R-Tree Database Design and API

Version 1.0

1 Design

The R-Tree database for $\mathcal{AXL}$ is designed to implement,

1. Antonin Guttman's design of R-Tree index;
2. Sequential scan over all index entries and data records;
3. Insertion/Deletion/Update of index entries and data records;
4. Finally, a BerkeleyDB compatible interface to support AXL queries.

This RTree implementation is based on the original spatial data structure described in the paper by Antonin Guttman "R-trees: A Dynamic Index Structure for Spatial Searching". This document describes the data structures involved in implementing the R-Tree. There are two existing online implementations. One from Dimitris Papadias and Yannis Theodoridis's research group (named as the Greek implementation later in this document), and another from General Electric company (named as the GE implementation later in this document). Both implementations have some shortcoming for developing a BerkeleyDB compatible interface for AXL.

- The Greek implementation is an academic experiment. It is an implementation with R-Tree index and records residing on secondary storage. However, there is no formal document describing how the secondary storage is organized (at least no comparable document as we will show below in this document), and the code has less comments than enough.
- The GE implementation follows Guttman's specification and has enough code comments. However, it is an implementation in primary storage only, which is unacceptable to this project.

Moreover, the most important reason is that neither of the above implementations support sequential scan over all leaf nodes in the R-Tree index. As it is almost impossible to modify the above implementations to satisfy our project's requirements, I have to implement a new R-Tree structure which is a **superset of Antonin Guttman's specification——that is,**

I chain all leaf nodes together and hence allow a sequential scan over all the records.

I have included both implementations in the R-Tree archive. One is under the subdirectory "Greek" and the other is under the subdirectory "GE". Users may compare them with the new implementation described in this document.

In order to fulfill specifications to allow for spatial data storage on the R-Tree, we needed to make sure we had an efficient data structure that allowed for operational functionality, including insertion, deletion, searching.

Like DB2/Oracle, each relation in our R-Tree database consists of two disk files, one for the data record, and one for the R-Tree index. Both files (index.db and data.db) begin with a super-structure.

- In index.db we have a super block that specifies the number of free blocks, the last block in use (i.e., end of file), R-Tree parameter m (i.e., minimal entries allowed in each R-Tree node), R-Tree parameter M (i.e., maximal entries allowed in each R-Tree node), and the head pointer of the leaf node chain.

  The block length is easily adjustable. A larger block length allows for more nodes to be pointed to per level. In our sample case, we have a block length of 1024.

- In data.db we have a supertuple that specifies the number of free tuples, the last tuple in use.

In index.db, the root node always resides in the block follows the superblock (if count from zero, superblock=0, root=1). In each node, the basic entry we use (represented by *dp* in the diagrams) consists of 5 long integers (20 bytes). Four long integers are for the entry's rectangle coordinates (upper left corner (x_{ul}, y_{ul}) and lower right corner (x_{lr}, y_{lr})), and one long integer for the offset pointer (for non-leaf nodes this pointer points to another node inside index.db, or for leaf nodes this pointer points to a tuple inside data.db).

Each node in index.db also has an overhead entry holding:

1. (1 byte) type of the node: 'l' for leaf node, 'n' for non-leaf node.
2. (4 bytes) number of entries currently in use. It is a unsigned integer.
3. (2*4=8 bytes) if current node is a leaf node (i.e., node type is 'l'), then there are two pointers *prevlptr* and *nextlptr* pointing to the previous node and next node in the leaf chain. The value include the pointers could be 0 which means end of chain (block 0 is the superblock, definitely not a leaf node).

Therefore, for a 1024 block size, we can have 50 entries per block (13 bytes for the overhead and 50*20 bytes for the entries). In this paper we define M as the maximal and m as the minimal number of entries in a node.

Free block management is achieved by maintaining a free block chain. For index.db, the super block's fb points to the head of the chain (i.e., the first free block). Each free block's first 4-byte is a pointer to the next free block. If the pointer is 0, then the end of the chain is met. It is simple to put a new block X into the chain, we put the current head of the chain in X and record X as the new head of the chain.

