THE RD-TREE: AN INDEX STRUCTURE FOR SETS

Joseph M. Hellerstein
University of Wisconsin, Madison

Avi Pfeffer
University of California, Berkeley

Abstract

The implementation of complex types in Object-Relational database systems requires the development of efficient access methods. In this paper we describe the RD-Tree, an index structure for set-valued attributes. The RD-Tree is an adaptation of the R-Tree that exploits a natural analogy between spatial objects and sets. A particular engineering difficulty arises in representing the keys in an RD-Tree. We propose several different representations, and describe the tradeoffs of using each. An implementation and validation of this work is underway in the SHORE object repository.

1. INTRODUCTION

Traditional relational database systems (RDBMSs) have excellent query processing capabilities, but suffer from a rigid and semantically impoverished data model. Work in Object Oriented databases (OODBMSs) has stressed the need for a richer data model that allows complex types. Among the requirements of this model is support for **set-valued attributes,** which are record elements of type set-of-x, where x is some type known to the system. These sets naturally occur in association with single objects, as, for example, the set of courses taken by a student, or the set of keywords in a document.

Much work has been done in carefully defining data models and languages for OODBMSs, but less attention has been paid to actually processing queries in such a system. There are several commercial OODBMS products that have limited or non-existent query processing and optimization facilities.

Object-Relational database systems (O-R systems) such as Postgres, Starburst and the commercial products UniSQL and Illustra [STON93], attempt to combine the richness of the OODBMS data model with the query processing performance of RDBMSs. In order to achieve this they require efficient support for manipulation of complex objects. In particular, they must be able to evaluate predicates involving set-valued attributes efficiently. Natural queries on sets are not well supported in current O-R systems, because there are no efficient access methods for set valued attributes.

In this paper we describe the RD-Tree, an index structure for sets. The RD-Tree is a variant of the R-Tree, a popular access method for spatial data [GUTT84]. RD stands for "Russian Doll", which describes the transitive containment relation that is fundamental to the tree structure. We discuss the engineering issues involved in representing the keys in an RD-Tree, and propose several representations. RD-Trees have been implemented in Illustra and in the SHORE object repository, and we plan extensive tests to demonstrate the validity of this approach and evaluate the various key representations.

1.1. SAMPLE QUERIES

To illustrate the types of queries that involve set predicates, we consider a sample database with two class definitions:

STUDENT = [*name*: text, *passed*: set of COURSE]
COURSE = [*name*: text, *department*: text, *number*: int, *prerequisites*: set of COURSE]

The predicates we want to evaluate on this database can be divided into several categories:

1) **superset predicates**

select STUDENT.name
from STUDENT
where STUDENT.passed $\supseteq$ {CS186, CS162}†

This query selects all students who have passed CS186 and CS162. In the predicate of this query we are searching for supersets of a given set. An RD-Tree on the STUDENT.passed attribute will greatly facilitate evaluation of this predicate.

2) **subset predicates**

select COURSE.name, COURSE.department, COURSE.number
from COURSE
where COURSE.prerequisites $\subseteq$ {CS186, CS182}

This query selects all courses that can be taken by a student who has passed CS186 and CS182. In the predicate of this query we are searching for subsets of a given set. Unfortunately the RD-Tree does not work well on this type of predicate. However, an inverted RD-Tree as described in section 4 can be used to evaluate subset predicates.

3) **overlap predicates**

select STUDENT.name
from STUDENT
where STUDENT.passed $\cap$ {CS150, CS186, CS162} $\geq$ 2

This query selects all students who have passed at least two of CS150, CS186 and CS162. RD-Trees are effective for overlap queries. If the degree of overlap required is equal to the cardinality of the given set, the predicate is equivalent to a superset predicate.

4) joins

select STUDENT.name, COURSE.name
from STUDENT, COURSE
where COURSE.prerequisites $\subseteq$ STUDENT.passed

† This is a minor abuse of notation. Sets of courses are normally represented by sets of object ids of course tuples.

