Ph.D. Comprehensive Exam

Scot Anderson

January 12, 2006

Contents

1 Concepts of Indices: Review

Long before databases came into use systems of organizing information used the concepts of an index. Examples of indices include the list of key words at the back of most books, library card catalogs and topic indices found in phone books. An index optimizes the data search process by providing a smaller, ordered subset of the data or meta data.

With the advent of computers came the process of structuring and managing information. Similar types of information such as addresses have similar data: street number, street name, city, state and zip. Each piece is stored in a *field*. Each address would then have the same set of fields. We store information in *rows*, often called *records*, where each row is an element in a *set* of information (e.g. each row may hold an address). The names of the fields describe the information stored in each field and are often times strongly typed. A single set of information looks like a *table* and is often identified as such. Although these sets can be operated on and related to one another using relational algebra, the result is just another set. Thus we focus on the organizing and accessing of information in a set.

Searching a table of information can be time consuming especially if the information is not ordered by the search parameters. On the other hand, keeping the information ordered in multiple ways while inserting, updating and deleting rows may be extremely time consuming without some organizational scheme. Indices are employed to optimize the search process and solve this problem. They allow the use of a *search key* made up of a field or set of fields used to look up a record.

To complete the background to indices we are going to review the following concepts: primary indices, dense and sparse indices, multilevel indices and hashing from [18, 22].

1.1 Primary and Secondary Indices

A search key along with pointers to one or more records comprise an *index record* or *index entry*. The sorted list of these entries forms an index.

Definition 1 *Given a file containing a set of records, the primary index defines the sequential ordering of those records within the file. Primary indices are also called clustering indices.*

Definition 1 gives us the idea that there is some piece of information in each record that provides some natural ordering called the search key. If that information is not unique than there will be an arbitrary ordering of non-unique records. Consider a list of customers sorted by state, city. There may be hundreds of customers just in Lincoln, Nebraska. Searching using state and city as the search key would then require an evaluation of each record that contains the search key. Clearly there are many ways to order a set of records, and the choice of the primary index may be related to tasks that require sequentially processing the table.

The second type of index that naturally comes to mind, is given in Definition 2 below.

Definition 2 *Given a file containing a set of records, secondary indices define search keys whose order is different from the sequential order of the file.*

1.2 Dense and Sparse Indices

When we have an index record for every search key found in its associated table, the index is called a *dense index*. The index entry for dense *primary* indices has a search key and a single pointer to the first record in the table containing the search key. The index entry for dense *secondary* indices has a search key and a list of pointers to records containing the search key. Formally we have the Definition 3 below:

Definition 3 *Let S_k be the set of all search keys in the table. A primary index is* dense *if and only if there exists a bijection from the index records to search keys in S_k. A secondary index is dense if and only if there exists a bijection from the index record pointers to records in the table.*

To find a single record using a dense, primary index, we lookup the search key and follow the pointer to the first record. We examine each record to find the single record matching our criteria until we: 1) find the record sought or 2) find a record with a different search key.

When we have an index record that maps to more than one search key in its associated table, the index is called a *sparse index*. Unlike dense indices we can only use sparse indices with primary indices since each search key must have an entry for secondary indices. Formally we define sparse indices as follows.

Definition 4 *Let S_k be the set of all search keys in the table. An index is* sparse *if it is not secondary and the mapping from the index records to S_k is not onto.*

Searching a sparse index is slightly more difficult. Given an index I with n index records $i_1, ..., i_n$ and a search key s_k we want to find the record i_j such that $searchKey\,(i_j) \leq s_k$ and $searchKey\,(i_{j+1}) > s_k$. After the correct index record is found, we proceed to the table using the pointer(s) to find the correct record.

Dense indices are searched somewhat faster than sparse indices. However the dense indices takes more space and impose greater maintenance costs on inserts and deletes.

In each database request, time is dominated by retrieving the data from disk. Since data is retrieved one page at a time, the compromise is to use a sparse index with one index entry per block stored on disk. This way we find the block that contains the record we need, and retrieve just that block.

It is possible for a single search key to span several pages. For secondary indices this is not a problem since we wouldn't necessarily be searching sequentially to start with. With the primary indices we are left to continue searching page after page until we find the correct record. There are two observations relevant to this situation: There may be a better index to use for our search or the primary index is too broad in that it does not impose sufficient order on the records. I.e. if we have an index that has two index records for a table with 1 million records, we need to find better search key criteria to build the index.

The opposite problem is to have an index that is too large to fit into memory. When indices grow too large, we consider multilevel indices.

1.3 Multilevel Indices

Suppose we have an index that will not fit in memory or would simply take to much memory. Using a standard binary search algorithm to find the index record in an index with $p = 100$ pages, would require

$$O\left(\lceil \log_2 100 \rceil\right) = 7 \tag{1}$$

block reads.

We can reduce the size of the search problem by making an index of the index. By treating the first level index that has grown to large as just another table, and creating an index on it, we build a multilevel index. (See Figure 1) Obviously we can repeat the process of building indices of indices until we have a small enough outer index to fit in memory.

For the index mentioned above, assuming the second level index will fit in memory we search the in-memory index to find the block containing the index-record we need. Loading that block we search it sequentially or using a binary search to find the pointer to the record we desire. We retrieve the desired data by loading that block. Thus we have only one block loaded for the index lookup, and one block loaded for the data. This saves at best 6 loads in the search process over (1).

We will discuss the B-tree which is similar to a multilevel tree in the second part of the exam.

1.4 Hashing

A hashing approach to indices computes the location containing the record from the search key. The "location" generally stores many records and is denoted in the literature as a *bucket*. A bucket is typically a disk page, but could be chosen to be larger or smaller than a

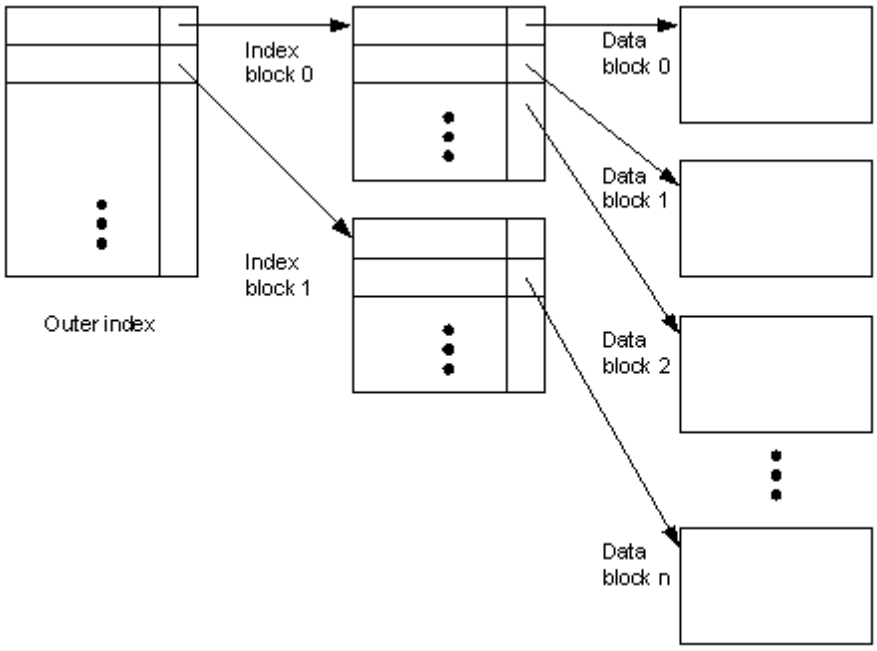

Figure 1: Multilevel Index

disk page.

Let K denote the set of all search key values, and let B denote the set of all bucket addresses. Let h denote a function called the *hash function* such that $h : K \rightarrow B$. If we didn't put requirements on hash functions, we could end up with a function that maps all the search keys in K to a single bucket in B. Thus we require hash functions to have the following two properties:

1. The distribution of search keys in the buckets is uniform.

2. The distribution of search keys in the buckets is random.

We need both of these properties to hold. Consider the following hash function that maps a name to a bucket based on the first letter of the name:

$$h : Names \rightarrow firstLetter(Name) \tag{2}$$

There will be 26 buckets, but the distribution of letters in the English language (or any

other language) is not uniform, thus some buckets, b_q for instance, will have very few if any names in it, while b_r may be overloaded. In this case not only is it not uniform, but as a consequence, it is not random either. If we made a hash function that mapped records to buckets based on age, we could make the buckets have a uniform distribution in increments of 10. So that buckets b_i contains names of people such that their age is

$$Name \subseteq b_i : 10\,(i-1) \leq age < 10i \tag{3}$$

Clearly this is uniform, but if we are looking at employees, buckets $1, 2$ will be empty compared to $3-7$. Thus (3) is not random.

1.4.1 Bucket Overflows

Each bucket has a capacity f_r which denotes the number of records that will fit in the bucket. We denote the number of records in a database as n_r. Then the number of buckets n_b must be chosen so the following holds:

$$n_b > \frac{n_r}{f_r} \tag{4}$$

If this condition does not hold, then we have a *bucket overflow*. Bucket overflow may be caused by some buckets having more records than other buckets. This situation is called *bucket skew*. Although theoretically the properties listed above will prevent bucket skew, in practice they only reduce bucket skew. To protect against bucket overflow due to skew, we choose a fudge factor (usually about 0.2) and choose n_b as follows:

$$n_b = \frac{n_r}{f_r}\,(1+d) \tag{5}$$

Despite adding a cushion of buckets, the database may still grow beyond the capacity of the index.

One method to deal with bucket overflow is to allow the system to create additional

buckets called overflow buckets. The original bucket links to the overflow bucket in a process called *chaining*. This approach is analogous to forming another layer in the multilevel approach. If the bucket is the size of a page, then you may have to load two pages if the first page is full and another has been chained to it. Chaining can reach arbitrary depths and reduce performance. This is often called *closed hashing*.

A second method allows records to overflow to other buckets either through computing a second hash function or linearly walking through the buckets to find one that has room. With a bad hash function or an unlikely choice of data, you could fill a single bucket and then continuously overflow to other buckets after that. The result is that you may have to load every block in the index before finding the entry desired. This method is called *open hashing* and has the advantage of not growing. However it severely hampers insert and delete operations and consequently is seldom used for databases.

1.4.2 Dynamic Hashing

So far we have considered n_b, the number of buckets, to be static. Allowing n_b to be dynamic solves the overflow problem, but introduces a different problem. How do we change the hash function to match the size of the index, specifically the number of buckets? If the database system adds new buckets, how does it distribute index records to these buckets uniformly given that the other buckets are now nearing capacity?

