Ingres R-Tree Detailed Design Specification

© 2008 Ingres Corporation

<table>
<thead>
<tr>
<th>Project Name</th>
<th>Ingres R-Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Author</td>
<td>Steve Wonderly</td>
</tr>
<tr>
<td></td>
<td>(<a href="mailto:steve.wonderly@ingres.com">steve.wonderly@ingres.com</a>)</td>
</tr>
<tr>
<td>Last Saved Date</td>
<td>September 17, 2008</td>
</tr>
<tr>
<td>Revision</td>
<td>1.5</td>
</tr>
<tr>
<td>Template Revision</td>
<td>1.3</td>
</tr>
</tbody>
</table>
Change History:

<table>
<thead>
<tr>
<th>Revision Date</th>
<th>Last Revision By</th>
<th>Description of Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>1995/02/22</td>
<td>S. Wonderly</td>
<td>Original document</td>
</tr>
<tr>
<td>1995/03/11</td>
<td>S. Wonderly</td>
<td>Added Algorithms</td>
</tr>
<tr>
<td>1995/04/18</td>
<td>S. Wonderly</td>
<td>Added Locking And Concurrency (B-link tree); cleaned up algorithms</td>
</tr>
<tr>
<td>1996/05/15</td>
<td>S. Wonderly</td>
<td>Removed range_granularity; showed bidp/tidp ahead of MBR; updated Algorithms, esp. Searching and Deletion; added Catalog Information.</td>
</tr>
<tr>
<td>1996/06/28</td>
<td>S. Wonderly</td>
<td>Revised Insert, HandleOverflow, and AdjustTree algorithms to account for logging and recovery.</td>
</tr>
<tr>
<td>2008/09/15</td>
<td>S. Wonderly</td>
<td>Format for Ingres Open Source Community</td>
</tr>
</tbody>
</table>
**Table of Contents**

1 INTRODUCTION.................................................................................................4
   1.1 BACKGROUND .........................................................................................4
   1.2 BACKGROUND PAPERS ........................................................................5

2 REQUIREMENTS.................................................................................................7

3 SPECIFICATION...............................................................................................8
   3.2 HILBERT R-trees ..................................................................................10
   3.3 THE R-TREE STRUCTURE .....................................................................10
   3.4 BASIC PAGE LAYOUT .........................................................................11
   3.5 R-TREE FILE FORMAT .......................................................................13
   3.6 ALGORITHMS .....................................................................................15
   3.7 RECOVERY AND CONCURRENCY ......................................................20
   3.8 MODULE STRUCTURE .......................................................................28
   3.9 CATALOG INFORMATION .................................................................29
   3.10 DATA STRUCTURE DETAIL...............................................................31
1 INTRODUCTION

Spatial data access methods provide an efficient means of storing and retrieving \textit{n-dimensional} points, lines, or polygons. These methods optimize spatial queries, which retrieve all objects that overlap a specified search region or point. R-trees provide the best tradeoff between performance and implementation complexity.

The R-tree is a multi-level tree structure designed to handle \textit{n-dimensional} objects. Much of the R-tree structure and its associated algorithms are similar to the B-tree (balanced tree structure). However, there are some major differences:

1. The B-tree keeps scalar values (keys) in the pages of the tree, whereas the R-tree stores \textit{n-dimensional} values.
2. When searching a B-tree, candidates are found in only one page of the tree, that is, the key ranges are mutually exclusive. In an R-tree, several index pages may contain candidates that overlap the specified search region, because the index page regions may overlap.
3. The page-splitting algorithm for B-trees splits a page on overflow, turning one page into two, a 1-to-2 splitting policy. The R-tree defers the split, waiting until it turns two pages into three, a 2-to-3 splitting policy.

1.1 BACKGROUND

INGRES has four basic storage structures:

HEAP Tuples are in no specific order. There is no rapid method for finding a tuple (unless a secondary index exists). The search starts at the beginning of the table, and every tuple in that table is read. There is one main data page and all the other pages in the table are overflow pages chained to the data page.

HASH Tuples are stored in the file based on a hash algorithm that is applied to the key. This speeds up data retrieval if the search condition is on the key, because the hash algorithm limits the number of main pages that must be searched.

ISAM The Indexed Sequential Access Method structure contains three types of pages: index pages, main data pages and overflow pages. Index pages have information about ranges of values for the key and indicate which main data page holds tuples with that key value. A main data page may have one or more keys on it, but there will never be two separate main data pages for the same key. When a main data page runs out of space, overflow pages are chained to it.

BTREE

This is a dynamic version of ISAM, which has data pages and index pages. The three types of index pages for a B-tree are: \textit{root}, \textit{index} and \textit{leaf}. The \textit{root} is the topmost index page. \textit{Leaf} pages, the bottom index layer, point to data pages instead of to other index pages. The pages of other layers are simply called \textit{index} pages. The data tuples pointed to by the index are always guaranteed to be ordered by key.
The $B$ in B-tree stands for balanced, which means that all searches read the same number of index pages to get to the data, regardless of the value of the key. Searches on very large B-tree tables are incredibly efficient. They may require reading five levels of index pages and the data page, instead of searching 50,000 pages (as in a heap). However, updating a B-tree can be much slower than any other structure, when an index page is full and has to be split.

A B-tree index can exist apart from any data pages as a secondary index. A secondary index can be added to any of the four basic storage structures. The leaf pages of a secondary index point to data pages in the actual table, instead of to the index itself.

This project adds the R-tree as another type of secondary index, not a new storage structure. This implementation approach speeds development time by avoiding several areas of INGRES, such as the modify and create table commands, modifydb, etc. Most of the benefits of the R-tree are still retained. The base table can be a heap or ISAM structure, sorted by Hilbert values, to implement clustering of the data. The addition of an R-tree storage structure can be done in the future.

### 1.2 BACKGROUND PAPERS

Several papers contributed substantially to this design and are mentioned here.

#### 1.2.1 “Requirements for a spatial index in OpenINGRES,” Peter van Oosterom, October 14, 1994.

#### 1.2.1.1 Summary

The spatial index should be similar to other access methods (B-tree, hash, etc.).

- The spatial index should be very efficient in solving range-queries (return all objects that overlap the specified region) for large numbers of objects (for example, 1,000,000 or more polygons).
- Possible to cluster the tuples by the spatial index, that is, the index can be used as a primary index. This means that objects that are close in 2-D space are also close on disk to minimize disk accesses.
- The query optimizer should use the spatial index when generating a query plan.
- Suggested syntax:

  ```
  modify test_table to rtree on location
  create index test_index on test_table location with structure=rtree
  ```

  The spatial index must be generic, that is, work for all spatial data types: point, (long) line, (long) polygon, and box. It could also be used for new user-defined spatial types (for example, arc or spline) and even 3-D types. A flexible solution is to provide:

- Indexes on only 2-D/3-D boxes
- Functional indexes for other types. That is, an index is not put on an attribute of type polygon, but on a function that converts polygon to box. This is also useful for non-spatial applications, for example, two scalar attributes (income, age) could be converted into a type on which the spatial index is defined.
1.2.2 “INGRES 7.x Storage Structures,” Teresa Seputis.

The document describes the storage structures for heap, hash, ISAM, and B-tree. Information from this document is used to ensure that the new R-tree implementation is close to the existing B-tree implementation. [In usilsu49:/devsrc/techdoc/dbms/dbutil/storage_struct.mt]


This document provides a good introduction to and tutorial on R-trees as well as information about the use of Hilbert values for the ordering and splitting of index pages.

1.2.4 “Performance of B-tree Concurrency Control Algorithms,” V. Srinivasan and Michael J. Carey, University of Wisconsin.

This paper reports the performance of various concurrency (locking) algorithms using a detailed simulation model of B-tree operations in a DBMS.
2 REQUIREMENTS

The following are the requirements for the initial release of the Ingres spatial index.

