INDEXING TECHNIQUES FOR HISTORICAL DATABASES

by

Curtis Kolovson and Michael Stonebraker

Memorandum No. UCB/ERL M89/34

6 April 1989

INDEXING TECHNIQUES FOR HISTORICAL DATABASES

by

Curtis Kolovson and Michael Stonebraker

Memorandum No. UCB/ERL M89/34

6 April 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

INDEXING TECHNIQUES FOR HISTORICAL DATABASES

by

Curtis Kolovson and Michael Stonebraker

Memorandum No. UCB/ERL M89/34

6 April 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

INDEXING TECHNIQUES FOR HISTORICAL DATABASES

Curtis Kolovson and Michael Stonebraker

Computer Science Division
Department of Electrical Engineering
and Computer Science
University of California
Berkeley, California 94720

Abstract

Database management systems that maintain historical data provide the facility of storing past states of the database for subsequent retrieval. Such systems may periodically transfer historical data from magnetic disk to optical disk. In order to efficiently maintain historical data, a database management system needs indexing mechanisms that provide efficient access paths to historical data that may be stored on optical disk. These indexes must be space-efficient, and may themselves be partially or completely contained on either medium. In this paper we propose two indexing structures based on R-Trees that are well-suited for historical data which may span magnetic disk and optical disk media. We compare the performance of these indexes to two other indexing candidates that are each contained entirely on one medium. Our test results indicate that our proposed indexes perform well when compared to an index that is contained entirely on optical disk. We conclude by discussing our plan for future work to design other types of indexing structures for historical databases.

1. Introduction

Most current database management systems do not provide access to *historical* data, i.e. past states of the database. Typically, such data must be painfully extracted from database log files or audit trail files. Depending on the age of the historical data and the extent of the on-line portion of the log or audit trail, such an operation may require the mounting of one or more magnetic tapes or disk packs which contain the desired data.

This research was supported by a Hewlett-Packard Laboratories Resident Fellowship.

With the advent of write-once read-many (WORM) optical disk technology, support for large historical data archives in a database management system may soon become cost-effective. Since WORM optical disk block contents may not be modified after their initial writing, the issue arises as to how best organize a historical database archive and associated indexes on such a write-once storage medium, given that the contents of the historical data archive are not known in advance and the archive grows incrementally over time.

We believe that there are advantages to allowing indexes for historical data relations to span magnetic disk (MD) and optical disk (OD) media, as opposed to being exclusively restricted to either medium. The two major advantages are:

(1) improved search and insert performance as compared to indexes that are completely contained on optical disk, and

(2) reducing the cost per bit of disk storage required for indexes as compared to storing historical data indexes entirely on magnetic disk.

1.1. Previous Related Work

The advent of optical disk technology spurred interest in databases that do not delete the past states of data [COPE82], [MAIE82]. Although many have dealt with the semantic issues concerning historical databases [MCKE86], [SNOD86], implementation ideas and indexing techniques for historical databases are much scarcer.

Lum et al. proposed a simple indexing scheme to support historical data relations [LUM84], but this technique required following a chain of historical tuple instances in descending time order to locate a particular tuple. Extremely bad performance is inevitable when a query requests data far in the past.

Several data storage organizations for historical data are also proposed in [AHN86]. However, his approach for a secondary index technique was to use a conventional multi-attribute index augmented with *Tmin* and *Tmax* index record fields (*Tmin* and *Tmax* refer to the transaction commit times of the transactions which inserted and deleted the given version of a tuple, respectively). However, Ahn limited his consideration to indexes maintained on magnetic disk only. Since the size of such indexes may become prohibitively large, we feel that historical indexes which are not at least partly contained on optical disk will be of limited utility.

Vitter proposed an indexing structure intended to be completely contained on optical disk, which he referred to as an *Allocation Tree* [VITT85]. Allocation Trees have the property that they provide fast access to the last item inserted into a list on a write-once memory, and were proposed as a mechanism for efficiently accessing the most-recently allocated disk page in an optical disk file system. Allocation Trees are also a candidate for an indexing technique for supporting historical data searches involving time-range queries. The Allocation Tree index will be described in more detail in Section 2.7.

Easton proposed a variation of B-Trees [BAYE72] for write-once optical disks, which he calls *Write-Once Balanced Trees* (WOBT) [EAST86]. As in Vitter's proposal, Easton's WOBT is designed to be contained entirely on optical disk. Moreover, to retrieve successively older versions of a data or index record, one must follow disk page pointers that point back to previously written disk pages. The WOBT index is better suited for *snapshot* retrievals that access all data that was valid at a given time in the past, but does not perform as well on queries that access all versions of a tuple with a given key value.

1.2. Our Hypothesis

Our hypothesis that motivated the set of experiments which are the subject of this paper is that suitably designed *composite* index structures, i.e. indexes which may span magnetic and optical disk, will outperform an index structure that is contained entirely on optical disk in terms of search performance, and will approach the performance of an index that is entirely contained on magnetic disk. This work investigates the performance of several indexing strategies for supporting historical database archives. A simulated workload was used to drive implementations of these strategies and produce performance statistics regarding indexes that were contained in each of three storage media environments: (1) a magnetic disk-based historical data index, (2) an optical disk-based historical data index, and (3) a historical data index that may span magnetic disk and optical disk.

The first environment was included in the tests to provide what we expected to be an upper bound for our performance statistics. The index structure that we selected for the second environment is based on the Allocation Tree index [VITT85]. With regard to the third environment, two page movement policies for migrating historical database indexes from magnetic disk to an optical disk-based archive were implemented for this study. These page movement policies were previously proposed in [STON87]. The index structures that we utilized in the first and third environments mentioned above are variations of the R-Tree index [GUTT84].

