASS statement. This SQL statement is described in "Syntax for Creating a New Operator Class" on page 3-31.

You must list the **RtreeInfo** support function in the eighth position, after **Union**, **Size**, and **Inter**, and the four optional bulk-loading support functions. If you do not provide the four optional bulk-loading support functions in your DataBlade module, specify NULL in the fourth, fifth, sixth, and seventh positions in the CREATE OPCLASS statement.

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

Tip: It is useful 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 it describes 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 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.

The **RtreeInfo** support function determines, for a given strategy function, which strategy function should actually be called when the R-tree access method is working on an internal nonleaf page. It also provides support for nearest-neighbor searches. You must define the **RtreeInfo** function if your DataBlade module is going to support bounding-box-only R-tree indexes or nearest-neighbor searches.

The R-tree access method uses the four optional support functions (**SFCbits**, **ObjectLength**, **SFCvalue**, and **SetUnion**) to increase the performance of initial R-tree index creation by performing fast bulk loading of data into the index from a populated table. First, the R-tree access method groups together the rows that belong to the same page. At the same time, the access method identifies the neighbors of each page. Once this process is completed, the R-tree access method stores all the rows in a singly linked list of leaf pages, filled as compactly as possible. As the leaf pages become full, the access method recursively builds the pages at the higher levels. The R-tree access method repeats this process until all the rows are written into the leaf pages.

The R-tree access method uses this method of building R-tree indexes only if you specify the optional support functions in the appropriate operator class. If you do not specify these support functions, then the R-tree access method uses a slower method to create the R-tree index.

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

The Union Function

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. In other words, the **Union** function tells the R-tree access method the data size of a bounding box.

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 SQL 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.

The return value of the **Union** function is not used by the R-tree access method. The **Union** function should call the **mi_db_error_raise()** DataBlade API function to return errors.

For variable UDTs, the third parameter of the **Union** function is not initialized; it contains a valid **mi_lvarchar** data type with slightly more memory than necessary allocated to it. Be sure you set the size in the function to the size, in bytes, of the largest possible result.

The result returned in the third parameter of the **Union** function must be a fixed size and not a large object. Set its size 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.

For sample C code of the **Union** function, see "Union Support Function" on page A-11. C code uses the DataBlade API to interact with the database server.

The Size Function

The R-tree access method uses the **Size** function to evaluate different ways to group objects by comparing the sizes of bounding boxes around objects or groups of objects. It does this when it decides where to place a new data object and when it splits a page. Ideally, a disk page is divided into two pages whose overall bounding boxes are as compact and small as possible.

For sample C code of the **Size** function, see "Size Support Function" on page A-12. C code uses the DataBlade API to interact with the database server.

Signature of the Size Function: The SQL 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 return value of the **Size** function is not used by the R-tree access method. The **Size** function should call the **mi_db_error_raise()** DataBlade API function to return errors.

Calculating the Size of a Bounding Box: Write the **Size** function to *always* return a different value as a bounding box expands or shrinks by the addition or removal of objects inside it. This means that you should add a compensating factor when

calculating the size to take care of degenerate bounding boxes. A *degenerate bounding box* is one that has one or more sides of 0 length.

Assume your data is in a two-dimensional space and you decide to use a simple *length* times *width* calculation to compute the size of a bounding box. If the width of the bounding box subsequently shrinks to 0, then the size of the bounding box is 0. However, if it was the length of the original bounding box that shrunk to 0, then the size would also be 0, breaking the rule that different bounding boxes return different sizes. Figure 3-2 describes this situation.

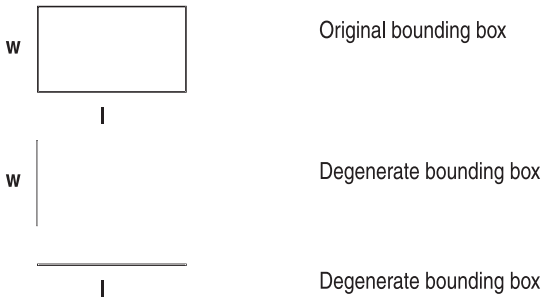


Figure 3-2. Size Calculation of Degenerate Bounding Boxes

In this situation, a better formula for calculating the size of a bounding box would be:

(*length* times *width*) plus (*length* plus *width*)

This formula for the **Size** function always returns a larger value if the box changes by the inclusion of a new item and returns a smaller value if it shrinks because something inside was removed.

The Inter Function

The SQL 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. The R-tree access method uses the resulting bounding box in a subsequent call to the **Size** function to find out how much two bounding boxes overlap.

The return value of the **Inter** function is not used by the R-tree access method. The **Inter** function should call the **mi_db_error_raise()** DataBlade API function to return errors.

For variable length UDTs, the third argument of the **Inter** function is not initialized; it contains a valid **mi_lvarchar** data type. You must set the size in the function to the size, in bytes, of the largest possible result.

For sample C code of the **Inter** function, see "Inter Support Function" on page A-13. C code uses the DataBlade API to interact with the database server.

The RtreeInfo Function

The **RtreeInfo** support function defines the switching semantics for the strategy functions in your DataBlade module. The R-tree access method calls the **RtreeInfo** function, if it exists, to determine, for a given strategy function, which strategy function it should actually call when working on an internal nonleaf page.

Earlier versions of the R-tree access method required DataBlade module strategy functions to test whether a page stored a bounding box or not to determine if that page was a leaf page or an internal page (only internal pages used to store bounding boxes). In the current version of the R-tree access method, if your DataBlade module implements the **RtreeInfo** function with the **strat_func_substitutions** operation, by default, indexes are created as bounding-box-only R-tree indexes; leaf pages store only bounding boxes (and not data objects).

For this version of the R-tree access method, if you are supporting bounding-box-only indexes, you must use a different method to specify how strategy functions behave when called on an internal nonleaf page or on a leaf page. To better understand why you might want your strategy function to behave differently on an internal nonleaf page or on a leaf page, see the following example and "Internal Uses of the Strategy Functions" on page 3-22. This section describes why each of the four required strategy functions sometimes uses different strategy functions on internal nonleaf pages and which function is actually used on the internal nonleaf pages. If necessary, you must redesign your strategy functions if you want them to behave differently for leaf and nonleaf pages. This is because you cannot distinguish between leaf and nonleaf items in a bounding-box-only index.

For example, suppose you have a strategy function in slot 5 named **MyEqual**, which is a variation on the **Equal** function. When this function is called on a nonleaf page, you want it to behave like **Contains**; you cannot eliminate nonleaf items by testing their bounding boxes for equality, because the test is too stringent. But when **MyEqual** is called on a leaf page, you do want it to test for equality. If the leaf pages contain the complete objects (the index is not a bounding-box-only index), you can implement this behavior switch yourself in the **MyEqual** function by checking to see if one or both operands are bounding boxes. However, with a bounding-box-only index, the leaf pages hold only the objects' bounding boxes. In this case, an implementation of **MyEqual,** which performs a **Contains** check whenever it is called with bounding boxes, would be inefficient because it would force the R-tree access method to make the extra step of retrieving a complete object from the table. Instead, a candidate data object could be eliminated immediately by performing an equality check on its leaf page bounding box.

To detect whether the operands are leaf or nonleaf data, and switch behavior accordingly, use the **RtreeInfo** support function, as described in this section, or design your own strategy functions to make this determination.

Important: If you create an **RtreeInfo** support function that defines the switching semantics of your strategy functions, you must modify your DataBlade module code to ensure that the strategy functions in slots 5 and up do not try to determine whether they are being executed on an internal or leaf page based on whether the input is a bounding box.

Important: If the R-tree access method detects an **RtreeInfo** support function that implements the **strat_func_substitutions** operation, the R-tree access method sets the default mode of index creation to "bounding-box-only."

The R-tree access method checks for the **RtreeInfo** function when it creates an R-tree index and updates the root page with the information. This means that if you create an **RtreeInfo** support function that defines the switching semantics of your strategy functions, you must update existing R-tree indexes so they know about it.

Use the **oncheck** utility to update any existing indexes, using the following syntax:

```
oncheck -ci -u "info_anchor_update"
{database[:[owner.]table[,fragdbs|#index]]}
```

Arguments of the RtreeInfo Support Function: Write the **RtreeInfo** support function to take four arguments.

Argument	Signature	Description
First	mi_lvarchar *dummy_obj	Should be NULL.
Second	mi_lvarchar *operation_ptr	A pointer to an MI_LVARCHAR structure that contains a string that represents the information needed from the DataBlade module. When writing the **RtreeInfo** function to return the internal-page equivalents of strategy functions, the string is **strat_func_substitutions**.
Third	mi_lvarchar *opclass_ptr	This argument points to an MI_LVARCHAR structure that contains a string that represents the name of the operator class.
Fourth	mi_lvarchar *answer_ptr	This argument points to an MI_LVARCHAR structure that contains a pointer to the structure that returns information to the R-tree access method. If *answer_ptr* is NULL, then the R-tree access method calls the **RtreeInfo** function to determine if a particular operation is supported by your DataBlade module. If the operation is not supported, set the return value of the function to RLT_OP_UNSUPPORTED. If the operation is supported, set the return value of the function to MI_OK. If *answer_ptr* is not NULL, fill in the array of integers with the slot numbers of the internal-page equivalent strategy functions. (This array is allocated by the R-tree access method). Then set the return value of the function to MI_OK.

SQL Definition of the RtreeInfo Support Function: Use the following CREATE FUNCTION SQL statement template to create the **RtreeInfo** support function after you write and compile the code:

```
CREATE FUNCTION rtreeInfo(UDT, pointer, pointer, pointer)
  RETURNS INT WITH (NOT VARIANT, PARALLELIZABLE)
  EXTERNAL NAME '$INFORMIXDIR/extend/bladedir/xxx.bld(funcname)
  LANGUAGE C;
```

In the statement template, the text *UDT* refers to *user-defined type* or the data type you want to index with the R-tree access method; *bladedir* refers to the name of your DataBlade module under the **extend** directory; *xxx* refers to the name of the shared object that contains the code for your DataBlade module; and *funcname* refers to the name of the function within the shared object that contains the code for the **RtreeInfo** function.

When you create the operator class with the CREATE OPCLASS statement, include the **RtreeInfo** support function in the eighth position, after the three required support functions **Union**, **Size**, and **Inter**, and the four optional bulk-loading support functions **SFCbits**, **ObjectLength**, **SFCvalue**, and **SetUnion**. If you do not provide the four optional bulk-loading support functions in your DataBlade module, specify NULL in the fourth, fifth, sixth, and seventh positions in the CREATE OPCLASS statement.

