The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes

John T. Robinson
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Abstract - The problem of retrieving multikey records via range queries from a large, dynamic index is considered. By *large* it is meant that most of the index must be stored on secondary memory. By *dynamic* it is meant that insertions and deletions are intermixed with queries, so that the index cannot be built beforehand. A new data structure, the *K-D-B-tree*, is presented as a solution to this problem. K-D-B-trees combine properties of K-D-trees and B-trees. It is expected that the multidimensional search effieciency of balanced K-D-trees and the I/O efficiency of B-trees should both be approximated in the K-D-B-tree. Preliminary experimental results that tend to support this are reported.

1. Introduction

Consider the following problem:

1. There are index records of the form

   $key_0, key_1, \ldots, key_{K-1}, location,$

   where key_i is an element of a finite totally ordered set $domain_i$, K is a constant, and *location* gives the location of a database record with these keys.
2. It is desired to retrieve records based on queries of the form

   $min_i \leq key_i \leq max_i, \qquad 0 \leq i < K-1,$

   i.e., *range* queries.
3. Insertions and deletions of records are randomly intermixed with queries, implying that whatever data structure is used to implement the index must be built and maintained dynamically.
4. The number of records in the index is so large that it is necessary to store most of the index on secondary memory (disk or drum) while it is being used, due to limitations on the size of primary memory.

In the case that $K=1$, the most efficient solution to this problem is probably the *B-tree* (see [Bayer and McCreight 72]), or one of its variants (see [Comer 79] for a survey). In the case that (4) is omitted, or alternatively, (3) is omitted so that the data structure can be built statically and then mapped onto pages (see [Bentley 79] for an example), there are a number of solutions (see [Bentley and Friedman 79] for a survey), such as the *K-D-tree* (see [Bentley 75]). Also, it should be mentioned that static solutions can often be converted to pseudo-dynamic solutions by various techniques, such as using overflow areas for inserted records and marking deleted records. In such cases, however, periodic reorganization of the index will usually be necessary to maintain efficiency.

This research is supported in part by the National Science Foundation under Grant MCS 78-236-76 and the Office of Naval Research under Contract N00014-76-C-0370.

Here, a new data structure, the *K-D-B-tree*, is presented as a solution to the above problem. K-D-B-trees, like B-trees, are multiway trees with fixed-size nodes that are always totally balanced in the sense that the number of nodes accessed on a path from the root node to a leaf node is the same for all leaf nodes. Each node is stored as a page so that efficient use can be made of secondary memory with paging. Unlike B-trees, 50% utilization of pages cannot be guaranteed, although it is expected that in practice the number of pages less than half full should be small (as compared to the total number of pages). Preliminary experimental results support this: as "random" records are inserted, storage utilization seems to stay around 60% for cyclic 2-D-B-trees and 3-D-B-trees (see Section 4 for the definition of cyclic, and Section 6 for details of the experiments). These percentages can hopefully be increased if reorganization techniques discussed in Section 5 are used (none of these tehcniques are currently implemented).

K-D-B-trees partition search spaces (subsets of $domain_0$ x $domain_1$ x ... x $domain_{K-1}$) in a manner similar to K-D-trees: a search space is partitioned into two subspaces based on comparison with some element of a single domain. Like K-D-trees, various strategies can be used to select the domain and the element in the domain. Some of these are discussed in Section 4.

In the next section the K-D-B-tree structure is defined. Algorithms for queries, insertions, and deletions are presented in Sections 3, 4, and 5, respectively. Also discussed in Section 5 are reorganization techniques, analagous to catenations and underflows in B-tree algorithms. The combination of B-tree properties and K-D-tree properties in the K-D-B-tree leads one to expect that the I/O efficiency of B-trees and the multidimensional search efficiency of K-D-trees might both be approximated in K-D-B-trees. Some preliminary experimental results that tend to support this are reported in Section 6. Section 7 contains conclusions and a discussion of further research.