Section 2 outlines two algorithms for migrating indexes on historical data relations from a magnetic disk to an optical disk. Results of the index performance tests are presented in Section 3. Section 4 contains a summary of our conclusions.

2. Vacuuming Algorithms for Indexes on Historical Data Relations

2.1. Definition: Data and Index Vacuuming

We have defined the term *vacuum* to refer to the transfer from magnetic to optical disk of either historical data records or index records that reference historical data records.

2.2. Assumptions

There are many alternatives to managing current and historical data in a database management system. We have not assumed any particular underlying database management system in our study of index performance, other than to assume that current and historical data are each maintained in separate relations, referred to as *current data relations* and *historical data relations*, respectively. Updating or deleting a *current* data tuple results in a *historical* data tuple being appended to the historical data relation, whereas inserting a new tuple appends that tuple in a current data relation and has no effect on historical data relations. Current and historical relations may have separate and possibly different indexes associated with them. The justification for this is that there should not be an adverse impact on the performance of operations on current data relations introduced by the support of historical data relations. Also, the queries that are performed against the current and historical relations may be different, which would favor having different indexes on each of the relations. However, it may be advantageous to allow "recent" historical data to remain in the current data relation on magnetic disk until such time as a system process transfers a collection of such historical tuples to the historical relation on optical disk. Current data relations and their associated indexes reside on magnetic disk, whereas historical data relations are contained on optical disk.

2.3. Two Vacuuming Algorithms

Two index vacuuming algorithms which were proposed in [STON87] have been implemented for this study. Both of these algorithms produce variations of an R-Tree index which span magnetic and optical disks. An R-Tree was chosen as a basis from which to design such indexes, since it provides fast access to two-dimensional data objects. Given that historical data may be regarded as existing in a multi-dimensional space, with *time* as one dimension, an R-Tree index is a logical choice of index structure for such a collection of data. All of the R-Tree variations in this study use one dimension of the index for indexing the time ranges defined by the Tmin and Tmax fields in each tuple, and another dimension for indexing an interval data attribute of a relation. In our R-Tree performance tests, we used the "Linear Cost Node Splitting Algorithm" [GUTT84], and a minimum node fill requirement ensuring that each node other than the root was at least half full. The index structures compared in this study are described in the following sections.

2.4. Index MD-RT: The Single MD R-Tree Index

The Single MD R-Tree Index is identical to the proposal in [GUTT84]. An R-Tree is a height-balanced tree with index records in its leaf nodes containing pointers to data objects. The nonleaf nodes contain pointers to other index records, and also contain values which bound the index interval in a given dimension. In all of our tests involving R-Trees, we used two-dimensional R-Trees. Thus, all of the nonleaf nodes contained both minimum and maximum values for the horizontal and vertical dimensions. R-Trees strive to maintain minimum coverage of the key space being indexed as well as minimum overlap of the key space among sibling index records. The R-Tree search algorithm descends the tree from the root in a manner similar to the B-Tree search algorithm. However, more than one subtree under a node visited may need to be searched, since subtrees may contain overlapping regions.

The following two *MD/OD R-Tree* indexes are R-Trees which may span magnetic and optical disk, i.e. certain of their nodes may be stored either on magnetic or optical disk.

2.5. Index MD/OD-RT-1: The Single MD/OD R-Tree Index

The Single MD/OD R-Tree Index is constructed from a standard R-Tree by the following simple vacuuming algorithm. Whenever the R-Tree index on magnetic disk reaches a threshold size near its maximum allotted size, the Vacuum Cleaner Process (VCP) moves some fraction of the left-most (oldest) leaf pages to the archive. In our implementation of this algorithm, up to one quarter of the leaf nodes on magnetic disk are candidates for vacuuming during each index vacuuming operation of the VCP. Following the vacuuming of the leaf nodes, the VCP then vacuums all parent nodes of the vacuumed nodes that point entirely to nodes on the archive. This nonleaf node vacuuming is applied recursively to higher level nodes. The root node is never a candidate for vacuuming.

2.6. Index MD/OD-RT-2: The Dual MD/OD R-Tree Index

The Dual MD/OD R-Tree Index consists of a pair of R-Tree indexes, both rooted on magnetic disk. The first R-Tree is contained completely on magnetic disk, and the second is rooted on magnetic disk and has its lower levels on optical disk. The Dual MD/OD R-Tree Index is constructed from a standard R-Tree by the following vacuuming algorithm. The VCP is invoked when the size of the first R-Tree index on magnetic disk reaches a threshold near its maximum allotted size. When first invoked, the VCP vacuums all of the first R-Tree's nodes, except its root node, to the optical disk and allocates a root node on magnetic disk for the second R-Tree. Then, a new R-Tree is constructed on magnetic disk as new historical data index records are inserted into the index. Subsequently, each time the VCP is invoked, it vacuums all of the first R-Tree's nodes except the root node, and inserts the immediate descendants of the first root into the corresponding level of the second R-Tree on magnetic disk. As more vacuuming operations occur, the number of magnetic disk nodes of the second R-Tree will increase, due to conventional R-Tree node splitting which may propagate up to the root node.

Over time, there would continue to be two R-Trees. The first would be completely on magnetic disk and periodically archived. Insertions are made to the first R-Tree while searches are performed by descending both R-Trees.

2.7. Index OD-AT: The Allocation Tree Index

The version of Allocation Trees implemented for this performance study is the first of the two types that Vitter described in [VITT85], which has some similarities to trees developed by Rathmann [RATH84]. Allocation Tree records are inserted in a breadth-first manner, and the tree is searched in a depth-first manner. A simple characterization of Allocation Trees is a linked list of increasingly deeper depth-first search trees, where each successive search tree is one level deeper than its predecessor in the list.