C Code Example for the RtreeInfo Support Function: You can use the following sample C code to help write your own **RtreeInfo** function.

```
/*****************************************************************************
* Description: Example of new support function used to return             *
*              requested Information to R-tree.                           *
*                                                                         *
* Arguments:                                                              *
*                                                                         *
* dummy_obj - (is NULL)                                                   *
*                                                                         *
* operation_ptr - ptr to string that represents the operation.            *
*                                                                         *
* opclass_ptr - ptr to string that represents the opclass name.           *
*                                                                         *
* answer_ptr - pointer to the pointer to the structure used to            *
*               return information to R-tree.                             *
*               answer_ptr is a "pointer to a pointer" to make            *
*               the interface generic to support later                    *
*               operations to implement which the blade might             *
*               need to allocate memory and return its address            *
*               to R-tree. For the operation                              *
*               "strat_func_substitutions", memory is allocated           *
*                  by R-tree.                                             *
*                                                                         *
*                                                                         *
* Support function slot no: 8                                             *
*                                                                         *
* Return values: MI_OK - Success, operation supported.                    *
*                MI_ERROR - Error.                                        *
*                RLT_OP_UNSUPPORTED - operation not supported.            *
*                                                                         *
*****************************************************************************
*/

     #define RLT_OP_UNSUPPORTED 1
     mi_integer
     rtreeInfo (mi_lvarchar *dummy_obj, mi_lvarchar *operation_ptr,
                  mi_lvarchar *opclass_ptr, mi_lvarchar *answer_ptr)
     {
     mi_integer status = MI_OK;
     mi_string *operation = NULL, *opclassname = NULL;
                 /* opclassname may be used if required */

         operation = mi_lvarchar_to_string(operation_ptr);
         if (operation == NULL)
             {
             status = MI_ERROR;
             goto bad;
             }

         opclassname = mi_lvarchar_to_string(opclass_ptr);
         if (opclassname == NULL)
             {
             status = MI_ERROR;
             goto bad;
             }

         if (!strcmp(operation,"strat_func_substitutions"))
             {
             mi_integer *answer = NULL;
```

```
        if (answer_ptr == NULL)
            {
            status = MI_OK;
            goto done;
            }/* Option is supported */

        /* For operation "strat_func_substitutions" memory
         * for 64 slots is allocated by R-tree. For later
         * operations, we might need to allocate the return
         * structure and set its address.
         */
         answer =(mi_integer*)
                    mi_get_vardata((mi_lvarchar*)
                        (mi_get_vardata(answer_ptr)));

        if (answer == NULL)
            {
            status = MI_ERROR;
            goto bad;
            }

        /* Provide mapping for strategy functions to be used at
         * internal nodes.
         * If the mapping changes for the opclasses I support,
         * use the opclassname
         */
        if (!strcmp(opclass,"my_opclass1"))
            {
            answer[0] = 0;
            answer[1] = 2;
            answer[2] = 2;
            answer[3] = 0;
            answer[4] = 4;
            answer[5] = 4;
                 /* as many slots as strategy functions. max is 64 */
            }
         else if (!strcmp(opclass,"my_opclass2")) {
            answer[0] = 0;
            answer[1] = 2;
            answer[2] = 2;
            answer[3] = 0;
            answer[4] = 4;
            }
         else /* for all other opclasses that I support */
            {
            answer[0] = 0;
            answer[1] = 2;
            answer[2] = 2;
            answer[3] = 0;
            }

        status = MI_OK;
        }
    else
        status = RLT_OP_UNSUPPORTED;
         /* Only "strat_func_substitutions" is
          * supported, as yet. */

done:
bad:
    if (opclassname)
        mi_free(opclassname);
    if (operation)
        mi_free(operation);
    return status;
}
```

The SFCbits Function

The R-tree secondary access method uses the **SFCbits** function to determine the number of bits required by the internal space-filling curve (SFC) algorithm to represent the spatial key. An example of a space-filling curve is the Hilbert function.

The **SFCbits** support function is optional. If you create it and specify it in the operator class with the other optional support functions, the R-tree secondary access method uses a fast bulk-loading algorithm to initially create an R-tree index. If you have not specified this function in the operator class, then the access method uses a slower method to create R-tree indexes.

The SQL signature of the **SFCbits** support function must be:

```
SFCbits (UDT, POINTER) RETURNS INTEGER
```

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

The sample C signature of the **SFCbits** function for a variable length UDT is:

```
mi_integer SFCbits(mi_lvarchar *object, mi_integer *bits)
```

Write the **SFCbits** function to return, in the second parameter, the number of bits required to build a spatial key on the data type you want to index. This value must be either 32 or 64.

The return value of the **SFCbits** function is not used by the R-tree access method. The **SFCbits** function should call the **mi_db_error_raise()** DataBlade API function to return errors.

For sample C code of the **SFCbits** function, see SFCbits Support Function in Appendix A. C code uses the DataBlade API to interact with the database server.

The ObjectLength Function

The R-tree secondary access method uses the **ObjectLength** function to determine the maximum size, in bytes, of the objects stored in the column that is being indexed with an R-tree index.

The **ObjectLength** support function is optional. If you create it and specify it in the operator class with the other optional support functions, the R-tree secondary access method uses a fast bulk-loading algorithm to initially create an R-tree index. If you have not specified this function in the operator class, then the access method uses a slower method to create R-tree indexes.

The SQL signature of the **ObjectLength** support function must be:

```
ObjectLength (UDT, POINTER) RETURNS INTEGER
```

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

The sample C signature of the **ObjectLength** function is:

```
mi_integer ObjectLength(mi_lvarchar *object, mi_integer *obj_max_length)
```

The first parameter of the **ObjectLength** function contains the name of the data type to be indexed; it does not contain a row value. For example, if the data type to be indexed is **MyPoint**, the parameter contains the string `MyPoint`.

Write the **ObjectLength** function to return, in the second parameter, the maximum possible size, in bytes, of the objects in the column to be indexed.

The return value of the **ObjectLength** function is not used by the R-tree access method. The **ObjectLength** function should call the **mi_db_error_raise()** DataBlade API function to return errors.

For sample C code of the **ObjectLength** function, see ObjectLength Support Function in Appendix A. C code uses the DataBlade API to interact with the database server.

The SFCvalue Function

The R-tree secondary access method uses the **SFCvalue** function to determine the sort values of an array of objects of the data type of the column that is being indexed with an R-tree index.

The **SFCvalue** support function is optional. If you create it and specify it in the operator class with the other optional support functions, the R-tree secondary access method uses a fast bulk-loading algorithm to initially create an R-tree index. If you have not specified this function in the operator class, then the access method uses a slower method to create R-tree indexes.

The SQL signature of the **SFCvalue** support function must be:

```
SFCvalue (UDT, INTEGER, POINTER) RETURNS INTEGER
```

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

The sample C signature of the **SFCvalue** function is:

```
mi_integer SFCvalue(mi_lvarchar *objects, mi_integer array_size,
                    void *spatialKey)
```

Write the **SFCvalue** function to store an array of **mi_lvarchar** pointers in the data portion of the first parameter. Each **mi_lvarchar** pointer points to a data object in the table for which the R-tree access method needs to compute a sort value.

The second parameter is the number of elements in the array.

The third output parameter is an array of either 32-bit or 64-bit values, depending on the number of bits specified in the corresponding **SFCbits** function. This array stores a spatial key for each data object. The number of elements in this array is always the same as the number of elements in the array of the first parameter. The R-tree secondary access method automatically allocates enough space for the array of the third parameter.

The return value of the **SFCvalue** function is not used by the R-tree access method. The **SFCvalue** function should call the **mi_db_error_raise()** DataBlade API function to return errors.

For sample C code of the **SFCvalue** function, see "SFCValue Support Function" on page A-17. C code uses the DataBlade API to interact with the database server.

The SetUnion Function

The R-tree secondary access method uses the **SetUnion** function to determine the union of all the elements in an array of objects of the data type of the column that is being indexed with an R-tree index.

The **SetUnion** support function is optional. If you create it and specify it in the operator class with the other optional support functions, the R-tree secondary access method uses a fast bulk-loading algorithm to initially create an R-tree index. If you have not specified this function in the operator class, then the access method uses a slower method to create R-tree indexes.

The SQL signature of the **SetUnion** support function must be:

```
SetUnion (UDT, INTEGER, POINTER) RETURNS INTEGER
```

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

The sample C signature of the **SetUnion** function is:

```
mi_integer SetUnion (mi_lvarchar *objects, mi_integer array_size,
                     void *UnionObject)
```

Write the **SetUnion** function to store an array of **mi_lvarchar** pointers in the data portion of the first parameter. Each **mi_lvarchar** pointer points to objects in the table for which the R-tree access method needs to compute the union. Each of the objects is either a data object or a bounding box.

The second parameter is the number of elements in the array.

The third output parameter is a single object that contains the union of all the objects in the input array of the first parameter. The R-tree secondary access method uses the **Union** support function to automatically allocate enough space for the output value.

The return value of the **SetUnion** function is not used by the R-tree access method. The **SetUnion** function should call the **mi_db_error_raise()** DataBlade API function to return errors.

For sample C code of the **SetUnion** function, see "SetUnion Support Function" on page A-18. 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. The example is based on the objects of the sample DataBlade module, described in Appendix A.

The SQL statements to register the **Size**, **Inter**, **SFCbits**, **ObjectLength**, **SFCvalue**, and **SetUnion** support functions with the database server are similar to the SQL statement to register the **Union** function.

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

The following SQL statement shows how to register the **Union** support function with the database server:

```
CREATE FUNCTION Union (MyShape, MyShape, MyShape)
RETURNS INTEGER
WITH
(
    NOT VARIANT
)
EXTERNAL NAME "$INFORMIXDIR/extend/shapes.3.0/shapes.bld (MyShapeUnion)"
LANGUAGE C;
```

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.

For the sample C code of the **MyShapeUnion** function, see "Union Support Function" on page A-11. C code uses the DataBlade API to interact with the database server. Sample C code to implement the **Size** and **Inter** functions is also provided in that appendix.

For more information on the DataBlade API, refer to the *IBM Informix DataBlade API Programmer's Guide*.

For more information and examples on how to create user-defined functions, refer to *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Strategy Functions

Strategy functions are user-defined functions that can be used in queries to select data. Registering them as strategy functions with 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 (boxes, region)`, the query optimizer considers using the R-tree index to evaluate the query.

You can include up to 64 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 detail in later sections of this chapter, with an example of creating the **Contains** strategy function.

Tip: It is useful 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 it describes 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 the following table shows.

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
...	...	...	...
64	Available for use	Same function	Same function
			You can use the **RtreeInfo** function to redefine these switching semantics.

The first column of the table 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 specifies the function that the R-tree access method uses as the commutator of a particular strategy function. The **Within** and **Contains** functions are commutators of each other. Other functions, including those numbered 5 and up, are assumed to be their own commutators. This means that the R-tree access method assumes that when it calls the function, the access method can reverse the order of the arguments without changing the results of the function. Strategy functions should be implemented with these commutator substitutions in mind.

In certain cases, the query optimizer uses the commutator functions as substitute functions. 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, because 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 64 can have commutator functions specified by the COMMUTATOR = FUNCTION modifier of the CREATE FUNCTION statements used to register the functions in SQL. If you do not specify a commutator function, the query optimizer does 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.3.0/shapes.bld (MyShapeContains)"
LANGUAGE C;
```

The strategy functions in slots 5 through 64 can also have negator functions specified by the NEGATOR = FUNCTION modifier of the CREATE FUNCTION statements used to register the functions in SQL. The R-tree access method cannot process queries with a negated strategy function, such as `NOT Separated(A,B)`. However, if the **Separated** strategy function declares the **Overlap** function as its negator, the query optimizer is able to substitute the predicate `Overlap(A,B)` for the `NOT Separated(A,B)`, which allows the use of an R-tree index on column **A**.

The fourth column 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 tests 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 test 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.

Similarly, an index search that uses the **Contains** function must substitute the **Equal** function for nonleaf index pages because a qualifying index entry could be in any subtree whose bounding box contains the search object.

Tip: The **RtreeInfo** function allows you to specify which function you want the R-tree access method to call for nonleaf data.

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 might also need to be updated. Updates usually mean 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 different from the old entry. If they are both equal, the access method does not perform the update.

The Overlap Function

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

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

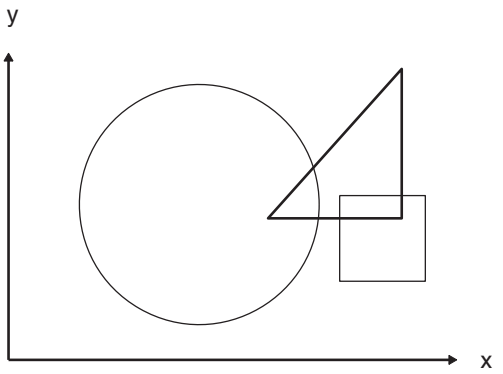


Figure 3-3. Example of a Circle That Overlaps a Triangle

The signature of the **Overlap** function must be:

```
Overlap (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 **Overlap** 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 data objects overlap; and if they do, then test if the data 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 3-4 shows that the first bounding box test eliminates the box-circle overlap immediately, but the second data object test is required to find out if the triangle and circle overlap. In this case, they do not.

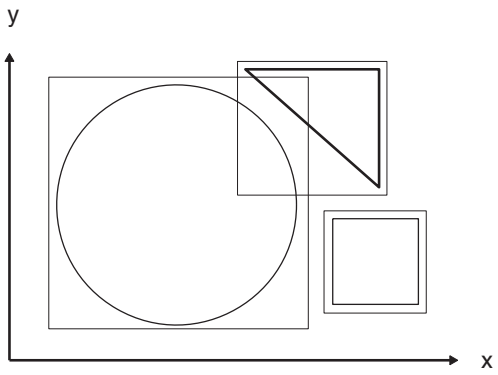


Figure 3-4. Bounding Box Example of the Overlap Function

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

The Equal Function

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** function 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.

The Contains Function

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

Figure 3-5 shows a circle that contains a box. The circle, however, does not contain the triangle, because part of the triangle lies outside the circle.