Extendable hashing (Figure 2) uses a hash function that produces a *b-bit* integer from a search key. Buckets are then created on demand based on the first i bits of the hash value. These bits are used as an offset into an additional table of bucket addresses. Each entry in the table has a value that indicates how many of the *b-bits* of the hash value are needed to correctly select the bucket and a pointer to the bucket. The value determining the number of bits changes as the database shrinks and grows.

When splitting a bucket there are two cases: (1) If only one entry in the table points to this bucket, then a new bucket and a new table entry must be created. The contents of the

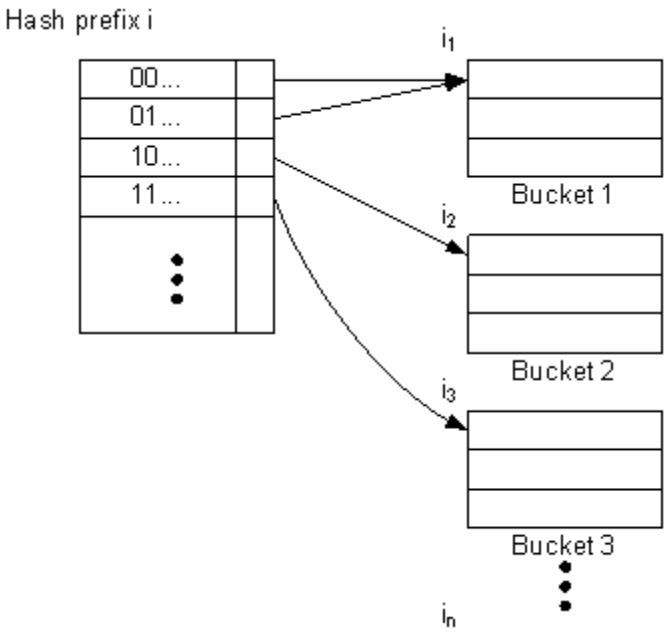

Figure 2: Extendable Hashing

bucket is then distributed between the old and new bucket. (2) There are two table entries pointing to this bucket. Here all that is needed is to create an additional bucket and point one of the table entries to the new bucket and redistribute the contents.

Removing buckets can be done when a bucket becomes empty. Coalescing buckets can be done based on a number of factors, most prominent of which is the used capacity of the buckets.

2 Three Indices

First I will discuss an index used in the MySQL relational database. Relational database systems are ubiquitous in just about every common data storage application. It resides at the back end of most accounting software packages, customer relationship management (CRM) systems, enterprise resource planning (ERP) software and almost any other business application that must store significant amounts of data. Of course it has been used to store scientific information as well.

MySQL uses a unique multi index approach to indexing a single table. Most MySQL indexes (PRIMARY KEY, UNIQUE, INDEX, and FULLTEXT) are stored in B-trees. Exceptions are that indexes on spatial column types use R-trees, and that MEMORY tables also support hash indices [24]. We describe and review the B-Tree index in Section 2.1. For our second index starting in Section 2.2 we discuss two related indices: R-trees and a descendent used in constraint databases, Parametric R-trees (PR-trees). Finally in Section 2.4 we discuss a specialized type of index for selectivity estimation called min-skew.

2.1 B-Tree Index

1. Describe for each index some example database applications. Explain how the database applications benefited by using the index.

B-trees are primarily used in relational databases. The primary goal and benefit of indices for databases is to find and manage (via inserts, deletes, and updates) information quickly. We already gave several examples above for relational databases.

The B-Tree as first introduced in 1972 by Bayer and McCreight [2]. Although Bayer never made clear what the B stands for, it is conjectured to stand for Bayer, Balanced or Boeing for whom he was working at the time. The tree is made up of two different types of nodes: tree nodes and leaf nodes. Although the root node is considered a tree node, it may be a leaf node if there are not enough elements for it to have children.

A tree node has search keys K_i, for $1 \leq i < m$ and pointers P_i, B_i, for $1 \leq i < m$, with a final pointer P_m. Figure 3 shows a tree node. Every search key has two pointers prior

P_1B_1	K_1	P_2B_2	K_2	...	$P_{m-1}B_{m-1}$	K_{m-1}	P_m

Figure 3: Tree Node

to it in the node, and there is an additional pointer after the last search key. P_i points to another node in the tree where each search key will be less than the search key K_i. B_i points

to a record containing search key K_i in the case of a primary index, or a bucket of pointers in the case of a secondary index where each pointer in the bucket points to a record in the table containing search key K_i. Each tree node may have between $\lceil \frac{m}{2} \rceil$ and m children. The methods for maintaining this constraint are discussed with the insertion and deletion of nodes. The root node is the exception since if we only have $m + 1$ records it must have exactly 2 children. We see that the root node exceptions for two cases: (1) If $2 \leq \lceil \frac{m}{2} \rceil$, the root may have fewer than $\lceil \frac{m}{2} \rceil$ children and (2) if the number of index entries is less than $m+1$, we have an exception and the root node may be a leaf node that still has the structure of a tree node with null pointers to no children.

A leaf node does not point to another node so its structure is slightly different than a tree node. Figure 4 shows a leaf node.Each pointer P_i in this case points to a record

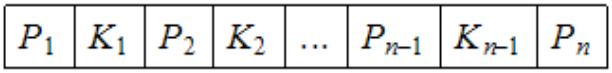

Figure 4: Leaf Node

containing search key K_i for primary indices or a bucket of pointers which each point to a record containing search key K_i for secondary indices. Although the nodes are the same size we can fit more search keys into a leaf node because it has fewer pointers per search key. Thus we have the condition that $m < n$. The exact relationship will depend upon the size of the search keys and the choice of n. Each leaf node may have between $\lceil \frac{n}{2} \rceil$ and n search keys unless the root node is the only node.

Now that we have well defined nodes, we give the properties that must be satisfied by a B-Tree [25]:

1. The root is either a tree leaf or has at least two children.

2. Each node (except the root and tree leaves) has between $\lceil \frac{m}{2} \rceil$ and m children, where $\lceil x \rceil$ is the ceiling function.

3. Each path from the root to a tree leaf has the same length.

Example 5 *Suppose we have a table of account records given in Table 1. Then the B-Tree*

Account #	**Town**	**Amount**
A-217	Brighton	750
A-101	Downtown	500
A-105	Clearview	300
A-110	Downtown	600
A-215	Mianus	700
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700
A-222	Redwood	700
A-305	Round Hill	350

Table 1: Account Records

for an index on the town is given in Figure 5.

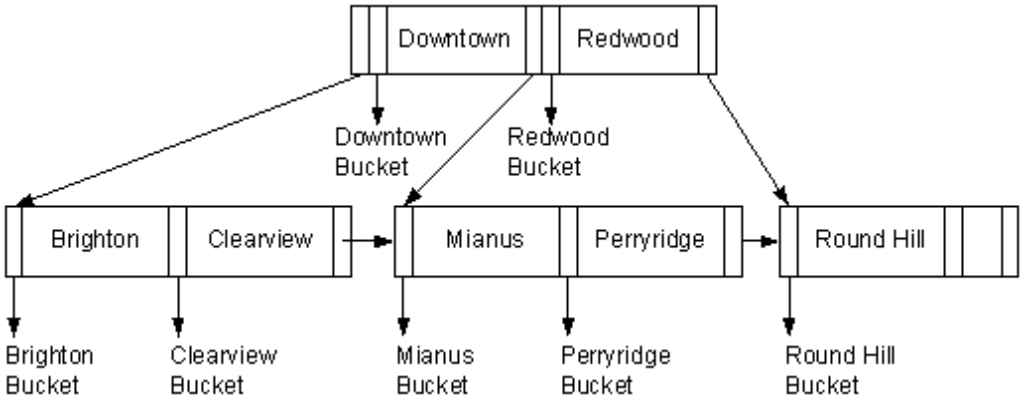

Figure 5: B-Tree of Accounts

2. Evaluate the space requirements of each index and the process required to build it.

Space Requirements: Let the size of the search key field including the pointers be k_s. For each node there will be at least $\lceil \frac{m}{2} \rceil$ fields occupied in the node. Given that each node

could be at minimum capacity, the space requirement for i index records in a primary index is

$$\begin{aligned} space_{primary} &= O(2ik_s) \\ &= O(i) \end{aligned} \tag{6}$$

For a primary index, the value of i may be much smaller than r, the number of records in the table. If the index is a secondary index, then the pointer will be to a bucket and these must be included in the size of the index. If we include the buckets as part of the index with r as the number of records in the table, we have

$$space_{secondary} = O(2ik_s + r) \tag{7}$$

Since $i \leq r$ the above equation reduces to

$$\begin{aligned} space_{secondary} &= O(2ik_s + r) && (8) \\ &\leq O(2rk_s + r) && (9) \\ &= O(r(2k_s + 1)) && (10) \\ &= O(r) && (11) \end{aligned}$$

in the worst case. Note that r does not represent the size of the table, but the number of records in the table and in general the size of the index will still be much smaller than the size of the table.

Creating a B-tree: The process to create a B-tree is simply to allocate an empty node as the root of the tree. After this we may call insert on the B-tree to add information.

3. Describe the operations that can be performed on the index. Include in the description the computational complexity of each operation.

For a node to be in an illegal state it must have less children or search keys than allowed or it must have more than is allowed after an insertion has completed. Clearly we will run into the former problem on a delete, and the latter problem on an insert. The height of the tree is an important factor in determining the running time of the insert, delete and search operations.

Insertions: There are three steps to an insert:

1. Search for the position into which the node should be inserted, and insert the value.
2. If the insertion did not overfill the node, then the process is finished.
3. If the node has more search keys or children than allowed, split the node, move to the parent node and check to see if splitting the node caused an overfill of the parent node. Loop on this third step until you have a node that does not need to be split.

In the worst case you would need to split every node all the way up to the root. Thus the cost of an insert in the worst case is on the order of the height of the tree traversed times the number of elements traversed in each split. Thus the running time given by [7] is

$$RunningTime = O\left(t \log_t n\right) \tag{12}$$

where n is the number of nodes and t is the least number of children (that is $t = \lceil \frac{m}{2} \rceil$).

Deletions: There are three steps to the delete operation described in [7] where leaves always have enough entries to allow a deletion See Appendix A for pseudocode:

1. If the key k is in node x and x is a leaf, delete k from x.
2. If the key k is in node x and x is an internal node, do the following
   (a) If the child y that precedes k in node x has at least t keys, then find the successor k' of k in the subtree rooted at y. Recursively delete k', and replace k by k' in x. (Finding k' and deleting it can be done in a single downward pass).