Allocation Trees, as described by Vitter, store data records within the index structure. R-Trees, on the other hand, may store either data records or data page pointers in its leaf nodes depending on whether the index is being used as a primary or secondary index, respectively. In order to make meaningful comparisons of the performance characteristics of these two indexes, we must store either data records or data page pointers in both indexes. We decided to do the latter, and thus considered both the Allocation Trees and R-Trees in our experiments to be secondary indexes. Therefore, the Allocation Trees which we implemented had data page pointers, hereafter referred to as *external data descriptors*, of 32 bytes (two 4-byte words for Tmin and Tmax, one 4-byte tuple identifier, and 20 bytes for a key descriptor and value). These are to be distinguished from *internal index descriptors* which point to other Allocation Tree nodes on the archival optical disk medium.

All leaf nodes in Allocation Trees contain external data descriptors only, whereas all nonleaf nodes may contain both external data and internal index descriptors. Thus, in this sense Allocation Trees are unlike B-Trees or R-Trees, both of which store pointers to other index pages in nonleaf nodes, and data page pointers in the leaf nodes. In Allocation Trees, any proportion of data to index descriptors in the nonleaf nodes is permitted, except that there must be at least two internal index descriptors per nonleaf page[1]. Five percentages of internal index descriptors in the nonleaf nodes were used in our tests: 2%, 25%, 50%, 75%, and 100%.

2.8. Branching Factors

Five page sizes were tested for all of the indexing structures. We assumed that logical pages are always physically contiguous on one track on both magnetic and optical disk, and may be read in one I/O operation on either type of disk. The maximum possible branching factors (the maximum number of internal index entries per nonleaf page) for the Allocation Tree and R-Tree indexes are shown in Figure 1.

[1]Strictly speaking, at least one is required, but a minimum of two guarantees reasonable search performance.

3. Performance Tests

The test database employed in our performance tests was simulated by constructing indexes generated using a random number generator for both the time domain (Tmin and Tmax) and data fields. The number of index records generated ranged from 30,000 to 100,000. We restricted ourselves to this range due to our limited magnetic disk resources. In the case of both the R-Tree and Allocation Tree indexes, each time range interval limit pair, specified by Tmin and Tmax, was generated by incrementing its predecessor by a random amount uniformly distributed over the range [1000, 10000] while satisfying the obvious constraint that Tmin be less than Tmax.

Since R-Trees may be used to index two-dimensional data objects, where one dimension may index *interval* data and the other dimension may index either *interval* or *point* data, we were faced with the decision as to what type of data to index. Interval indexes in both dimensions are

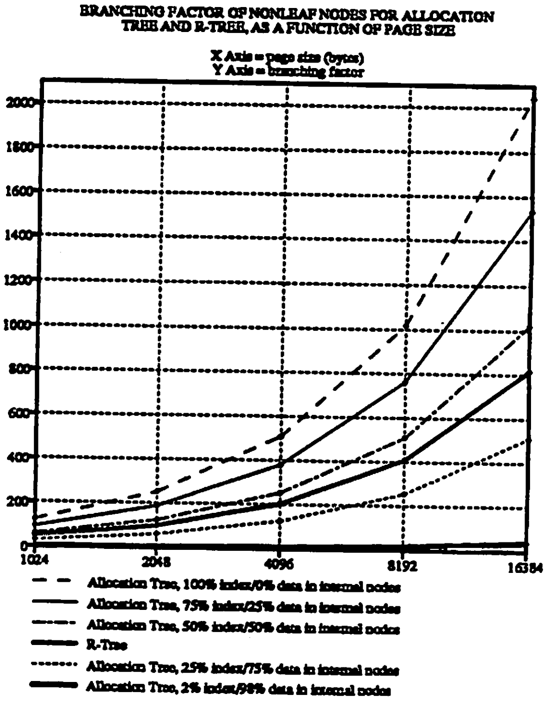


Figure 1: Branching factors for various page sizes and indexes

required for indexing spatial data objects. We expect historical indexes on time intervals and point data to be the more common variety in practice. An example of this type of index, presented in [STON87], may be created by (using POSTQuel):

```
index on archive EMP is EMP-INDEX
        (I with interval-ops,
        salary with float-ops)
        using R-tree and fill-factor = .8
```

The above index supports time intervals and salary data. A query such as:

```
retrieve (EMP.name)
        using EMP[T1,T2]
        where EMP.salary = 10000
```

to find all the names of employees who were active at some time within the time interval represented by [T1,T2] and whose salary was $10,000 would be converted internally within POSTGRES into:

```
retrieve (EMP.name)
        where interval(T1,T2)
         overlaps EMP.I
        and EMP.salary = 10000
```

which can be efficiently executed using the R-Tree index as an access path. However, for the purposes of this study, we decided to index interval data in both dimensions, as this is the more difficult case for R-Trees to deal with efficiently. An example of a query that would benefit from such an index is:

```
retrieve (EMP.name)
        using EMP[T1,T2]
        where EMP.performance
         overlaps [25%, 75%]
```

This query finds the names of all employees who were active at some time within the time interval represented by [T1,T2] and whose performance interval rating overlapped the interval [25%, 75%].

Therefore, in our tests the R-Tree index records had an additional index interval range defined for an integer data attribute where each record's interval limit values were uniformly distributed over the range of [1, 2*10**9] and satisfying the condition that the lower limit be less than the upper limit. Such a wide variation between adjacent interval values in one dimension constitutes a nearly worst-case input for R-Trees.