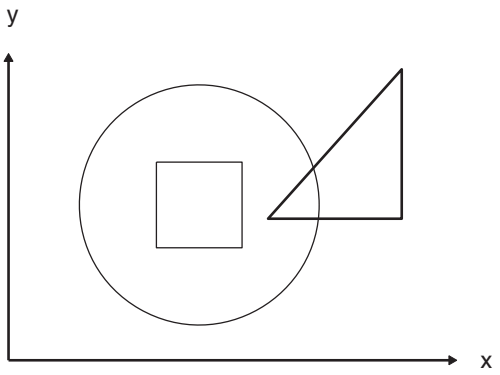


Figure 3-5. Example of a Circle That Contains a Box

The signature of the **Contains** function 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 box of the second object; and if it does, then test if the first data object contains the second data 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 3-6 shows that the first bounding box test eliminates the box-circle containment immediately, but the second data object test is required to find out if the circle contains the triangle. In this case, it does not.

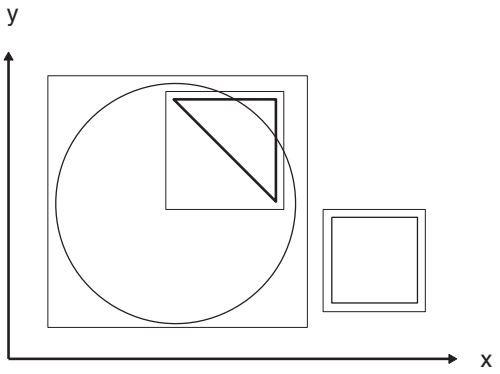


Figure 3-6. Bounding Box Example of the Contains Function

If you allow loose, or inexact, bounding boxes, be careful when you calculate the containment of bounding boxes. For example, Figure 3-7 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.

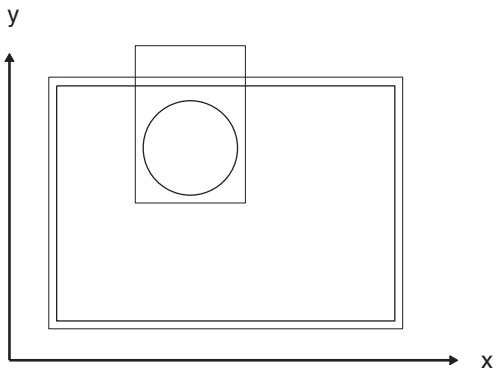


Figure 3-7. Containment and Loose Bounding Boxes

In this case, a preliminary test for bounding box containment returns inaccurate results unless you used 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 3-7.

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, and vice versa. For an example of how to specify a commutator when you create a function, see "Example of Creating a Strategy Function" on page 3-29.

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

The Within Function

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 3-8 shows a box that is within, or contained by, a circle. The triangle, however, is not within either the circle or the box, because all or part of the triangle lies outside both the circle and the box.

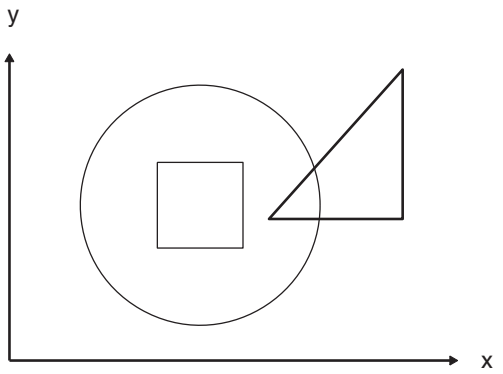


Figure 3-8. Example of a Box That is Within a Circle

The signature of the **Within** function 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 box of the second object; and if it is, then test if the first data object is contained in the second data 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 3-9 shows that the first bounding box test eliminates the box-circle containment immediately, but the second data object test is required to find out if the triangle is within the circle. In this case, it is not.

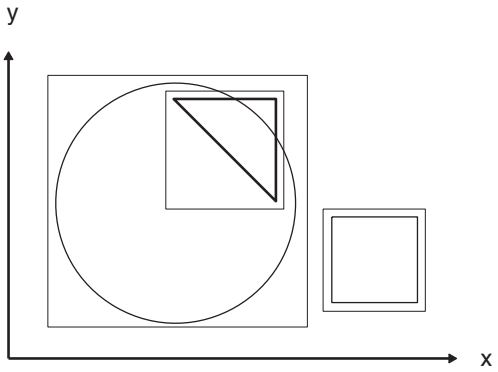


Figure 3-9. Bounding Box Example of the Within Function

If you allow loose, or inexact, bounding boxes, be careful when you calculate the containment of bounding boxes. For example, Figure 3-10 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.

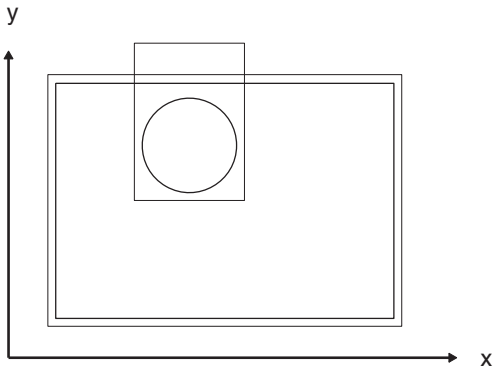


Figure 3-10. Containment and Loose Bounding Boxes

For more information on loose bounding boxes, refer to "Loose Bounding Box Calculations" on page 3-7.

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. For an

example of how to specify a commutator when you create a function, see "Example of Creating a Strategy Function" on page 3-29.

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 60 nonrequired strategy functions for an operator class. This means that together with the four required functions, you can have a total of 64 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 that appendix.

The SQL statements to register the **Overlap, Equal,** and **Within** strategy functions with the database server are similar to the SQL statement to register the **Contains** function.

Tip: The DBDK automatically generates the SQL statement to create the function.

The following SQL statement shows how to register the **Contains** strategy function with the database server:

```
CREATE FUNCTION Contains (MyShape, MyShape)
RETURNS BOOLEAN
WITH
(
    COMMUTATOR = Within,
    NOT VARIANT
)
EXTERNAL NAME "$INFORMIXDIR/extend/shapes.3.0/shapes.bld (MyShapeContains)"
LANGUAGE C;
```

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.

For the sample C code of the **MyShapeContains** function, see "Contains Strategy Function" on page A-8. C code uses the DataBlade API to interact with the

database server. Sample C code to implement the **Overlap**, **Equal**, and **Within** functions is also provided in that appendix.

For more information on the DataBlade API, refer to the *IBM Informix DataBlade API Programmer's Guide*.

For more information and examples on how to create user-defined functions, refer to *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Selectivity and Cost Functions

For the optimizer to accurately assess the cost of using an R-Tree index, your DataBlade module must provide selectivity and per-row cost functions. If these functions are not present, or only one of the functions is present, the cost of using an R-Tree index defaults to 50, except when the nearest neighbor strategy function is used. When the nearest neighbor strategy function is used, the server always uses the R-tree index.

Selectivity is defined as the number of rows in the result set divided by the total number of rows in the table queried (and must be between 0.0 and 1.0):

The per-row cost function calculates the cost of evaluating the predicate of the query for each row (and must be greater than or equal to 0).

For information about how to write selectivity and cost functions, see the *IBM Informix DataBlade API Programmer's Guide*, which describes how to create selectivity and cost functions for an expensive UDR. For a general description of how the query optimizer uses cost and selectivity for UDRs, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

The paper, *Accurate Estimation of the Cost of Spatial Selections by A. Aboulnaga and J. F. Naughton*, might also provide useful information. It is available in the proceedings of the IEEE International Conference on Data Engineering, San Diego, California, 2000.

The cost of using the R-tree index is calculated when you run UPDATE STATISTICS. See "Updating Statistics" on page 4-2 for more information about how statistics are gathered.

You register the selectivity and per-row cost functions when you register the strategy functions for the R-tree index. For example:

```
-- The selectivity function for the strategy function equal
CREATE FUNCTION GeoObjectEqualSelectivity(pointer, pointer)
RETURNS float
WITH (not variant, parallelizable) EXTERNAL NAME
'$INFORMIXDIR/extend/GEO/geodetic.bld(GeoObjectEqualSelectivity)' LANGUAGE
c;


-- The per-row cost function for the strategy function equal
CREATE FUNCTION GeoObjectEqualCost(pointer, pointer)
RETURNS int
WITH (not variant, parallelizable) EXTERNAL NAME
'$INFORMIXDIR/extend/GEO/geodetic.bld(GeoObjectEqualCost)' LANGUAGE c;


--Register the selectivity and per-row cost functions as
--you register the strategy function equal
CREATE FUNCTION equal(GeoObject, GeoObject) RETURNS Boolean
```

```
WITH (not variant, parallelizable,
selfunc=GeoObjectEqualSelectivity,
costfunc=GeoObjectEqualCost)
EXTERNAL NAME
'$INFORMIXDIR/extend/GEO/geodetic.bld(GeoObjectEqual)' LANGUAGE c;
```

It is recommended that you specify the selectivity and per-row cost functions with each strategy function that you register. If you have already registered a strategy function and you want to add the selectivity and per-row cost functions, use the ALTER FUNCTION statement as shown in the following example:

```
ALTER FUNCTION Contains(GeoObject, GeoObject) WITH
(ADD selfunc= GeoObjectContainsSelectivity);

ALTER FUNCTION Contains(GeoObject, GeoObject) WITH
(ADD costfunc= GeoObjectContainsCost);
```

Important: Do not set the selectivity or per-row cost at a constant value; this will cause the cost of using an R-tree index to be set at 50. (If required, you can set your selectivity and per-row cost functions to return a constant value.)

Syntax for Creating a New Operator Class

After you create all the required support and strategy functions, you are ready to create 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
    {, support, 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 previously created Four strategy functions are required; any others are optional.	You can list a maximum of 64 functions. You must include the following four strategy functions: **Overlap**, **Equal**, **Contains**, and **Within**. You can name them whatever you choose, but they must be listed as the first, second, third, and fourth functions, respectively.
support	The names of the three required support functions you previously created; the four support functions for bulk-loading are optional. The support function **RtreeInfo** is also optional but must be in the eighth position if specified.	You must include the following three support functions: **Union**, **Size**, and **Inter**. You can name them whatever you choose, but they must be listed as the first, second, and third functions, respectively. You can optionally include the four bulk-loading support functions: **SFCbits**, **ObjectLength**, **SFCvalue**, and **SetUnion**. You can name them whatever you choose, but they must be listed as the fourth, fifth, sixth, and seventh functions, respectively. If you do not specify the four optional bulk-loading support functions and you do specify **RtreeInfo**, put NULL in positions four, five, six, and seven.

If you use the 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 because the DBDK automatically generates 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 shows how to create the **MyShape_ops1** operator class:

```
CREATE OPCLASS MyShape_ops1
FOR RTREE
STRATEGIES (Overlap, Equal, Contains, Within)
SUPPORT (Union, Size, Inter);
```

The strategy functions are called **Overlap**, **Equal**, **Contains**, and **Within**. The support functions are called **Union**, **Size**, and **Inter**.

The following example shows how to create an operator class that also supports bulk loading:

```
CREATE OPCLASS MyShape_ops2
FOR RTREE
STRATEGIES (Overlap, Equal, Contains, Within)
SUPPORT (Union, Size, Inter, SFCbits, ObjectLength, SFCvalue, SetUnion);
```

Note the additional optional bulk-loading support functions **SFCbits**, **ObjectLength**, **SFCvalue**, and **SetUnion**.

The following example shows how to create an operator class that does not support bulk loading but does include the **RtreeInfo** support function:

```
CREATE OPCLASS MyShape_ops3
FOR RTREE
STRATEGIES (Overlap, Equal, Contains, Within)
SUPPORT (Union, Size, Inter, NULL, NULL, NULL, NULL, RtreeInfo);
```

Important: You cannot alter an existing operator class that has only the **Union**, **Size**, and **Inter** support functions defined to add the bulk-loading support functions. Instead, you must create a new operator class to use these support functions for bottom-up building of R-tree indexes.

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

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

Setting Up Nearest-Neighbor Searching

To enable users of a datablade module to perform nearest-neighbor searches, your datablade module must provide one or more strategy functions in your R-tree operator class, which are set up as nearest-neighbor functions.

You need to provide documentation to your users that explains how to perform nearest-neighbor searches.