This query selects all (STUDENT, COURSE) pairs where the student has satisfied the prerequisites for the course. It can be executed as a nested-loop join where the outer relation is COURSE and the inner relation is STUDENT. The join then becomes a series of superset predicates for which an RD-Tree index on STUDENT.passed can be used.

1.2. RELATED WORK

There has been a fair amount of work done on access methods for nested attributes. Bertino and Kim [BERT89] proposed three indexing mechanisms for complex objects: the nested index, path index and multiindex. However, these access methods are not designed to support efficient evaluation of set predicates.

Ishikawa *et al.* [ISHI93] examined the use of signature file techniques for testing set inclusion. They provide a probabilistic algorithm that attempts to quickly determine whether one set is a subset of another. In some cases the algorithm will return "false", in others it will return "don't know" and other methods must be used for determining the answer. Thus signature files provide a useful filter for restricting set predicates. However, they are not an indexing method, as the entire signature file must be scanned when evaluating a predicate.

Signature file techniques and RD-Trees can coexist. Signatures provide a way to quickly resolve a comparison of two specific sets, while the RD-Tree is an access method that guides the DBMS to the appropriate data. The RD-Tree can use signatures to perform individual set comparisons.

Inverted files are a popular technique for single-element lookups in a collection of set-valued attributes. See, for example, [BROW94]. Inverted files can be extended to search for supersets of a given set S with n elements, by performing a single-element lookup for each element in S and then taking the n-way intersection of the results. This can become inefficient if n is large, especially if the n result sets are large.

2. THE RD-TREE STRUCTURE

The structure of an RD-Tree is similar to that of an R-Tree. Leaf nodes in an R-Tree contain entries of the form (*data object*, *bounding box*). The bounding box is the smallest n-dimensional rectangle that contains the data object. Non-leaf nodes in an R-Tree contain entries of the form (*child pointer*, *bounding box*). In this case the bounding box is the smallest n-dimensional rectangle that contains all the bounding boxes of the entries in the child node.

Any data object indexed by the R-Tree is contained in its own bounding box and in the bounding box of all its ancestors in the tree. This transitive containment relation is at the heart of the R-Tree structure. It allows entire branches of the tree to be rejected when searching for a particular region in space. RD-Trees rely on a similar transitive containment relation, the set inclusion relation.

We refer to the sets being indexed by the RD-Tree as **base sets,** and the elements which comprise them as **base elements.** The set of all base elements is the **universe.** Every base set has a **bounding set,** which is the smallest set containing the base set that satisfies certain properties. This is analogous to the idea that the bounding box of a polygon is a rectangle, which is a polygon with special properties. The particular properties that the bounding set must satisfy depends on the choice of representation of keys, to be discussed in section 3.

Leaf nodes in an RD-Tree contain entries of the form (*base set*, *bounding set*). Non-leaf nodes contain entries of the form (*child pointer*, *bounding set*). The bounding set of a non-leaf node entry must contain all the bounding sets in the child node. Thus it is the bounding set of the union of the bounding sets in the child node. Each base set is contained in its bounding set, and each bounding set is contained in the bounding set of its parent, so the transitive set inclusion relation that is crucial to the RD-Tree structure exists.

To find all base sets which are supersets of a given set, begin at the root node of the RD-Tree. On any non-leaf node, examine the bounding set of each entry. If the bounding set is not a superset of the given set, then none of the base sets descended from it can be either, so the branch of the tree rooted at that entry can be discarded. If the bounding set is a superset of the given set, then its child node must be examined. On a leaf node, the base sets can be directly examined, but in some cases it may be advantageous to examine the bounding sets first.

Overlap searches are performed similarly. If the bounding set of a non-leaf entry does not overlap with the given set by the required amount, then the entire branch descended from that entry can be discarded from the search. If the bounding set of the entry being examined does overlap by the given amount, then its child node must be searched.