Searching [Guttman: section 3.1] To search, we specify a rectangular coordinate S, starting at the root node T of the given R-Tree index. We return all entries that contain those coordinates within T. If T is not a leaf, we check each entry E to see if E overlaps S. For all containing entries, we recursively search upon the tree whose root node is pointed to by E. If T is a leaf we check all entries E to see if they match. If they do, E is a qualifying record and we return it.

Insertion [Guttman: section 3.2] Insertion is similar to the common BTree implementation where new index records are added to leaves and overflowed nodes are split resulting in node propagation up the tree. To insert a node E, first invoke *ChooseLeaf* which descends from the node until it finds the leaf with the least enlargement result, and set N to be the root node. If it is a leaf node and isn't full, we install E, otherwise we invoke *SplitNode* to get two new nodes L and LL both containing E and the M entries of the old node. We propagate the changes by calling *AdjustTree*, which ascends from a leaf node L to the root, adjusting covering rectangles and propagating node splits as necessary. If the node propagation caused the root to split, we make a new root where the children are the two resulting nodes.

Deletion [Guttman: section 3.3] In deletion we specify index record E and root node T and invoke *FindLeaf* to locate the leaf node L containing E. If T isn't a leaf node, it checks to see if each entry F in T contains the specified E. For each entry, we recursively call *FindLeaf* to parse the entries to find the node that points to E, or until all containing entries have been checked. For each T that is a leaf, we check for entries matching with E. If E is found we return T. We stop if the record was not found. Otherwise, we remove E from L/T. Afterwards we propagate the changes up with *CondenseTree*, passing L. *CondenseTree* will eliminate the node L if it has too few entries and propagate the

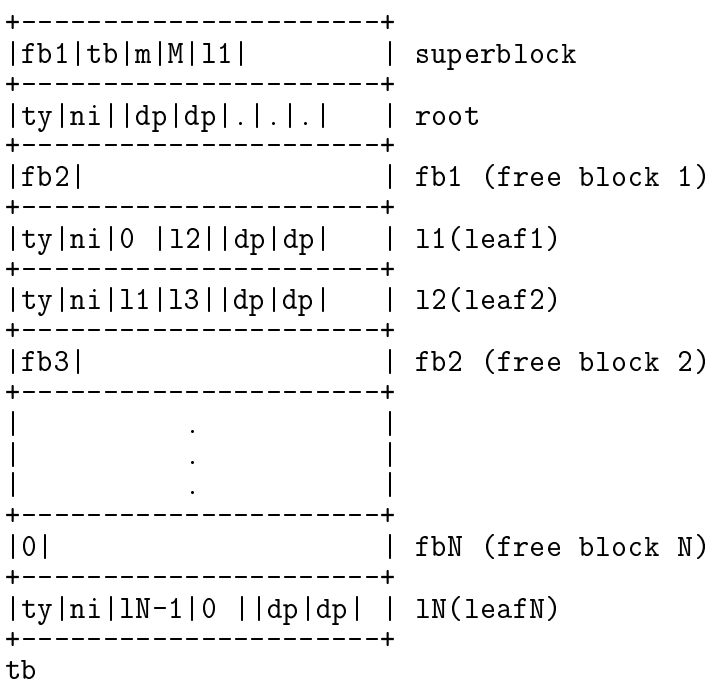
node elimination upwards as needed. All entries in the eliminated nodes are recorded in a temporary buffer Q. If the root node has only one child after the tree has been adjusted make the child the new root. Finally all the eliminated entries in buffer Q are re-inserted into the index again (This is the standard behavior specified by Antonin Guttman).

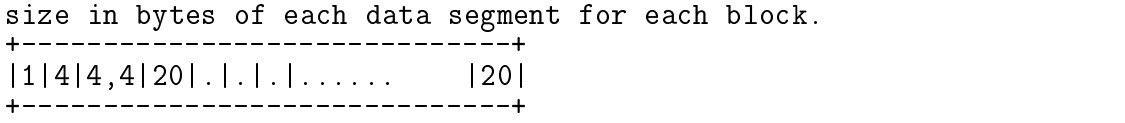
Node Splits [Guttman: section 3.5] In the scenario where we have to split a node because either we have N (number of entries) $< m$ or $N > M$. To split the tree we have three options: an exhaustive algorithm, a quadratic cost algorithm, and a linear-cost algorithm. The exhaustive algorithm is the most straightforward way, but not surprisingly unfeasible for databases with many nodes. The number of possibilities is approximately $2^M - 1$.