Setting Up a Strategy Function for Nearest-Neighbor Searching

For each nearest-neighbor strategy function, there must exist a separate distance-measuring function of the same name but with a different signature. The R-tree access method calls *only* the distance-measuring function associated with the strategy function; the strategy function itself should not be called directly. The appearance of the strategy function in a query allows the query planner to set up a scan using the related R-tree index. You must raise an error if a user calls the strategy function directly, with a message such as, "An attempt was made to use the nearest-neighbor function *name* as a filter during a non-index table scan. Nearest-neighbor queries require an index scan."

You must also set up the **RtreeInfo** support function (described in "Support Functions" on page 3-9) to indicate that the strategy function is for nearest-neighbor searches, as "Setting RtreeInfo to Indicate Nearest-Neighbor Functions" on page 3-34 shows.

The Distance-Measuring Function

The distance measuring function is not itself a part of the operator class.

The first and second arguments of the distance function must be the same as the first and second arguments of the strategy function. The third argument must be INTEGER and the return value DOUBLE PRECISION. For example, for the strategy function **Nearest**, created by the following SQL statement:

```
CREATE FUNCTION Nearest(UDT, UDT)
  RETURNS BOOLEAN
  WITH (NOT VARIANT);
```

The associated distance function, **Nearest**, looks like this:

```
CREATE FUNCTION Nearest(UDT, UDT, INTEGER)
  RETURNS DOUBLE PRECISION
  WITH (NOT VARIANT);
```

where *UDT* is a user-defined data type, such as the point data type, **ST_Point**, from the IBM Informix Spatial DataBlade Module.

In C, the distance function declaration looks like this:

```
mi_double_precision *Nearest(UDT *x1,
                             UDT *x2,
                             mi_integer flags,
                             MI_FPARAM *fp);
```

The first two arguments are the objects or locations between which the function calculates the distance (or the bounding-boxes of the objects, as "Distance Function: Using Bounding Boxes" on page 3-34 describes).

The third argument is not used in this version of the R-tree access method.

The DOUBLE PRECISION return value is not interpreted by the R-tree access method.

Distance Function: Using Bounding Boxes

Optionally, you can provide a distance function (paired with a strategy function) that calculates distances between bounding boxes rather than exact distances between objects. The distances calculated this way are imprecise, but the function runs more quickly. For example, the IBM Informix Spatial DataBlade Module provides the **SE_Nearest** and **SE_NearestBBox** functions so that the users can choose whether to run searches using precise or estimated distances.

In this case, set the **RtreeInfo** support function to match the strategy function with the operation key **bbox_only_distance** as the following section, Setting RtreeInfo to Indicate Nearest-Neighbor Functions shows.

Setting RtreeInfo to Indicate Nearest-Neighbor Functions

This C code fragment shows how to set the **RtreeInfo** support function to indicate that a strategy function is a nearest-neighbor function, and that a nearest-neighbor function exists that makes approximate distance calculations. To do this, use the operation keys (*operation_ptr* arguments), **nearest_neighbor_functions**, and **bbox_only_distance**, respectively. You can combine this fragment with the example shown in "C Code Example for the RtreeInfo Support Function" on page 3-16.

For each operation (**nearest_neighbor_functions** and **bbox_only_distance**), if the *answer_ptr* argument is NULL, the function should return either MI_OK or RLT_OP_UNSUPPORTED, depending whether that operation is supported.

If the *answer_ptr* argument is not NULL, it is a pointer to a pointer to an MI_LVARCHAR containing an array of 64 MI_BOOLEANS, one for each strategy function slot (allocated by the caller). For the **nearest_neighbor_functions** operation, the **RtreeInfo** function should fill in either MI_TRUE or MI_FALSE for each entry corresponding to a nearest-neighbor strategy function. For the **bbox_only_distance** operation, the **RtreeInfo** function should fill in MI_TRUE to indicate that the distance function uses bounding-box measurements only or MI_FALSE to indicate that exact calculation distance calculations are required. If the **bbox_only_distance** operation is not supported, the R-tree access method assumes that exact distance calculations are required.

```
...
else if (matches(operation, "nearest_neighbor_functions"))
  {
  /*
  ** Indicate which strategy functions are nearest-neighbor
  ** functions. In this case, the 6th strategy function.
  */
  mi_boolean *answer = NULL;

  if (answer_ptr == NULL)
      goto done; /* Operation is supported */

  /* Memory for 64 booleans is allocated by R-tree */
  answer = (mi_boolean*) mi_get_vardata((mi_lvarchar*)
           mi_get_vardata(answer_ptr));

  answer[0] = MI_FALSE; /* intersect */
  answer[1] = MI_FALSE; /* equal     */
  answer[2] = MI_FALSE; /* contains  */
  answer[3] = MI_FALSE; /* inside    */
  answer[4] = MI_FALSE; /* outside   */
  answer[5] = MI_TRUE;  /* nearest   */

  }

else if (matches(operation, "bbox_only_distance"))
  {
  /*
  ** Indicate which nearest-neighbor distance functions
  ** do their calculation using only bounding box information,
  ** giving an approximate distance. In this case, the 7th
  ** strategy function.
  */
  mi_boolean *answer = NULL;

  if (answer_ptr == NULL)
      goto done; /* Operation is supported */

  /* Memory for 64 booleans is allocated by R-tree */
  answer = (mi_boolean*) mi_get_vardata((mi_lvarchar*)
  (mi_get_vardata(answer_ptr));

  if (answer == NULL)
      {
       status = MI_ERROR;
       goto bad;
      }

  answer[0] = MI_FALSE; /* intersect   */
  answer[1] = MI_FALSE; /* equal       */
  answer[2] = MI_FALSE; /* contains    */
  answer[3] = MI_FALSE; /* inside      */
  answer[4] = MI_FALSE; /* outside     */
  answer[5] = MI_FALSE; /* nearest     */
  answer[6] = MI_TRUE;  /* nearest_bbox*/


  }
```

Creating Registration Scripts for Dependent DataBlade Modules

After you create one or more user-defined data types, an operator class, and other objects, use the DBDK to package all the objects into an installable module.

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

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

The dependency is signaled by importing an interface object provided by the IBM 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 IBM 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 DBDK, 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 IBM 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 BladeSmith project file.

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

For details on how to use BladeSmith, see the *IBM Informix DataBlade Developers Kit User's Guide*. For more information on BladeManager, refer to the *IBM Informix DataBlade Module Installation and Registration Guide*.

Repairing R-tree Indexes After Migrating to a Different Version of a DataBlade Module

After you migrate to a different version of a DataBlade module, you must synchronize R-tree index information and operator class functions for every R-tree index that uses the migrated DataBlade module.

To repair R-tree indexes after migrating a DataBlade module:

1. Determine which R-tree indexes are dependent on the migrated DataBlade module.
2. Run the following command for every affected R-tree index:

   oncheck -u update_capsules *database***:***table***#** *index*

   Identify each index with its name, the name of the table it is based on, and the name of the database containing that table.
3. Restart the database server.

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
- Returning the Coordinates of the Root Bounding Box
- Estimating the Size of an R-Tree Index
- R-Tree Index and Logging
- "Cannot Rename Databases that Use the Secondary Access Method" on page 4-10
- "Drop R-Tree Indexes Before Truncating a Table" on page 4-10
- System Catalogs
- Checking R-Tree Indexes with the oncheck Utility

Performance Tips

This section discusses tips on how to improve 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 a User-Defined Data Type" on page 3-3, which describes performance considerations when designing the user-defined data type of the column that is indexed with an R-tree index.

For other performance issues that are also relevant to R-tree indexes, refer to the *IBM Informix Performance Guide*.

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 than 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 more slowly than if there are no statistics on the indexed column at all.

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

Important: Be sure to always run UPDATE STATISTICS after you load data into a table that has an R-tree index. Without the new statistics, the query optimizer might think the table is small and never 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 you run UPDATE STATISTICS on a column of user-defined type, the Informix server calls the **statcollect()** user-defined routine (if present) to gather statistics. See the *IBM Informix User-Defined Routines and Data Types Developer's Guide* and the *IBM Informix DataBlade API Programmer's Guide* for more information about the **statcollect()** routine.

When you run UPDATE STATISTICS on a column with an R-tree index, the DataBlade module that implements the user-defined type determines how statistics are gathered to assess the cost of using the R-tree index.

If the DataBlade module provides functions to evaluate selectivity and per-row cost, the following formula is used to calculate the cost of using an R-tree index:

Cost = filtering cost + refinement cost + data-access cost

Where:

- filtering cost = selectivity * (number of rows in table/average number of rows per page)

- refinement cost = selectivity * number of rows * per-row cost
- data-access cost = selectivity * number of data pages

This approach assumes that IO cost is significantly greater than the cost of evaluating the filters. See "Selectivity and Cost Functions" on page 3-30 for information about adding selectivity and per-row cost functions.

If the DataBlade module does not provide functions to evaluate selectivity and per-row cost, the cost is set at 50. The documentation for the DataBlade module should state which method is used.

The following statistics are generated when the UPDATE STATISTICS command is executed on a column 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 *IBM 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 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 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. Because 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 that affects 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 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 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 has high 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 faraway lakes do not appear.

An example of data that has low 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. Checking for bounding boxes overlapping a particular place on earth does not eliminate many satellites, unless time can also be used for finer resolution. Airline routes behave similarly.

Clustering Spatial Data on the Disk

If the rows of a table with an R-tree index are clustered on disk the same way as the corresponding entries in the R-tree index that indexes the column, the performance of the retrieval of the data is improved. This section describes how you can cluster existing spatial data on the disk to reflect the ordering in the R-tree index.

Important: Because the following procedure requires that the data in the original table be temporarily deleted, make a backup copy of the table either by loading all the rows into a new table or by taking a full backup of the database.

To cluster existing spatial data on the disk to reflect the ordering in an R-tree index:

1. Create a new table that is exactly the same as the original table and insert all rows from the original table into the new table.

   For example, if the original table is called **circle_tab**, the following SQL statements create an exact copy called **circle_tab_temp** and insert all rows from the **circle_tab** table into the **circle_tab_temp** table:

   ```
   CREATE TABLE circle_tab_temp
   (
       id        INTEGER,
       circles   MyCircle
   ```

```
);

INSERT INTO circle_tab_temp
SELECT * FROM circle_tab;
```

2. Create an R-tree index on the relevant column of the new table.
3. Update statistics on the new table.
4. Drop the R-tree index on the original table and delete all rows.
5. Insert all rows from the new table back into the original table with a SELECT statement that returns all rows in the new table *and* uses the R-tree index at the same time. Be careful that you design this SELECT statement carefully so it satisfies both restrictions.

   You might consider using the **Overlap** strategy function in your query, passing as the second parameter the coordinates of the entire space in which the spatial objects in the table exist. Because each spatial object obviously overlaps with the entire possible space, the query returns every row in the table. In addition, because the **Overlap** strategy function is specified in the WHERE clause of the query, the query must use the R-tree index.

   For example, assume all the spatial objects in the table exist within a box defined by the coordinates `(-1000,-1000,1000,1000)`. In this case, the query might look like the following example:

   ```
   INSERT INTO circle_tab
   SELECT * FROM circle_tab_temp
   WHERE Overlap(circles, 'box(-1000,-1000,1000,1000)::MyBox');
   ```
6. Create a new R-tree index on the appropriate column of the original table.
7. Drop the new table.

If your original table is fragmented, be sure to use the same fragmentation scheme throughout the procedure. In other words, fragment the new table and its index the same way the original table and index are fragmented and make sure that the data is re-inserted into the correct fragment of the original table.

Subsequent updates will gradually degrade the clustering of data achieved with this procedure.

Returning the Coordinates of the Root Bounding Box

The root page of an R-tree index contains the bounding box that encloses all the objects in the index. It is often useful to know the exact coordinates of this bounding box. For example, Step 5 in the procedure described in the section "Clustering Spatial Data on the Disk" on page 4-4 uses this measurement. One common use of this information is to set the scale of a display screen before retrieving data.

To return the coordinates of the root bounding box, use the **rtreeRootBB()** function.

Syntax

The syntax of the **rtreeRootBB()** function is:

```
execute function rtreeRootBB ( index_name, spatial_datatype );
```

The arguments are described in the following table.

Arguments	Purpose
index_name	The name of the R-tree index for which you want to find the coordinates of the root bounding box
spatial_datatype	The spatial data type of the column that is indexed with the R-tree index named *index_name*

If the R-tree index is fragmented, then the **rtreeRootBB()** function returns the union of the root bounding box for each index fragment.