1. The spatial index should be very efficient in solving range-queries (return all objects that overlap the specified region) for large numbers of objects (e.g., 1,000,000 or more objects). The query optimizer must be able to use the spatial index when generating a query plan (strategy).

2. The spatial index will be defined for all 2-D spatial types: point, segment, (long) line, (long) polygon, box, and circle. In addition, the index will function for new user-defined types (UDTs) that provide the needed predicates, specifically the new overlaps, which is logically equivalent to "intersects OR inside."

3. The spatial index will be implemented as an R-tree, specifically the Hilbert R-tree, which outperforms older ones and can achieve a very high packing density.
3 SPECIFICATION

The storage structure and module structure of the R-tree closely parallel the B-tree secondary index and make use of algorithms generally available from the research community. This simplifies program maintenance and takes advantage of reusing code.

An Ingres R-tree is created using the create index statement:

```
create index indexname on table_name(column_name) with
    structure = rtree,
    range = ((min_x, min_y), (max_x, max_y))
```

So the syntax for “create index” in the SQL Reference Manual on page 7-62 is changed to read:

3.1.1 Syntax

```
[exec sql] create [unique] index [schema.]indexname
    on table_name
    (column_name {, column_name})
    [with with-clause]
```

A with-clause consists of the word with followed by a comma-separated list of any of the following items:

- `structure = btree | isam | hash | rtree`
- `key = (columnlist)`
- `fillfactor = n`
- `minpages = n`
- `maxpages = n`
- `leaffill = n`
- `nonleaffill = n`
- `location = (locationname {, locationname})`
- `allocation = n`
- `extend = n`
- `compression [= ([no] key [, [no] data])] | nocompression`
- `[no]persistence`
- `unique_scope = statement | row`
- `range = ((min_x, min_y), (max_x, max_y))`

The following text with the dotted underlining should be added to the “Create Index Statement Parameters” table:
### 3.1.2 Create Index Statement Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>column_name {, column_name}</code></td>
<td>A list of columns from the specified table to be included in the index. If you use the <code>key</code> option, then the columns specified as keys must head this list and must appear in the same order in which they are specified in the <code>key</code> option. If the structure is <code>rtree</code>, then only one column can be named.</td>
</tr>
<tr>
<td><code>structure</code></td>
<td>This option specifies the storage structure of the index. The default is <code>isam</code> if this is not included. If the structure is <code>rtree</code>, then <code>unique</code> cannot be specified.</td>
</tr>
<tr>
<td><code>key</code></td>
<td>Specifies the columns on which the index is keyed. If this is not included, the index is keyed on the columns in the index definition. If the structure is <code>rtree</code>, then only one column can be named.</td>
</tr>
<tr>
<td><code>compression</code></td>
<td>Specifies whether the index key and data are to be compressed. By default indexes are not compressed. If the structure is <code>rtree</code>, then <code>compression</code> cannot be specified. For details about compression, refer to “The Compression Option” on page 7-264.</td>
</tr>
<tr>
<td>`unique_scope = row</td>
<td>statement`</td>
</tr>
<tr>
<td><code>range = ((min_x, min_y), (max_x, max_y))</code></td>
<td>For <code>rtree</code> indexes only: specifies the minimum and maximum values of the indexed column. The values must be the same data type as the indexed column, either <code>integer4</code> or <code>float8</code>. The <code>range</code> parameter must be specified if the structure is <code>rtree</code>.</td>
</tr>
</tbody>
</table>
3.2 HILBERT R-TREES

The performance of the R-tree index depends on how well the spatial objects are clustered in a page. The INGRES R-tree uses a space-filling curve (or fractal), the Hilbert curve, to impose a linear ordering on the data rectangles.

A space-filling curve visits all the points in a grid exactly once and never crosses itself. The Hilbert curve achieves the best clustering among several methods. The following figure and description are from the paper by Kamel and Faloutsos.

The basic Hilbert curve on a $2 \times 2$ grid, denoted by $H_1$, is shown in Figure 1. To derive a curve of order $i$, each vertex of the basic curve is replaced by the curve of order $i-1$, which may be appropriately rotated and/or reflected. The figure also shows the Hilbert curves of order 2 and 3. When the order of the curve tends to infinity, the resulting curve is a fractal, with a fractal dimension of 2.

The path of a space-filling curve imposes a linear ordering on the grid points. Figure 1 shows one such ordering for a $4 \times 4$ grid (curve $H_2$). For example the point $(0,0)$ on the $H_2$ curve has a Hilbert value of 0, while the point $(1,1)$ has a Hilbert value of 2. The Hilbert value of a rectangle is defined to be the Hilbert value of its center.

3.3 THE R-TREE STRUCTURE

The design of the R-tree structure needs to:

- Provide for range-query searches, using the Minimum Bounding Rectangle (MBR) of the index page.
- Support deferred splitting on insertion, using the Hilbert value of the inserted object as the key.

So, for every page of the tree, the R-tree stores (a) the page’s MBR, and (b) the Largest Hilbert Value (LHV) of the objects that belong to its sub-tree. A leaf page contains entries of the form

$$(tidp, \ MBR_{obj})$$

where $tidp$ (tuple ID ptr) is a pointer to the object on the data page and $MBR_{obj}$ is the MBR of the spatial object $(x_{low}, y_{low}, x_{high}, y_{high})$. A non-leaf (index) page contains entries of the form

$$(bidp, \ MBR_{page}, \ LHV)$$

---

Ingres R-Tree Detailed Design Specification

Page 10 of 35
where bidp is a pointer to the sub-tree page, \( MBR_{\text{page}} \) is the MBR that encloses all the children of that page, and \( LHV \) is the largest Hilbert value among the spatial objects enclosed by \( MBR_{\text{page}} \). Figure 2 shows the structure of the R-tree.

**Figure 2. R-tree Structure**

3.4 BASIC PAGE LAYOUT

3.4.1 Logical Page Layout

A storage structure page consists of three sections (see Figure 3):

**fixed header section** – this section contains the fixed portion of the page (page_main through page_next_line).

**page_line_tab** – array of pointers to tuples on the page. This section contains the offset, in bytes, from the start of the page to the start of this tuple. There is an entry in this array for each tuple on the page. This array is where the tid points. A tid has two components: (1) page number (0 to 8,388,607 or 23 bits) and (2) offset in page_line_tab[] (0 to 511 or 9 bits). Therefore, because of size constraints on the tid, a page may never have more than 512 tuples on it. The last page_line_tab array element has the offset to the first free space where another tuple would be placed.

**data tuple section** – this section contains the actual tuple. It is just binary data on the page. ADF (in conjunction with information from iirelation and iattribute) places interpretation (including compression or decompression) on this data.
The fixed header section is at the top of the page. The page_line_tab array starts right after the header section, and grows "down" the page as additional tuples are added. The data tuple section starts at the bottom of the page and grows "up" the page. When the amount of space left between the last data tuple and the last page_line_tab element is smaller than the size required to hold a tuple and page_line_tab entry, the page is full.

### 3.4.2 Physical Page Format

The pages in a disk file all have a basic format, regardless of storage structure (see Figure 4).