Each of the four index types described above were compared for insert and search performance, as well as index space requirements. To measure search performance, a large number of random interval searches were generated over the entire time interval contained in the index.

Each performance test consisted of two phases, an *insert phase* and a *search phase*. Each test program began with an empty index tree. During the insert phase, an input data file was read and an index tree was constructed. Insert performance was measured for the last 10% of the records, when the tree was nearly its final size. During the search phase the program performed a series of 100 random searches. The following queries were used to generated these searches (in SQL):

```
Query 1:
  select *
  from test_relation
  where tmax >= rand_time_min
        and tmin <= rand_time_max;

Query 2:
  select *
  from test_relation
  where tmax >= rand_time_min
        and tmin <= rand_time_max
        and vmax >= rand_val_min
        and vmin <= rand_val_max;
```

In both queries, tmin, tmax, vmin, and vmax are attributes of test_relation. The rand_time_min/max and rand_val_min/max correspond to randomly generated search intervals, respectively. Query 1 generates a random search interval and Query 2 generates a random search "rectangle", i.e. a two-dimensional search space composed of two search intervals. The random search intervals were generated by creating a random interval with a length that was uniformly distributed over the range [0, 50%] of the difference of the maximum and minimum values of the index records in a given dimension. Each query was repeated 100 times using different random search arguments. A series of 100 query executions retrieved approximately 20% and 10% of the index records when processing Query 1 and Query 2, respectively.

Two sets of experiments were performed. In the first set we varied the page size of the index, using sizes of 1, 2, 4, 8, and 16 Kb, where in each of these tests the number of index records was 50,000. In the Allocation Tree tests, for each page size we varied the percentage of space in the non-leaf nodes used for internal index descriptors, as previously mentioned. In the second set of tests, we varied the number of index records while using a page size of 1024 bytes. We repeated the tests with indexes consisting of from 30,000 to 100,000 records, in increments of 10,000 records. In the Allocation Tree tests in which we varied the number of index records, we chose to have 50% of the nonleaf page space used for internal index descriptors, which provides a good balance between insert and search performance. In the tests involving the MD/OD R-Tree indexes, the size of the magnetic disk area usable for the index was 256 Kb. Our choices for the number of index records and the size of the magnetic disk area usable for the index were dictated by our limited magnetic disk resources. Although these parameter values may be small with respect to "realistic" historical data indexes, they were large enough for analyzing the performance characteristics of the various indexes.

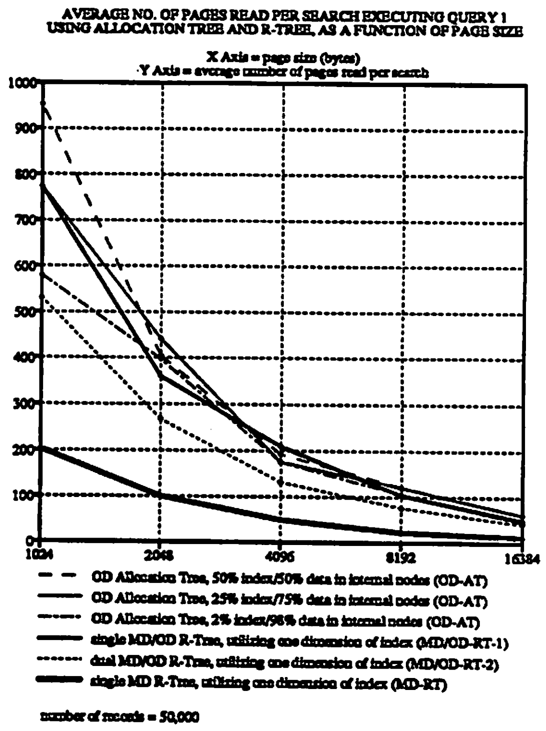
3.1. Performance Test Results: Varying the Page Size

The set of graphs to be discussed next are all plotted as a function of the page size. Figures 2 and 3 show the *weighted* average number of index pages read per search to find all the qualifying index records, for all of the R-Tree variations and for Allocation Tree indexes which had 2%, 25%, or 50% of the nonleaf node space used for index descriptors. The weighted average number of pages read per search was computed using the following formula:

weighted average # of pages read per search =
(average # of MD pages read per search) / W
+ (average # of OD pages read per search)

The weighting factor, W, is used to normalize the average number of magnetic and optical disk I/Os per search. We chose a weighting factor, W, of 7 since the average device block transfer time for a magnetic disk is typically 30ms, while the corresponding statistic for a 2 Gb write-once optical disk is approximately 210 ms [LMSI87]. The value of such a scaling factor may decrease in the future as optical disk performance improves.

The curves in Figures 2 and 3 show the performance data collected for one-dimensional and two-dimensional searches corresponding to Query 1 and Query 2, respectively. Note that the curves for the Allocation Tree indexes are the same in both graphs, since the same number of index pages are read when an Allocation Tree is used as the access path to execute either Query 1 or 2, i.e. all the records that satisfy the given time range search argument are examined when executing either query. MD-RT and MD/OD-RT-2 outperform all of the other indexes in both Figures 2 and 3. MD/OD-RT-1 has approximately the same search performance as the OD-AT indexes in Figure 2, but performs only slightly better than the OD-AT variations in Figure 3.