2. The K-D-B-Tree Structure

Define a *point* to be an element of $domain_0$ x $domain_1$ x ... x $domain_{K-1}$, and a *region* to be the set of all points $(x_0, x_1, ..., x_{K-1})$ satisfying

$$min_i \leq x_i < max_i, \qquad 0 \leq i \leq K\text{-}1,$$

for some collection of min_i, $max_i \in domain_i$. Points can be represented most simply by storing the x_i, and regions by storing the min_i and max_i.

Below, it will be required that certain regions be disjoint, and that their union be a region -- thus the strict inequality on the right hand side of the region definition above. However, it will also be required that the union of certain regions be all of $domain_0$ x $domain_1$ x ... x $domain_{K-1}$. It is therefore necessary to create for each domain a special element ∞_i, which is greater than all elements of $domain_i$, and to allow the max_i to assume these values. It is also convenient to define $-\infty_i$ as the minimum of $domain_i$.

Like B-trees, K-D-B-trees consist of a collection of pages and a variable *root ID* that gives the page ID of the root page. There are two types of pages in a K-D-B-tree.

1. Region pages: region pages contain a collection of (*region*, *page ID*) pairs.
2. Point pages: point pages contain a collection of (*point*, *location*) pairs, where *location* gives the location of a database record. The (*point*, *location*) pair is in fact an index record.

The following set of properties define the K-D-B-tree structure. The algorithm for range queries given in the next section depends only on these properties, and the algorithms for insertions and deletions are designed so as to preserve these properties.

1. Considering each page as a node and each page ID in a region page as a node pointer, the resulting graph structure is a multi-way tree with root *root ID*. Furthermore, no region page contains a null pointer, and no region page is empty (note that this, together with the fact that point pages do not contain page IDs, means that

the point pages are the leaf nodes of the tree).

2. The path length, in pages, from root page to leaf page is the same for all leaf pages.
3. In every region page, the regions in the page are disjoint, and their union is a region.
4. If the root page is a region page (it may not exist, or if there is only one page in the tree it will be a point page), the union of its regions is $domain_0$ x $domain_1$ x ... x $domain_{K-1}$.
5. If (*region*, *child ID*) occurs in a region page, and the child page referred to by *child ID* is a region page, then the union of the regions in the child page is *region*.
6. Referring to (5), if the child page is a point page, then all the points in the page must be in *region*.

Figure 1 illustrates an example 2-D-B-tree.

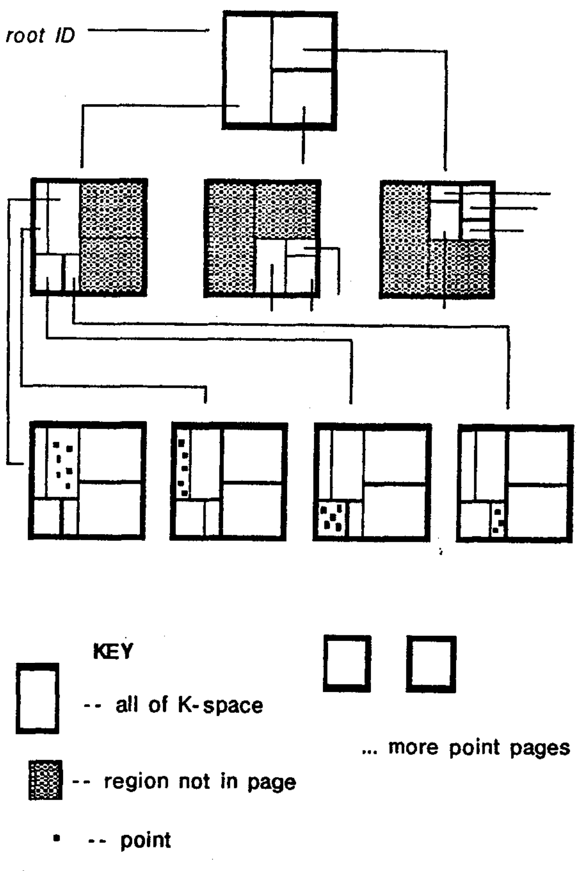

Figure 1. Example 2-D-B-Tree

3. Queries

A range query can be expressed by specifying a region, the *query region*. It is convenient to think of regions as a cross-product of intervals I_0 x I_1 x ... x I_{K-1}. If some of the intervals of a query region are full domains, the query is a *partial range* query; if some of the intervals are points and the rest are full domains, the query is a *partial match* query; if all of the intervals are points, the query is an *exact match* query.