The format of the return value of the **rtreeRootBB()** function is defined by the output function of the specified data type. The output function of the spatial data type is a user-defined routine that specifies how to convert between the internal representation of the data type to its external representation. This output function must be able to display the bounding box of the corresponding data type as well as the data type itself.

Example

Assume the table **circle_tab** contains a column of data type **MyCircle** indexed with an R-tree index called **circle_tab_index5**. To return the coordinates of the root bounding box, execute the following statement:

```
EXECUTE FUNCTION rtreeRootBB ( 'circle_tab_index5' , 'MyCircle' );
```

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 4-6 shows how to estimate index size by performing a series of calculations.
- "Using the oncheck Utility to Calculate Index Size" on page 4-7 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 estimates only 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 appear in index leaf pages.

   If you are indexing a user-defined data type, the size of the key value is the value of the INTERNALLENGTH variable of the CREATE OPAQUE TYPE statement.
2. 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

3. 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
   ```

4. 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 percent because index leaf pages are usually just over half full.

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

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

   leaves = *rows* / *leafpagents*

   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 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 percent full. Your R-tree index might 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 existing 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 **shapes** database, run the following command as user **informix** at the UNIX shell or Windows command prompt:

```
oncheck -pT shapes:circle_tab
```

For more information on the **oncheck** utility, refer to your *IBM Informix Administrator's Guide*.

R-Tree Index and Logging

The R-tree secondary access method uses the extensible log manager of the Informix database server to perform logical logging of its operations. These logical-log records can be used to recover an R-tree index after a database server failure or to abort the R-tree operations after a rollback.

The R-tree secondary access method creates its own logical-log records for only some of the R-tree index operations, in particular:

- Insertion of an item into a leaf page
- Deletion of an item from a leaf page

The R-tree secondary access method does not create its own logical-log records for the following operations; instead, the access method allows the extensible log manager to create the logical-log records:

- Insertion of an item into or deletion from an internal branch page
- Creation of a new page due to split of a page
- Update of the bounding box or other metadata in a page
- Update of the child page of an internal branch page
- Update of the root page number in the root page when a new root page is created

The following R-tree operations are not logged at all:

- CREATE INDEX statement to create an R-tree index
- Any operation on an R-tree index of a temporary table

Description of the R-Tree-Specific Logical-Log Records

As described in the preceding section, the R-tree secondary access method creates its own logical-log records for only two types of R-tree operations: insertion of an item into a leaf page and deletion of an item from a leaf page. For all other logged R-tree operations, the R-tree secondary access method allows the extensible log manager to create the logical-log record. This section describes the format of the two logical-log records created by the R-tree secondary access method.

The first six columns of the R-tree-specific logical-log records are the standard columns displayed for all logical-log records. You can identify these log records as R-tree log records because the third column always has a value of `RTREE`. The R-tree-specific information is contained in the seventh column of the log record. An eighth column is also displayed, although its value is always `0`.

For detailed information about the standard first six columns of logical-log records, refer to the *IBM Informix Administrator's Guide* for your database server.

Logical-Log Records of Insertions of Items into a Leaf Page

The format of the seventh column of the logical-log record of an insertion into an R-tree leaf page is as follows:

```
LEAFINST [page number, base table rowid, base table fragid, delete flag]
```

The following example shows an actual log record of this type displayed with the **onlog** utility:

```
c104     192  RTREE    8        0  c040     LEAFINST [9,257,1048960,0]   0
```

Logical-Log Records of Deletions of Items from a Leaf Page

The format of the seventh column of the logical-log record of a deletion from an R-tree leaf page is as follows:

```
LEAFDEL [page number, base table rowid, base table fragid, delete flag]
```

The following example shows an actual log record of this type displayed with the **onlog** utility:

```
288 192  RTREE    8        0  1c4      LEAFDEL  [39,258,1048960,0]   0
```

Using the onlog Utility to View R-Tree Logical-Log Records

This section describes how you can use the **onlog** utility to view R-tree logical-log records. The following procedure first shows how to force the log manager to start using a new logical log file; this is done for ease of searching the logical log file for R-tree-specific records.

To use the onlog utility to view R-tree log records:

1. Log in as the **informix** user.
2. Execute the following utility at the operating system prompt:

   ```
   onmode -l
   ```

   This utility forces the log manager to switch to the next available logical log.
3. Execute the following utility to find the unique identifier of the logical log file that the log manager will next use:

   ```
   onstat -l
   ```

   In the output of the **onstat** utility, look under the **Logical Logging** heading for the list of logical log files currently in use. Find the log file that has a value of `0` in the **used** column.

   The following sample **onstat** output shows that the logical file with a unique identifier of 11 will be the next logical log file that the log manager uses:

   ```
   address  number   flags    uniqid   begin        size     used    %used

   a13a6a4  1        U-B----  7        100a03      10000      655     6.55
   a13a6c0  2        U-B----  8        103113      10000       62     0.62
   a13a6dc  3        U-B----  9        105823      10000      500     5.00
   a13a6f8  4        U-B---L  10       107f33      10000      197     1.97
   a13a714  5        U---C--  11       10a643      10000        0     0.00
   a13a730  6        U-B----  6        10cd53      10000       57     0.57
   ```
4. Execute SQL commands that manipulate an R-tree index. For example, create a table with a column of a spatial data type and then create an R-tree index on the column.
5. Execute the **onlog** utility, specifying a particular log file with the **-n** option so you can search for R-tree entries.

   For example, the following sample use of the **onlog** utility shows how to view the log file whose unique id is 11 and pipe the output to the UNIX **grep** command to search for the term RTREE:

   ```
   onlog -n 11 | grep RTREE
   ```

   The following sample output shows both log records made by the extensible log manager and log entries made by the R-tree access method:

   ```
   addr     len  type     xid      id link
   5680     436  HINSERT  6        0  5328     600002   102      391
   6050     372  BLDCL    6        0  5680     700002 6 6 2056 0 polyidx
   61c4     36   CHALLOC  6        0  6050     800035   6
   61e8     40   PTEXTEND 6        0  61c4     700002   5        800035
   e4a4     64   HUPDAT   6        0  e460     100056   80e      0        94
   94  2
   e4e4     36   COMMIT   6        0  e4a4     07/23/1999 21:08:30
   f018     40   BEGIN    6        2  0        07/23/1999 21:08:30 12       rk
   f040     932  HINSERT  6        0  f018     100085   101      888
   f3e4     72   HUPDAT   6        0  f040     600002   101      0        812
   812 3
   f42c     140  HINSERT  6        0  f3e4     600002   801      96
   10018    928  RTREE    6        0  f42c     LEAFINS  [802,880,257,1048709 0]
   10074    116  HUPDAT   6        0  f4b8     600002   801      0        96
   96  2
   100e8    932  HINSERT  6        0  10074    100085   102      888
   11018    928  RTREE    6        0  100e8    LEAFINS  [803,880,258,1048709 0]
   ```

```
11048    84   HUPDAT   6        0  1048c    600002   801      0        96
96  3
1109c    932  HINSERT  6        0  11048    100085   201      888
11440    72   HUPDAT   6        0  1109c    700002   101      0        812
812 3
11488    140  HINSERT  6        0  11440    700002   801      96
12018    928  RTREE    6        0  11488    LEAFINS  [802,880,513,1048709 0]
120d0    116  HUPDAT   6        0  11514    700002   801      0        96
96  2
12144    36   COMMIT   6        0  120d0    07/23/1999 21:08:30
```

Cannot Rename Databases that Use the Secondary Access Method

You cannot rename a database if the database has any tables that were created using the primary access method (also known as virtual table interface) or indexes using the secondary access method (also known as virtual index interface). R-tree indexes are implemented using the secondary access method. Therefore, you cannot rename databases with R-tree indexes.

Drop R-Tree Indexes Before Truncating a Table

Before truncating a table that contains an R-tree index, you must drop the R-Tree index.

After you issue a TRUNCATE command, you can re-create the R-tree index.

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        rtree
am_owner       informix
am_id          2
am_type        S
am_sptype      D
am_defopclass  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 *IBM 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 operator 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;sfcbits;objectlength;sfcvalue;setunion;
```

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 **MyShape_ops** operator class also defines the bottom-up building support functions **SFCbits**, **ObjectLength**, **SFCvalue**, and **SetUnion**.

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.3.0/shapes.bld(MyShapeWithin)
```

The result shows that a **Within** function exists in the database for the MyShape data type.

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  rtree_ops
opclassid    2

opclassname  myshape_ops
opclassid    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 the *IBM 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.

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

Checking R-Tree Indexes with the oncheck Utility

The **oncheck** utility is a database server utility that checks and displays information about database server disk structures. You can use all the default options of the **oncheck** utility to check R-tree indexes.

For R-tree indexes, you can use the default **oncheck** options to check that the bounding boxes for each item on a given page are contained in the master

bounding box for the page. You can also check for possible incomplete splits, which can be detected by the **oncheck** utility by comparing some internal information between pages. You can also use the **oncheck** utility to check that the bounding box of a parent entry on a given page matches the bounding box of the child page. Finally, you can check that all leaf pages that have a right sibling contain a right-pointing link that points to the correct leaf page.

When you check an R-tree index with the default options of the **oncheck** utility, the database server takes a shared lock on the index fragment currently being checked.

Important: If you specify the **-u "rtree_cleanup"** option, described later in this chapter, the database server takes an exclusive lock on the index fragment currently being cleaned up.

The following **oncheck** options check and display information for an R-tree index.

Option	Purpose
-ci, -cI	Performs standard index checking with minimal output Both options display the same output.
-pT	Performs some index checking and displays only index summary information
-pk, -pK	Performs index checking of each page in the index Primarily displays internal page information about the root and branch pages, although it also displays minimal information about the leaf pages. Both options display the same output.
-pl, -pL	Similar to **-pk** and **-pK** except that it displays additional information about leaf pages Both options display the same output.
-u *parameter*	Depending on the *parameter* you specify, restricts the checking of an R-tree index to specified levels and pages or performs a cleanup of an R-tree index This option applies to R-tree indexes only. You cannot use this option to check other types of indexes.

For information about the exact syntax of oncheck options, refer to the *IBM Informix Administrator's Guide* for your database server.

Checking Pages with the -ci and -cI Options

The **-ci** and **-cI** options tell the **oncheck** utility to walk through the R-tree structure, checking that the bounding box in the parent page matches the *master* bounding box on the child page for all nonleaf pages. In addition, the utility checks that the master bounding box on each page contains all of the bounding boxes for items on the page. Finally, the utility checks that the right-pointing links point to a valid R-tree page.

The following example shows how to use the **-ci** and **-cI** options:

```
oncheck -cI rtree_db:polygons
```

In the example, the **oncheck** utility is checking any R-tree indexes that exist on the **polygons** table in the **rtree_db** database.

Checking Pages with the -pT Option

The **-pT** option performs standard R-tree index checks and prints out a summary of information about the index for each index fragment. In addition, this option also displays information about the indexed table.

The following example shows how to use the **-pT** option:

```
oncheck -pT rtree_db:polygons
```

The following example shows the type of output that **oncheck -pT** displays:

```
Tree Depth: 3
Internal Pages: 11
Leaf Pages: 125
Empty Pages: 0
Total Pages: 136
Root page items: 10
Leaf Page Tuples: 1000
Internal Page Tuples: 135
Avg. Leaf Page Tuples per Leaf Page: 8.0
Space utilization:
Total Space: 278528
Free Space: 82880
Total Page Overhead: 59028
User Data Space: 136620
User Free Space: 119380
User Page Overhead: 36500
Total: user data: 49.1%, free 29.8%, overhead 21.2%
User Pages Only: data 53.4%, free 42.9%, overhead 14.3%
```

Checking Pages with the -pk and -pK Options

The **-pk** and **-pK** options display detailed information about the root and branch pages in an R-tree index. These options also display minimal information about the leaf pages.

The **-pk** and **-pK** options of the **oncheck** utility display the following type of information about root and branch pages:

- **Level**. The level of the page within the R-tree structure
  The root page is at level 0.
- **Pagenum**. Unique identifier of the page
- **Usage**. The percent of the total space on the page that is currently in use
- **Number of children**. The number of entries on the page
- **Right**. The page number of the right sibling
  If the page does not have a right sibling, then this value is `-1`.
- **Bounding box.** The global bounding box on the page (root page only)
- **Children.** A list of the page's children

The following example shows how to use the **-pK** option:

```
oncheck -pK rtree_db:polygons
```

The following partial example shows the type of output that **oncheck -pK** displays:

```
Node: Level 0, Pagenum 31, Usage  51.2%, No. of Children 10, right -1
X(2.49752E-05,1) Y(-1,1) Z(-1,1) A(any) T(any)
Child 10, Fullness 0x0
X(0.000161568,1) Y(-1,1) Z(-1,1) A(any) T(any)
.
.
.
Node: Level 1, Pagenum 136, Usage  37.7%, No. of Children 7, right -1
Child 104, Fullness 0x0
X(0.0547637,0.73305) Y(-1,-0.670752) Z(-0.583419,0.588895) A(any) T(any)
.
.
.
```

The example shows output for a root page (level 0) and a branch page (level 1).

The example displays only one child for each page; the output for the remaining children is similar.

Checking Pages with the -pl and -pL Options

The **-pl** and **-pL** options display similar information about the root and branch pages as the **-pk** and **-pK** options. In addition, the **-pl** and **-pL** options also display detailed information about the leaf pages in an R-tree index followed by information about the data objects on the leaf page.

The **-pl** and **-pL** options of the **oncheck** utility display the same information listed in "Checking Pages with the -pk and -pK Options" on page 4-14 about the root, branch, and leaf pages. In addition, for each data object on a leaf page, the following information is displayed:

- **size**. The size of the data object in bytes
- **rowid**. The row ID of the data object in the indexed table
- The bounding box of the data object

The following example shows how to use the **-pL** option:

```
oncheck -pL rtree_db:polygons
```

The following example shows the type of output about leaf pages that **oncheck -pL** displays:

```
Node: Level 2, Pagenum 143, Usage  44.3%, No. of Children 5, right -1
Data record on page 143: size 136, rowid 1048992/30467
X(0.893479,1) Y(-0.176591,0.267366) Z(-0.0306181,0.388314) A(any) T(any)
Data record on page 143: size 136, rowid 1048992/16386
X(0.916716,1) Y(-0.399292,0.126833) Z(0.00581815,0.025057) A(any) T(any)
```

The example displays only two of the five children of the leaf page; the output for the remaining children is similar.

Other Options with -u

Use the **-u** option of the **oncheck** utility to restrict the checking of an R-tree index to specific levels or pages. You can also use this option to perform a cleanup of the index. Unlike the other default options of the **oncheck** utility, the **-u** option always takes at least one parameter, enclosed in double quotes. The available parameters are described later on in this section.

You must use the **-u** option of the **oncheck** utility in combination with one of the default options (**-pk**, **-pK**, **-pl**, **-pL**, **-ci**, or **-cI**).

The **-u** option applies to R-tree indexes only. You cannot use this option to check other types of indexes, such as B-tree.

The following table describes the parameters you can specify with the **-u** option of the **oncheck** utility.

Parameter	Description
slevel(*N***)**	Starts checking at the level in the R-tree structure specified by the value of *N* By default, the **oncheck** utility starts checking at level 0 or at the root page.
elevel(*M***)**	Stops checking the R-tree structure after you check level *M* By default, the **oncheck** utility stops checking at the last level of the R-tree structure.
spage(*pg***)**	Starts checking only when a page number matches *pg* By default, the **oncheck** utility starts checking at the root page.
rtree_cleanup	Cleans up an R-tree index Cleaning up an index includes freeing unused pages, tightening bounding boxes, and merging almost-unused pages. If you specify this parameter, the database server takes an exclusive lock on the index fragment currently being cleaned up. You cannot specify any of the other **-u** parameters with the **rtree_cleanup** parameter.

The preceding parameters apply to each fragment. For example, if you specify `-u "spage(5)"`, each fragment is checked starting at page 5, assuming it exists in the fragment.

The following example shows how to use the **-pk** option in combination with the **-u** option to check only those pages in levels 2 or higher in all the R-tree indexes that exist on the **polygons** table in the **rtree_db** database:

```
oncheck -pk -u "slevel(2)" rtree_db:polygons
```

The following example shows how to combine two parameters in the **-u** option to specify where the **oncheck** utility should start and stop checking the R-tree index:

```
oncheck -pk -u "slevel(2),elevel(5)" rtree_db:polygons
```

The following example shows how to perform a cleanup of all R-tree indexes on the **polygons** table:

```
oncheck -pk -u "rtree_cleanup" rtree_db:polygons
```

Appendix A. Shapes3 Sample DataBlade Module

This appendix describes the Shapes3 sample DataBlade module used in the examples in this guide.

Sample DataBlade modules are provided as downloadable examples as part of the IBM Informix Developer Zone at http://www.ibm.com/developerworks/db2/zones/informix/library/samples/db_downloads.html.

The downloadable example provides instructions on how to install the DataBlade module on your database server. It includes the C code used to create the data types and functions that make up the DataBlade module and a description of how the module works. It also provides all the SQL scripts needed to register the DataBlade module in your database.

The first section of this appendix, "Description of the Sample DataBlade Module" on page A-1, describes the data types and operators the sample DataBlade module provides. The second section, "Sample C Code" on page A-3, provides the C code to create the strategy and support functions defined in the operator class. The header file **shape.h** that describes common elements is also included at the end of this appendix.

Description of the Sample DataBlade Module

This section describes the data types, operators, and operator class 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 is the supertype in the type hierarchy and the MyPoint, MyCircle, and MyBox data types are the subtypes.

The following example creates a table called **box_tab** that has a column called **boxes** of data type MyBox:

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

The following INSERT statements show how to insert two 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(10,10,40,40)
```

The following 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,-20,5,9)
```

Operator Class

The sample DataBlade module defines the **MyShape_ops** operator class that you should use when you create R-tree indexes on columns of data type MyBox, MyCircle, and MyPoint.

The sample DataBlade module defines the **MyShape_ops** operator class as follows:

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

The operator class specifies the four required strategy functions (**Overlap**, **Equal**, **Contains**, and **Within**), the three required support functions (**Union**, **Size**, and **Inter**), as well as the four optional bulk-loading support functions (**SFCbits**, **ObjectLength**, **SFCValue**, and **SetUnion**.)

The following example shows how to specify the **MyShape_ops** operator class when you create an R-tree index:

```
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 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 MyShape also implements the same function for the subtypes MyPoint, MyCircle, and MyBox.

This section includes C code for the following objects:

- shape.h Header File
- Overlap Strategy Function
- Equal Strategy Function
- Contains Strategy Function
- Within Strategy Function
- Union Support Function
- Size Support Function
- Inter Support Function
- SFCbits Support Function
- ObjectLength Support Function
- SFCValue Support Function
- SetUnion Support Function

shape.h Header File

```
#ifndef SHAPES_BLADE_H
#define SHAPES_BLADE_H

/*****************************************************************************
**
**  Project:
**
**       Shapes.3.0 DataBlade
**
**  File:
**
**       shape.h
**
**  Description:
**
**       This is the header file for the Shapes DataBlade.
**       It contains constants, structure definitions, and function
**       prototypes.
**
*****************************************************************************/
```

```
#include <mi.h>

/*
 * Convenience typedefs. Saves typing!
 */
typedef mi_double_precision mi_double;
typedef mi_unsigned_char1   mi_uchar;

/*
 * Datablade version. This string is returned by the ShapeRelease UDR.
 */
#define BLADE_VERSION "Shapes DataBlade version 3.0"

/*
 * Data structure version. Also serves as a magic number.
 */
#define SHAPE_VERSION 0x53687033      /* 'Shp3' in ascii hex */

/*
 * Subtype tag definitions.
 */
#define MyPointTag    1
#define MyCircleTag   2
#define MyBoxTag      3
#define MyHeaderTag   4
#define LastTag       4

/*
 * Size of spatial key generated by SFCvalue routine.
 */
#define SPATIAL_KEY_BITS 32

/*
 * Mathematical constants
 */
#define MyEpsilon     0.000001
#define Pi            3.14159265358979323846

/*
 * Tracing-related macros
 */
#define TRACE_CLASS "Shapes"
#define TRACE_LEVEL 20

#define SHAPE_TRACE_ENTER(fn) DPRINTF(TRACE_CLASS, TRACE_LEVEL, ("Enter " #fn))
#define SHAPE_TRACE_EXIT(fn)  DPRINTF(TRACE_CLASS, TRACE_LEVEL, ("Exit  " #fn))
#define SHAPE_TRACE(args)     DPRINTF(TRACE_CLASS, TRACE_LEVEL, args)

/*
 *  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

/*
 * Data structures.
 */

/*
 * The data structures for the supertype (MyShape) and its subtypes
 * (MyPoint, MyBox, MyCircle) all share a common header, called
 * MyShapeHdr. This contains a version number, a tag which indicates
 * what the subtype is, and a bounding box. This structure is also
 * what gets stored in R-Tree internal-node pages, with the tag field
 * set to MyHeaderTag.
 */

typedef struct
```

```
{
    mi_integer    version;
    mi_integer    tag;          /* type of this object */
    mi_double     xmin, ymin;   /* bounding box */
    mi_double     xmax, ymax;
}
MyShapeHdr;

/*
 * Data structures for each subtype's actual geometry data.
 */
typedef struct
{
    mi_double  x;
    mi_double  y;
}
MyPointData;

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

typedef struct
{
    MyPointData  c;            /* center */
    mi_double    r;            /* radius */
}
MyCircleData;


/*
 * MyShape is the structure which contains both the header information
 * and the geometry data; it is the full definition of a shape object.
 */
typedef struct
{
    MyShapeHdr hdr;
    mi_char    data[8];      /* start of subtype geometry data */
}
MyShape;


/*
 * Typedefs for the function dispatch tables.
 */
typedef mi_boolean (*operatorFunction)   (MyShape*, MyShape*);
typedef operatorFunction* functionTable;


/*
 * Function prototypes for the functions in the function dispatch tables.
 */

mi_boolean CircleIBox    (MyShape *obj1, MyShape *obj2);
mi_boolean CircleICircle (MyShape *obj1, MyShape *obj2);
mi_boolean CircleXBox    (MyShape *obj1, MyShape *obj2);
mi_boolean CircleXCircle (MyShape *obj1, MyShape *obj2);
mi_boolean BoxICircle    (MyShape *obj1, MyShape *obj2);
mi_boolean BoxIBox       (MyShape *obj1, MyShape *obj2);
mi_boolean BoxXBox       (MyShape *obj1, MyShape *obj2);
mi_boolean PointXBox     (MyShape *obj1, MyShape *obj2);
mi_boolean PointXCircle  (MyShape *obj1, MyShape *obj2);
mi_boolean PointXPoint   (MyShape *obj1, MyShape *obj2);

mi_boolean Dispatch      (functionTable  tab,
                          mi_boolean     commutative,
                          MyShape       *obj1,
                          MyShape       *obj2);
```

```
/*
 * Function dispatch tables.
 * These are essentially NxN matrices (where N is the number of subtypes),
 * with only the upper diagonal of each matrix filled in.
 */

static operatorFunction intersectTable[] =
{
                    /* PointT   = 1                */
    PointXPoint,    /*                PointT   = 1 */
    PointXCircle,   /*                CircleT  = 2 */
    PointXBox,      /*                BoxT     = 3 */
                    /* CircleT  = 2                */
    NULL,
    CircleXCircle,  /*                CircleT  = 2 */
    CircleXBox,     /*                BoxT     = 3 */
                    /* BoxT     = 3                */
    NULL,
    NULL,
    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 */
};


/*
 * Miscellaneous internal subroutines
 */
mi_lvarchar *MyShapeInCommon    (mi_integer   tag,
                                 mi_lvarchar *text,
                                 MI_FPARAM   *fp);

mi_lvarchar *MyShapeRecvCommon  (mi_integer   tag,
                                 mi_sendrecv *recv_data,
                                 MI_FPARAM   *fp);

void         CheckVersion       (mi_integer   v);


#endif
```