(b) Symmetrically, if the child z that follows k in node x has at least t keys, then find the successor k' of k in the subtree rooted at z. Recursively delete k', and replace k by k' in x. (Finding k' and deleting it can be done in a single downward pass).

(c) Otherwise, if both y and z have only $t-1$ keys, merge k and all of z into y, so that x loses both k and the pointer z, and y now contains $2t-1$ keys. Then, free z and recursively delete k from y.

3. If the key k is not present in internal node x, determine the root $c_i[x]$ of the appropriate subtree that must contain k, if k is in the tree at all. If $c_i[x]$ has only $t-1$ keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then, finish by recursing on the appropriate child of x.

   (a) If $c_i[x]$ has only $t-1$ keys but has an immediate sibling with at least t keys, give $c_i[x]$ an extra key by moving a key from x down into $c_i[x]$, moving a key from $c_i[x]$'s immediate left or right sibiling up into x, and moving the appropriate child pointer from the sibling into $c_i[x]$. That is we perform a rotation so that child i of x has enough elements that we can delete one.

   (b) If $c_i[x]$ and both of $c_i[x]$'s immediate siblings have $t-1$ keys, merge $c_i[x]$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

To illustrate the deletion steps consider the following example.

Example 6 *Suppose we have the B-tree shown in Figure 6 Figures 7-12 show each case in the deletion algorithm. Each figure is labeled with the element deleted and the case used to delete it. Case 2b is not shown as it is symmetric to case 2a. Highlighted nodes have changes from the previous figure.*

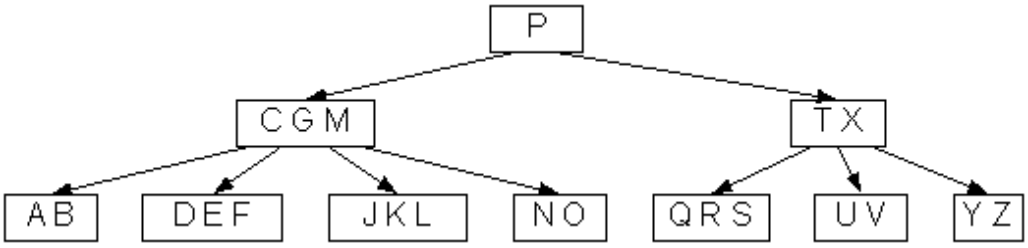

Figure 6: Initial tree

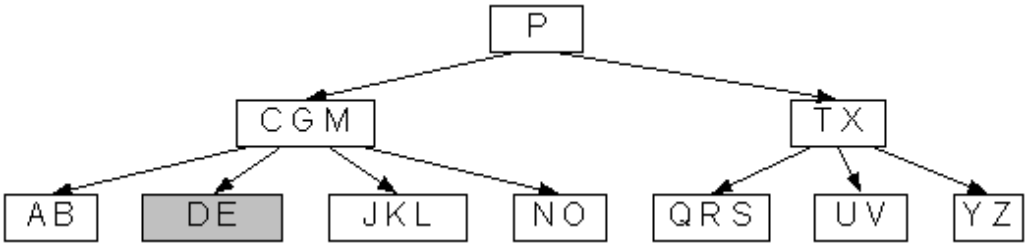

Figure 7: F deleted: case 1

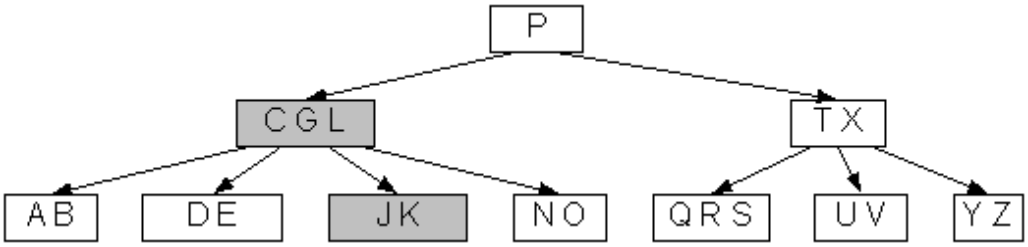

Figure 8: M deleted: case 2a

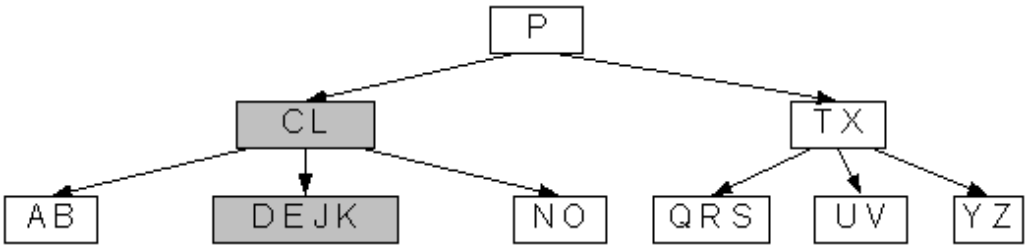

Figure 9: G deleted: case 2c

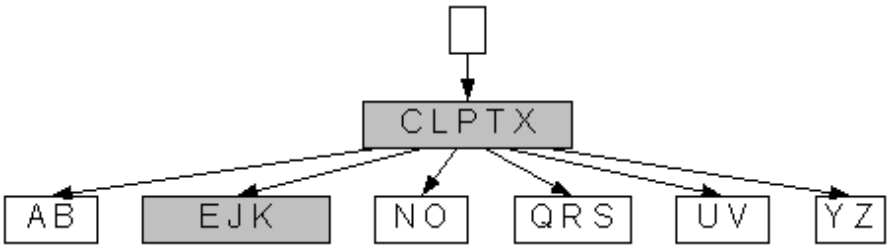

Figure 10: D deleted: case 3b

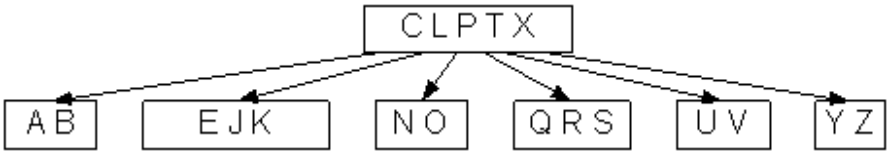

Figure 11: Tree shrinks in height after D is deleted

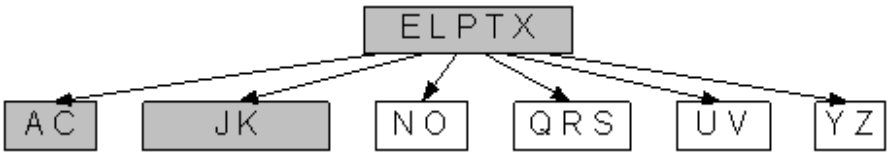

Figure 12: B deleted: case 3a

Here again in the worst case you would need to traverse the tree to find a leaf node, and merge it back up to the top. Thus the worst case running time is $O(\log n)$. Since we have to deal with at least t children or $t-1$ keys at each level we have the running time given in [7] as:

$$RunningTime = O(t \log_t n) \tag{13}$$

In each of the above cases the search time matched the possible number of merges and splits that may also occur in the worst case.

Searches: In the above deletions and insertions we referred to finding or searching for a specific node. Clearly when searching the tree structure, the worst case requires searching from the root to a leaf node or the height of the tree times every key. Thus the running time given in [7] is:

$$RunningTime = O(t \log_t n)$$

4. Where applicable, describe extensions to query languages that use the index.

To the best of my knowledge and research, there are no extensions to SQL or relational algebra that works with relational databases that have extensions based on the B-tree.

2.2 R-tree Index

1. Describe for each index some database applications. Explain how the database applications are benefited by using the index.

R-trees was introduced by Antonin Guttman to provide an index structure for spatial data objects. Although multi-dimensional indexing techniques existed, at that time, some lacked the ability to be efficiently searched when stored in secondary memory. Others were designed in a way that caused difficulty in answering spatial queries [14].

R-trees were specifically designed for spatial databases and have only a few other applications. Spatial access is a rich area that is a specialization of multidimensional indices. An

excellent review of both the general multidimensional access methods and the subset of those designed for spatial data is given by Gaede and Günther [12]. Roussopoulos and Leifker [21] describe the use of R-trees to index pictoral databases. Bohm et. al. [4] review several indexing methods including R-trees for multimedia applications and note that R-trees are among a group of indices that support high dimensional indexing. Mamoulis et. al. [16] give an interesting application of data mining periodic spatial temporal patterns. Within spatial applications R-trees have been used to perform a couple of additional operations to those we mention in response to question 4 below.

1. Nearest Neighbor and k-nearest neighbor [20]
2. Window Queries or range queries (count of objects within a window).

Pagel et. al. [17] presents a closer look at different types of window queries. Most of the operations defined for spatial applications use extensions of the R-tree to optimize performance. R-trees have also been extended to deal with temporal data as well.

An R-tree is a height-balanced tree similar to a B^+-tree with index records in its leaf nodes containing pointers to data objects. If the index is stored on disk, then nodes may correspond to disk pages. The benefits of the index were to provide a fundamentally unique way to search and manage spatial data. Many indices have been based on the R-tree that now support both spatial and temporal indices. I will introduce the R-tree here as a stepping stone to the PR-tree described below for constraint databases.

Leaf nodes contain *index records* of the form

$$(I, tuple - identifier) \tag{14}$$

where the $tuple - identifier$ refers to a tuple in the database and I is an n-dimensional minimum bounding rectangle (MBR). I originally was defined as a tuple of intervals

$$I = ([l_1, u_1], ..., [l_n, u_n]) \tag{15}$$

where each interval $[l_i, u_i]$ defined the extent in the i^{th} dimension.

Non-leaf nodes contain tuples of the form

$$(I, child - pointer) \tag{16}$$

where I is identical to leaf nodes and $child - pointer$ is a pointer to a lower node in the tree in which all nodes are contained within the extent of I.

Let M be the maximum number of entries that will fit in one node and let $m \leq \frac{M}{2}$ be the minimum number of entries in a node. An R-tree satisfies the following properties from [14].

1. Every leaf node contains between m and M index records unless it is the root.
2. For each index record $(I, tupel - identifier)$ in a leaf node, I is the smallest rectangle that spatially contains the n-dimensional data object represented by the indicated tuple.
3. Every non-leaf node has between m and M children unless it is the root.
4. For each entry $(I, child - pointer)$ in a non-leaf node, I is the smallest rectangle that spatially contains the rectangle in the child node.
5. The root node has at least two children unless it is a leaf.
6. All leaves appear on the same level.