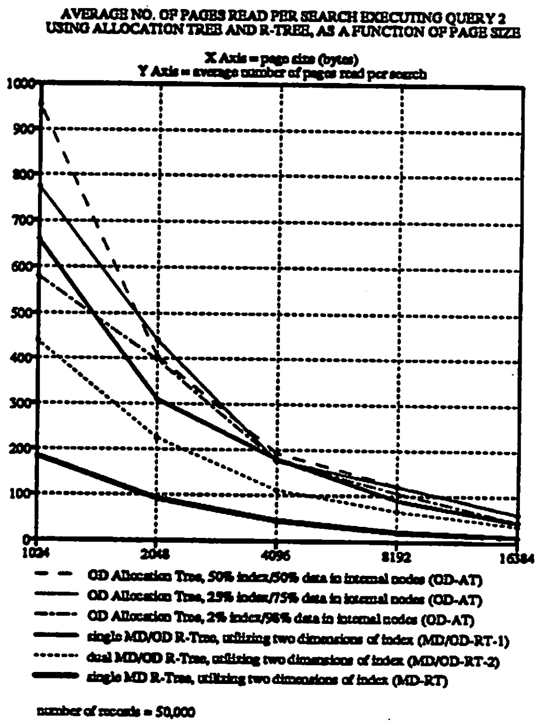


Figure 2 (top) and Figure 3 (bottom):
No. of Pages Read per Search

An interesting point to notice about Figures 2 and 3 is the rather slight improvement in the two-dimensional search performance over that of the one-dimensional search performance. Clearly, the R-Trees do provide some benefit, but because the data values being indexed vary independently of the time domain values, the R-Trees do not provide a substantial improvement in search performance in the two-dimensional searches. However, Allocation Trees do not provide any improvement at all in two-dimensional search performance as compared to one-dimensional searches.

Although the MD-RT index provided the best performance, this is more a result of the effect of the weighting factor, W, than the number of read operations performed when using this index as an access method. In particular, MD-RT required the highest number of read operations. This is due to the randomizing effect of page splits that occur when the R-Tree is entirely contained on a magnetic disk medium, as in MD-RT. The R-Tree search algorithm descends the tree from the root in a manner similar to the B-Tree search algorithm. However, more than one subtree under a node visited may need to be searched, since subtrees may contain overlapping regions. As has been pointed out by [SELL87], "...since it is very hard to control the overlap during the dynamic splits of R-Trees, efficient search degrades...". Our results verify this observation, as is shown in Table 1, which shows the distribution of TIDs per R-Tree index page for each of the R-Tree variations employed in our tests. A TID represents a *Tuple Identifier*, which in the R-Tree experiments was merely a counter that was incremented by one for each new tuple inserted. As a separate experiment, the data presented in Table 1 was collected from test runs of MD-RT, MD/OD-RT-1, and MD/OD-RT-2, each with an index file consisting of 30,000 records, and with a page size of 1 Kb. Since the index records were inserted into the R-Tree indexes in approximately sorted order with respect to the time domain[2], one might expect a lower range of TID values per page than those shown in Table 1 in the absence of page splits. However, another contributing factor to the wide range of TID values per page was due to the excessive partitioning of the data due to the lack of multi-dimensional clustering. R-Trees are best suited to indexing data that exhibits a high degree of natural clustering in multiple dimensions, e.g. two-dimensional geometric objects. This natural clustering provides the R-Tree index an opportunity to partition the data into rectangles so as to minimize both the coverage and overlap of the entire set of rectangles corresponding to the leaf nodes and the enclosing rectangles that correspond to the nonleaf nodes of the index.

Returning to the effect of page splits, notice that the decrease in the average range of TIDs per page from MD-RT to MD/OD-RT-1, and from MD/OD-RT-1 to MD/OD-RT-2 are direct side-effects of the vacuuming algorithms. Thus, an added benefit to the search performance of R-Trees is provided by the vacuuming of the index pages, which makes them no longer eligible for inserting new records onto, and hence no longer candidates for page splits. The *batch vacuuming algorithm* employed by MD/OD-RT-2 vacuums the entire magnetic disk R-Tree other than its root node once the R-Tree has reached its maximum size on magnetic disk. This vacuuming algorithm results in an R-Tree structure which experiences fewer page-splits per page than either MD-RT or MD/OD-RT-1. The reason for this is because the R-Tree nodes on magnetic disk have a shorter

[2]The time ranges as defined by Tmin and Tmax between adjacent records may overlap.

Index Type	Number of Leaf Pages	Average Number of TIDs per Page	Maximum Range of TIDs per Page	Average Range of TIDs per Page
MD-RT	829	36.19	29960	27624.35
MD/OD-RT-1	816	36.77	29946	18110.30
MD/OD-RT-2	819	36.63	9215	7821.82

Table 1: Distribution of Tuple IDs per Page for the R-Tree indexes

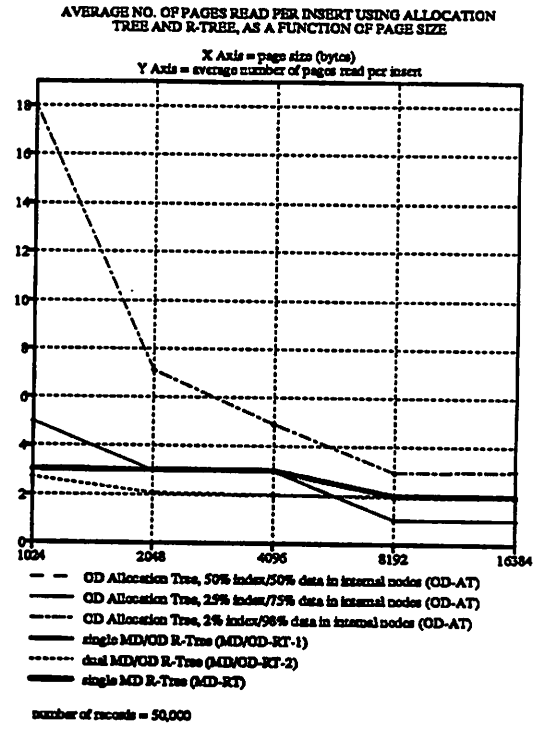
magnetic disk occupancy time on the average, as compared to MD-RT and MD/OD-RT-1. In Table 1, note that the average number of TIDs per page is nearly identical in all of the R-Tree organizations tested, namely about 36 TIDs per page, which corresponds to 72% space utilization (R-Trees with a page size of 1 Kb may hold up to 50 index records), for both magnetic disk and optical disk pages.