- **page_main** (i4) – Main page forward pointer (page number)
- **page_ovfl** (i4) – Next overflow page pointer (page number)
- **page_page** (i4) – Actual page number of this page
- **page_stat** (i2) – Bitmap of page status,
  - DMPP_DIRECT x'0001' – Index page of ISAM
  - DMPP_PRIM x'0002' – Primary page
  - DMPP_OVFL x'0004' – Overflow page
  - DMPP_FULLCHAIN x'0008' – Overflow chain is full
  - DMPP_DATA x'0020' – Data page
  - DMPP_LEAF x'0040' – B-tree/R-tree leaf page
  - DMPP_FREE x'0080' – B-tree/R-tree unused page
  - DMPP_MODIFY x'0100' – Used by buffer manager to keep track of modified pages (meaningful only while page is in memory).
  - DMPP_CHAIN x'1000' – B-tree page on overflow chain
  - DMPP_INDEX x'2000' – B-tree/R-tree index page

---

**Figure 3. Logical Page Layout**

<table>
<thead>
<tr>
<th>HEADER SECTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>page_line_tab[0]</td>
</tr>
<tr>
<td>page_line_tab[2]</td>
</tr>
<tr>
<td>free space</td>
</tr>
<tr>
<td>data tuple 1</td>
</tr>
<tr>
<td>data tuple 0</td>
</tr>
</tbody>
</table>

**Figure 4. Physical Page Format**

<table>
<thead>
<tr>
<th>page_main</th>
<th>page_ovfl</th>
</tr>
</thead>
<tbody>
<tr>
<td>page_page</td>
<td>page_stat</td>
</tr>
<tr>
<td>page_tran_id</td>
<td></td>
</tr>
<tr>
<td>page_log_addr</td>
<td></td>
</tr>
<tr>
<td>page_version</td>
<td>page_next_line</td>
</tr>
<tr>
<td>page_line_tab[2]</td>
<td>...</td>
</tr>
<tr>
<td>data tuple 1</td>
<td></td>
</tr>
<tr>
<td>data tuple 0</td>
<td></td>
</tr>
</tbody>
</table>
• DMPP_ASSOC x‘4000’ – B-tree associated (or current) data page
• DMPP_SPRIG x‘8000’ – B-tree/R-tree parent of leaf page

**page_seq_number** (i2) – The sequence number associated with a deferred update cursor within a transaction. This field is only meaningful to DMF while in memory.
**page_tran_id** (DB_TRAN_ID) – The transaction id that last changed this page. This field is only meaningful to DMF while in memory.
**page_log_addr** (LG_LSN) – The log record address (log sequence number) of the most recently updated record on the page. This field is only meaningful while in memory.
**page_checksum** (i2) – Checksum of all bytes on the page.
**page_next_line** (i2) – Next available line number in table. Also the `page_line_tab[page_next_line]` contains the number of free bytes left on the page.
**page_line_tab[]** (array of i2) – Array of offsets from beginning of page to first byte of the record.

### 3.5 R-TREE FILE FORMAT

The R-tree storage structure consists of several different types of pages.

- **root page** – Highest index page.
- Other levels of **index pages** (depending on size of table) – The sprig page is simply the lowest level of index page. It is possible for the root page to be the sprig page in a small table.
- **leaf page** – Indicates the line and data page containing a tuple with the specific MBR.
- **data page** – Contains the whole tuple in a B-tree, heap, hash, or ISAM table. The R-tree secondary index does not contain any data pages, instead the leaf pages point to the data pages in the actual table.
- **free list page** – Head of free list (a list of pages available for use).
- **free page** – An unused page that can be formatted into any page type.

The **root page** is always on page zero. Other than that, the pages may be in any order. The **page_stat** field indicates the page type.

- root page – DMPP_INDEX (x‘2000’)
- other index pages – DMPP_INDEX (x‘2000’)
- lowest level index (parent of leaf) – DMPP_INDEX (x‘2000’) & DMPP_SPRIG (x‘8000’)
- leaf page – DMPP_LEAF (x‘0040’)
- data page – DMPP_DATA (x‘0020’) & DMPP_ASSOC (x‘4000’) & DMPP_PRIM (x‘0002’)
- free list page – DMPP_INDEX (x‘2000’)
- free page – DMPP_FREE (x‘0080’)

The data on a non-leaf index page is formatted differently than on a leaf page or a data page.
INDEX PAGE – A B-tree index page contains pairs of entries. Each pair—a key value and a page pointer—constitutes a range for a key. If the search key falls in that range, then the pointer on the lower end of that range indicates which page to use. In contrast, the entries in an R-tree index page consist of three parts—an MBR to define the search range, a pointer, and the LHV. If a search rectangle overlaps one or more MBRs, then the pointer(s) indicate which page(s) to use. When inserting a new entry, the LHV values establish an ordering that ensures good clustering among the pages.

LEAF PAGE (see Figure 5) – A B-tree leaf page contains a tid for each key value. That tid contains two pieces of information—page number and line number (index into page_line_tab array of the data page). Both pieces of tid information are used to find the tuple on the data page. Similarly, an R-tree index page contains a tid for each MBR value.

**Figure 5. Format of a leaf page**

DATA PAGE – Since there are no data pages in a secondary index, the leaf pages of an R-tree will point to data pages in the base table.
3.6 ALGORITHMS

3.6.1 Introduction

A B-tree:
• splits a page on overflow
• turns one page into two, a 1-to-2 splitting policy.

A Hilbert R-tree:
• defers the split, overflowing first onto siblings
• turns two pages into three, a 2-to-3 splitting policy
• or can have an s-to-(s + 1) splitting policy.

The splitting policy affects both space utilization and disk access speed (see Table 1).

Table 1. Effect of split policy on space and cost

<table>
<thead>
<tr>
<th>Split Policy</th>
<th>Space utilization</th>
<th>Disk accesses per insertion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-to-2</td>
<td>65.5%</td>
<td>3.23</td>
</tr>
<tr>
<td>2-to-3</td>
<td>82.2%</td>
<td>3.55</td>
</tr>
<tr>
<td>3-to-4</td>
<td>89.1%</td>
<td>4.09</td>
</tr>
<tr>
<td>4-to-5</td>
<td>92.3%</td>
<td>4.72</td>
</tr>
</tbody>
</table>

3.6.2 An Empty R-Tree

When an R-tree index is created, it has one root page (which doubles as an index page), one leaf page, one free header page (with no entries on the free page list), and one free map page.

Initially, the R-tree root page points to a leaf page with both x and y ranges of \(-\infty\) to \(+\infty\), which is represented via \(bt\_sequence[-2]\) and \(bt\_sequence[-1]\).

3.6.3 Searching

Start from the root and descend the tree, examining all pages that overlap the query rectangle. At the leaf level, report the first entry that overlaps the query window.

Algorithm Search (page \(N\), MBR \(w\))
/* positions the index to the first entry whose MBR overlaps the query window \(w\) */
S1. Search non-leaf pages:
   If page \(N\) is the root page,
      \(Lvl = 0\); otherwise \(Lvl++\).
   Invoke Search for first entry whose MBR overlaps the query window \(w\).

S2. Search leaf pages:
   Position at first entry that overlaps the query window \(w\) as candidate.
   Save Page[\(Lvl\)], Slot[\(Lvl\)], \(Lvl\) in the Record access Control Block (RCB).
Algorithm GetNext ( )
/* starting at the current position, returns the next qualifying entry and saves the new position */
G1. Find next qualifying entry on page:
   Slot[Lvl]++
   If Slot[Lvl] is a valid slot on Page[Lvl],
      if Entry[Slot[Lvl]] qualifies,
         if at Leaf page,
            return Leaf page;
      else
         PushStack,
         repeat from G1.
G2. Past end of slots--pop up a level:
   Save Page[Lvl] as LastPage.
   PopStack.
   at end: return EndOfData.
   If Entry[Slot[Lvl]] still points to LastPage,
      repeat from G1.
G3. Index page modified--find entry pointing to previous position:
   Slot[Lvl]++.
   If Slot[Lvl] is valid,
      if Entry[Slot[Lvl]] points to LastPage,
         repeat from G1,
      else
         repeat from G3.
G4. Index split occurred--look on next index page:
   If Page[Lvl].Next = 0,
      return Error--entry is lost.
   Save Page[Lvl].Next as SaveNext.
   Unfix Page[Lvl].
   Page[Lvl] = SaveNext.
   Fix Page[Lvl].
   Slot[Lvl] = -1
   Repeat from G3.