When an object is inserted into an R-Tree, the position of the object in the tree must be determined. On each non-leaf node, the insertion algorithm examines all the bounding boxes and chooses the one whose bounding box needs least enlargement to include the new object. An alternative heuristic would be to choose the rectangle with the largest overlap with the new object. The analogous heuristic in the RD-Tree is to choose the bounding set whose intersection with the set to be inserted has the greatest cardinality.

Similarly, the algorithms to split a node in an R-Tree involve heuristics to make the two new nodes as disjoint as possible and minimize the areas of their bounding boxes. Analogous heuristics exist for the RD-Tree. It is desirable to place two entries whose bounding boxes have large intersection in the same node, and entries whose bounding boxes have little intersection in different nodes. The quadratic-cost R-Tree node splitting algorithm has a direct analog in RD-Trees. The linear-cost algorithm relies on the geometrical concepts of "high" and "low" and is not immediately generalizable to RD-Trees. However, for some representations of bounding sets described below, a linear-cost splitting algorithm can be used.

2.1. AN EXAMPLE

To illustrate how an RD-Tree can be used to index sets, consider an example with six sets containing integers from 0 to 9:

S1 = {1, 2, 3, 5, 6, 9}
S2 = {1, 2, 5}
S3 = {0, 5, 6, 9}
S4 = {1, 4, 5, 8}
S5 = {0, 9}
S6 = {3, 5, 6, 7, 8}
S7 = {4, 7, 9}

In this simple example, we define the bounding box of a set to be the set itself, and use an RD-Tree with up to two entries in each node. In practice RD-Trees would have a much larger number of entries in a node. The RD-Tree index to these sets is shown in figure 1.

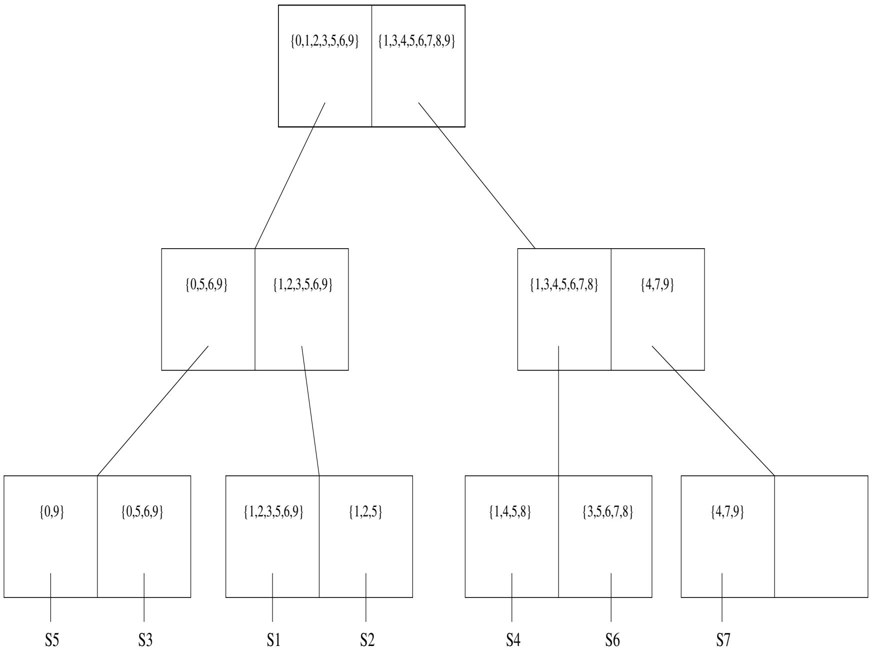

Figure 1. A sample RD-Tree

A search for all supersets of {2, 9} will begin at the root of the tree. Since {1, 3, 4, 5, 6, 7, 8, 9} is not a superset of {2, 9}, the right subtree is discarded. {0, 1, 2, 3, 5, 6, 9} is a superset of {2, 9} so the left child of the root is examined next. {0, 5, 6, 9} is not a superset of {2, 9}, so the left subtree is rejected, but {1, 2, 3, 5, 6, 9} is a superset, so its child is examined. This is a leaf node, so the base sets can be examined directly, revealing the answer S1.