Figure 4 shows the average number of pages read per insert. The notable feature of this graph is that OD-AT with 2% indexing descriptors in the nonleaf nodes requires a larger number of reads per insert, especially with page sizes less than 8 Kb. Figures 2-4 illustrate the trade-off between search and insert performance in Allocation Trees (OD-AT), which is controlled by the ratio of internal index descriptors to external data descriptors in the nonleaf nodes. A lower ratio provides better search performance but has more costly inserts, whereas a higher ratio has better insert performance but not as good search performance. From these graphs, it appears that a good balance can be achieved by having 25% or 50% of nonleaf node space used for indexing descriptors.

Figure 5 shows the sizes of the various index structures. The R-Tree indexes are generally smaller than the Allocation Trees, with the exception of MD/OD-RT-2 with page sizes of 8 Kb and 16 Kb. This is due to the fact that the size of the optical disk portion of the second R-Tree in MD/OD-RT-2 increases more sharply with the larger page sizes, i.e. with page sizes of 8 Kb and 16 Kb, as is made clear in Figure 5. This phenomenon may be explained by Table 2. Table 2 shows that the space utilization, measured as the percentage of the total number of index record entries that are used, decreases for MD/OD-RT-2 as the page size increases beyond 4 Kb. Obviously, as the page size increases, the *batch vacuuming algorithm* will vacuum pages that have fewer entries in use, on the average. This may be alleviated by modifying the simple *batch vacuuming algorithm* to only vacuum pages that have at least some minimum threshold percentage of their space utilized.

3.2. Performance Test Results: Varying the Amount of Data

The set of graphs to be discussed next are all plotted as a function of the number of index records. In each of these tests, the page size of the indexes was 1 Kb. For all of the tests involving OD-AT discussed in this section, the internal



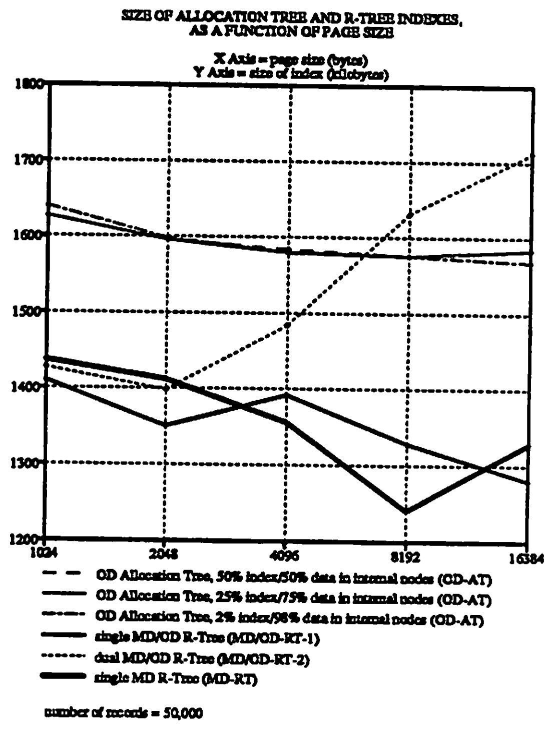


Figure 4 (top): No. of Pages Read Per Insert
Figure 5 (bottom): Index Space Requirements

Page Size	Index Type	Space Utilization
1 Kb	MD-RT	71%
2 Kb	MD-RT	71%
4 Kb	MD-RT	72%
8 Kb	MD-RT	79%
16 Kb	MD-RT	73%
1 Kb	MD/OD-RT-1	72%
2 Kb	MD/OD-RT-1	74%
4 Kb	MD/OD-RT-1	70%
8 Kb	MD/OD-RT-1	73%
16 Kb	MD/OD-RT-1	76%
1 Kb	MD/OD-RT-2	72%
2 Kb	MD/OD-RT-2	71%
4 Kb	MD/OD-RT-2	66%
8 Kb	MD/OD-RT-2	60%
16 Kb	MD/OD-RT-2	57%

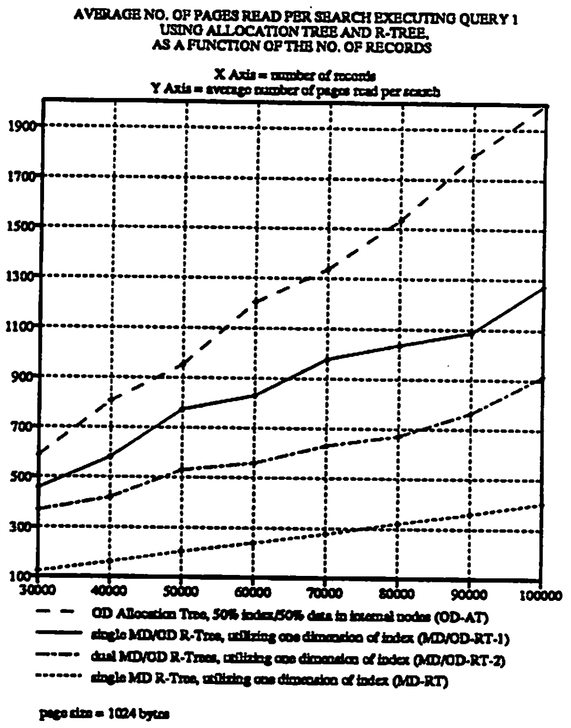
Table 2: Space Utilization of R-Tree indexes for various Page Sizes

(nonleaf) nodes contain 50% internal index descriptors and 50% external data descriptors. Figures 6 and 7 show the average number of pages read per one-dimensional and two-dimensional search, respectively. These graphs correspond to Figures 2 and 3, which were plotted as a function of page size, and confirm those results. These graphs indicate that the performance of OD-AT tends to degrade more sharply than that of MD/OD-RT-1 or MD/OD-RT-2 as the number of records increases.