The algorithm to output all records satisfying a range query specified by *query region* is as follows.

Q1. If *root ID* is the null page ID, terminate. Otherwise, let *page* be the root page.

Q2. If *page* is a point page, then for each (*point*, *location*) pair in *page* with *point* a member of *query region*, retrieve and output the database record at *location*.

Q3. Otherwise, for each (*region*, *child ID*) pair in *page* such that the intersection of *region* and *query region* is non-empty, set *page* to be the page referred to by *child ID*, and recurse from (Q2).

The experimentally observed performance of various queries is given in Section 6.

4. Insertions

First, it is necessary to define the *splitting of a region* along element x_i of $domain_i$. Let the region be I_0 x I_1 x ... x I_{K-1}. If $x_i \notin I_i$, the region is not changed by splitting. Otherwise, let $I_i = [min_i, max_i)$; splitting the region results in two new regions:

1. I_0 x ... x $[min_i, x_i)$ x ... x I_{K-1},
2. I_0 x ... x $[x_i, max_i)$ x ... x I_{K-1}.

Region (1) is called the *left* region and region (2) the *right* region. If $x_i \notin I_i$ since $x_i < min_i$, the region is said to lie to the *left* of x_i; if $x_i \geq max_i$, the region is said to lie to the *right* of x_i. A point $(y_0, y_1, ..., y_{K-1})$ is said to lie to the *left* of x_i if $y_i < x_i$, and to the *right* of x_i otherwise.

A point page is split along x_i by creating two new point pages, the *left page* and the *right page*; then transferring all the (*point*, *location*) pairs in the page to either the left or right page depending on whether *point* lies to the left or the right of x_i; and then deleting the old page. See Figure 2.

A region page is split along x_i by creating two new region pages, again called the left page and the right page; filling these pages with regions derived from the old region page; and then deleting the old page. This procedure takes place as follows. For each (*region*, *page ID*) in the old region page:

```
S1. If region lies to the left of $x_i$, add
    (region, page ID) to the left page.
S2. If region lies to the right of $x_i$, add
    (region, page ID) to the right page.
S3. Otherwise:
    S3.1. Split the page referenced by page ID
          along $x_i$, resulting in pages with IDs
          left ID and right ID.
    S3.2. Split region along $x_i$, resulting in
          regions left region and right region.
    S3.3. Add (left region, left ID) to the left
          page, and (right region, right ID) to
          the right page.
```

Note that this procedure is recursive due to (S3.1). See Figure 3.