Insertion of the set S8 = {2, 4, 8, 9} would begin by choosing the right subtree at the root, because {2, 4, 8, 9} has three elements in common with {1, 3, 4, 5, 6, 7, 8, 9} and only two with {0, 1, 2, 3, 5, 6, 9}. Descending to the right child, we find that {2, 4, 8, 9} has two elements in common with both {1, 3, 4, 5, 6, 7, 8} and {4, 7, 9}. Since {4, 7, 9} is smaller, we choose the right branch again. Finally, the child node is a leaf, and S8 is inserted into the empty space.

2.2. STORING VARIABLE LENGTH KEYS

A technical issue that must be addressed when implementing RD-Trees is the fact that the keys describing the bounding sets may not all be the same size. In an R-Tree the bounding box is specified by a fixed number of *n*-dimensional points, so the keys are all the same size. However, some of the bounding set representations described in section 3 allow variable length keys.

This problem complicates the RD-Tree in several ways. First of all, the maximum number of keys that will fit on a page becomes variable, making analysis and tuning more difficult. A more serious difficulty is that an entry in a non-leaf node may change size whenever a base set is inserted into or deleted from one of its descendants. This may require that the entry be moved, and in some cases it will no longer fit in its node.

When this situation occurs, we remove the growing entry from the node it is currently in, and split the node. It is then unclear into which new node the growing entry should be inserted. Rather than attempt to calculate which node to use, we take advantage of a feature of R*-Trees [BECK90] which is available in SHORE. R*-Trees allow *forced reinserts:* that is, any entry can be inserted at any level of the tree. The tree chooses which node at that level will be used.

3. REPRESENTATION OF KEYS

The keys in an RD-Tree describe the bounding sets of entries. The precise definition of the bounding set depends on the choice of representation of the keys. A good representation will satisfy several criteria:

size A good key will be small, so as to allow as many entries as possible in a node. This increases the **fanout** of the tree and reduces its height.

completeness

A good key will represent a set as completely as possible. In other words, the bounding set that it describes should contain as few elements as possible that are not in the set being bounded. The **lossiness** of a representation is a measure of the "noise-to-signal" ratio of the representation. It is defined by

$$\frac{|\mathit{bounding\ set} - \mathit{bounded\ set}|}{|\mathit{bounding\ set}|} .$$

This is the probability that a single element will be in the bounding set but not in the bounded set. During a search for supersets of a set containing a single element, a branch of the tree will be selected for searching if the bounding set of the root of the branch contains the element. The lossiness of the representation indicates the probability that the branch is being searched unnecessarily. In a search for supersets of a set of cardinality n, the probability that the search of a branch is unnecessary equals the n-th power of the lossiness. Therefore lossiness is less of a factor when searching for supersets of large sets.

computation cost

A good key will allow efficient computation of the set inclusion and intersection operations. These operations are performed many times during a search and are critical to the performance of the RD-Tree.

3.1. COMPLETE REPRESENTATION

The first class of representations considered define the bounding set of a set to be exactly the same as the set itself. These representations have zero lossiness. The disadvantage of these representations is that they are either too large to be practical, or make computation of the basic set operations very expensive.

The simplest approach is to directly represent the sets in the RD-Tree nodes. A set will typically be a list of simple objects if the base elements are of simple type, or a list of object ids if the base elements are complex objects. The list may be sorted to improve efficiency of computing the set operations.

This approach is impractical in all but the simplest databases, as the keys quickly become large. Even if the base sets are all small, their union may be large, and keys in higher levels of the tree represent unions of many sets. As the key size grows, fanout decreases, and when a key becomes larger than half a page, the index can no longer function.

A solution to this problem is to store keys externally to the RD-Tree, and to store a pointer to each entry's key in the tree. This may be combined with the previous approach so that small keys are stored directly in the tree, while large ones are referred to by pointer. While this does guarantee high fanout, it makes computation of set operations expensive. As keys become larger than half a page, every key comparison will require reading a page from disk.