Figure 13 shows an example of spatial objects with solid thin lines, and MBRs for those objects in dotted lines. Figure 14 shows the R-tree for those objects.

2. Evaluate the space requirements of each index and the process required to build it.

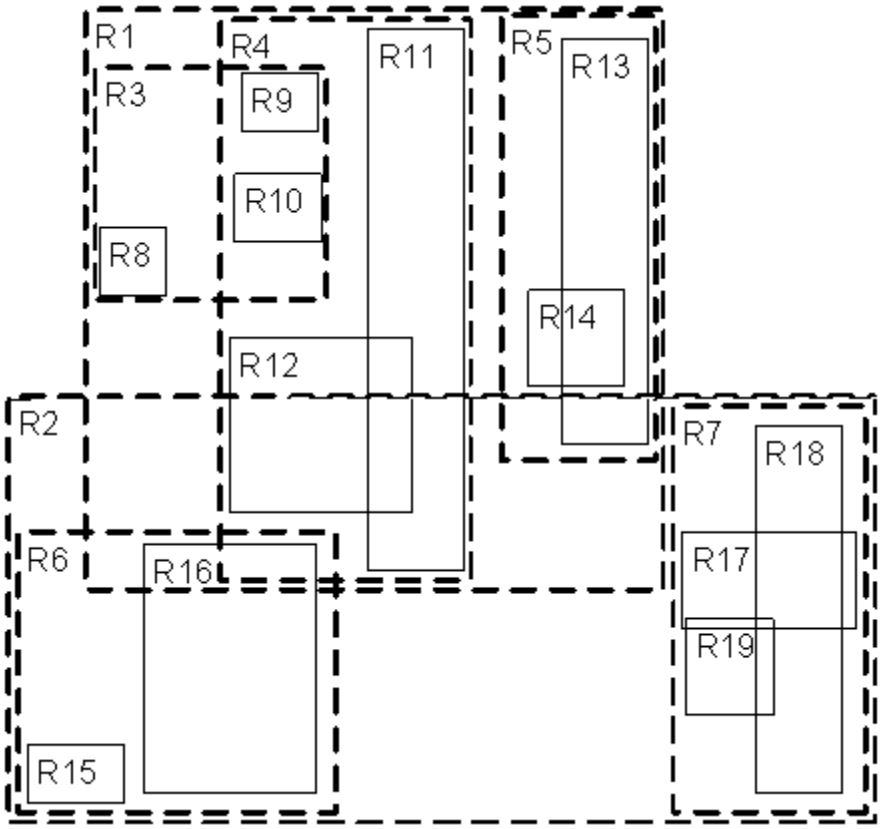

Figure 13: Spatial Objects and MBRs

Build Process: There is no special build process other than to create an empty tree consisting of a root node and then start inserting objects into it. The insert process is evaluated below.

Space Requirements: The space requirements depend upon the number of nodes created for N nodes and the height of the R-tree which is given by Guttman [14] as:

$$h = \lceil \log_m N \rceil - 1 \tag{17}$$

where m is the branching factor. The leaf nodes contain all the objects so we have $\lceil \frac{N}{m} \rceil$ leaf nodes. For each layer traversed up the tree the number of nodes decreases by a factor of m.

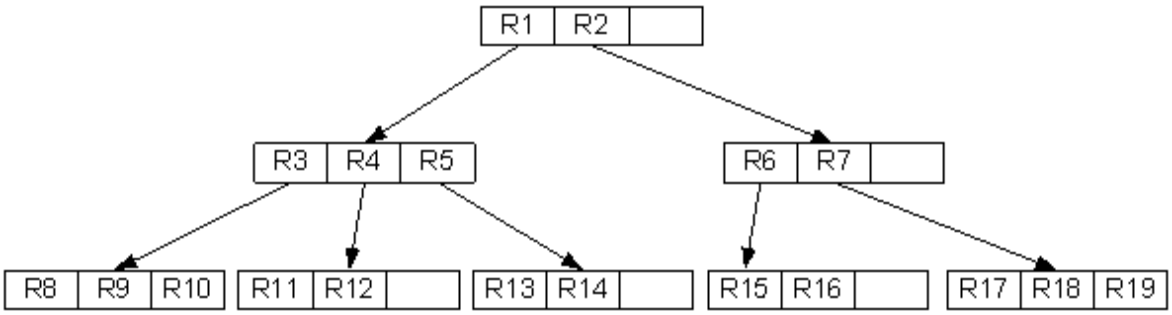

Figure 14: R-Tree

Thus the space requirements for N nodes is give by Guttman [14] as:

$$space = \sum_{i=1}^{h} \left\lceil \frac{N}{m^h} \right\rceil + 1 \tag{18}$$

where we add 1 for the root node.

3. Describe the operations that can be performed on each index. Include in the description the computational complexity of each operation.

R-trees support the four standard operations: Search, Insert, Delete and Update. The algorithms for Search, Insert and Delete are included in Appendix B. Just like the B^+-tree, the R-tree algorithms maintain a balanced tree with data in the leaf nodes.

2.2.1 Searches

As Figure 13 shows, internal node's MBRs may overlap. Thus a search must traverse all paths that contain or overlap the query space. It is easy to construct a set of objects such that every object overlaps every other object. In such a case if you query for a point in the region where all the objects overlap, every leaf node must be visited. In the worst case, the search is no better than a sequential search running time of $O(N)$ where N is the number of objects. However in general, objects do not overlap every other object and in many cases do not overlap at all. In the latter case we are back to the B-Tree search time of $O(m \log_m N)$,

and on average the search time is very close to:

$$SearchTime = O\left(m \log_m N\right) \tag{19}$$

2.2.2 Inserts

Just as there may be more than one leaf node containing a query space, so there may be more than one node into which an object's index entry may be inserted. Finding a node whose MBR contains the object would be the ideal case and even then there may be more than one such leaf node. At any point in the traversal of internal nodes, it is not possible to know which of the possible child nodes leads to a leaf node with a MBR that contains the new object. The heuristic used in [14], finds the MBR that needs the least enlargement if no node's MBR contains the object, or the MBR with the smallest area if more than one node's MBR contains the object. In either case a complete traversal of the tree from root to leaf is required, and possibly several splits back up the tree.

The cost of an insert depends on the cost of several steps. To find the node is just a traversal of the tree and costs $O\left(m \log_m N\right)$. Adding the entry costs $O\left(1\right)$. Traversing the tree back to the root to adjust the MBRs costs $O\left(\log_m N\right)$. The cost of splitting a node on the traversal up the tree depends on the choice of algorithms to split a node. The preferred choice is to use an algorithm that is linear in the number of entries in the node $O\left(m\right)$. This algorithm is fast, but does not always produce a good split. A second algorithm is quadratic in the number of entries and produces slightly better splits. Either one could be considered constant in the number of nodes split since the number of entries is fixed by m. Consequently we have an insert cost of the traversal down plus the traversal up times the cost of the split as:

$$\begin{aligned} InsertTime &= O\left(m \log_m N + m \times \log_m N\right) && (20) \\ &= O\left(m \log_m N\right) && (21) \end{aligned}$$

in the worst case.

2.2.3 Deletes

Deletions can be performed similar to the B^+-tree deletion by borrowing entries from sibling nodes or merging sibling nodes that become underfull. This would involve a search for a node costing $O(m \log_m N)$ followed by a deletion $O(1)$ and then some merging which could be done in $O(M) = O(1)$ for a total cost similar to B^+-trees of $O(m \log_m N)$ [22]. The original R-tree presented in [14] used a more expensive method (shown in Appendix B). Instead of borrowing or merging an underfull node, the underfull node is eliminated and the orphans collected in a set. Traversing up the tree to make sure the parents are not now underfull, nodes are removed and orphaned objects continue to be collected. After the root is reached or we no longer have underfull nodes, the orphans are reinserted into the tree. Thus it is possible to completely rebuild half of the tree during a delete using this method. We chose the previous method and give the running time of deletes as:

$$DeleteTime = O(m \log_m N) \tag{22}$$

2.2.4 Updates

Like many indices, updates to the search key (in this case the spatial dimensions) requires a deletion followed by an insertion. Other types of updates can be done in place. Given that a delete and insertion both cost $O(m \log_m N)$, we have a cost for updates as:

$$O(m \log_m N) \tag{23}$$

4. Where applicable describe an extension to a query language that uses the index.

Spatial databases in general have provided a rich (possibly too rich for this paper) field for extending query languages. In general SQL has been the language of choice to extend, however spatial data is best presented in graphical form and is often stored in graphical form. Thus the result of a query is often times a picture instead of a relation and the input for a query is interpreted from mouse actions on a pictoral display. I chose to describe an extension called Spatial SQL [10]. In this same paper there is a graphical addition called the Graphical Presentation Language (GPL) dealing with the form of the data in the presentation.

The goal of Spatial SQL is to preserve as much as possible the form of SQL while adding new domains, operators and predicates [10] (which the author oddly calls relations).

1. Domains: Essentially types of data that can be stored in a database that the system recognizes as spatial fields. These are denoted $Spatial\_x$ where $x \in \{0, 1, 2, 3\}$ and correspond to points, lines, areas and volumes.
2. Operators: Similar to aggregation operators in where they appear, operators take a spatial domain and map to either another spatial domain or a real number. The mappings are listed after the operators below.
   (a) dimension: $Spatial \rightarrow \mathbb{R}$
   (b) boundary: $Spatial\_x \rightarrow Spatial\_y$
   (c) interior: $Spatial\_x \rightarrow Spatial\_x$
   (d) length, area, volume: $spatial \rightarrow \mathbb{R}$
   (e) distance, directions: $Spatial \times Spatial \rightarrow \mathbb{R}$
3. Relationships are predicate symbols that defining relationships between two spatial objects
   (a) disjoint
   (b) meet
   (c) overlap
   (d) inside/contains
   (e) covers/coveredBy
   (f) equal

All operators listed in (2) use prefix notation, and predicate symbols listed in (3) use infix notation.

Example 7 *Suppose we create a table*

CREATE TABLE city (

name char(20)

geometry spatial_2

point spatial_0);

Then we could perform a query below to determine the distance.

SELECT distance(finish.point,start.point)

*FROM (SELECT * FROM city WHERE name='Lincoln') as start*

*(SELECT * FROM city WHERE name='Omaha') as finish*

We note that the extension above is essentially syntactic sugar, and that behind all these extensions must reside a foundation for handling multi-dimensional objects. Further we note that all these functions are now available in constraint databases [18] which we will look at next as an extension to R-trees.

2.3 Parametric R-trees

1. Describe for each index some example database applications. Explain how the database applications are benefited by using the index.