The quadratic-cost algorithm attempts to find a small-area split for the database. It is efficient, effective, but not complete in search space (not guaranteed to find the smallest area possible). It works by picking two of $M + 1$ entries that would waste the most space if paired together. For each step, we call PickNext and select the next entry that has the largest difference between that new entry and the groups; $ABS(MAX(d1 - d2))$ where $d1$ is the distance between the object and the first group and $d2$ the same for the second group.

We use quadratic-cost algorithm in the implementation.

Graphical Representation of the databases: Here is a graphical representation of index.db.



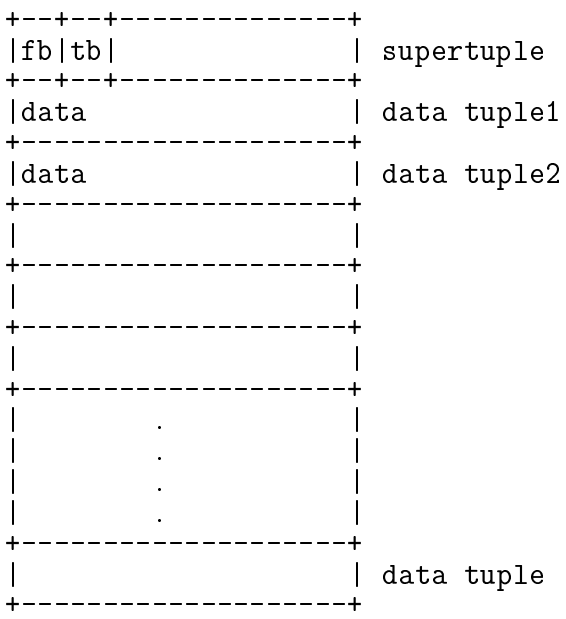


- Under superblock
  * fb means head of free block list. It points to the first free block.
  * tb means total number of blocks in current file, i.e., the block after the last block in used.
  * m means minimum number of entries per node
  * M means maximum number of entries per node
  * lptr means the head of the leaf node chain.

The second block in index.db is always the root block.

- Under root,
  * ty means node type, i.e., if it is a leaf or non-leaf node.
  * ni means number of entries in use in the node.
  * dp means data structure or leaf/non-leaf node pointer. We can have multiple pointers per row which represent the nodes on the R-tree. If the block you are currently on is a non-leaf node block, then dp will point to another block in index.db. If the block you are currently on is a leaf node block, then dp will point to a block in data.db.

Here is a graphical representation of data.db.




```
- Under supertuple
  * fb means head of free block list. It points to the next free block.
  * tb means total number of blocks in current file

- Under a free tuple
  * The first integer is a pointer pointing to next free tuple in
    the free-chain.  If the pointer is 0, it means the end of the
    chain.  Then the next free tuple will be grabbed from the end
    of the storage file which is recorded in tb.
```

2 Implementation

- `rtree.c` implements R-Tree index.
- `block.c` implements node-level operations in the R-Tree index.
- `rectangle.c` implements operations related to rectangles.
- `interface.c` implements BerkeleyDB compatible operations including cursor operations.
- `tupledb.c` implements data record storage.
- `tupledb.h` specifies definitions and macros for data record storage.
- `rtree.h` specifies definitions and macros for R-Tree index storage.

3 API: Application Programming Interface

3.1 rtree_create

```
#include <rtree.h>
int rtree_create(RTree **rtree, char *indexFileName, char *recordFileName, int flags);
```

Description

The *rtree_create* function creates handle for an R-Tree relation. A pointer to this structure is returned in the memory referenced by AXL.

The currently supported R-Tree index entry format consists of two parts: (i) 4 long integers representing the coordinates of upperleft corner (x_{ul}, y_{ul}) and lowerright corner (x_{lr}, y_{lr}). (ii) 1 long integer representing the file offset to the child node. The index is stored in *indexFileName*. The details format of the file is explain in design section of this document.