Overlap Strategy Function

```
/*****************************************************************************
**
** Function name:
**
**      MyShapeOverlap
**
** Description:
**
**      Entrypoint for the SQL routine "Overlap (MyShape,MyShape)
**      returns boolean".  This is an Rtree strategy function.
**
** Special Comments:
**
**      Because MyShape and its subtypes are variable length opaque
```

```
**      datatypes, the UDT instances are passed in from the server
**      wrapped in mi_lvarchars.
**
** Parameters:
**
**      mi_lvarchar *in1, *in2   UDT instances to be spatially compared.
**      MI_FPARAM   *fp          UDR function parameter & state info.
**
** Return value:
**
**      mi_boolean               True if the two shapes overlap.
**
*****************************************************************************/

UDREXPORT mi_boolean
MyShapeOverlap (mi_lvarchar *shape1,
                mi_lvarchar *shape2,
                MI_FPARAM   *fp)

{
    mi_boolean  bbox_overlap;
    mi_boolean  retval;
    MyShape *s1 = (MyShape *) mi_get_vardata (shape1);
    MyShape *s2 = (MyShape *) mi_get_vardata (shape2);

    SHAPE_TRACE_ENTER (MyShapeOverlap);

    CheckVersion (s1->hdr.version);
    CheckVersion (s2->hdr.version);

    /*
     * First check if bounding boxes overlap.
     */
    bbox_overlap = (s1->hdr.xmin <= s2->hdr.xmax && s2->hdr.xmin <= s1->hdr.xmax &&
                    s1->hdr.ymin <= s2->hdr.ymax && s2->hdr.ymin <= s1->hdr.ymax);

    /*
     * If bounding boxes do not overlap then it is not possible for
     * the actual shapes to overlap.
     */
    if (!bbox_overlap)
    {
        retval = MI_FALSE;
        goto OverlapDone;
    }

    /*
     * If bounding boxes overlap and one or both of the objects are
     * R-Tree internal nodes there are no actual geometries to test.
     */
    if (s1->hdr.tag == MyHeaderTag || s2->hdr.tag == MyHeaderTag)
    {
        retval = MI_TRUE;
        goto OverlapDone;
    }

    /*
     * Both objects are 'real' objects or objects on R-Tree leaf nodes.
     */

    retval = Dispatch (intersectTable, MI_TRUE, s1, s2);

OverlapDone:

    SHAPE_TRACE_EXIT (MyShapeOverlap);

    return 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_lvarchar *in1, *in2   UDT instances to be compared.
**      MI_FPARAM   *fp          UDR function parameter & state info.
**
** Return value:
**
**      mi_boolean               The comparison result.
**
*********************************************************************/

UDREXPORT mi_boolean
MyShapeEqual (mi_lvarchar *shape1,
              mi_lvarchar *shape2,
              MI_FPARAM   *fp)
{
    /* Call Compare to perform the comparison. */
    return (mi_boolean) (0 == MyShapeCompare (shape1, shape2, fp));
}
```

Contains Strategy Function

```
/*********************************************************************
**
** Function name:
**
**      MyShapeContains
**
** Description:
**
**      Entrypoint for the SQL routine "Contains (MyShape,MyShape) returns
**      boolean".  This is an Rtree strategy function.
**
** Special Comments:
**
**      Because MyShape and its subtypes are variable length opaque
**      datatypes, the UDT instances are passed in from the server
**      wrapped in mi_lvarchars.
**
** Parameters:
**
**      mi_lvarchar *in1, *in2   UDT instances to be spatially compared.
**      MI_FPARAM   *fp          UDR function parameter & state info.
**
** Return value:
**
**      mi_boolean               True if shape2 is completely inside
**                               shape1.  If shape1 is a non-region
**                               subtype (e.g. a point), returns NULL.
**
*********************************************************************/

UDREXPORT mi_boolean
MyShapeContains (mi_lvarchar *shape1,
                 mi_lvarchar *shape2,
```

```
                MI_FPARAM   *fp)

{
    mi_boolean  bbox_overlap;
    mi_boolean  retval;
    MyShape *s1 = (MyShape *) mi_get_vardata (shape1);
    MyShape *s2 = (MyShape *) mi_get_vardata (shape2);

    SHAPE_TRACE_ENTER (MyShapeContains);

    CheckVersion (s1->hdr.version);
    CheckVersion (s2->hdr.version);

    /*
     * If shape1 is a non-region shape (e.g. point) it is not
     * possible for shape1 to contain shape2 so return NULL.
     */
    switch (s1->hdr.tag)
    {
        case MyHeaderTag:
        case MyBoxTag:
        case MyCircleTag:
            break;

        case MyPointTag:
        default:
            mi_fp_setreturnisnull((fp), 0, MI_TRUE);
            retval = MI_FALSE;
            goto ContainsDone;
    }

    bbox_overlap = (s1->hdr.xmin <= s2->hdr.xmax &&
                    s2->hdr.xmin <= s1->hdr.xmax &&
                    s1->hdr.ymin <= s2->hdr.ymax &&
                    s2->hdr.ymin <= s1->hdr.ymax);

    /*
     * If bounding boxes do not overlap then it is not possible for
     * shape1 to contain shape2.
     */
    if (!bbox_overlap)
    {
        retval = MI_FALSE;
        goto ContainsDone;
    }

    /*
     * If bounding boxes overlap, and one or both objects are internal
     * index nodes, we cannot rule out the possibility that objects
     * in the subtree below this node satisfy the spatial test,
     * so return true.
     */
    if (s1->hdr.tag == MyHeaderTag || s2->hdr.tag == MyHeaderTag)
    {
        retval = MI_TRUE;
        goto ContainsDone;
    }

    /*
     * Both objects are actual shapes so perform an exact geometric test.
     * Note operand order is reversed so we can simply use the insideTable.
     */
    retval = Dispatch(insideTable, MI_FALSE, s2, s1);

ContainsDone:

    SHAPE_TRACE_EXIT (MyShapeContains);

    return retval;
}
```

Within Strategy Function

```
/*********************************************************************************
**
** Function name:
**
**      MyShapeWithin
**
** Description:
**
**      Entrypoint for the SQL routine "Within (MyShape,MyShape)
**      returns integer".  This is an Rtree strategy function.
**
** Special Comments:
**
**      Because MyShape and its subtypes are variable length opaque
**      datatypes, the UDT instances are passed in from the server
**      wrapped in mi_lvarchars.
**
** Parameters:
**
**      mi_lvarchar *in1, *in2   UDT instances to be spatially compared.
**      MI_FPARAM   *fp          UDR function parameter & state info.
**
** Return value:
**
**      mi_boolean               True if
**
*********************************************************************************/

UDREXPORT mi_boolean
MyShapeWithin (mi_lvarchar *shape1,
               mi_lvarchar *shape2,
               MI_FPARAM   *fp)

{
    mi_boolean  bbox_overlap;
    mi_boolean  retval;
    MyShape *s1 = (MyShape *) mi_get_vardata (shape1);
    MyShape *s2 = (MyShape *) mi_get_vardata (shape2);

    SHAPE_TRACE_ENTER (MyShapeWithin);

    CheckVersion (s1->hdr.version);
    CheckVersion (s2->hdr.version);

    /*
     * If shape2 is a non-region shape (e.g. point) it is not
     * possible for shape1 to be within shape2 so return NULL.
     */
    switch (s2->hdr.tag)
    {
        case MyHeaderTag:
        case MyBoxTag:
        case MyCircleTag:
            break;

        case MyPointTag:
        default:
            mi_fp_setreturnisnull((fp), 0, MI_TRUE);
            return MI_FALSE;
    }

    bbox_overlap = (s1->hdr.xmin <= s2->hdr.xmax &&
                    s2->hdr.xmin <= s1->hdr.xmax &&
                    s1->hdr.ymin <= s2->hdr.ymax &&
                    s2->hdr.ymin <= s1->hdr.ymax);
    /*
     * If bounding boxes do not overlap then it is not possible for
     * shape1 to be within shape2.
     */
    if (!bbox_overlap)
```

```
    {
        retval = MI_FALSE;
        goto WithinDone;
    }

    /*
     * If bounding boxes overlap, and one or both objects are internal
     * index nodes, we cannot rule out the possibility that objects
     * in the subtree below this node satisfy the spatial test,
     * so return true.
     */
    if (s1->hdr.tag == MyHeaderTag || s2->hdr.tag == MyHeaderTag)
    {
        retval = MI_TRUE;
        goto WithinDone;
    }

    /*
     * Both objects are actual shapes so perform an exact geometric test.
     */
    retval = Dispatch (insideTable, MI_FALSE, s1, s2);

WithinDone:

    SHAPE_TRACE_EXIT (MyShapeWithin);

    return retval;
}
```

Union Support Function

```
/*****************************************************************************
**
** Function name:
**
**      MyShapeUnion
**
** Description:
**
**      This is an R-Tree support function which enables
**      the server to maintain an R-Tree index.  It computes the
**      union of two objects' bounding boxes.
**
** Special Comments:
**
**      Because MyShape and its subtypes are variable length opaque
**      datatypes, the UDT instances are passed in from the server
**      wrapped in mi_lvarchars.
**
** Parameters:
**
**      mi_lvarchar *in1, *in2   UDT instances to be unioned together.
**      mi_lvarchar *out         Resulting union.
**      MI_FPARAM   *fp          UDR function parameter & state info.
**
** Return value:
**
**      mi_integer               MI_OK if success, MI_ERROR if problems.
**
*****************************************************************************/

UDREXPORT mi_integer
MyShapeUnion (mi_lvarchar *shape_in1,
              mi_lvarchar *shape_in2,
              mi_lvarchar *shape_out,
              MI_FPARAM   *fp)
{
    MyShapeHdr *h1;
    MyShapeHdr *h2;
    MyShapeHdr *h3;

    SHAPE_TRACE_ENTER (MyShapeUnion);
```

```
    h1 = (MyShapeHdr *) mi_get_vardata (shape_in1);
    h2 = (MyShapeHdr *) mi_get_vardata (shape_in2);
    h3 = (MyShapeHdr *) mi_get_vardata (shape_out);

    CheckVersion (h1->version);
    CheckVersion (h2->version);

    if (h1 == h2)
    {
        /*
         * This is a 'self-union', which is how the R-Tree determines how
         * big your header structure is.  This situation will occur just
         * once, on the first index insert operation.
         */
        h3->version = SHAPE_VERSION;
        h3->tag     = MyHeaderTag;
        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 R-Tree reuses variables to save memory.
         * This means we have to be careful not to prematurely overwrite
         * any elements of h1 or h2 as we assign values to h3.
         * The following algorithm is safe in this regard.
         */
        h3->version = SHAPE_VERSION;
        h3->tag     = MyHeaderTag;
        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;
    }

    /*
     * Set the size of the mi_lvarchar to tell the R-Tree how
     * big each element to be stored on internal node pages will be.
     * IMPORTANT NOTE: You must do this in every Union() call,
     * not just the first one (where h1 == h2).
     */
    mi_set_varlen (shape_out, sizeof(MyShapeHdr));

    SHAPE_TRACE_EXIT (MyShapeUnion);

    return MI_OK;
}
```

Size Support Function

```
/*******************************************************************************
**
** Function name:
**
**      MyShapeSize
**
** Description:
**
**      This is an R-Tree support function which enables
**      the server to maintain an R-Tree index.  It computes the
**      size of an object's bounding box.
**
** Special Comments:
**
**      Because MyShape and its subtypes are variable length opaque
**      datatypes, the UDT instance is passed in from the server
**      wrapped in an mi_lvarchar.
```

```
**
** Parameters:
**
**      mi_lvarchar *shape       MyShape UDT whose bbox size is to be computed.
**      mi_double   *bbox_size   Return value, size of UDT's bbox.
**      MI_FPARAM   *fp          UDR function parameter & state info.
**
** Return value:
**
**      mi_integer               MI_OK if success, MI_ERROR if problems.
**
*****************************************************************************/

UDREXPORT mi_integer
MyShapeSize (mi_lvarchar *shape,
             mi_double   *bbox_size,
             MI_FPARAM   *fp)
{
    mi_double   length;
    mi_double   width;
    MyShapeHdr *hdr = (MyShapeHdr *) mi_get_vardata (shape);

    SHAPE_TRACE_ENTER (MyShapeSize);

    length = hdr->xmax - hdr->xmin;
    width  = hdr->ymax - hdr->ymin;

    if (length < 0 && width < 0)
    {
        /*
         * No intersection case.
         * R-Tree preceded this Size() call with an Inter() call that
         * detected no intersection between two bounding boxes.
         */
        *bbox_size = 0;
    }
    else
    {
        /*
         * Normal case.
         * Take care to always return a different value as a bounding box
         * expands or shrinks.  The following algorithm (area + extent) will
         * correctly account for zero-width or zero-height bounding boxes.
         */

        *bbox_size =  (length * width) + (length + width);
    }

    SHAPE_TRACE_EXIT (MyShapeSize);

    return MI_OK;
}
```