Algorithm PushStack ( )
/* pushes current page’s information on a stack in the RCB, unfixes current page, uses current index entry to fix the appropriate child page */
Pu1. Push current page’s information on the stack:
   Slot[Lvl+1] = -1.
   Unfix Page[Lvl].
   Lvl++.
   Fix Page[Lvl].
Algorithm PopStack()

/* Pops current page’s information from the stack in the RCB, unfixes current index page, fixes parent page. If current page is a leaf page, leave it locked for “serializable” isolation level. */

Po1. Pop current page from the stack:
    If Page[Lvl] is not a Leaf page or isolation level is “dirty read”,
        unfix Page[Lvl].
    Lvl--.
    Fix Page[Lvl].

3.6.4 Insertion

To insert a new spatial object $o$ with minimum bounding rectangle $r$ in the R-tree, the Hilbert value $h$ of the center of the new MBR is used as a key. In each index level, choose the page with the smallest LHV greater than $h$. If the leaf page is full, HandleOverflow pushes entries onto a sibling or splits to create a new page. AdjustTree is called to propagate the split or overflow to the parent level. Finally, Insert inserts the object into the chosen page.

As with B-tree, R-tree recovery does not back out a page split once it is complete. R-tree logs a split operation within a mini-transaction. When the split is complete, the mini-transaction is ended and locks on the pages are released. This method enhances concurrency within the R-tree index nodes.

Algorithm Insert: dm2r_put(object $o$)

/* inserts a new object $o$ with MBR $r$ in the Hilbert R-tree. $h$ is the Hilbert value of the object’s mbr. */

I1. Find the appropriate leaf page: dm1m_allocate($o$, $h$)
    Find the mbr $r$ of the object $o$.
    Calculate the Hilbert value $h$ of the centroid of $r$.
    Invoke ChooseLeaf: dm1m_search($o$, $h$) to select a leaf page $N_0$
    in which to place $o$.

I2. Make sure leaf page has room:
    If $N_0$ is full, invoke HandleOverflow($N_0$, $BM$), which will return new
    leaf if split was inevitable and the Log Sequence Number of the
    Begin Mini-transaction, $BM$.

I3. Propagate changes upward:
    Form a set $N$ that contains $N_0$, its cooperating siblings $N_1...N_{s-1}$ (if overflow),
    and the new leaf $N_s$ (if split occurred).
    Invoke AdjustTree($N$, $o$).

I4. Grow tree taller:
    If page split propagation caused the root to split,
    create a new root whose children are the two resulting pages,
    adjust the MBR and LHV of the root page and unlock it.
I5. **Insert the object:** \texttt{dm1m_put(object o, leaf page \(N_i\))}
   
   Insert object \(o\) into the page being updated, \(N_i\)
   
   Adjust the MBR and LHV of page \(N_i\)
   
   [If using record-level locking,]
   
   lock the record for object \(o\) and unlock leaf page \(N_i\)]
   
   If MBR or LHV of page changed,
   
   invoke \texttt{AdjustTree(\(N_i\))} to adjust MBR/LHV of parent.

Algorithm **ChooseLeaf:** \texttt{dm1m_search(object o, int \(h\))}

/* returns the leaf page in which to place a new object \(o\) */

C1. **Initialize:**
   
   Set \(N_0\) to be the root page.

C2. **Leaf check:**
   
   If \(N_0\) is a leaf, return \(N_0\).

C3. **Choose sub-tree:**
   
   If \(N_0\) is a non-leaf page, choose the entry \((bidp, MBR, LHV)\)
   
   with the minimum \(LHV\) value greater than or equal to \(h\).

C4. **Descend until a leaf is reached:**
   
   Set \(N_0\) to the page pointed to by \(bidp\) and repeat from C2.

### 3.6.5 Overflow handling

The overflow handling algorithm treats the overflowing pages either by moving some of the entries to one of the \(s-1\) cooperating siblings or by splitting \(s\) pages to \(s+1\) pages.

Algorithm **HandleOverflow** (page \(N_0\), LG\_LSN \(BM\))

/* return the new page if a split occurred */

H1. Lock the \(s-1\) cooperating siblings to search for space.
   
   Begin a mini-transaction by calling \texttt{dm0l_bm(BM)}.

H2. Let \(E\) be a set that contains all the entries from \(N_0\)
   
   and its \(s-1\) cooperating siblings \(N_1,\ldots,N_{s-1}\).

H3. If at least one of the \(s-1\) cooperating siblings is not full,
   
   write “before images” of the cooperating siblings,
   
   distribute \(E\) evenly among the \(s\) pages according to the Hilbert values.

H4. If all the \(s-1\) cooperating siblings are full,
   
   create a new page \(N_s\) to the right of the rightmost cooperating sibling,
   
   write “before images” of the cooperating siblings,
   
   distribute \(E\) evenly among the \(s+1\) pages according to the Hilbert values.

H5. Write “after images”
   
   If a new page was created,
   
   return \(N_s\).
Algorithm AdjustTree(set \( N \), object \( o \))
/* \( N \) is a set of pages that contains the page being updated \( N_0 \), its cooperating siblings \( N_1...N_{s-1} \) (if overflow has occurred), and newly created page \( N_s \) (if split has occurred).

The routine ascends from leaf level towards the root, adjusting MBR and LHV of pages that cover the pages in \( N \) siblings. It ends the current mini-transaction, unlocks appropriate pages, and propagates any splits. */

A1. If reached root level, stop.
A2. Propagate page split upward
   Let \( N_i \) be the page in \( N \) where object \( o \) should be placed, by Hilbert value.
   Lock \( P_0 \), the parent page of \( N_0 \).
   Convert locks on the pages in \( N \)—except \( N_i \)—from exclusive to shared.
   If \( N_0 \) has been split:
      let \( N_s \) be the new page and let \( P_n \) be the parent of \( N_s \),
      if \( P_n \) is full, invoke \HandleOverflow\( (P_n, BM) \),
      insert \( N_i \) into \( P_n \) or one of its siblings according to its Hilbert value.
      if \( P_n \) is split, let \( P_s \) be the new page.
A3. Adjust the MBRs and LHVs in the parent level:
   Let \( P \) be the set of parent pages for the pages in \( N \).
   Adjust MBRs and LHVs of the entries in \( P \) corresponding to the pages in \( N \).
   If a mini-transaction \( BM \) was started,
      end it by calling \dm0L_em\( (BM) \) and unlock the pages in \( N \)—except \( N_i \).
A4. Move up to next level:
   Let \( N \) become the set of parent pages \( P \), with \( N_s = P_s \), if \( P_n \) was split.
   Repeat from A1.

3.6.6 Deletion

Because of the static nature of spatial data, the best algorithm for deletion is to leave empty R-tree pages, instead of collapsing them. This is consistent with the INGRES B-tree implementation. The index tree can be rebuilt if too many pages are left empty because of deletions. The B-link locking algorithm adopted makes this a very good solution.

Algorithm Delete (object \( o \)):
D1. Find the host leaf:
   Perform an exact match search (by Hilbert value and TID pointer) to find the
   leaf page \( L \) that contains \( o \).
D2. Delete \( o \):
   Remove \( o \) from page \( L \) and adjust the page’s MBR.
   Note that if \( o \) is the highest entry in the page, we do not modify the LHV of the page, only
   its MBR. This approach tends to re-insert a deleted record onto the same page it
   occupied, rather than it spilling onto the next page, which could cause more page splits
   toward the right and more empty pages toward the left.
D3. If \( L \) underflows:
   Leave \( L \) empty.
Note that the research papers indicate that an R*tree page should “borrow” entries from cooperating siblings when underflow occurs and should merge $s + 1$ pages to $s$ pages when all cooperating siblings underflow. However, the R-link locking mechanism makes this extremely difficult, if not impossible, without locking an entire branch of the tree, from the root down. We believe it is far better to allow empty pages to remain in the R-tree index.