Figure 8 shows the average number of pages read per insert, and Figure 9 shows the sizes of the various indexes. Both of these graphs feature curves that are all within a narrow range. In Figure 9, the Allocation Tree index is increasingly larger than the R-Tree indexes as the number of records increases, by an amount ranging between approximately 100 to 400 Kb, for a given number of records.

4. Conclusions

We have verified our hypothesis that suitably designed index structures which are allowed to span magnetic and optical disk media may outperform an index structure that is completely contained on optical disk, and may approach the performance of an index that is entirely contained on magnetic disk.



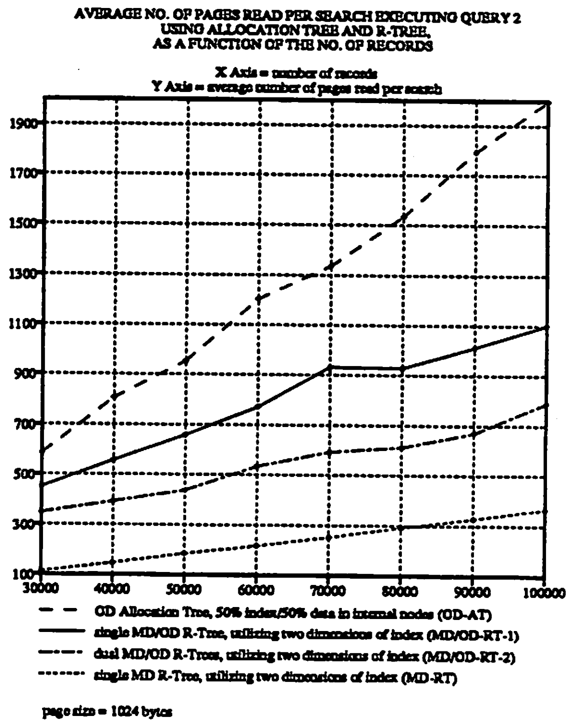
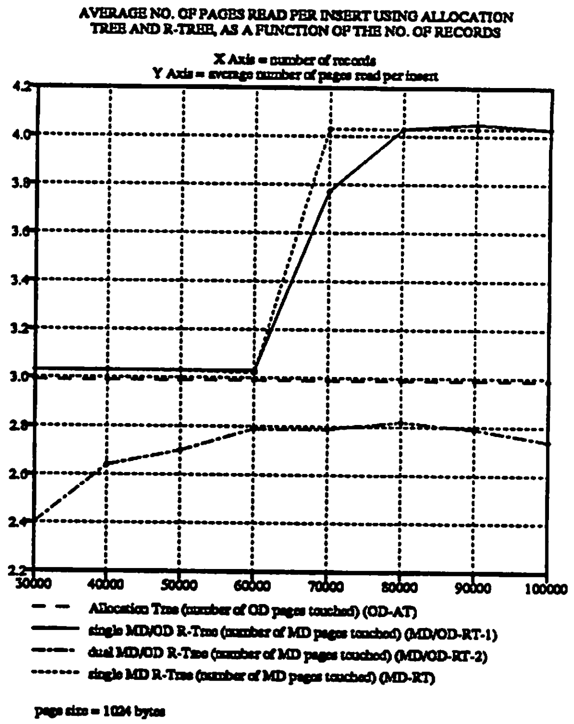


Figure 6 (top) and Figure 7 (bottom): No. of Pages Read per Search

The two R-Tree variations that we have proposed which may span magnetic and optical disk have been shown to be useful for indexing historical data relations. One variation of the R-Tree, MD/OD-RT-2, always outperformed the Allocation Tree index in terms of the average number of read operations per search, in some cases by more than a factor of two. Also, the periodic transfer of R-Tree indexes from magnetic disk to write-once optical disk actually produced better search performance of the index in terms of the average number of read operations per search. This improvement in search performance is due to a reduction in the average number of page splits per index page in the R-Trees that span magnetic and optical disk media, since once an index page has been written on a write-once optical disk, no new records may be inserted on that page and hence that page can no longer be split by the *Insert* procedure.

Comparing the performance of our two R-Trees variations that span magnetic and optical disk media to an R-Tree index that is completely contained on magnetic disk, the weighted average number of read operations per search for the MD-only R-Tree is always below the corresponding statistic for the two MD/OD R-Trees. This is due to the effect of the weighting factor, since most of the MD/OD read operations per search occur on optical disk. Comparing the MD-only R-Tree to the MD/OD R-Trees in terms of the average number of read operations per search, we found that the MD-only R-Tree required more read operations than either of the two MD/OD R-Trees for all page sizes. The reason for this is primarily due to the randomizing effect of page splits on the MD-only R-Tree, which causes significant overlap of the rectangular partitions in the nonleaf nodes.