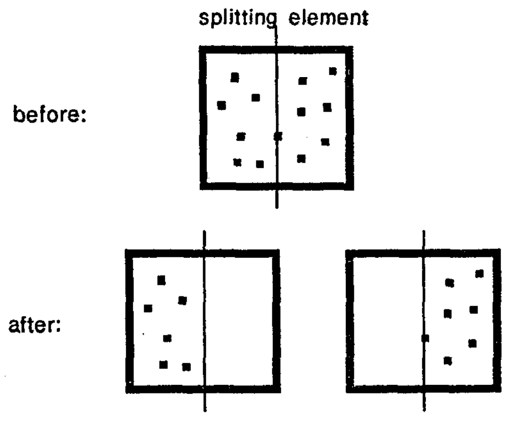

Figure 2. Splitting a point page

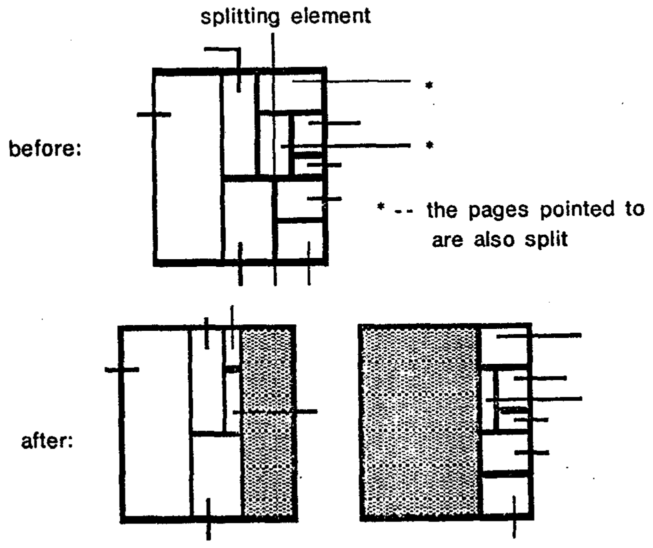

Figure 3. Splitting a region page

The algorithm for inserting an index record (*point*, *location*) is as follows.

I1. If *root ID* is null, create a point page containing (*point*, *location*), set *root ID* to the ID of this page, and terminate.

I2. Otherwise, do an exact match query on *point*, which finds a point page that *point* should be added to if the K-D-B-tree structure is to be preserved. If *point* is already in the page, do something special (like generating an error, or modifying link fields in database records), and terminate.

I3. Add (*point*, *location*) to the point page. If the page does not overflow, terminate. Otherwise, let *page* be the point page.

I4. Let the ID of *page* be *old ID*. Pick a domain, $domain_i$, and an element x_i in this domain, such that *page* split along x_i will result in two pages that are not overfull (since the number of points or regions in *page* need only be decreased by one to avoid overflow, it is easy to see that this is always possible). Split *page* along x_i, giving left and right pages with IDs *left ID* and *right ID*.

I5. If *page* was the root page, go to (I6). Otherwise, let *page* be the parent page of *page* (this parent page was found during the exact match query step above). Replace (*region*, *old ID*) in *page* with (*left region*, *left ID*) and (*right region*, *right ID*), where *left region* and *right region* are obtained by splitting *region* along x_i. If this causes *page* to overflow, repeat from (I4); otherwise terminate.

I6. Create a new region page containing the regions $(domain_0 \times \ldots \times [-\infty_i, x_i) \times \ldots \times domain_{K-1}, left\ ID)$, $(domain_0 \times \ldots \times [x_i, \infty_i) \times \ldots \times domain_{K-1}, right\ ID)$, and set *root ID* to its ID.

Variations of the above algorithm result from the way $domain_i$ and x_i are chosen in (I4). One way of choosing $domain_i$ is to do so cyclically, as follows (this was the method used in the experiments described in Section 6). Store in each page a variable *splitting domain*, initialized to 0 in a root page when a new root page is created. When a page splits, an element of $domain_{splitting\ domain}$ is used, and the new pages have *splitting domain* set to (*splitting domain* + 1) MOD *K*. This results in the regions that correspond to leaf pages being arranged as in Figure 4, and similarly for other levels (there are exceptions to this arrangement due to effects of step (S3) in splitting). This method is analagous to the cyclic choice of domains in K-D-trees (see [Bentley 75]).

This cyclic method might be modified if something is known about queries. For example, suppose $K = 2$, and that "most" queries are partial match queries or partial range queries on $domain_0$ only. In such a case it would be desirable to use $domain_0$ several times in a row before incrementing *splitting domain*, resulting in a splitting pattern like Figure 5.

Alternatively, suppose something about the distribution of index record points is known, so that it is possible to give, for any interval in any $domain_i$, the expected number of index records (at some point) that have key_i values in this interval. Call this the length of the interval. In such a case it would be desirable (if nothing is known about queries) to have regions as "*K*-cubical" as possible. This means a region $I_0 \times I_1 \times \ldots \times I_{K-1}$ would be split using that $domain_i$ such that I_i is the longest of the *K* intervals, and the splitting point would be selected so that the sub-intervals of I_i that are produced are of equal length (assuming this does not leave an overfull page).