If the universe of base elements is fixed and of small size, the sets can be represented by a bitmap. This has the advantages that keys have a fixed size no matter how large the set they represent, and that set computations are cheap bitwise operations. The universe must be small enough for several bitmaps to fit on each page. Also, there must be a guarantee that new elements will not be added to the universe after the tree has been created. If these conditions hold this is a very practical approach.

3.2. BLOOM FILTERS AND SIGNATURES

A variation on the bitmap approach represents the keys by a bit vector that is smaller than the size of the universe. In addition, new elements may be added to the universe at any time. This approach has the same advantages as the bitmap approach: fixed size keys and cheap computations. However, a degree of lossiness is introduced into the representation by the fact that individual bits no longer represent unique elements.

A set can be represented by a Bloom filter. This is a vector in which all bits are initially set to zero. Every element in the set is hashed to a particular bit in the filter, which is then set to one. Elements that hash to the same position cannot be differentiated from each other. Thus each bit represents all the elements that hash to that bit, and the lossiness of the Bloom filter is equal to

$$1 - \frac{\textit{size of bit vector}}{\textit{cardinality of universe}} .$$

If the universe is significantly larger than the bit vector, the lossiness will be close to 1. Therefore this approach is only effective for queries that search for supersets of fairly large sets.

A more sophisticated technique is to use signatures as described in [ISHI93]. Every element has a signature, which is a pattern of bits in the bitmap that are set when that element is present. The signature of a set is a bitwise *or* of the signatures of the elements of the set. Since elements are mapped to collections of bits rather than individual bits, each element can have a unique signature even in a universe much larger than the size of the bitmap. Therefore the lossiness of this representation is much lower than the lossiness of Bloom filters. Nevertheless, a small amount of lossiness still exists due to the fact that the combined signature of all elements in a set may also include the signature of other elements.

3.3. RANGESETS

An alternative representation of sets that allows efficient processing of set operations is the **rangeset** [HELL93]. A **range** is an ordered pair of integers (a, b) where $a \le b$, that represents the set of integers x such that $a \le x \le b$. A rangeset is an ordered list of disjoint ranges $((a_1, b_1), \ldots, (a_n, b_n))$ such that $b_i < a_j$ whenever $i < j$. It represents the union of the sets represented by the ranges.

Any set of integers can be represented by a rangeset. For example, the sets of section 2.1 would be represented as

S1 = ((1, 3), (5, 6), (9, 9))
S2 = ((1, 2), (5, 5))
S3 = ((0, 0) (5, 6) (9, 9))
S4 = ((1, 1) (4, 5) (8, 8))
S5 = ((0, 0) (9, 9))
S6 = ((3, 3), (5, 8))
S7 = ((4, 4), (7, 7) (9, 9))

A set of integers may also be approximated by a rangeset consisting of fewer ranges. For example, S1 may be approximated by $((1, 6), (9, 9))$, and S6 may be approximated by $((3, 8))$.

A rangeset representation of RD-Tree keys would fix a maximum number of ranges n allowed in a rangeset describing a key. The bounding set of a set S is the smallest set containing S that can be described by a rangeset containing n or fewer ranges. Given any set S and a maximum number of ranges n, the bounding set of S can be computed by the algorithm described in the appendix.

If the universe consists of objects of some type other than integer, elements in the universe can be mapped to unique integers in the range $(1, maxid)$, where *maxid* is the cardinality of the universe. A procedure for performing this mapping is described in [HELL93]. Once this mapping has been performed, sets in the universe can be represented by rangesets of integers.

The lossiness of a rangeset representation depends on the **correlation factor** of the universe, and on the choice of mapping of base elements to the integers. The correlation factor is the degree to which elements in the universe associate into groups. In other words, it describes the degree to which the presence of one element in a set influences the probability that another element will be present. Elements which are closely correlated should be mapped to integers that are close together. Ranges of integers will then represent groups of elements that commonly occur together. Sets will naturally cluster into ranges so that they can be well approximated by rangesets. We are currently investigating ways of choosing an effective mapping that takes advantage of high correlation.