The currently supported R-Tree record format is fixed length tuples stored in *recordFileName*. The details format of the file is explain in design section of this document.

The flags value must be set to 0.

Return Value

The *rtree_create* function returns a non-zero error value on failure and 0 on success.

Errors

The *rtree_create* function may fail and return a non-zero value -1 for disk access errors and memory exhaustion errors.

Sample Call

```
RTree *rtree;

if((ret = rtree_create(&rtree, "index.db", "data.db", 0) != 0)
    return ERROR;
```

3.2 RTree->open

```
#include <rtree.h>
int rtree->open(RTree **rtree, char *indexFileName, char *recordFileName, int flags);
```

Description

The RTree->open interface opens the database represented by name for both reading and writing.

The *indexFileName* and *recordFileName* must already been created by the *rtree_create* routine. RTree->open is merely for re-open the existing database.

The flags parameter must be set to 0 currently.

Return Value

The RTree->open function returns a non-zero error value on failure and 0 on success.

Errors

The RTree->open function may fail and return a non-zero value -1 for disk access errors and memory exhaustion errors.

Sample Call

```
RTree *rtree;

if((ret = rt->open(&rt, "index.db", "data.db", 0)) != 0)
    return ERROR;
```

3.3 RTree->close

```
#include <rtree.h>
int RTree->close(RTree *rtree, int flags);
```

Description

The RTree->close function closes any open cursors, frees any allocated resources.

The flags parameter must be set to 0 currently.

Once RTree->close has been called, regardless of its return, the RTree handle may not be accessed again until the handle is re-opened.

Return Value

The RTree->close function returns a non-zero error value on failure and 0 on success.

Errors

The RTree->close function may fail and return a non-zero value -1 for disk access errors and memory exhaustion errors.

Sample Call

```
if((ret = rt->close(rt, 0)) != 0)
    return ERROR;
```

3.4 RTree->del

```
#include <rtree.h>
int RTree->del(RTree *rel, Rectangle *mbr, int flags);
```

Description

The RTree->del function removes the key/data pair from a relation. The key/data pair associated with the specified *mbr* is discarded from the relation. In the presence of duplicate key values, only the first records associated with the designated key will be discarded.

The flags parameter is currently unused, and must be set to 0.

Return Value

The RTree->del function returns a non-zero error value on failure, 0 on success, and returns DB_NOTFOUND if the specified key did not exist in the file.

Errors

The RTree->del function may fail and return a non-zero value -1 for disk access errors and memory exhaustion errors.

Sample Call

```
if((ret = rt->del(rt, &rectangle, 0)) != 0)
    return ERROR;
```

3.5 RTree->get

```
#include <rtree.h>
int RTree->get(RTree *rtree, Rectangle *mbr, char *tuple, size_t tplsz, int flags);
```

Description

The RTree->get function retrieves key/data pair from the database. The data record associated with the rectangle *mbr* is returned in the placeholder *tuple*. The caller must pre-allocate the placeholder with at least size *tplsz*, as the routine simply memcpy *tplsz* bytes into the placeholder from data record storage.

In the presence of duplicate key values, RTree->get will return the first data item for the designated key. Retrieval of duplicates requires the use of cursor operations. See RTreeCursor->c_get for details.

The flags parameter must be set to 0.

Return Value

If the requested key is not in the database, the RTree->get function returns DB_NOTFOUND.

Otherwise, the RTree->get function returns a non-zero error value on failure and 0 on success.

Errors

The RTree->get function may fail and return a non-zero error for errors specified for other Berkeley DB and C library or system functions.

Sample Call

```
RTree *rtree;
Rectangle mbr;
size_t tplsz = 6;
char *tuple=malloc(tplsz);

......

if((ret = rtree->get(rtree, &mbr, tuple, tplsz, 0)) != 0)
    return ERROR;

......
free(tuple);
```

3.6 RTree->put

```
#include <rtree.h>
int RTree->put(RTree *rtree, Rectangle *mbr, char *tuple, size_t tplsz, int flags);
```

Description

The RTree->put function stores key/data pairs into the R-Tree index and data record.