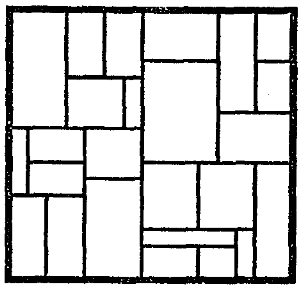

Figure 4. Cyclic splitting pattern

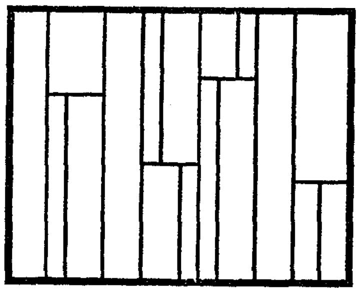

Figure 5. Domain 0 priority splitting

In general, though, given the splitting domain, the splitting point should be chosen so as to put approximately the same number of points or regions in each new page, and to minimize the number of times step (S3) is performed. Some care is needed, however: in some cases, the splitting domain may have to be re-chosen, as shown in Figure 6.

The observed efficiency of insertions, in terms of page accesses, is reported in Section 6, along with storage utilization measurements for growing cyclic K-D-B-trees.

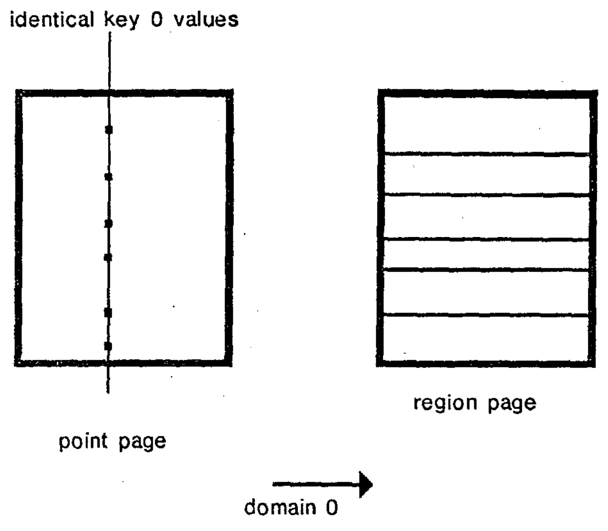

Figure 6. Pages that can't be split along domain 0

5. Deletions and Reorganization

Since the K-D-B-tree structure, as defined in Section 2, does not preclude empty point pages, and does not require anything about storage utilization, the basic deletion algorithm is very simple: find the index record (*point, location*) with an exact match query, and remove (*point, location*) from the point page.

Unless there are very few deletions, or by chance insertions take place that "fill in the holes" left by deletions, this basic deletion algorithm will be unacceptable due to the resulting low storage utilization. In B-tree algorithms, this problem is solved by what are here considered to be reorganization techniques. Referring to Figure 7, if the pointers to pages *B* and *C* in page *A* are adjacent, two types of reorganization may take place: catenation, in which the information in pages *B* and *C* is combined into one page, and underflow, in which the information in pages *B* and *C* is re-distributed between them.

This reorganization is local in that it involves only pages *A* and its children (reorganization involving more than two child pages of *A* may also be desirable for B-trees -- see [Comer 79]). Exactly the same type of local reorganization may be performed for K-D-B-trees, approximately, providing

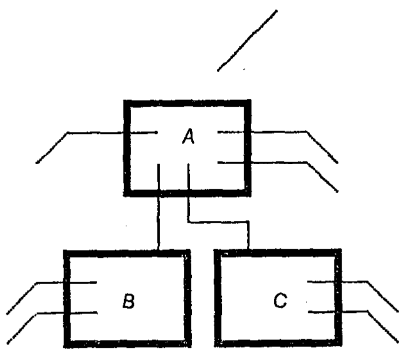

Figure 7. Three pages

the union of the regions corresponding to the pointers to pages *B* and *C* is also a region. "Approximately," because unless pages *B* and *C* are point pages, redistributing the information they contain between them may involve splitting step (S3).