D4. Adjust MBR in parent levels:
   Form a set $S$ that contains $L$.
   Invoke $\text{AdjustTree}(S)$.

### 3.7 RECOVERY AND CONCURRENCY

#### 3.7.1 Introduction

Recovery and concurrency are inextricably linked. The algorithms have been reworked several times in order to satisfy the needs of both high concurrency and recoverability. A method similar to the B-link locking algorithm is used to provide a high level of concurrency in the Ingres R-tree. Recovery of the R-tree uses a combination of logical logging and physical logging. The locking and logging methods are described below along with an example.

#### 3.7.2 The B-link locking algorithm

A B-link tree is a modification of the B-tree that uses sideways links to chain pages together at all levels of the tree, not just the leaf level. A page in a B-link tree contains a high key and a link to the right sibling. The link allows a page split to occur in two phases: a half-split, followed by the insertion of the index entry into the parent page. After a half-split and before the parent page has been updated, the new page is reachable through the right link of the old page. Operations arriving at a newly split page with a search key greater than the high key use the link to get to the appropriate page, a link chase.

A reader descends the tree from the root to the leaf using shared (S) locks. At each page, the next page to be searched can be either a child or the right sibling of the current page if a half-split has taken place (the search key is greater than the high key on the page). A reader releases the lock on a page before getting a lock on the next page (no lock-coupling).

An updater behaves like a reader until it reaches the appropriate leaf page. Then it releases the S lock and gets an exclusive (X) lock on the same leaf. After getting the X lock, the updater may either find that the leaf is the correct one to update or that it has to perform one or more link-chases to get to the correct leaf. The updater uses X locks while performing link chases, releasing the lock on a page before asking for the next.

If a page split is necessary, the updater performs a half-split while holding an X lock. Then it converts the X lock to an S lock and holds the S lock while acquiring an X lock on the parent page (lock-coupling on the way up).
3.7.3 The Hilbert R-link algorithm

The B-link tree algorithm can be adapted to work with the Hilbert R-tree. A reader behaves like a reader using the B-link algorithm, that is, releasing a lock on a page before getting a lock on its descendant—no lock-coupling.

An R-tree updater behaves like a reader until it reaches the appropriate leaf page. It gets an exclusive (X) lock on the leaf instead of a shared lock. After getting the X lock, the updater may either find that the leaf is the correct one to update or that it has to perform one or more link-chases to get to the correct leaf. The updater uses X locks while performing link chases, releasing the lock on a page before asking for the next.

If a page split is necessary, the updater performs a half-split while holding X locks on the leaf pages involved. It holds the X locks while acquiring an X lock on the parent page (lock-coupling on the way up).

The R-link implementation cannot convert the leaf page X locks to S locks before getting the X lock on the parent, the way the B-link algorithm does. R-tree must hold the X lock in order to prevent a reader—using “overlaps” tests on MBRs—from reaching a leaf page where a half-split has just occurred before the parent is updated. Otherwise, the reader could pop up to the parent page and could miss an overlapping object, because the parent does not yet point to the leaf page containing it. A reader using “overlaps” tests on MBRs cannot tell when a leaf split has occurred, so it does not do a chain-chase. Instead it pops up to the parent and continues testing the next entry to see if it overlaps.

3.7.4 Logging and Recovery

Recovery of the R-tree uses a combination of logical logging—for puts and deletes—and physical logging of before and after images—for page splits. The usage of logical logging reduces the size and amount of log records for most updating. The recovery code for insert and delete is very straightforward and looks exactly like the corresponding B-tree method. However, recovery code for doing page splits is quite complicated and, since page splits are infrequent compared to other updates, the R-tree uses physical logging of before and after images.

Like B-tree, R-tree uses mini-transactions to accomplish logging of page splits. When a page split is complete, the mini-transaction is ended. Thus, a page split is not backed out during undo recovery once it is complete. This allows early releasing of locks on index pages and, thus, higher concurrency. R-tree splits occur in a bottom-up fashion, starting when a leaf page is full and propagating upward to the root, splitting the root page if necessary. For a B-tree, this method of splits is relatively straightforward, with a mini-transaction used for each level of the tree.

An R-tree has an additional requirement: to update the MBRs of the index pages after adding a new object to the leaf page. So the bottom-up method of splitting pages is used first and then another pass of the R-tree must be made to update MBRs of index pages. Thus, some or all of the pages in a path from the leaf to the root may be locked and unlocked more than once, but no better alternative seems possible.

The general method of propagating a split and adjusting the tree is:
1. While holding locks at the leaf level, start a mini-transaction and write before images to the log.
2. Create a new page if necessary, redistribute the entries, and write after images.
3. Lock the parent level and convert X locks to S locks on the child level, except for the page that will hold the object being inserted.
4. Replace entries in the parent to account for the updated pages.
5. Insert an entry in the parent for the newly split page, splitting the parent page first if necessary.
6. Commit the mini-transaction and release the S locks at the child level.

If the parent page overflowed, repeat the split process at the parent level. After the split is complete, to the root if necessary, add the object:

7. Add the object to the leaf page and update the MBR of the page.
8. If the MBR of the page changes, adjust its MBR and repeat the process at the parent level, with the key of the leaf page as the object to be replaced in the parent.

Note that steps 1–6 are inside the mini-transaction and steps 7–8 are outside. During recovery, an undo skips the updates inside the mini-transaction—the page split is not undone. However, if the transaction that initiated the object’s insert is not complete, the updates of steps 7–8 are backed out.

An example of the R-link locking mechanism and the logging method follows.

3.7.4.1 Insert into leaf page

This example uses the R-tree pictured below in Figure 6. Assume that a data tuple entry is to be inserted into page 4 (because its Hilbert value lies within page 4’s LHV) and assume that pages 4 and 5 are full. The Insert, HandleOverflow, and AdjustTree algorithms are reproduced here interspersed with a description of the locking that takes place (shown inside boxes). The example assumes a 2-to-3 splitting policy throughout.
Algorithm Insert: dm2r_put (object o)

I1. Find the appropriate leaf page: dm1m_allocate(o, h)

Find the mbr r of the object o.
Calculate the Hilbert value h of the centroid of r.
Invoke ChooseLeaf: dm1m_search(o, h) to select a leaf page N₀ in which to place o.

S-0 (shared lock on 0), search page 0 by Hilbert, R-0 (release lock on 0)
S-2, search page 2, R-2
X-4 (exclusive lock on 4), search page 4

I2. Make sure leaf page has room:

If N₀ is full, invoke HandleOverflow(N₀, BM), which will return new leaf if split was inevitable and the Log Sequence Number of the Begin Mini-transaction, BM.

Algorithm HandleOverflow (page N₀, LG_LSN BM)

H1. Lock the s − l cooperating siblings to search for space.

Begin a mini-transaction by calling dm0l_bm(BM).

X-5

Begin Mini-transaction
H2. Let $E$ be a set that contains all the entries from $N_0$ and its $s - 1$ cooperating siblings $N_1...N_{s-1}$.

H3. If at least one of the $s - 1$ cooperating siblings is not full, write “before images” of the cooperating siblings, distribute $E$ evenly among the $s$ pages according to the Hilbert values.

H4. If all the $s - 1$ cooperating siblings are full, create a new page $N_s$ to the right of the rightmost cooperating sibling, write “before images” of the cooperating siblings, distribute $E$ evenly among the $s + 1$ pages according to the Hilbert values.

H5. If a new page was created, return $N_s$.