The choice of the maximum number of ranges allowed in the rangesets describing bounding sets involves a tradeoff between decreasing lossiness and increasing fanout. Allowing more ranges reduces lossiness by making the bounding set approximate the bounded set more closely. However, it also increases the size of the key, thereby reducing fanout. The optimal key-size will vary from key to key, as some sets can be approximated closely by a small number of ranges whereas others cannot. We are studying the use of both fixed-size and variable-size rangesets in RD-Trees.

3.4. COMBINED REPRESENTATIONS

Each representation has its advantages and disadvantages. The best representation for a key may vary from place to place within an RD-Tree. For example, sets on leaf nodes are likely to be small while sets near the root are likely to be large. Small sets may be best represented directly, while large ones are probably best represented by one of the approximations described above.

In addition, some of the representation methods have parameters that can be tweaked to different values at different places in the tree. One example is the maximum number of ranges allowed in a rangeset, as described above. Another is the size of the bit vector used by the signature representation. The lossiness of a signature increases with the cardinality of the set being represented, and decreases with the size of the bit vector. Therefore smaller bit vectors may be more appropriate at lower levels of the tree, where sets being represented are smaller.

An RD-Tree that combines different representations may prove to be more efficient than one that uses a single representation. The granularity of variation of representation may be the individual entry, the node, or the tree level. Whatever the granularity, a record must be kept at appropriate points in the tree indicating the representation being used and the values of the parameters.

Another option is to store more than one representation for each key. A key for an entry could consist of a hint, in the form of one of the approximate bounding sets described above, together with a pointer to the complete set description. When performing a search on some predicate, first the predicate is evaluated for the bounding set. If the answer is negative, the entry can be rejected. If it is positive, the answer is checked against the complete set description. If the answer is still positive, the subtree rooted at that entry is searched. If it is now negative, an unnecessary search of the subtree has been avoided.

4. INVERTED RD-TREES

As mentioned above, RD-Trees do not perform well on subset predicates. If the bounding set of an entry is a subset of a given set, then all the base sets reached via that entry are also subsets. However, the branch of the tree rooted at that entry must still be explored in order to reach those base sets. If the bounding set is not a subset of the given set, the branch cannot be rejected, because it is possible that one of its descendants is a subset.

The inverted RD-Tree allows subset predicates to be evaluated. Non-leaf nodes in an inverted RD-Tree contain entries of the form (*child pointer*, *key*), where *key* is the intersection of all keys in the child node. Thus the transitive containment relation is inverted, as parent keys are contained in the keys of all their children. If the key of any entry is not the subset of a given set, then none of its children can be either, and the branch of the tree rooted at that entry can be rejected from the search. An example of an inverted RD-Tree indexing the sets of section 3.1 is shown in figure 2.

A combined normal/inverted RTree could also be constructed, by keeping both the union and intersection keys for each pointer. Unless the keys are stored externally, the combined RD-Tree would have lower fanout due to larger key size.

For both inverted and combined RD-Trees, a heuristic must be chosen for splitting tree pages. For the inverted tree, maximum size of intersection is the criterion that will produce the best overlap within a page, but it also maximizes the key size for representations with variable length keys. For the combined tree, minimizing the symmetric set difference $(A - B) \bigcup (B - A)$ appears to be a natural heuristic.

Mathematically, an inverted RD-Tree is equivalent to an RD-Tree on the complements of the base sets. This is because the complement of the intersection of two sets is the union of the complements of those sets. Theoretically, the inverted RD-Tree should perform as well on subset predicates as an RD-Tree performs on superset predicates.