If the union of two (or more) regions is a region, the regions are said to be *joinable*. This is analagous to adjacency in B-trees. A problem is that cases may arise in which there is a region that is not joinable to any other region in the page, as is the case with region *A* in Figure 8 (because B-trees are one-dimensional, this problem never occurs for B-tree reorganization). A solution is reorganization based on more than two regions. For example, in Figure 8, all of the regions or points in the three pages corresponding to regions *A*, *B*, and *C* could be combined into one page, and this page split once or twice if necessary to prevent overflow. If no splitting is necessary, regions *A*, *B*, and *C* are replaced with their union; if splitting occurs once, *A*, *B*, and *C* are replaced with two regions, etc.

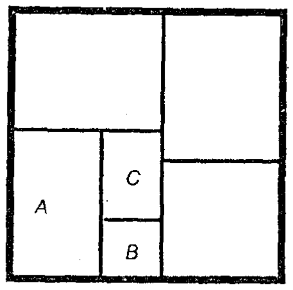

Figure 8. Region A not joinable with another

Generalizing, an outline of the algorithm to "reorganize page *P*" is as follows (*P* could be an underfull point page produced by a deletion, or an underfull region page produced by previous reorganization).

1. Let *page* be the parent page of *P*, containing (*region*, *ID*), where *ID* refers to *P*.
2. Find ($region_1$, ID_1), ($region_2$, ID_2), ..., in *page* such that *region*, $region_1$, $region_2$, ..., are joinable (this is always possible -- in the worst case, this will be all the regions of *page*).
3. Catenate the pages with IDs *ID*, ID_1, ID_2, ..., and then repeatedly split this page and resulting pages until no page is overfull.
4. Replace (*region*, *ID*), ($region_1$, ID_1), ($region_2$, ID_2), ..., in *page* with the resulting new regions and page IDs.

Another possible use of reorganization is during insertions, since step (S3) can leave empty or near-empty point pages. This should probably be done only at the point page level, since reorganization itself makes use of step (S3) when performed at higher levels. However, almost all pages of a K-D-B-tree are point pages (see Tables 1 and 2 in Section 6 for examples) -- perhaps reorganizing only at the point page level during insertions would significantly increase storage utilization. At the time of this writing, K-D-B-tree reorganziation was not yet implemented; the problem was still being studied.

6. Preliminary Experiments

A major difference between K-D-B-trees and B-trees with respect to insertions is step (S3), which forces pages at lower levels to split even though they are not overfull. An immediate question is how badly step (S3) affects performance, in terms of storage utilization and page accesses. Therefore, it was thought desirable to first gain some knowledge of the performance of "basic" K-D-B-trees: K-D-B-trees without deletions and without reorganization.

Statistics gathered from various experiments are given in Tables 1 and 2. The K-D-B-trees were generated as follows.

1. Each domain was the set of floating point numbers in [0,1).
2. The index record points were pseudo-randomly generated, uniformly distributed in *K*-space.
3. The splitting domain, $domain_i$, was chosen cyclically, and the splitting element was chosen as the median of the key_i for a point page and the median of the min_i for a region page.
4. Nine trees each were generated for *K* = 2, 3, by inserting 10,000 records, with various page sizes and record sets (Table 1).
5. One tree each was generated for *K* = 2, 3, by inserting 100,000 records (Table 2).