The flags parameter must be set to 0 or one of the following values:

DB_APPEND
: Insert the tuple to the existing database, no matther whether the key *mbr* already exists in the database.

DB_NOOVERWRITE
: Insert the tuple to the existing database, or do nothing if the key *mbr* already exists in the database.

The default behavior of the RTree->put function is to enter the new key/data pair, replacing the first previously existing key *mbr*.

Return Value

The RTree->put function returns a non-zero error value on failure, 0 on success.

Errors

The RTree->put function may fail and return a non-zero value -1 for disk access errors and memory exhaustion errors.

Sample Call

```
RTree *rtree;
Rectangle mbr;
char tuple[6]="hello";

......

if((ret = rtree->put(rtree, &mbr, tuple, 6, 0)) != 0)
    return ERROR;
```

3.7 RTree->cursor

```
#include <rtree.h>
int RTree->cursor(RTree *rtree, RTreeCursor **cursorp, int flags);
```

Description

The RTREE->cursor function creates a cursor and copies a pointer to it into the memory referenced by "cursorp".

A cursor is a structure used to provide sequential access through a database.

Currently the flags value must be set to 0.

Return Value

The RTREE->cursor function returns a non-zero error value on failure and 0 on success.

Sample Call

```
RTree *rtree;
RTreeCursor *rtc;

......

if((ret = rtree->cursor(rtree, &rtc, 0)) != 0)
    return ERROR;
```

3.8 RTreeCursor->c_get

```
#include <rtree.h>
int RTreeCursor->c_get(RTreeC *cursor, char *tuple, size_t tplsz, int flags);
```

Description

The flags parameter must be set to one of the following values:

DB_FIRST
: The cursor is set to reference the first tuple of the relation, and that pair is returned.

  If the database is empty, RTreeCursor->c_get will return DB_NOTFOUND.

DB_NEXT
: If the cursor is not yet initialized, DB_NEXT is identical to DB_FIRST.

  Otherwise, the cursor is moved to the next tuple of the database, and that pair is returned.

  If the cursor is already on the last record in the database, RTreeCursor->c_get will return DB_NOTFOUND.

Return Value

If no matching keys are found, RTreeCursor->c_get will return DB_NOTFOUND. Otherwise, the RTreeCursor->c_get function returns a non-zero error value on failure and 0 on success.

If RTreeCursor->c_get fails for any reason, the state of the cursor will be unchanged.

Error

The RTreeCursor->c_get function may fail and return a non-zero value -1 for disk access errors and memory exhaustion errors.

Sample Call

```
RTreeCursor *rtc;
size_t tplsz = 6;
char *tuple=malloc(tplsz);

......
```

```
if((ret = rtc->c_get(rtc, tuple, 6, DB_NEXT)) == DB_NOTFOUND)
    .......
free(tuple);
```

3.9 RTreeCursor->c_del

```
#include <rtree.h>
int RTreeCursor->c_del(RTreeC *cursorp, int flags);
```

Description

The RTreeCursor->c_del function deletes the key/data pair currently referenced by the cursor.

The flags parameter is currently unused, and must be set to 0.

Return Value

If the cursor is not yet initialized or the database is currently empty, the RTreeCursor->c_del function will return DB_NOTFOUND. Otherwise, the RTreeCursor->c_del function returns 0 on success.

Sample Call

```
RTreeCursor *rtc;

......

if((ret = rtc->c_del(rtc, 0)) == DB_NOTFOUND)
   .......
```

3.10 RTreeCursor->c_close

```
#include <rtree.h>
int RTreeCursor->c_close(RTreeCursor *cursorp);
```

Description

The RTreeCursor->c_close function discards the cursor.

Once RTreeCursor->c_close has been called, regardless of its return, the cursor handle may not be used again.

Return Value

The RTreeCursor-¿c_close function returns a non-zero error value on failure and 0 on success.

Sample Call

```
RTreeCursor *rtc;

......

if((ret = rtc->c_close(rtc)) != 0)
    return ERROR;
```

Contents