However, in most practical domains, base sets will usually be small compared to the universe. In such situations the intersection of all sets on a node is likely to be very small. The intersections will degenerate to the null set on higher levels of the tree, making the index useless. In domains with high correlation factor, where certain groups

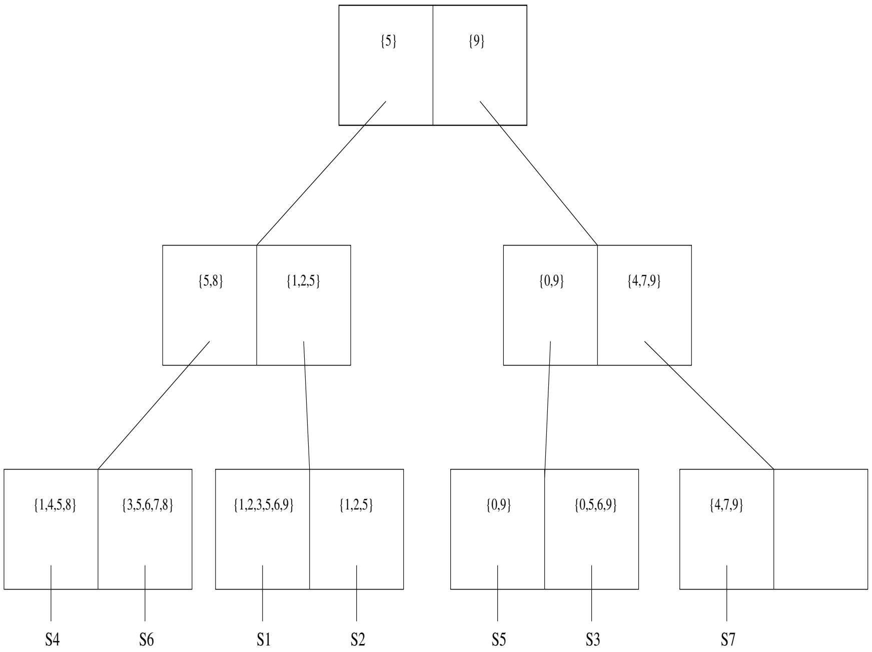

Figure 2. An inverted RD-Tree

of elements are shared by many sets, intersections will degenerate more slowly, making the use of inverted RD-Trees more reasonable. It remains to be seen whether the inverted RD-Tree can be effective in a practical application.

5. PROPOSED PERFORMANCE STUDY

We plan a detailed performance study that implements several of the representations described above and analyzes their performance on a variety of data. We will also compare the performance of RD-Trees and inverted RD-Trees to that of traditional set access methods such as signature files.

We will test our implementation on synthetic data that varies the following parameters:

size of universe
average base set size
variance of base set size
number of base elements in universe
frequency distribution of base elements
correlation factor

We will also test the RD-Tree on real undergraduate enrollment data from the University of Wisconsin. This database is similar to the one described in section 1.1, and will allow us to perform the queries presented there.

6. CONCLUSION

We have proposed and implemented the RD-Tree, an index structure for set-valued attributes. Studies are currently underway to test the effectiveness of this access method. Once the basic validity of our approach has been demonstrated, we will undertake a more detailed analysis to determine the most efficient implementation of RD-Trees in various domains.

The performance of an RD-Tree in a domain is likely to depend on the correlation factor of elements in the domain. An interesting direction of future research is an investigation into this concept. Methods are needed to analyze and characterize the correlation factor of a universe, and to determine which elements are correlated with each other. We believe that a deep understanding of these issues will serve to elucidate the structure of collections of sets, and have practical applications for databases that manage such collections.

REFERENCES

[BROW94] Eric W. Brown, James P. Callan, and W. Bruce Croft, "Fast Incremental Indexing for Full-Text Information Retrieval", *Proc. 20th Conference on Very Large Databases*, Santiago, Chile, September 1994.

[BECK90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger, Bernhard "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles" , *Proc. ACM-SIGMOD International Conference on Management of Data*, Atlantic City, N.J., Nay 1990.

[BERT89] E. Bertino and W. Kim, "Indexing Techniques for Queries on Nested Objects", *IEEE Trans. on Knowledge and Data Engineering* 1(2):196-214, June 1989.

[GUTT84] Antonin Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching", *Proc. ACM-SIGMOD International Conference on Management of Data*, Boston, Mass., June 1984.