K	PAGE SIZES[A]	RUN	PAGES AT EACH LEVEL[B]	STORAGE UTILIZATION	PAGES ACCESSED/ INSERTION[C]
2	12, 21	1	1, 11, 96, 756	0.62	1.18, 3.85
		2	1, 11, 100, 783	0.59	1.19, 3.86
		3	1, 12, 95, 773	0.60	1.28, 3.85
2	25, 42	1	1, 19, 360	0.66	1.13, 2.93
		2	1, 22, 373	0.63	1.12, 2.93
		3	1, 20, 361	0.65	1.12, 2.93
2	51, 85	1	1, 4, 164	0.72	1.04, 2.72
		2	1, 5, 179	0.65	1.05, 2.70
		3	1, 5, 177	0.66	1.05, 2.72
3	9, 15	1	1, 6, 30, 188, 1179	0.53	1.33, 4.61
		2	1, 6, 33, 196, 1206	0.51	1.34, 4.71
		3	1, 5, 32, 192, 1166	0.53	1.33, 4.63
3	18, 31	1	1, 4, 45, 576	0.54	1.16, 3.53
		2	1, 4, 51, 564	0.55	1.16, 3.59
		3	1, 3, 43, 557	0.56	1.16, 3.58
3	36, 63	1	1, 11, 275	0.57	1.07, 2.83
		2	1, 11, 265	0.59	1.06, 2.85
		3	1, 10, 242	0.65	1.06, 2.85

[A] Page sizes = R, P, where R is maximum number of regions in a region page, P is maximum number of points in a point page.

[B] For example, "1, 4, 164" means 1 page at level 1 (root page), 4 pages at level 2, and 164 pages at level 3 (point pages).

[C] Pages accessed = W, R, where W is pages written, R is pages read, averaged over 10,000 insertions.

Table 1. K-D-B-Trees of Size 10,000

K	PAGE SIZES[A]	SIZE	PAGES AT EACH LEVEL[B]	STORAGE UTILIZATION	PAGES ACCESSED/ INSERTION[C]
2	25, 42	20,000	1, 2, 40, 714	0.66	1.09, 3.36
		40,000	1, 4, 80, 1458	0.65	1.09, 4.00
		60,000	1, 7, 122, 2187	0.65	1.13, 4.00
		80,000	1, 9, 165, 2904	0.65	1.18, 4.00
		100,000	1, 12, 209, 3662	0.64	1.18, 4.00
3	36, 63	20,000	1, 20, 514	0.61	1.15, 2.92
		40,000	1, 2, 45, 1060	0.59	1.16, 3.30
		60,000	1, 2, 63, 1547	0.61	1.15, 4.01
		80,000	1, 4, 89, 2084	0.60	1.16, 4.01
		100,000	1, 4, 108, 2594	0.60	1.15, 4.00

A, B As in Table 1.

C As in Table 1, averaged over last 20,000 insertions.

Table 2. K-D-B-Trees of Size 100,000

K	PAGE SIZES[A]	RUN[B]	RANGE[C]	RECORDS FOUND/ QUERY[D]	PAGES READ/ QUERY[D]	QUERY EFFICIENCY[E]
2	25, 42	1	0 X 1	0	22	0
			.1 X .1	100	11	0.34
			.01 X 1	99	25	0.15
			.3 X .3	908	52	0.66
			.1 X .9	905	56	0.61
2	25, 42	2	0 X 1	0	22	0
			.1 X .1	98	12	0.33
			.01 X 1	100	26	0.15
			.3 X .3	903	55	0.65
			.1 X .9	900	59	0.61
3	18, 31	1	0 X 1 X 1	0	73	0
			0 X 0 X 1	0	12	0
			.2 X .2 X .2	80	27	0.19
			.02 X .4 X 1	79	47	0.11
			.008 X 1 X 1	79	75	0.07
			.5 X .5 X .5	1270	170	0.47
			.25 X .5 X 1	1262	152	0.52
			.125 X 1 X 1	1263	149	0.53
3	18, 31	2	.0 X 1 X 1	0	74	0
			0 X 0 X 1	0	13	0
			.2 X .2 X .2	81	28	0.18
			.02 X .4 X 1	79	46	0.11
			.008 X 1 X 1	79	78	0.06
			.5 X .5 X .5	1252	170	0.46
			.25 X .5 X 1	1259	149	0.52
			.125 X 1 X 1	1243	146	0.53

A As in Table 1. B Refers to K-D-B-tree produced for Table 1.