The Parametric R-tree (PR-Trees) introduced and discussed in [5, 18] were designed specifically for moving objects in constraint databases. Many different database systems now have support grafted into them to support spatial data. We are seeing increased support for moving objects. Indexing moving objects has given rise to many different indices that fall into three broad categories:

1. Trees: Of which there are many variations. Trees are very popular because they scale to work in main memory or in secondary memory and the structure may be searched quickly.

2. Hashes: Generally these hashes are variations of extendable hashing or linear hashing.

For in memory indices, these provide superior speed but do not scale well outside of main memory.

3. Space filling curves: These provide a unique style of indexing that is somewhere between hashing and trees. In fact the Quadtree is a direct descendent of space-filling curves.

The purpose of the broad taxonomy of indices given above is to emphasize that *trees* in some form or another dominate indices. In multidimensional data, usually some descendent of the R-tree is used, and for relational databases usually some type of B-tree is used.

The spatial temporal database application that tracks moving points has all the same goals of spatial databases with the added dimension of time. In fact there is no reason to treat time differently from the other dimensions when the objects rarely move. When objects move and change their motion at frequent intervals, indexing schemes for spatial databases fail to accurately maintain knowledge of the objects location at any given time. This can be seen most clearly in the simple example of a movie scene.

Example 8 *During a scene objects appear to move about as their physical location changes over time. For the human eye it is enough to provide* 30 *frames per second (fps) for the brain to interpret the images as smooth flowing motion. Disregarding for the moment that in many applications (such as high energy physics) the human eye is not fast enough, consider the difficulty of indexing objects through a* 30 *second scene at* 30 *fps. If there are* 10 *objects, we have*

$$10 \times 30 \times 30 = 9000 \tag{24}$$

different tuples. That is fine for a commercial, but what about a 1.5 hour movie? That gives us

$$10 \times 1.5 \times 60 \times 60 \times 30 = 1,620,000 \tag{25}$$

tuples if all we have is 10 *objects in all our scenes.*

In the above example we consider the movie as a window (a very small window) on a database of moving objects. Since the actual database may need to track thousands or even millions of objects at all times in a much larger window it becomes necessary to organize spatial temporal data differently than spatial or relational data. This is where special spatial temporal indices benefit spatiotemporal database applications.

We can solve these problems with appropriate modeling in constraint databases. Constraint databases allow you to model the motion as a function of time and therefore you need not update the position of an object because that position can be determined from its motion as stored in the constraint database. Although it is possible to model non-linear motion in constraint databases using an approximation technique, precise knowledge of motion is not always known from the collection of data. Consequently moving objects are often times modeled as linearly moving objects. Updates then occur to the object to provide a piecewise linear model of the motion. Consider the following example from [18].

Example 9 *The Airplane Relation in Table 2 describes the movement of eight airplanes as parametric rectangles.*

ID	X	Y	T
$r4$	$[7t+30, 7t+50]$	$[6t+80, 6t+100]$	$[0, 10]$
$r5$	$[12t+30, 12t+40]$	$[5t+50, 5t+65]$	$[0, 10]$
$r6$	$[6t+75, 7t+90]$	$[8t+70, 8t+80]$	$[2, 10]$
$r7$	$[0, 15]$	$[5t+40, 5t+55]$	$[0, 9]$
$r8$	$[0, 12]$	$[4t+20, 4t+40]$	$[1, 10]$
$r9$	$[30, 50]$	$[7t+10, 7t+20]$	$[0, 10]$
$r10$	$[-5t+80, -5t+100]$	$[2t, 3t+20]$	$[0, 10]$
$r11$	$[-6t+60, -6t+70]$	$[-3t+30, -2t+40]$	$[0, 10]$

Table 2: Airplane Relation

The minimum bounding parametric rectangles are given by Table 3. This gives a PR-tree as shown in Figure 15.

2. Evaluate the space requirements of each index and the process to build it.

ID	X	Y	T
$r1$	$[7t+30, 7t+90]$	$[5t+50, 6t+100]$	$[0, 10]$
$r2$	$[0, 50]$	$[5t+10, 5t+55]$	$[0, 10]$
$r3$	$[-6t+60, -5t+100]$	$[0, t+40]$	$[0, 10]$

Table 3: Airplane MBPR

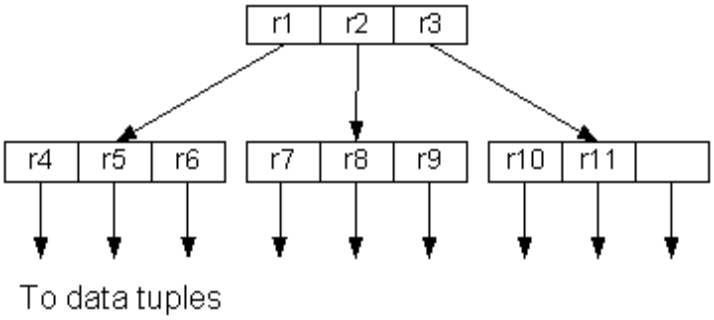

Figure 15: Airplane PR-tree

2.3.1 Minimum Bounding Parametric Rectangles

First we need to define parametric rectangles and minimum bounding parametric rectangles.

Definition 10 *A parametric rectangle is a d-dimensional rectangle defined by the extreme points in each dimension where the extreme points are linear functions of time:*

$$extreme\ points = x_i^{[}(t), x_i^{]}(t) \tag{26}$$

Definition 11 *The minimum bounding parametric rectangle (MBPR) of a set of parametric rectangles S in d dimensions is a parametric rectangle r such that*

1. *r contains all the parametric rectangles in S*
2. *The area of the project of r onto the (x_i, t) space is minimized for each $i = 1, ..., d$.*

The algorithm to find the MBPR of a set S is given in Appendix C

Theorem 12 *[18]The minimum bounding parametric rectangle R of a set S of n, d-dimensional parametric rectangles can be computed in $O(d\ n \log n)$ time.*

The parametric R-tree is defined as follows:

Definition 13 *A parametric R-tree is an index structure for parametric rectangles. It is a height-balanced tree in which each node has between $M/2$ and M children, where M depends upon the page size. Each node of a PR-Tree is associated with a minimum bounding parametric rectangle.*

Build Process: The process to build the PR-tree is to create a root node first and then insert each additional object using the insert algorithm for every other object.

Space Requirements: Suppose we have n moving objects in a PR-tree. Each entry in a node has a linear function for the lower bound and upper bound of each dimension, a lower and upper bound constant for the time and a pointer. Each node has at most M entries. Thus the space requirements for each node will be at most $O(M \times (d+2))$. Since d is a constant and usually a small constant we have $O(M \times (d+2)) \leq O(M)$. Of course both M and d are constants in reality and thus we have constant size nodes but we leave the space requirement as $O(M)$. Since each object is represented by a leaf node, the size of the structure will be the number of leaves, n, plus the number of internal nodes times their size:

$$\begin{aligned} O(n) + O\left(M \times \frac{n}{2}\right) &= O\left(n + \frac{M+2}{2}\right) \\ &= O(n) \end{aligned} \tag{27}$$

The constant hidden in the O notation reflects the fact that a tree has precisely $\left\lfloor \frac{n}{2} \right\rfloor$ internal nodes for n leaves.

3. Describe the operations that can be performed on the index. Include in the description the computational complexity of each operation.

2.3.2 Searching the PR-Tree

Revesz [18][1] shows that searching the PR-tree can be checked in $O(d)$ if two d-dimensional linear parametric rectangles have a non-empty intersection. Thus searching the index for the different types of non-empty intersection defined for spatial queries where we always choose one child, may be done in $O(M \log_M n)$ time. Similar to R-trees however multiple children may have non-empty intersections and thus in the worst case we may have to traverse the tree to every node which degenerates to a sequential search of $O(n)$ time.

2.3.3 Insertion

The insertion algorithm traverses the tree from the root to a leaf node expanding the MBPR at each level. Since a key issue in search performance is minimizing the number of nodes one has to visit, we choose the child node that needs the least volume enlargement. This means calculating the least volume enlargement for each for between $\frac{M}{2}$ and M parametric rectangles at each internal node of the tree. When an appropriate leaf node is found, and the number of elements in the node is less than M, the node is inserted and the process finishes. If the node has M elements the node must be split, and the split propagated up the tree.

When a node must be split, we find the two elements r_i, r_j in the node such that the volume of the MBPR of r_i and r_j is the largest in comparison with the volume of r_i and r_j. We can state this as follows:

$$\max_{r_i, r_j \in E} \left(Vol\left(FindMBPR(r_i, r_j)\right) - \left(Vol\left(r_i\right) + Vol\left(r_j\right)\right)\right) \tag{28}$$

After the first entries for the two nodes have been chosen, we insert the rest of the entries based on the least enlargement principle. For pseudocode see Appendix C.

Theorem 14 *The insertion of a parametric rectangle into a PR-tree can be done in* $O(M^2 d \log_M n)$ *time where* M *is the maximum number of children per node,* d *is the dimension, and* n *is*

[1] by Theorem 14.2.2

the number of parametric rectangles. [18][2]

Proof. As we go down the tree we need at each level to find the appropriate subtree and update the MBPR of the chosen subtree. The computation of two parametric rectangles takes $O(d)$ time. The computation of the volume of a parametric rectangle depends on its dimension and takes $O(d)$ time. There are at most M entries in a node. Hence computing their enlargements and selecting the minimum out of these requires $O(dM)$ time at each level. The height of a PR-tree with n parametric rectangles is $\log_M n$. Therefore, going down the PR-tree to find an appropriate leaf node requires $O(dM \log_M n)$ time.

If we add the new entry to a node that has less than M entries, then the splitting is not required and we are done. Otherwise, we need to split the last non-leaf node and then propagate the split upward. Each split requires finding the pair of nodes that are least desirable together. This requires M^2 computations of MBPRs of two parametric rectangles; hence this can be done in $O(dM^2)$ time. To add each of the $M-2$ other entries to either of the two requires $O(dM)$ time. Because the height of a PR-tree with n parametric rectangles is $\log_M n$, the total time for splitting including propagation upward is $O(dM^2 \log_M n)$.