Despite the good performance of R-Trees in this study, there are two potential shortcomings of the R-Tree index that are detrimental to its search performance when the data being indexed consists of time intervals in one dimension and interval data in the other dimension. These shortcomings were evident in our study, since we constructed nearly a worst-case input for R-Trees. The first of these problems is that R-Trees work best on collections of data that exhibit a natural multi-dimensional clustering, but historical databases may consist of relations whose data attribute values vary independently of their time domain attribute values, and therefore may exhibit only one-dimensional clustering. When an R-Tree is used to index a collection of data that lacks natural multi-dimensional clustering, the result is that the R-Tree algorithms excessively partition the data into a set of minimally enclosing multi-



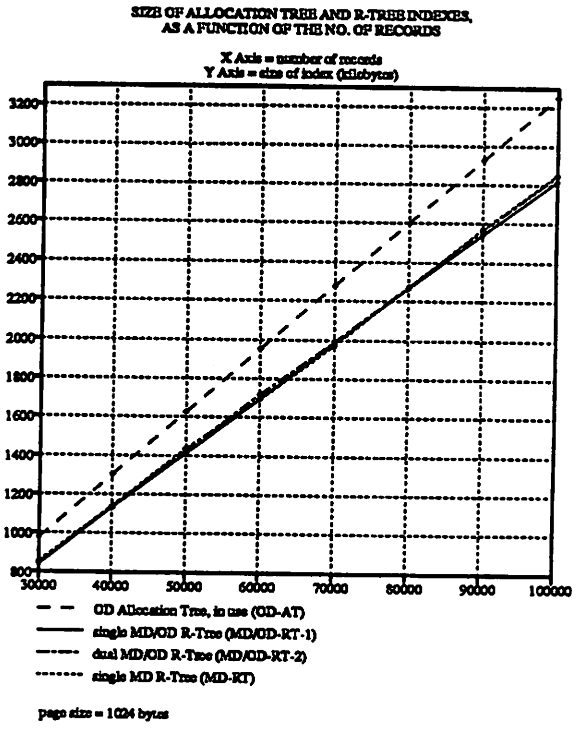


Figure 8 (top): No. of Pages Read Per Insert
Figure 9 (bottom): Index Space Requirements

dimensional spaces. The second problem with R-Trees for historical databases is that the effect of page splits causes a good deal of overlap in the search regions of the nonleaf nodes. R+-Trees [SELL87] have been proposed to address this shortcoming of R-Trees. Although R+-Trees would likely provide an improvement to search performance for indexing historical databases over R-Trees, they still would not improve the problem of excessive partitioning due to the potential lack of natural multi-dimensional clustering of the interval data being indexed. These factors suggest that new indexing structures may be useful for historical databases.

Although our results indicate that R-Trees may not be the optimal indexing structure for indexing time intervals and *interval* data, we believe that they may provide good support for indexing time intervals and *point* data, which we expect to be more common in practice. Thus, we plan to experiment with R-Trees that index historical point data, i.e. one-dimensional R-Tree indexes. We also plan to design and measure the performance of new indexing structures in conjunction with several vacuuming algorithms in subsequent performance tests, to determine how they compare to those discussed in this paper in terms of search, insert, index size, and space utilization performance. Our goal is to design indexing structures that achieve good overall performance for a broad range of database applications that require access to historical data.

References

[AHN86] Ahn, I., "Performance Modeling and Access Methods for Temporal Database Management Systems", PhD Thesis, Computer Science Department, University of North Carolina at Chapel Hill, July 1986.

[BAYE72] Bayer, R., and McCreight, E., "Organization and Maintenance of Large Ordered Indexes", Acta Informatica, 1, No. 3 (1972), pp. 173-189, Springer-Verlag.

[COPE82] Copeland, G., "What if Mass Storage Were Free?", IEEE Computer, 15, No. 7, July 1982, pp. 27-35.

[EAST86] Easton, M., "Key-Sequences Data Sets on Indelible Storage", IBM Journal of Research and Development, 30, No. 3, May 1986, pp. 230-241.

[GUTT84] Guttman, A., "R-Trees: A Dynamic Index Structure for Spatial Searching", Proc. of ACM SIGMOD Conf., Ed. B. Yormark. Association for Computing Machinery. Boston, MA: June 1984, pp. 47-57.

[LMSI87] Laser Magnetic Storage International Co., "LaserDrive 1200 Intelligent Digital Optical Disk Drive User Manual", LMSI Co., Colorado Springs, CO, 1987.

[LUM84] Lum, V., et al., "Designing DBMS Support for the Temporal Dimension", Proc. of ACM SIGMOD Conf., Ed. B. Yormark. Association for Computing Machinery. Boston, MA: June 1984, pp. 115-130.

[MAIE82] Maier, D., "Using Write-Once Memory for Database Storage", Proc. ACM Symposium on Principles of Data Base Systems, March 1982, pp. 239-246.

[MCKE86] McKenzie, E., "Bibliography: Temporal Databases", ACM SIGMOD Record, 15, No. 4, pp. 40-52.

[RATH84] Rathmann, R. "Dynamic Data Structures on Optical Disks", Proc. of IEEE Data Engineering Conf., IEEE. Los Angeles, CA: April 1984.

[SELL87] Sellis, T., Roussopoulos, N., and Faloutsos, C., "The R+-Tree: A Dynamic Index for Multi-Dimensional Objects", Tech. Report UMIACS-TR-87-3, CS-TR-1795, Dept. of Computer Science, U. of Maryland, College Park, MD. February 1987.

[SNOD86] Snodgrass, R., "Research Concerning Time in Databases: Project Summaries", ACM SIGMOD Record, 15, No. 4, pp. 19-39.

[STON87] Stonebraker, M., "The POSTGRES Storage System", Proc. of the 1987 VLDB Conf. Brighton, England: September 1987.

[VITT85] Vitter, J., "An Efficient I/O Interface for Optical Disks", ACM Trans. on Database Systems, 10, No. 2, June 1985, pp. 129-162.