C Ranges of 0x1, 0x1x1, 0x0x1 are partial match queries.

D Average of 100 queries. E As defined in text.

Table 3. Queries on K-D-B-trees of Size 10,000

The results of the insertion experiments may be summarized by noting that storage utilization was 60% ± 10%, the average number of pages written per insertion was slightly over one, and the average number of pages read per insertion was close to the height of the tree.

Query experiments were also performed, on four of the trees produced for Table 1, by specifying various ranges, and then randomly generating 100 queries with the given ranges. The average number of pages read for queries with the same ranges was computed, but this number is fairly meaningless taken by itself; it really should be compared with the total number of pages in the tree, and with the "size" of the query. Therefore, the *query efficiency* of a query was defined to be

$$\frac{(R_Q / R_T) \times P_T}{P_Q},$$

where R_Q is the number of records found by the query, R_T is the total number of records in the tree, P_Q is the number of pages accessed by the query, and P_T is the total number of pages in the tree. The idea behind this definition is that a query that retrieves R_Q records out of R_T records accesses a fraction R_Q/R_T of the database; if the database is stored as P_T pages, $(R_Q/R_T) \times P_T$ should be the ideal number of page accesses. This definition ignores storage utilization considerations (to get query efficiency as compared to queries on the "perfect" K-D-B-tree, multiply the above by storage utilization). Average query efficiency, as defined above, is usually quite a bit lower than 1.0 for obvious reasons (always accessing the full root page, rather than R_Q/R_T of the page; partial intersection of the query region with page regions).

The results of the query experiments are given in Table 3. Note that exact match queries were not included, since the number of page accesses in this case is just the height of the tree (available from Table 1). Also note that the number of records found by partial match queries was zero (making query efficiency zero), since domains were floating point numbers, and the random number generator did not repeat.

8. Conclusions

Preliminary experimental results tend to support the expectation that the K-D-B-tree is an efficient search structure for large multidimensional dynamic indexes, at least for $K = 2, 3$. One could not hope for much better than the page access statistics of Tables 1 and 2. Furthermore, the query efficiency for full range queries seems quite good.

If partial match or partial range queries are more important than full range queries, though, other search structures might be better. For example, suppose all queries are partial match or partial range queries in which one key is specified. In such a case a collection of K B-trees, one for each domain, would in general be far better: informally, one would expect O(log N) performance as compared to at best $O(N^{k-1/k})$ performance. But if queries are small full range queries, one would expect these to be reversed.

Clearly much work remains, notably implementation of and experimentation with reorganization. Also important is the investigation of point distributions other than uniform in K-space. Two limiting cases might be mentioned. First, if the points are highly linearly correlated, the resulting K-D-B-tree will essentially be a B-tree (on any one of the domains). Second, if the cardinality of one of the domains, say $card(domain_i)$, is extremely small, the resulting K-D-B-tree will essentially be a collection of $card(domain_i)$ (K-1)-D-B-trees. But there are many other cases that need to be investigated.

References

[Bayer and McCreight 72] Bayer, R. and McCreight, E. Organization and maintenance of large ordered indexes. *Acta Informatica 1*, 3 (1972), 173-189.

[Bentley 75] Bentley, J. L. Multidimensional binary search tress used for associative searching. *Comm. ACM 18*, 9 (1975), 509-517.

[Bentley 79] Bentley, J. L. Multidimensional binary search trees in database applications. *IEEE Trans. Software Eng. SE-5*, 4 (1979), 333-340.

[Bentley and Friedman 79] Bentley, J. L. and Friedman, J. H. Data structures for range searching. *Computing Surverys 11*, 4 (1979), 397-409.

[Comer 79] Comer, D. The ubiquitous B-tree. *Computing Surveys 11*, 2 (1979), 121-138.