Finally, note that the PR-tree remains balanced after each insertion because the height of the PR-tree increases only when we split the root node, and that increases each path from the root to a leaf by one. ■

2.3.4 Deletion

Deletions in the PR-tree are accomplished differently than the R-tree. If deletion is desired, a secondary search tree (or hash table) of identifiers with pointers to their associated parametric rectangles in the PR-tree are created for each item on insertion. Then to delete parametric rectangle R with identifier i we search the secondary search tree for i and follow the pointer to R. We must delete both the node in the secondary tree and R from the PR-tree. If the parent node p of R has at least $\frac{M}{2}$ entries after deletion we simply recompute

[2] Theorem 17.2.1

the MBPR for p which takes $O(\log_M n)$ time. However if p has less than $\frac{M}{2}$ entries we do the following: Let p' be the parent of p. Delete p and insert the children of p into the subtree rooted at p'. Revesz [18] gives the running time as:

$$O\left(dM^3 \log_M n\right) \tag{29}$$

For pseudocode see Appendix C.

2.3.5 Updates

Updates are accomplished by deleting and reinserting a node. See Insertions and Deletions above.

4. Where applicable, describe extensions to query languages that uses the index.

Although SQL may be used to query constraint databases, datalog appears to be a favorite method. Since constraint databases natively handle spatial temporal data including moving objects, the indexing method doesn't specifically extend the query language. All the spatial query types introduced for R-trees may be used with the addition of a time parameter. However spatial temporal data in general allows for new types of queries that relate to the new time dimension available. The concept of finding aggregations of max and min of a moving query rectangle over time were introduced and discussed in [19, 6]. Other operations including max in both time and variable space have not been explored in the literature possibly due to the large running time complexity. Tao and Papadias [23] introduced a special case of max aggregation over time by considering dynamic aggregation as a running total within a query window that is valid for the current time only.

2.4 Min-Skew BSP

1. Describe for each index some example database applications. Explain how the database applications are benefited by using the index.

Spatial selectivity estimation was first introduced by Belussi and Faloutsos [3]. Subsequently Archarya Poosala and Ramaswamy [1] gave the *Min-Skew* binary space partitioning index algorithm we review here.

This specialized index is designed specifically for *spatial* databases but may be used for multidimensional data in general. It has also been used in *spatiotemporal* databases. *Data mining* may use this index for approximate clustering as pointed out by Kollios et. al. [15]. The most common use however is with spatial or spatiotemporal data.

Selectivity estimation has been used in relational databases as a method to estimate query running time and query result set size prior to running a query. The goal of selectivity estimation in spatial applications is to partition the space into buckets such that each bucket has a *uniform distribution of objects*. That is we wish to minimize the spatial-skew of the distribution. Spatial-skew is defined in [1] as follows.

Definition 15 *Consider a grouping $\mathcal{G}$ with buckets B_i, $1 \leq i \leq \beta$. Let n_i be the number of points in B_i. The spatial-skew s_i of bucket B_i is the statistical variance of the spatial densities of all points grouped within that bucket. The spatial-skew S of the entire grouping is the weighted sum of spatial-skews of all the buckets: $\sum_1^\beta n_i \times s_i$.*

With a uniform distribution of objects in each bucket, if a query then selects an area a_i of a bucket i with area A_i containing O_i objects, then we can estimate the number of objects returned by a query for that bucket as q_i:

$$q_i = \frac{a_i}{A_i} O_i \tag{30}$$

Chen and Revesz [6] used the index for estimating the max count over time aggregation for moving points using the dual transform of a moving point. This index like most recent indices are specially designed for spatiotemporal databases. However generalizing the method to non-spatial, multi-dimensional data may make it useful for knowledge discovery and data mining. Due to the static nature of the index, it would not be useful for multimedia database

applications where the objects change their motion often, appear, disappear and reappear frequently.

In general, selectivity estimation in its various forms allows for the neglect of some aspects of the stored information while optimizing on other aspect(s). The goal is to approximate the answer to a specific type of query. If a particular application requires only that specific type of query, estimation methods may organize the data in such a way that not all information is preserved, but the information needed to answer the query is still available. Essentially this amounts to pruning or aggregating information in the index to obtain the minimum information needed. This type of highly specialized index has limited use, but provides *extremely fast search capabilities*.

Example 16 *Suppose we have a set of linearly moving object shown in Figure 16. We*

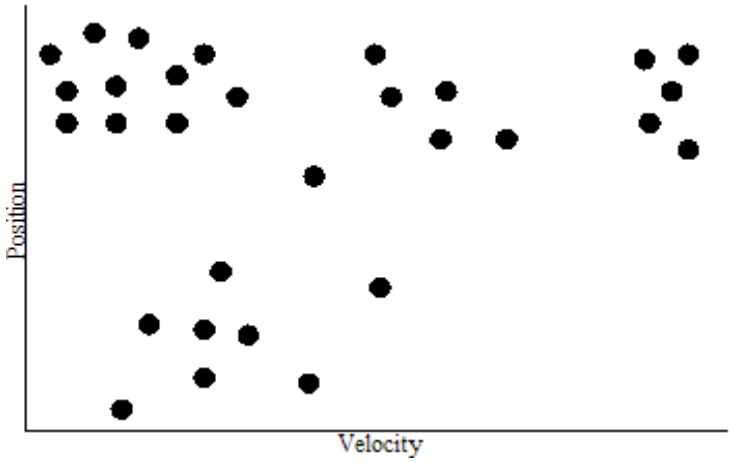

Figure 16: Spatial Distribution

partition the space for selectivity estimation using a binary space partition. That is we partition a bucket that reduces the distribution skew along one of the axes. This is shown in Figure 17 where we have 6 *partitions. The algorithm may be given a specific number of partitions, or it may stop when the maximum skew of any one bucket is under a given value.*

2. Evaluate the space requirements of each index and the process required to build it.

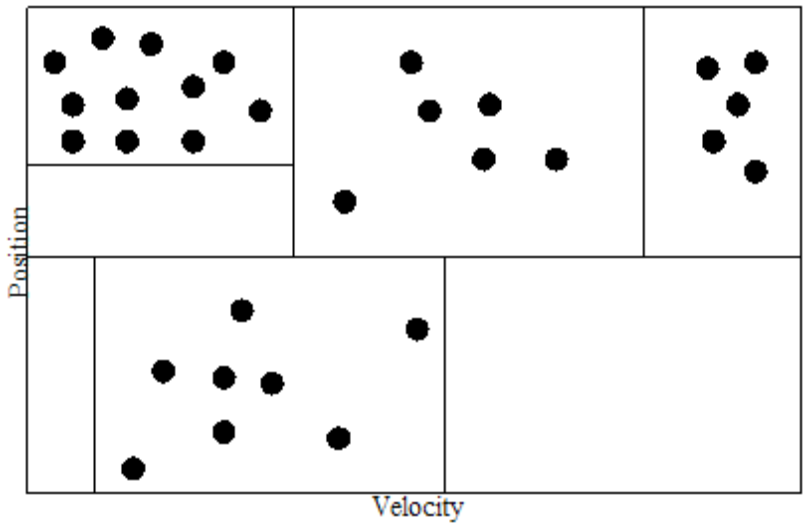

Figure 17: Binary Space Partition

2.4.1 Min-Skew Indexing Algorithm

For the moving points dataset we have a list of i-dimensional points. Each data point is a tuple of position and velocity values:

$$data\ point = (p_1, v_1, ..., p_n, v_n) \tag{31}$$

The algorithm first determines the space in question and overlays it with a grid of cells. In this case the grid is a 4-dimensional grid of hyper-cubes. The size of the cells determines the first level of approximation and each cell is given a count of the objects that overlap it. The histogram starts out with a bucket containing all the cells. Then the bucket is partitioned into n buckets. To chose the partition, we minimize the spatial-skew from Definition 15.

The algorithm to generate the index is given in Figure 18.

2.4.2 Min-Skew Space

Interestingly enough, the Min-Skew algorithm does not specify a structure to store the buckets or cells. Each bucket will have a lower and upper bound for each dimension and the number of objects contained in it when the indexing algorithm has completed. At that

Hyper-Bucket-Indexing(S,c,n,H)
input: S is the set of 4-dimensional points to be indexed, c Cell size, and n hyper-buckets.
output: The histogram H.
Place the points into cells
H gets all the cells in the 4-dimensional space.
while H has less than n buckets
    for each bucket B_i in H **do**
        Compute the spatial-skew of B_i and find the split point along its dimensions that will produce the maximum reduction in spatial-skew
    end for
    Pick the bucket B whose split will lead to the greatest reduction in spatial-skew.
    Split B into two buckets B_1 and B_2 and assign regions from B to B_1 and B_2.
end while
Assign each cell in the input to the hyper-bucket whose Minimum bounding rectangle contains the center of the rectangle

Figure 18: Hyper-bucket indexing algorithm.

point we may discard all the information used to build the index. Thus there is the space requirements to build the index and the space requirement for the index itself.

Suppose we are given n spatial objects in a database. Let l_i, u_i be the lower and upper bounds respectively of the space contained in the database. Let w_i be the width of each cell in the i^{th} dimension. Then we will have c number of cells defined as follows:

$$c = \prod_i \left(\left\lceil \frac{u_i - l_i}{w_i} \right\rceil \right) \tag{32}$$

The space requirements for the grid are then $O(c)$. The number of buckets β is a given parameter. Each bucket contains the upper and lower bounds and the number of objects overlapping it. Thus the space requirements for the buckets is $O(\beta)$.

3. Describe the operations that can be performed on the index. Include in the

description the computational complexity of each operation.

There are only two operations currently supported on this type of index on moving objects: Build and selection estimation. This index assumes static information such as spatial data that rarely if ever changes. Because it assumes all information is available at index build time, there is no way to insert or delete information from the index without upsetting the spatial skew. This is also applicable to what we might consider throw away data. That is data that we need to run several queries on quickly and then discard. We give as an example the one dimensional search algorithm and the max-count search problem from [6].

Let Q_1, Q_2 be two query points such that the answer to the query is the number of points between Q_1 and Q_2 at time t. Now a line through a point p with a slope of $-t$ is such that, points above the line are ahead of the point p and points below the line are behind the point p. Let $P(t) = v_x t + x_0$ and $Q(t) = 2t + 5$. Then $P(t)$ is in front of Q if and only if

$$\begin{aligned} P(t) &> Q(t) \\ v_x t + x_0 &> 2t + 5 \\ x_0 &> (2 - v_x) t + 5 \\ x_0 &> -t(v_x - 2) + 5 \end{aligned} \tag{33}$$

The last inequality is true if and only if the point $P(t)$ is above the line through Q with slope $-t$. This is similarly true for points below the line when switching the inequality.