Inter Support Function

```
/*****************************************************************************
**
** Function name:
**
**      MyShapeInter
**
** Description:
**
**      This is an R-Tree support function which enables
**      the server to maintain an R-Tree index.  It computes
**      the intersection of two objects' bounding boxes.
**
** Special Comments:
**
**      Because MyShape and its subtypes are variable length opaque
**      datatypes, the UDT instances are passed in from the server
**      wrapped in mi_lvarchars.
```

```
**
** Parameters:
**
**      mi_lvarchar *in1, *in2   UDT instances to be intersected.
**      mi_lvarchar *out         Resulting intersection.
**      MI_FPARAM   *fp          UDR function parameter & state info.
**
** Return value:
**
**      mi_integer               MI_OK if success, MI_ERROR if problems.
**
*****************************************************************************/

UDREXPORT mi_integer
MyShapeInter (mi_lvarchar *shape_in1,
              mi_lvarchar *shape_in2,
              mi_lvarchar *shape_out,
              MI_FPARAM   *fp)
{
    MyShapeHdr *h1;
    MyShapeHdr *h2;
    MyShapeHdr *h3;

    SHAPE_TRACE_ENTER (MyShapeInter);

    h1 = (MyShapeHdr *) mi_get_vardata (shape_in1);
    h2 = (MyShapeHdr *) mi_get_vardata (shape_in2);
    h3 = (MyShapeHdr *) mi_get_vardata (shape_out);

    CheckVersion (h1->version);
    CheckVersion (h2->version);

    h3->version = SHAPE_VERSION;
    h3->tag     = MyHeaderTag;

    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.
         * Indicate this by swapping xmin & xmax and ymin & ymax.
         * R-Tree will follow this Inter() call with a Size() call;
         * at that time we will return zero to indicate no intersection.
         * PROGRAMMING TIP:  There are several ways to indicate no
         * intersection.  You might also consider using a flag in
         * the header structure.
         */
        mi_double temp;
        temp = h1->xmax;
        h3->xmax = h1->xmin;
        h3->xmin = temp;
        temp = h1->ymax;
        h3->ymax = h1->ymin;
        h3->ymin = temp;
    }
    else
    {
        /*
         * Bounding boxes of the two shapes do intersect.
         * Like MyShapeUnion, h1 and h3 may both reference the same
         * structure, or h2 and h3 may both reference the same structure.
         * This means we have to be careful not to prematurely overwrite
         * any elements of h1 or h2 as we assign values to h3.
         * The following algorithm is safe in this regard.
         */
        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;
    }
```

```
    SHAPE_TRACE_EXIT (MyShapeInter);

    return MI_OK;
}
```

SFCbits Support Function

```
/*****************************************************************************
**
** Function name:
**
**      MyShapeSFCbits
**
** Description:
**
**      This is an R-Tree support function which enables the
**      server to use a fast method of building an R-Tree index.
**
** Special Comments:
**
**      The SQL function signature for this function is
**      "SFCbits (UDT, pointer)".  This requires an explanation:
**
**      The purpose of the first argument is to provide function signature
**      uniqueness, since you must declare a separate SFCbits function for
**      each subtype in that can participate in the opclass.
**
**      The second argument is declared to be an SQL pointer (i.e. void *);
**      in reality it is a pointer to an integer.  You must not allocate
**      space for this returned value; the server will allocate it for you.
**
** Parameters:
**
**      mi_lvarchar *udt         UDT instance
**      mi_integer  *bits        Returned value, size of spatial key.
**      MI_FPARAM   *fp          UDR function parameter & state info.
**
** Return value:
**
**      mi_integer               MI_OK if success, MI_ERROR if problems.
**
*****************************************************************************/

UDREXPORT mi_integer
MyShapeSFCbits (mi_lvarchar *shape,
                mi_integer  *bits,
                MI_FPARAM   *fp)
{
    SHAPE_TRACE_ENTER (MyShapeSFCbits);

    *bits = SPATIAL_KEY_BITS;

    SHAPE_TRACE_EXIT (MyShapeSFCbits);

    return MI_OK;
}
```

ObjectLength Support Function

```
/*****************************************************************************
**
** Function name:
**
**      MyShapeObjectLength
**
** Description:
**
**      This is an R-Tree support function which enables the
**      server to use a fast method of building an R-Tree index.
**
```

```
** Special Comments:
**
**      The SQL function signature for this function is
**      "ObjectLength (UDT, pointer)".  This requires an explanation:
**
**      The purpose of the first argument is to provide function signature
**      uniqueness, since you must declare a separate ObjectLength function
**      for each subtype in that can participate in the opclass.  In reality
**      the server will pass an lvarchar containing the subtype name,
**      and it will be lower case.
**
**      The second argument is declared to be an SQL pointer (i.e. void *);
**      in reality it is a pointer to an integer.  You must not allocate
**      space for this returned value; the server will allocate it for you.
**
** Parameters:
**
**      mi_lvarchar *typename    Type name of this UDT (e.g. 'MyShape')
**      mi_integer  *maxlen      Returned value, max length of object
**      MI_FPARAM   *fp          UDR function parameter & state info.
**
** Return value:
**
**      mi_integer               MI_OK if success, MI_ERROR if problems.
**
******************************************************************************/

UDREXPORT mi_integer
MyShapeObjectLength (mi_lvarchar *typename,
                     mi_integer  *maxlen,
                     MI_FPARAM   *fp)
{
    mi_char *col_type_name;

    SHAPE_TRACE_ENTER (MyShapeObjectLength);

    col_type_name = mi_lvarchar_to_string (typename);

    if (strcmp (col_type_name, "myshape") == 0)
    {
        /*
         * This is a supertype column.  It could contain any
         * combination of points, boxes, or circles, so return
         * the size of the largest possible subtype.
         */
        *maxlen = sizeof(MyShapeHdr) + sizeof(MyBoxData);
    }
    else if (strcmp (col_type_name, "mypoint") == 0)
    {
        *maxlen = sizeof(MyShapeHdr) + sizeof(MyPointData);
    }
    else if (strcmp (col_type_name, "mybox") == 0)
    {
        *maxlen = sizeof(MyShapeHdr) + sizeof(MyBoxData);
    }
    else if (strcmp (col_type_name, "mycircle") == 0)
    {
        *maxlen = sizeof(MyShapeHdr) + sizeof(MyCircleData);
    }
    else
    {
        mi_db_error_raise (NULL, MI_EXCEPTION,
                           "unknown column type name", (mi_char *) 0);
    }

    SHAPE_TRACE_EXIT (MyShapeObjectLength);

    return MI_OK;
}
```

SFCValue Support Function

```
/*****************************************************************************
**
** Function name:
**
**      MyShapeSFCvalue
**
** Description:
**
**      This is an R-Tree support function which enables the
**      server to use a fast method of building an R-Tree index.
**
** Special Comments:
**
**      The SQL function signature for this function is
**      "SFCvalue (UDT, integer, pointer)". This requires an explanation:
**
**      The purpose of the first argument is to provide function signature
**      uniqueness, since you must declare a separate ObjectLength function
**      for each subtype in that can participate in the opclass. In reality
**      the server will pass an lvarchar containing an array of UDTs.
**
**      The second argument is an integer containing the size of the
**      arrays in the first and third arguments.
**
**      The third argument is declared to be an SQL pointer (i.e. void *);
**      in reality it is a pointer to an array of spatial keys. This
**      array is allocated for you by the server. The array element size
**      will be the size that you returned in the SFCbits support function.
**
** Parameters:
**
**      mi_lvarchar *objects     Array of UDTs, wrapped in an mi_lvarchar.
**      mi_integer  *array_size  Size of arrays.
**      void        *keys        Array of spatial keys to be computed.
**      MI_FPARAM   *fp          UDR function parameter & state info.
**
** Return value:
**
**      mi_integer               MI_OK if success, MI_ERROR if problems.
**
*****************************************************************************/

UDREXPORT mi_integer
MyShapeSFCvalue (mi_lvarchar *objects,
                 mi_integer   array_size,
                 void        *keys,
                 MI_FPARAM   *fp)
{
    mi_unsigned_integer  *key_ptr = (mi_unsigned_integer *) keys;
    mi_lvarchar **shape_array = (mi_lvarchar **) mi_get_vardata (objects);
    mi_integer  i;

    SHAPE_TRACE_ENTER (MyShapeSFCvalue);

    for (i = 0; i < array_size; i++)
    {
#ifdef USE_HILBERT_KEY
        Compute32BitHilbertKey (shape_array[i], &key_ptr[i]);
#else
        Compute32BitMortonKey (shape_array[i], &key_ptr[i]);
#endif
    }

    SHAPE_TRACE_EXIT (MyShapeSFCvalue);

    return MI_OK;
}
```

SetUnion Support Function

```
/*********************************************************************************
**
** Function name:
**
**      MyShapeSetUnion
**
** Description:
**
**      This is an R-Tree support function which enables the
**      server to use a fast method of building an R-Tree index.
**
** Special Comments:
**
**      The SQL function signature for this function is
**      "SetUnion (UDT, integer, pointer)".  This requires an explanation:
**
**      The purpose of the first argument is to provide function signature
**      uniqueness, since you must declare a separate ObjectLength function
**      for each subtype in that can participate in the opclass.  In reality
**      the server will pass an lvarchar containing an array of UDTs.
**
**      The second argument is an integer containing the size of the
**      array in the first arguments
**
**      The third argument is declared to be an SQL pointer (i.e. void *);
**      in reality it is an instance of a 'header' subtype.  A 'header'
**      subtype is the data structure that contains just a bounding box;
**      it is the same thing as the 3rd argument of the Union support
**      function.  If your UDTs are variable length, this UDT instance
**      will be wrapped in an mi_lvarchar.  If your UDTs are fixed length
**      you will get a pointer to the structure itself.  In both cases
**      the server allocates memory for the structure for you.
**
** Parameters:
**
**      mi_lvarchar *objects     Array of UDTs, wrapped in an mi_lvarchar
**      mi_integer  *array_size  Size of array.
**      void        *union       Pointer to resultant union shape.
**      MI_FPARAM   *fp          UDR function parameter & state info.
**
** Return value:
**
**      mi_integer               MI_OK if success, MI_ERROR if problems.
**
*********************************************************************************/

UDREXPORT mi_integer
MyShapeSetUnion (mi_lvarchar *objects,
                 mi_integer   array_size,
                 mi_lvarchar *union_shape,
                 MI_FPARAM   *fp)
{
    mi_lvarchar **shape_array = (mi_lvarchar **) mi_get_vardata (objects);
    mi_integer    i;

    SHAPE_TRACE_ENTER (MyShapeSetUnion);

    MyShapeUnion (shape_array[0], shape_array[0], union_shape, fp);

    for (i = 1; i < array_size; i++)
    {
        MyShapeUnion (shape_array[i], union_shape, union_shape, fp);
    }

    SHAPE_TRACE_EXIT (MyShapeSetUnion);

    return MI_OK;
}
```

Appendix B. Accessibility

IBM strives to provide products with usable access for everyone, regardless of age or ability.

Accessibility features for IBM Informix Dynamic Server

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use information technology products successfully.

Accessibility Features

The following list includes the major accessibility features in IBM Informix Dynamic Server. These features support:

- Keyboard-only operation.
- Interfaces that are commonly used by screen readers.
- The attachment of alternative input and output devices.

Tip: The IBM Informix Dynamic Server Information Center and its related publications are accessibility-enabled for the IBM Home Page Reader. You can operate all features using the keyboard instead of the mouse.

Keyboard Navigation

This product uses standard Microsoft® Windows® navigation keys.

Related Accessibility Information

IBM is committed to making our documentation accessible to persons with disabilities. Our publications are available in HTML format so that they can be accessed with assistive technology such as screen reader software. The syntax diagrams in our publications are available in dotted decimal format.

You can view the publications for IBM Informix Dynamic Server in Adobe Portable Document Format (PDF) using the Adobe Acrobat Reader.

IBM and Accessibility

See the *IBM Accessibility Center* at http://www.ibm.com/able for more information about the commitment that IBM has to accessibility.

Notices

IBM may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created

programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

> © (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. (enter the year or years). All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml.

Adobe, Acrobat, Portable Document Format (PDF), and PostScript are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Index

L

M

N

O

P

Q

R

S

T

U

V

W

Printed in USA

SC23-9436-00