I3. (resumed) Propagate changes upward:
Form a set $N$ that contains $N_0$, its cooperating siblings $N_1...N_{s-1}$ (if overflow), and the new leaf $N_s$ (if split occurred).
Invoke \texttt{AdjustTree(N, o)}

Create page 7 (X-7), chain pages 5 $\rightarrow$ 7 $\rightarrow$ 6
Write “before images” for pages 4, 5, and 7
Split entries by Hilbert among pages 4, 5, and 7; update LHV's and MBRs
Algorithm AdjustTree(set $N$, object $o$)

A1. If reached root level, stop.

A2. Propagate page split upward
   Let $N_i$ be the page in $N$ where object $o$ should be placed, by Hilbert value.
   Lock $P_o$, the parent page of $N_o$.
   Convert locks on the pages in $N$—except $N_o$—from exclusive to shared.
   If $N_o$ has been split:
      let $N_i$ be the new page and let $P_o$ be the parent of $N_{s-1}$,
      if $P_o$ is full, invoke HandleOverflow($P_o, BM$), ...

Algorithm HandleOverflow (page $N_o$, LG_LSN $BM$)

H1. Lock the $s-1$ cooperating siblings to search for space.
   Begin a mini-transaction by calling dm0l_bm($BM$).

X-2, convert X-7 to S-7 and X-5 to S-5

H2. Let $E$ be a set that contains all the entries from $N_o$
    and its $s-1$ cooperating siblings $N_1...N_{s-1}$.

H3. If at least one of the $s-1$ cooperating siblings is not full,
   write “before images” of the cooperating siblings,
   distribute $E$ evenly among the $s$ pages according to the Hilbert values.

H4. If all the $s-1$ cooperating siblings are full,
   create a new page $N_i$ to the right of the rightmost cooperating sibling,
   write “before images” of the cooperating siblings,
   distribute $E$ evenly among the $s + 1$ pages according to the Hilbert values.

Create page 8 (X-8)
Write “before images” of pages 2, 3, and 8
Split entries by Hilbert among pages 2, 3, and 8; update LHV s and MBRs

H5. If a new page was created,
   return $N_i$.

A2. (resumed) Propagate page split upward
   insert $N_i$ into $P_o$ or one of its siblings according to its Hilbert value.
   if $P_o$ is split, let $P_o$ be the new page.

Insert entry for page 7 into page 3 and adjust MBR of page
A3. Adjust the MBRs and LHVs in the parent level:
Let $P$ be the set of parent pages for the pages in $N$.
Adjust MBRs and LHVs of the entries in $P$ corresponding to the pages in $N$.
If a mini-transaction $BM$ was started, end it by calling $\text{dm01_em}(BM)$ and unlock the pages in $N$—except $N_i$.

Update MBR and LHV for entries 4, 5, and 7 in pages 2 and 3
End mini-transaction; R-7, R-5

A4. Move up to next level:
Let $N$ become the set of parent pages $P$, with $N_s = P_s$, if $P_n$ was split.
Repeat from A1.

A1. If reached root level, stop.

A2. Propagate page split upward:
Let $N_i$ be the page in $N$ where object $o$ should be placed, by Hilbert value.
Lock $P_0$, the parent page of $N_0$.
Convert locks on the pages in $N$—except $N_i$—from exclusive to shared.
If $N_0$ has been split:
let $N_s$ be the new page and let $P_n$ be the parent of $N_{s-1}$,
if $P_n$ is full, invoke $\text{HandleOverflow}(P_n, BM)$, ...

X-0, convert X-8 to S-8, X-3 to S-3, and X-2 to S-2
Algorithm HandleOverflow (page $N_0$, LG_LSN BM)

H1. Lock the $s - l$ cooperating siblings to search for space.
   Begin a mini-transaction by calling dm01_bm(BM).

   | (No siblings to search)          |
   | Begin Mini-transaction           |

H2. Let $E$ be a set that contains all the entries from $N_0$
   and its $s - l$ cooperating siblings $N_1...N_{s-1}$.

H3. If at least one of the $s - l$ cooperating siblings is not full,
   write “before images” of the cooperating siblings,
   distribute $E$ evenly among the $s$ pages according to the Hilbert values.

H4. If all the $s - l$ cooperating siblings are full,
   create a new page $N_s$ to the right of the rightmost cooperating sibling,
   write “before images” of the cooperating siblings,
   distribute $E$ evenly among the $s + 1$ pages according to the Hilbert values.

   | Create page 9 (X-9), chain pages 0 → 9  |
   | Write before image of page 0           |
   | Split entries by Hilbert between pages 0 and 9; update LHVs and MBRs |

H5. If a new page was created,
   return $N_s$.

A2. (resumed) Propagate page split upward
   insert $N_i$ into $P_s$ or one of its siblings according to its Hilbert value.
   if $P_n$ is split, let $P_s$ be the new page.

   | Add new entry for page 8 into page 9 and adjust MBR of page |

A3. Adjust the MBRs and LHVs in the parent level:
   Let $P$ be the set of parent pages for the pages in $N$.
   Adjust MBRs and LHVs of the entries in $P$ corresponding to the pages in $N$.
   If a mini-transaction BM was started,
   end it by calling dm01_em(BM) and unlock the pages in $N$—except $N_i$.

   | Update MBR and LHV for entries 2, 3, and 8 in pages 0 and 9 |
   | End mini-transaction; R-8, R-3, R-2 |

A4. Move up to next level:
   Let $N$ become the set of parent pages $P$, with $N_s = P_s$; if $P_n$ was split.
   Repeat from A1.

A1. If reached root level, stop.
14. (resumed) Grow tree taller:
   If page split propagation caused the root to split,
   create a new root whose children are the two resulting pages,
   adjust the MBR and LHV of the root page and unlock it.

   Create a new page 10 (X-10), copy page 0 → 10
   Re-initialize page 0, add entries for pages 10 and 9
   End Mini-transaction, R-9, R-10, R-0

15. Insert the object: \( \text{dm1m\_put}(\text{object } o, \text{ leaf page } N_i) \)
   Insert object \( o \) into the page being updated, \( N_i \)
   Adjust the MBR and LHV of page \( N_i \)
   [If using record-level locking,
   lock the record for object \( o \) and unlock leaf page \( N_i \)]
   If MBR or LHV of page changed,
   invoke \( \text{AdjustTree}(N_i) \) to adjust MBR/LHV of parent.

   Insert \( A' \) into page 4
   Recalculate MBR and LHV of page 4, reflecting \( A' \)

3.8 MODULE STRUCTURE

As stated earlier, the R-tree module structure closely parallels the B-tree, for easier
maintenance and reuse of code.

R-tree source will be added in back/dmf/dmp. The description of the routines and
parameters are listed in the appendix.

\textbf{dm1m.c} – Routines to access and update tables using R-trees. \( [\text{The m is for multi-
dimensional or mbr (minimum bounding region)}] \)

The external routines defined in this file are:
- \textbf{dm1m\_allocate} – Allocates space for putting an R-tree record.
- \textbf{dm1m\_delete} – Deletes a record from an R-tree table.
- \textbf{dm1m\_bybid\_get} – Gets a record by BID.
- \textbf{dm1m\_get} – Gets a record from an R-tree table.
- \textbf{dm1m\_put} – Puts a record to an R-tree table.
- \textbf{dm1m\_rch\_update} – Updates all RCBs in transaction.
- \textbf{dm1m\_replace} – Replaces an R-tree record.
- \textbf{dm1m\_search} – Searches an R-tree for a given key.

\textbf{dm1mindex.c} – Routines needed to manipulate R-tree indices.