Adding two query points Q_1 and Q_2 to Figure 17, we show the query area as the grey band bounded by the lines through the query points in Figure 19. The query area intersects three buckets, but one of them is empty and does not contribute to the selection estimate. The top bucket actually contains all the objects, but will return a smaller value for that bucket since it does not cover the entire area of the bucket. The lower left bucket includes 3 out of 8 points, but will return more than 3 because it covers about half of the bucket. Thus

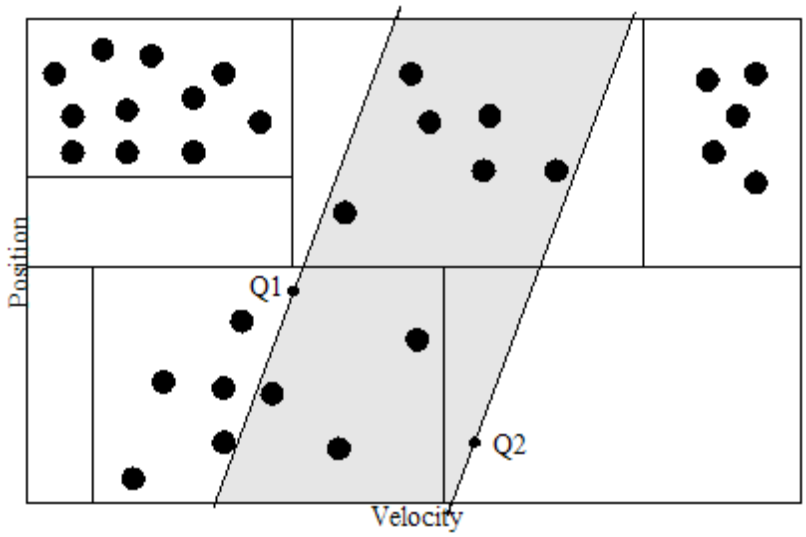

Figure 19: Query Area

we see in this example that the estimated result will be close to the actual result.

4. Where applicable, describe extensions to query languages that uses the index.

Selectivity estimation to find a fast estimate of the number of objects in spatial database is an addition that query languages may indirectly support using a new operator or by turning on the ability to estimate.

An extension to selectivity estimation is Max-Count over time described in [6] which extends a query language to include a max-count aggregation estimation operator. To understand how the operator implementation works is much easier than to wade through the math involved. If we allow time to change in the query depicted in Figure 19, then the lines will still go through Q_1 and Q_2, but the slopes of the lines will change as $-t$ changes. As long as the lines stay in the same buckets, we can write the area in each bucket as a function of t. The form of the function is given in [6] as:

$$Area\,(t) = at + \frac{b}{t} + c \tag{34}$$

where a, b, c are real constants corresponds to the different polygon shapes that the query

area may form. Thus we can maximize the sum of the areas to find an estimated max-count over time segments where Equation 34 does not change. The sides used to calculate the query area in a bucket change when a line crosses a vertex and thus Equation 34 will change whenever a line crosses a vertex of a bucket. The maximum number of vertices that the lines may cross is $O(B)$ and thus we can calculate the estimated max-count over time in $O(B)$.

3 Current trends in indexing research

The concepts of indices are most often associated with databases, however information retrieval applications such as search engines use indexing techniques to search unstructured or semi-structured data. This has been a hot area of research in the past few years in companies such as Google.com. Current trends include the indexing of all manner of media including voice and video. Research in this area is ongoing.

The concepts of the *semantic web* thought up by world wide web inventor Tim Burners-Lee is an active area of research that includes indexing annotated data using the resource definition framework (RDF). According to the W3C working group [13] the semantic web is still developing. "The Semantic Web is a vision: the idea of having data on the web defined and linked in a way that it can be used by machines not just for display purposes, but for automation, integration and reuse of data across various applications." There has also been efforts to annotate unstructured data. Dowman et. al. [9] recently introduced a method for temporally accurate, conceptual semantic annotation of broadcast news using speech recognition to match audio resources to textual news stories on the web. Ding et. al. [8] introduced swoogle: a crawler-based indexing and retrieval system for the Semantic Web. Finin et. al. [11] give a nice overview of information retreival and the Semantic Web. The few examples we cited above show that indexing different types of media is a trend that continues to develop.

Indexing is also developing in spatiotemporal applications as we have reviewed above.

Other areas such as knowledge discovery and datamining also are developing advanced indexing techniques for specialized areas of interest.

Instead of diving into these areas and providing additional references, I would like to identify two divergent underlying trends. The first trend is in developing indices for specialized applications. Some of our example indices we reviewed and the examples cited above show that indices have entered an era of specialization. Special indices such as R-trees, PR-trees, R^+-trees, R^*-trees, K-D-trees as well as space-filling curves and many others have been developed in an effort to solve specific types of spatial and spatiotemporal problems. This trend towards specialization in many applications has led leading database venders such as Oracle to include the capability to add external indices. Thus we have a trend toward diverging indices due to specialization.

The second trend is less a trend than a recognition of convergence to the optimal data structure: the tree. Within the divergent trend identified above most index methods for secondary storage have used some form of the tree data structure. This trend of optimizing the tree structure for special applications is likely to continue.

We conclude that just as for relational databases where the B^+-tree became a standard for relational database applications, eventually each separate type of application will evolve a standard index data structure. Because many types of information (spatiotemporal, audio, video, and constraint data) are continuously being stored and because the semantics of these data are trying to be captured in indices, we believe that the area of indices will be a rich field for research for some time to come.

References

[1] Acharya, S., Poosala, V., and Ramaswamy, S. Selectivity estimation in spatial databases. In *Proceedings of the ACM SIGMOD International Conference on Management of data* (1999), pp. 13–24.

[2] Bayer, R., and McCreight, E. M. Organization and maintenance of large ordered indices. *Acta Inf. 1* (1972), 173–189.

[3] Belussi, A., and Faloutsos, C. Estimating the selectivity of spatial queries using the 'correlation' fractal dimension. In *Proc. 21st Int. Conf. Very Large Data Bases, VLDB* (11–15 1995), U. Dayal, P. M. D. Gray, and S. Nishio, Eds., Morgan Kaufmann, pp. 299–310.

[4] Böhm, C., Berchtold, S., and Keim, D. A. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv. 33*, 3 (2001), 322–373.

[5] Cai, M., and Revesz, P. Parametric r-tree: An index structure for moving objects. In *Proceedings of the 10th COMAD International Conference on Management of Data* (2000), Tata McGraw-Hill, pp. 57–64.

[6] Chen, Y., and Revesz, P. Max-count aggregation estimation for moving points. In *Proceedings of the 11th International Symposium on Temporal Representation and Reasoning* (2004), pp. 103–108.

[7] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms*, 2nd ed. MIT Press, Massachusetts, 2001.

[8] Ding, L., Finin, T., Joshi, A., Pan, R., Cost, R. S., Peng, Y., Reddivari, P., Doshi, V., and Sachs, J. Swoogle: a search and metadata engine for the semantic web. In *CIKM '04: Proceedings of the thirteenth ACM international conference on Information and knowledge management* (New York, NY, USA, 2004), ACM Press, pp. 652–659.

[9] Dowman, M., Tablan, V., Cunningham, H., and Popov, B. Web-assisted annotation, semantic indexing and search of television and radio news. In *WWW '05: Proceedings of the 14th international conference on World Wide Web* (New York, NY, USA, 2005), ACM Press, pp. 225–234.

[10] Egenhofer, M. J. Spatial sql: A query and presentation language. *IEEE Transactions on Knowledge and Data Engineering 6*, 1 (1994), 86–95.

[11] Finin, T., Mayfield, J., Joshi, A., Cost, R. S., and Fink, C. Information retrieval and the semantic web. In *System Sciences, 2005. HICSS '05. Proceedings of the 38th Annual Hawaii International Conference on* (2005), pp. 113a–113a.

[12] Gaede, V., and Günther, O. Multidimensional access methods. *ACM Computing Surveys 30*, 2 (1998), 170–231.

[13] Group, W. W. The semantic web. www, 2005.

[14] Guttman, A. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference* (1984), B. Yormark, Ed., ACM Press, pp. 47–57.

[15] Kollios, G., Gunopulos, D., Koudas, N., and Berchtold, S. Efficient biased sampling for approximate clustering and outlier detection in large datasets. *IEEE Transactions on Knowledge and Data Engineering 15*, 5 (2003).

[16] Mamoulis, N., Cao, H., Kollios, G., Hadjieleftheriou, M., Tao, Y., and Cheung, D. W. Mining, indexing, and querying historical spatiotemporal data. In *KDD '04: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining* (New York, NY, USA, 2004), ACM Press, pp. 236–245.

[17] Pagel, B., Six, H., Toben, H., and Widmayer, P. Toward an analysis of range query performance in spatial data structures. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (May 1993), pp. 214–221.

[18] Revesz, P. *Introduction to Constraint Databases*. Springer-Verlag, 2002.

[19] Revesz, P., and Chen, Y. Efficient aggregation over moving objects. In *Proceedings of the 10th International Symposium on Temporal Representation and Reasoning, Fourth International Conference on Temporal Logic* (2003), pp. 118–127.

[20] Roussopoulos, N., Kelley, S., and Vincent, F. Nearest neighbor queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1995), pp. 71–79.

[21] Roussopoulos, N., and Leifker, D. Direct spatial search on pictorial databases using packed r-trees. In *SIGMOD '85: Proceedings of the 1985 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1985), ACM Press, pp. 17–31.

[22] Silberschatz, A., Korth, H. F., and Sudarshan, S. *Database System Concepts*, 4th ed. McGraw-Hill, 1221 Avenue of the Americas, New York, NY 10020, 2002.

[23] Tao, Y., and Papadias, D. Spatial queries in dynamic environments. *ACM Trans. Database Syst. 28*, 2 (2003), 101–139.

[24] Unattributed. Mysql 5.0 reference manual: How mysql uses indexes. url, mysql.com, 2005. http://dev.mysql.com/doc/refman/5.0/en/mysql-indexes.html.

[25] Weisstein, E. W. B-tree. Tech. rep., From MathWorld–A Wolfram Web Resource, 1999. http://mathworld.wolfram.com/B-Tree.html.