[HELL93] Joseph M. Hellerstein "Rangesets: A Data Representation for Quick Query Processing on Nested Sets" , working draft, University of Wisconsin, Madison, May 1993

[ISHI93] Yoshiharu Ishikawa, Hiroyuki Kitagawa, and Nobuo Ohbo, "Evaluation of Signature Files as Set Access Facilities in OODBs", *Proc. ACM-SIGMOD International Conference on Management of Data*, Washington, D.C., May 1993.

[STON93] Michael Stonebraker, "The Miro DBMS", *Proc. ACM-SIGMOD International Conference on Management of Data*, Washington, D.C., May 1993.

APPENDIX: ALGORTIHM FOR COMPUTING RANGESETS

<u>Problem</u>

Given a set S of n elements stored in sort-order, reduce it to a rangeset of $k < n$ ranges such that a minimal number of new elements are introduced.

Algorithm

Initialize: consider each element $a_m \varepsilon S$ to be a range $< a_m, a_m >$.

do {

find the pair of adjacent ranges with the least interval between them;
form a single range of the pair;

} until only k ranges remain;

Proof of optimality

We must show that greedy local choices lead to a globally optimal solution. Consider some optimal rangeset A. Assume A does not contain the initial greedy choice: that is, A does not group together the 2 adjacent elements a_i , a_{i+1} of least interval $\delta = a_{i+1} - a_i$. Then a_i is the endpoint of one range, and a_{i+1} is the start of another range. We can modify A by moving a_i to the range containing a_{i+1} ; call the resulting rangeset B . The number of "false" elements in B differs from that in A by a factor of $(a_{i+1} - a_i - 1) - (a_i - a_{i-1} - 1)$ · Since we assumed that δ was a minimal interval, this expression cannot be less than 0. So B must be an optimal rangeset, or our assumption that A was an optimal rangeset was false. Hence B is an optimal rangeset that begins with a greedy choice, and therefore making a greedy initial choice can never produce a suboptimal rangeset.

Moreover, once the greedy choice of a_i, a_{i+1} is made, the problem reduces to a subproblem of finding an optimal rangeset over the ranges

$$S' = \{ < a_1, a_1 > , \ldots , < a_{i-1}, a_{i-1} > , < a_i, a_{i+1} > , < a_{i+2}, a_{i+2} > , \ldots , < a_n, a_n > \} \cdot$$

By a similar argument to the one above, making a greedy choice here is once again no worse than an optimal solution to this subproblem. This argument continues inductively, demonstrating that a greedy choice at each step produces an optimal solution.

Running Time

One can first sort the intervals between the elements by size to find the smallest $(n - k)$ intervals in $O(nlog(n))$ time. Using this information, each run of the do loop takes constant time, so running time is $O(nlog(n))$.

Modified version for variable-length keys:

Given that the greedy algorithm above produces an optimal rangeset for a fixed number of ranges k, the following greedy algorithm produces a "good" rangeset of no more than k ranges. It produces a rangeset of fewest ranges under 2 constraints:

(1) the result has no more than k ranges ,

(2) an error-bound is not exceeded unless one has to do so to ensure (1). The error bound is expressed in terms of percentage bad info, i.e. ratio of false elements to total elements .

The rangeset of l ranges that is produced is an optimal rangeset of l ranges, by the proof above.

Given a set S, a number k, and an error-tolerance $t < 1$:

Initialize:

consider each element $a_m \varepsilon S$ to be a range $< a_m, a_m >$.
current_card = |S|;
max_card = $\frac{|S|}{1 - t}$;

num_ranges = |S|;

while ((num_ranges > k) || (current_card <= max_card)) {

group together the two adjacent ranges < $a_i - a_{i+r}$ >,
< $a_{i+r+1} - a_{i+r+s}$ > of least difference;
current_card += $a_{i+r+1} - a_{i+r} - 1$;
num_ranges--;

}

if (num_ranges > k)

ungroup the last group; /* since max_card has been exceeded */