- \textbf{dm1mxdelete} – Deletes a key,tid pair from a page.
- \textbf{dm1mxformat} – Formats an empty page.
- \textbf{dm1mxinsert} – Inserts a key,tid pair onto a page.
- \textbf{dm1mxjoin} – Joins index pages for modify to merge.
- \textbf{dm1mxnewroot} – Creates a new root due to a split.
- \textbf{dm1mxsearch} – Searches for a key or key,tid pair on a page.
- \textbf{dm1mxsplit} – Split a page into two pages.

Source that must be modified for \textbf{create index}:

\textbf{pslindx.yi} – YACC grammar for \textbf{create index}.
pslindx.c – Semantics for create index.

qeu? – Builds parameters.

qens – Calls dmf_call to create the index.

dmuindex.c – Contains the functions necessary to create an index on a table.
  • dmu_index – Routine to perform the normal create index operation.

dm2uind.c – Contains routines that perform the index table functionality.
  • dm2u_index – Create an index on a table.

dm2ucre.c - Create table utility operation.
  • dm2u_create – Create a table – makes the catalog entries.
    → dm2u_file_create – Creates the physical file.

dm2uuti.c - Routines that perform physical utility operations.
  • dm2u_load_table – Reads file and loads the sorter.
    → dm1mlbegin – Begin the R-tree build.
    → dm1mlput – Add to the R-tree.
    → dm1mlend – End the R-tree build.
  • dm2u_update_catalogs –Perform the catalog updates needed for modify index.

dm1mbuild.c - Routines to build an R-tree.
  • dm1mbbegin – Initializes R-tree file for building.
  • dm1mbput – Adds a record to a new R-tree and builds index.
  • dm1mbend – Finishes building R-tree table.

3.9 CATALOG INFORMATION

A new value (13) is added to the relspec column of the table $ingres.iiirelation to represent an R-tree structure.

<table>
<thead>
<tr>
<th>relid</th>
<th>relspec</th>
</tr>
</thead>
<tbody>
<tr>
<td>xfio_shape_ix</td>
<td>13</td>
</tr>
<tr>
<td>xfio_slc_ix</td>
<td>11</td>
</tr>
</tbody>
</table>

The value RTREE is displayed as the storage structure type that appears in the view $ingres.iiindexes. The following selection from iiindexes shows both an R-tree and a B-tree index.

<table>
<thead>
<tr>
<th>index_name</th>
<th>base_name</th>
<th>storage_structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>xfio_shape_ix</td>
<td>xfio_boundary</td>
<td>RTREE</td>
</tr>
<tr>
<td>xfio_slc_ix</td>
<td>xfio_boundary</td>
<td>BTREE</td>
</tr>
</tbody>
</table>

This storage structure type is also displayed by a help index command; for example:

<table>
<thead>
<tr>
<th>Name:</th>
<th>xfio_shape_ix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owner:</td>
<td>wonst02</td>
</tr>
<tr>
<td>Created:</td>
<td>16-may-1996</td>
</tr>
<tr>
<td>Location:</td>
<td>ii_database</td>
</tr>
<tr>
<td>Type:</td>
<td>secondary index on xfio_boundary</td>
</tr>
<tr>
<td>Version:</td>
<td>OPING1.2</td>
</tr>
</tbody>
</table>
A new table, $ingres.iirange$, is added to hold range information for all R-tree indexes in the database. The $iirange$ table is created by $createdb$ whenever a database is created.

The $iirange$ table has the following structure:
left “_ll” and upper-right “_ur” floating point values for each of the dimensions. The table can store ranges for up to four dimensions, though only 2-dimensional indexes are currently supported. In addition, the table stores the number of dimensions (currently 2) for the index and the size in bytes of the Hilbert values in the index (currently 6).

The following example illustrates the creating of an R-tree index:

```sql
create index xfio_shape_ix on xfio_boundary(shape)
  with structure=rtree,
  range=((100000000,400000000),(400000000,700000000));
```

which would result in the addition of the following row to `$ingres.iirange`:

<table>
<thead>
<tr>
<th>rng_baseid</th>
<th>rng_indexid</th>
<th>rng_ll1</th>
<th>rng_ll2</th>
<th>rng_ll3</th>
<th>rng_ll4</th>
</tr>
</thead>
<tbody>
<tr>
<td>174</td>
<td>177</td>
<td>1.000e+008</td>
<td>4.000e+008</td>
<td>0.000</td>
<td>0.000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rng_ur1</th>
<th>rng_ur2</th>
<th>rng_ur3</th>
<th>rng_ur4</th>
<th>rng_dim</th>
<th>rng_hilb</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.000e+008</td>
<td>7.000e+008</td>
<td>0.000</td>
<td>0.000</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

**3.10 DATA STRUCTURE DETAIL**

**3.10.1 Index/Leaf Page Format**

The R-tree index page has the same format as a B-tree page, which is a special structure overlaid on a page (DM1B_INDEX). That structure looks like a regular page until `page_checksum`, then it deviates. New tuples are added to the top of the free space on the index/leaf page. This is different from data pages, where new tuples are added at the end of the free space.

A B-tree or R-tree index or leaf page has the following format.

- `page_main` (i4) – Main page forward pointer (page number).
- `page_ovfl` (i4) – Next overflow page pointer (page number). Unused for R-tree
- `page_page` (i4) – Page number of this page.
- `page_stat` (i2) – Bitmap of page status.

- DMPP_DIRECT x'0001' – Index page of ISAM
- DMPP_PRIM x'0002' – Primary page
- DMPP_DATA x'0020' – Data page
- DMPP_LEAF x'0040' – B-tree/R-tree leaf page
- DMPP_FREE x'0080' – B-tree/R-tree unused page
- DMPP_MODIFY x'0100' – Used by buffer manager to keep track of modified pages. This is meaningful only while page is in memory and is meaningless once the page has been flushed to disk.
- DMPP_INDEX x'2000' – B-tree/R-tree index page
- DMPP_ASSOC x'4000' – B-tree associated (or current) data page
- DMPP_SPRIG x'8000' – B-tree/R-tree parent of leaf page
**page_seq_number** (i2) – The sequence number associated with a deferred update cursor within a transaction. This field is meaningful to DMF only while in memory and meaningless once page is flushed to disk.

**page_tran_id** (DB_TRAN_ID, 8 bytes long) – The transaction id that last changed this page. This field is only meaningful to DMF while in memory and meaningless once page is flushed to disk.

**page_log_addr** (LG_LSN, 8 bytes long) – The log record address of the most recently updated record on the page. This field is meaningful to DMF only while in memory and meaningless once page is flushed to disk.

**page_checksum** (i2) – Checksum of all bytes on the page.

**bt_padalign** (i2) – A filler to align things.

**bt_nextpage** (i4) – Sideways index/leaf page pointer—indicates next index or leaf page.

**bt_split_lsn** (LG_LSN, 8 bytes long) – The log address of the most recent split operation which affected the page.

**bt_unused** (char 28) – Spare room for future growth.

**bt_data** (i4) – Page number of associated data page. Unused for R-tree, since R-tree is for secondary indexes only.

**bt_kids** (i4) – Number of *tid/key* pairs on the page. This number does not include the DM1M_LRANGE and DM1M_RRANGE *tid/key* pairs. Those two special *tid/key* pairs are reserved to indicate the lowest and highest leaf pages in the chain and are not used for any of the other index pages. In other words, there are really bt_kids+2 entries in the **bt_sequence** array. The first two entries are reserved and have a special meaning on leaf pages. Those first two entries are not used on any other index pages and will contain a null *tid*.

**bt_sequence[]** (array of i2) – This array is similar to the page_line_table array on a data page. It is the offset from the start of the page to the start of the tuple. Unlike the page_line_table[] on a data page, the entire array is pre-allocated when the page is created. The first data tuple is added after the end of the **bt_sequence** array. All **bt_sequence** values should be less than or equal to the page size (2048 bytes). Whenever a new data tuple is added to the page, the members of the **bt_sequence** array are repositioned to ensure that the data tuples they point to are ordered.