A B-tree Algorithms

Below are the algorithms for search, delete, and insert. Each node has a boolean value indicating if it is a leaf: leaf[x] and a value indicating the number of entries in the node n[x]. The tree has a root pointer: root[x]. All the pseudocode given below is taken from [7]

```
B-tree-Create(T) //tree T
    x = Allocate-Node()
    leaf[x] = true
    n[x] = 0
    root[T] = x


B-tree-Search(x,k) //x-node, search key k
    i = 1
    while i ≤ n[x] and k > key_i[x]
        i = i+1
    if i ≤ n[x] and k = key_i[x]
        return (x,i)
    if leaf[x]
        then return null
        else return B-tree-Search(Child_i[x], k)

B-tree-Insert(T,k)
    r = root[T]
    if n[r] = 2t-1
    then {
        s = Allocate-Node()
        root[T] = s
        leaf[s] = false
        n[s] = 0
        child_1[s] = r
        B-tree-Split-Child(s,1,r)
        B-tree-Insert-NonFull(s,k)
    }
    else {
        B-tree-Insert-NonFull(r,k)
    }


B-tree-Insert-NonFull(x,k)
    i = n[x]
    if leaf[x]
    then {
        while i ≥ 1 and k < key_i[x]
        {
```

```
            key_{i+1}[x] = key_i[x]
            i = i - 1
        }
        key_{i+1}[x] = k
        n[x] = n[x] + 1
    }
    else {
        while i ≥ 1 and k < key_i[x]
        {
            i = i - 1
        }
        i = i + 1
        if n[child_i[x]] = 2t - 1
        then {
            B-Tree-Split-Child(x,i,child_i[x])
            if k > key_i[x]
                i = i + 1
        }
        B-tree-Insert-NonFull(child_i[x],k)
    }

B-tree-Delete(root[T],k)
    (x,i) = B-tree-Search(root[T], k)
    if x == null
        return null
//Step 1
    if leaf[x] == true
    then {
        delete key_i[x]
    }
//Step 2
    else {
    // 2.a
        if n[child_{i-1}[x]] ≥ t
            (y,j) = succ(child_{i-1}[x])
            k = key_j[y]
            B-tree-Detele(y,key_j[y])
    //2.b
        else if n[child_i[x] ≥ t
            (y,j) = pred(child_i[x])
            k = key_j[y]
            B-tree-Detele(y,key_j[y])
    //2.c
        else
            y = merge(child_{i-1}[x],k,child_i[x])
```

```
            for j=i to n[x]-1
                key_j = key_{j+1}
                child_j[x] = child_{j+1}[x]
            n[x] = n[x] - 1
            free(child_i[x])
    }
    //Step 3
    (c) = ChildContaining(x,k)
    if (c == null)
        return //we deleted it return
    if (leaf[c] != true)
        //step 3.a
        if n[child_i[c]] = t-1
            if (n[child_{i-1}[x]] > t-1)
                borrowleft(x,i)
            if (n[child_{i+1}[x]] > t-1)
                borrowright(x,i)
        //Step 3.b
        else
            c = merge(child_{i-1}[x],k,child_i[x])
            for j=i to n[x]-1
                key_j = key_{j+1}
                child_j[x] = child_{j+1}[x]
            n[x] = n[x] - 1
            free(child_i[x])
    B-tree-Delete(c,k)


borrowleft(x,i)
    y = child_{i-1}[x]
    z = child_i[x]
    k = key_i[x]
    key_i[x] = key_{n[y]}[y]
    insertKeyInFront(z,k)


borrowright(x,i)
    y = child_{i+1}[x]
    z = child_i[x]
    k = key_i[x]
    key_i[x] = key_1[y]
    addKeyToEnd(z,k)


succ(x)
```

```
    if (leaf[x] == true)
        return (x,n[x])
    else
        return succ(child_n[x])


pred(x)
    if(leaf[x] == true)
        return (x,1)
    else
        return succ(child_1[x])


B-tree-Split-Child(x,i,y) //where y is the i^th child of x and y is being split.
    z = Allocate-Node()
    leaf[z] = leaf[y]
    n[z] = t - 1 //where t is the minimum degree of a node
    for j=1 to t-1
        key_j[z] = key_{j+t}[y]
    if not leaf[y]
    {
        for j = 1 to t
            c_j[z] = c_{j+t}[y]
    }
    for j = n[x] downto i
        key_{j+1}[x] = key_j[x[
    key_i[x] = key_t[y]
    n[x] = n[x] + 1
```

B R-Tree Algorithms

Below are algorithms Search, Insert, and Delete adapted to pseudo code from [14]. The pseudo code exudes an arroma of java. Not all the methods used exist in the code, but those omitted should have clear meaning.

```
Search(SearchRectangle S, R-Tree T)
    ReturnVal = {}
    if (!isLeaf(T))
    For (Entry E : root(T))
        if (doesOverlap(E,S))
            ReturnVal = ReturnVal ∪ Search(E.Node, S)
    else
        For (Entry E : root(T))
            if (doesOverlap(E,S))
```

```
                ReturnVal = ReturnVal ∪ E
    return ReturnVal

Insert(Entry E, R-Tree T)
    Node L = chooseLeaf(E,T)
    if (hasRoom(L))
        L.add(E)
    else
        Node LL = splitNode(L,E) //Splits node L →L,LL and adds E to L
        If (isRoot(L))
            Create a new root and point to L and LL.
        AdjustTree(L,LL)


Delete(Entry E, R-Tree T)
    L = findLeaf(E)
    if (L == null)
        return
    L.remove(E)
    CondesnseTree(L)
    if (oneChildRoot(T))
        make child new root.
    return


ChoseLeaf(Entry E, R-Tree T)
    Set N=root(T)
    While (!isLeaf(N))
        Let F be the entry in N whose rectangle needs the least
            enlargement to include E. Resolve ties by using the
            rectangle with the smallest ares.
        N=F
    return N

AdjustTree(Node L, Node LL)
    N=L
    NN=LL
    While (!isRoot(N))
        P = parent(N)
        E_N=entry(N,P) //the entry of N in the parent.
        Adjust E_N so that it tightly encloses all entry rectangles in N
        If (NN != null)
            create E_NN with a pointer to NN
            if (isRoom(P))
                P.add(E_NN)
```

```
            else
                PP = SplitNode(P,E_NN)
        N = P
        NN = PP
    return
```

Eliminate underfull node and collect the orphans in Q. Walk up thetree doing this and then, reinsert the orphans. Deletes can be expensive.

```
CondenseTree(Node L, R-Tree T)
    N=L
    Q={} //the set of eliminated nodes
    while (!isRoot(N))
        P = parent(N)
        E_N=entry(N,P) //the entry of N in hte parent.
        if (entryCount(N) < m)
            delete E_N from P.
            Q = Q ∪ E_N
        else
            Adjust E_N so that it tightly encloses
           all entry rectangles in N
        N = P
    for (E in Q)
        insert(E,T)
FindLeaf(Entry E, Node T)
    if (!isLeaf(T))
        for (Entry F : T)
            if (F.contains(E))
                FindLeaf(E, F.Node)
    else
        for (Entry F : T)
            if (F == E)
                return F
    return null
Quadradic Split(Node L, Entry E)
    Set<Entry> M = Node.entries ∪ E
    L = new Node()
    LL = new Node()
    L.add(PickSeeds())
    LL.add(PickSeeds())
    while (M.count > 0)
        if (L.count - m = M.count) //the rest of the entries must go to L
            L.addAll(M)
        else if (LL.count - m = M.count) // the rest go to LL
            LL.addAll(M)
        else
```

```
            Entry EE = PickNext()
            if (L.coversAmount(EE) > LL.coversAmount(EE)))
                L.add(EE)
            else if (LL.coversAmount(EE) > L.coversAmount(EE))
                LL.add(EE)
            else if (L.area() < LL.area())
                L.add(EE)
            else if (LL.area() < L.area())
                LL.add(EE)
            else if (booleanRandomChoice)
                L.add(EE)
            else
                LL.add(EE)
    return LL //we assume L is passed by reference and is updated
PickSeeds(Set<Entry> M, Entry S1, Entry S2)
    int cd = infinity
    for all combinations of E_i, E_j ∈ M such that i ≠ j
        j = MBR(E_1, E_2)
        d = j.area − E_1.area − E_2.area
        if (d < cd)
            cd = d
            S1 = E_i
            S2 = E_j
return
LinearPickSeeds(Set<Entry> M, Entry S1, Entry S2)
    Mbr = MBR(M)
    diff[] = new int[dim]
    d=0
    lowest[] = {E_inf, E_inf, ..., E_inf}
    highest[] = {E_−inf, E_−inf, ..., E_−inf}
    for (i = 1; i ≤ dim; i++)
        for (Entry E : M)
            if (E.lowerBound[i]>highest[i].lowerBound[i])
                highest[i] = E
            if (E.upperBound[i]<lowest[i].upperBound[i])
                lowest[i] = E
    for (i = 1; i ≤ dim; i++)
        diff[i] = (highest[i].lowerBound[i] - lowest[i].upperBount[i]) / Mbr.width[i]
        if (diff[i] > d)
            S1 = highest[i]
            S2 = lowest[i]
return
PickNext(Set<Entry> M, Node L, Node LL)
    min_cost = infinity
    Entry pick
```

```
for (Entry E : M)
    d1 = area increase required if covered by L
    d2 = area increase requried if covered by LL
    if (|(d1 − d2)| < min_cost)
        pick = E
return pick
```

C MBPR Algorithms

This section gives pseudocode for the insert and delete operations of the PR-tree. This is just simple code that outlines the operations and you will find some concepts written in, but it is intended to give a feel for the algorithms.

```
FindMBPR(Set<ParametricRectangles> S)
t_min= min_{r∈S} (r, t^[)
t_max= max_{r∈S} (r, t^])
t_med= (t_max + t_min) /2
for each dimension x_i do
    Find S_i the set of extreme points of projections (x_i, t) of S
    Compute the convex hull H_i of S_i
    Find the edges of H_i that intersect with t_med
    Construct x_i^[, x_i^] for the MBPR

Insert(ParametricRectangle pr)
    //Find the appropriate leaf (simple search through the tree using
    //FindMBPR where we choose the node with the least enlarment of
    //the MBPR)
    Node n = findLeaf(pr)
    if (n.count < M/2)
        add the rectangle to the node
    else
        Node nn = split(n)
        n.parent.add(nn)

split(Node n)
    find r_i, r_j such that max_{r_i,r_j∈E} Vol(FindMBPR(r_i, r_j)) − (Vol(r_i) + Vol(r_j))
    Node n1.add(r_i)
    Node n2.add(r_j)
    for k ≠ i, j
        insert r_k into r_l where l is given by
        min_{l∈{i,j}} (Vol(FindMBR(r_l, r_k)) − Vol(r_k))
    n = n1
    return n2

Delete(SecondaryTree b, PRtree pt, id)
    SecondaryNode sn = b.find(id);
    PRNode p = sn.parametricrectangle.parent();
    p.delete(sn.parametricrectangle)

PRNode.delete(ParametricRectangle pr, ChildSet cs)
    remove pr
    if (childcount < M/2)
        p2 = this.parent();
```

```
    if cs == null
        cs = new Set<Child>
    cs.addAll(myChildren)
    p2.delete(p,cs)
else
    adjust MBPR(pr)
```