  - **bt_sequence[-2]** – The *tid.line_number* indicated by this pointer will have a non-zero value on the first leaf page (indicating a bottom range of negative infinity). The *tid.line_number* will be zero on all other pages. Note, the *tid.page_number* and *key* are not used for the entry.
  - **bt_sequence[-1]** – The *tid.line_number* indicated by this pointer will have a non-zero value on the last leaf page (indicating a top range of infinity). The *tid.line_number* will be zero on all other pages. Note, the *tid.page_number* and *key* are not used for the entry.
  - **bt_sequence[0]** – A pointer to the first (lowest Hilbert value) *tid/key* pair on this index page. A *key* in an R-tree is either an *mbr* (minimum bounding region) — for a leaf page — or an *mbr* and *lhv* (largest Hilbert value) — for an index page.
  - **bt_sequence[bt_kids-1]** – A pointer to the last (highest Hilbert value) *tid/key* pair on this index/leaf page.
### 3.10.1.1 Root, Index and SPRIG Page Formats

The R-tree root page is merely the highest level index page in the R-tree. If the R-tree is small, it may be the only index page in the R-tree. There will always be at least one index page. A sprig page is the lowest level of index page in the R-tree. DMF knows that the next lower level page it will read will be a leaf page, which requires different locking than an index page does.

An R-tree index or leaf page has the following format. The first group of fields is common to all INGRES pages.

- **page_main** – Always zero.
- **page_ovfl** – Always zero. An index page does not overflow—it splits instead.
- **page_page** – The root page always resides on page zero, so this value will always be zero for the root page. It will be the page number that DMF reads from disk for all other pages.
- **page_stat** – The DMPP_INDEX bit will always be set. However, if the next lower level page is a leaf page, then DMPP_SPRIG will also be set.
- **page_seq_number** – Meaningful to DMF only when the page is in memory.
- **page_tran_id** – Meaningful to DMF only when the page is in memory.
- **page_checksum** (i2) – Checksum of all bytes on the page.
- **bt_padalign** – Space holder, may contain anything.
- **bt_nextpage** – Sideways index page pointer. Permits the link chase described earlier in the section, “The B-link tree algorithm”.
- **bt_split_lsn** (LG_LSN, 8 bytes long) – The log address of the most recent split operation which affected the page.
- **bt_unused[]** – Space holder. It usually has blanks, but the contents do not matter.
- **bt_data** – Not used on index pages. Should contain zero.
- **bt_kids** – The number of bid/mbr/lhv entries in the bt_sequence array. (Sometimes the term bid is used to describe pointers to other index or leaf pages. A bid looks exactly like a tid, except that only the page number information is valid, not the line number.) This number does not include the DM1M_LRANGE[-2] or DM1M_RRANGE[-1] entries in the bt_sequence array.
- **bt_sequence[]** – Array of pointers to the bid/mbr/lhv entries. Each pointer is the number of bytes from the start of the page to the start of the record. When new records are added to the page, the bids are rearranged to keep the data ordered. It is much cheaper to move around members in an array of i2 than it is to move whole records around on the page.

### 3.10.1.2 Leaf Page Format

The leaf page is the bottom level of the R-tree. The pointers in a leaf page point to data pages in the actual table instead of other index pages.

- **page_main** – Always zero.
- **page_ovfl** – Unused for R-tree, always zero.
- **page_page** – Page number that DMF reads from disk.
- **page_stat** – The DMPP_LEAF bit will be set.
- **page_seq_number** – Meaningful to DMF only when the page is in memory.
- **page_tran_id** – Meaningful to DMF only when the page is in memory.
- **page_checksum** (i2) – Checksum of all bytes on the page.
- **bt_padalign** – Space holder, may contain anything.
- **bt_nextpage** – Sideways pointer to the next leaf page in the chain. Permits the link chase described earlier in the section, “The B-link tree algorithm”.

---

**Ingres R-Tree Detailed Design Specification**

Page 33 of 35
bt_split_lsn (LG_LSN, 8 bytes long) – The log address of the most recent split operation which affected the page.
bt_unused[] – Space holder. It usually has blanks, but the contents do not matter.
btdata – Not used for R-tree secondary index.
btkids – The number of mbr/tid pairs in the bt_sequence array. This number does not include the DM1M_LVALUE[-2] or DM1M_RVALUE[-1] entries in the bt_sequence array.
bt_sequence[] – Array of pointers to the mbr/tid records. Each pointer is the number of bytes from the start of the page to the start of the record. When new records are added to the page, the bt_sequence members are rearranged to keep the data ordered. It is much cheaper to move around members in an array of i2 than it is to move whole records around on the page. The bt_sequence array on leaf pages contains tids, not bids. That is, both the page and line numbers will be filled in.

3.10.1.3 Overflow Page

Ingres R-trees, unlike B-trees, do not use overflow pages. Instead, leaf pages are split. This means that index pages can have duplicate entries pointing to different leaf or index pages. However, since R-trees can have overlapping mbts anyway, the duplicate entries cause no further problems. This simplifies the R-tree code substantially, compared to the B-tree code.

3.10.2 Data Page Format

Since an R-tree is a secondary index, the data pages pointed to by leaf pages are the data pages found in other storage structures (heap, hash, ISAM, or B-tree). The page_next_line[] array grows down from the top of the page and actual tuple records grow up from the bottom of the page. A B-tree data page format is shown below.

page_main – This is always zero.
page_ovfl – Always zero. B-tree data pages cannot overflow.
page_page – The page number that DMF reads from disk.
page_stat – The DMPP_DATA bit will be set. Also, the DMPP_ASSOCIATED or DMPP_PRIM bits may be set.
page_seq_number – This is garbage after the page is written back to disk. It is meaningful to DMF when the page is in memory.
page_tran_id – This is garbage after the page is written back to disk. It is meaningful to DMF when the page is in memory.
page_checksum (i2) – Checksum of all bytes on the page.
page_next_line – Varies depending on the size of the record for each tuple. It must never exceed 512, as that is the maximum value allowed by the current TID structure.
However, it is possible to calculate the maximum tuple count for R-tree index pages as

\[
(DM\_PG\_SIZE - DM1B\_OVERHEAD) / (relwid + 2) - 2
\]

or

\[
(2048 - 84) / (22 + 2) - 2 = 1964/24 - 2 = 81 - 2 = 79
\]

and for R-tree leaf pages as

\[
(2048 - 84) / (16 + 2) - 2 = 1964/18 - 2 = 109 - 2 = 107.
\]
So the maximum fanout for index pages is 79 (at 100% index fill factor) and for leaf pages is 107 (at 100% leaf fill factor).

**page_line_tab[]** – The offset from the start of the page to the start of the record, in bytes. None of the entries in the array should ever exceed the page size of DM_PG_SIZE (2048).

### 3.10.3 Free Page List Format

This is an index page that is used exclusively to chain free pages to. There are only three valid fields on this page. The contents of all other fields should be ignored. The valid fields are:

- **page_main** – Pointer to first page in free list.
- **page_page** – The page number that DMF reads from disk.
- **page_stat** – The DMPP_INDEX bit should be set.

**NOTE**—DMF does not search down the whole free list to get a page. It takes the first free page, removes it from the chain, and sets **page_main** to point to the next page in the chain.

### 3.10.4 Free Page Format

Most of the fields on a free page have garbage in them. Those pages will be formatted when they are removed from the free list. The only fields that are dependable are:

- **page_main** – A forward pointer to the next page on the free list. This will be zero for the last free page in the list.
- **page_page** – The page number that DMF reads from disk.
- **page_stat** – No bits will be cleared from **page_stat**, but the DMPP_FREE bit will be set.