Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

M.L. Kersten

A time and space efficient implementation of a dynamic index in a main-memory DBS

Computer Science/Department of Algorithmics & Architecture Report CS-R8806 January

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

A Time and Space Efficient Implementation of a Dynamic Index in a Main-Memory DBS

Martin L. Kersten

Centre for Mathematics and Computer Science
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

ABSTRACT

Exploitation of a very large main-memory as the prime storage for medium sized databases poses challenging problems regarding the software architecture of such a DBMS. A central issue is how the database should be organized and what performance gains can be obtained from judicious use of the large memory and CPU power. In this paper we describe new algorithms and programming techniques which prove valuable in organizing access paths with minimal storage overhead and good performance characteristics.

1980 Mathematics Subject Classification: 69E20, 69H22.
CR Categories: E.2, H.2.2.
Keywords and Phrases : Data storage representations, access methods, main-memory database management.
Note: This paper is submitted for publication elsewhere.

1. Introduction

A growing awareness within the database community is that in the near future random access memory becomes a cost-effective storage medium [Garcia-Molina83]. In 1984 [DeWitt84] an estimate of 250,000 dollar is given for a one Gigabyte main-memory system by the end of this decade. Their guess has not been disproved since.

With the availability of a cheap large main-memory it becomes necessary to re-evaluate the architecture of our relational database systems. Enlarging the disk-buffer space in existing systems to keep the database in memory is not the best possible system architecture, because many CPU cycles remain being spent on buffer and 'file'-organization. Moreover, the ratio between database processing time and query plan generation may change in favour of the former. This means a drastic change in the way query optimizers should be constructed. In addition, the specific problems encountered in a main-memory system, such as the volatility of the storage medium, are not dealt with appropriately in a disk-based system.

One of the earlier attempts to implement a main-memory database system (MMDBMS) is the Silicon Database Machine [Leland85]. The first prototype was based on a multi-processor consisting of six M68000 with 5 Megabyte of main memory. A prototype database management system was developed for this hardware environment and exercised thoroughly. They obtained a speed-up of 2-7 times compared with the IDM/500. These rather discouraging results can be attributed to the slow inter-processor communication and the limited use of the main-memory characteristics. In particular, for recovery purposes, they simulated a traditional page-based filing system in main-memory which led to excessive copying overhead. The final prototype

Report CS-R8806
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

was constructed from five AT&T 321000-based single board computers, each with a one megabyte of local, on-board memory, and 64 megabytes of shared memory. Performance of the latter showed a factor 5 performance improvement over the Teradata machine composed of 20 processors [Leland87].

Another prototype MMDBS is described inBitton86 which forms the foundation of an interactive office system, called Office-By-Example. This project showed that it is important to consider the combined space and time requirements for execution of queries, as opposed to simple evaluating response times. At the technical level they discovered that memory allocation and management are crucial factors in a MMDBS environment.

A more ambitious project is the PRISMA/DB project in the Netherlands [Kersten87]. Here the database software is developed for a machine composed of 64 M68020's each equipped with 16 Megabyte of main-memory, and special purpose communication processors. One of its ultimate goals is to generate database systems from a high-level specification of their intended behaviour. Moreover, this system is implemented in an experimental object-oriented programming language. The study reported in this paper has been triggered by the design of this MMDBS.

An overview of the MMDBS developed at the University of Wisconsin-Madison is described in [Lehman86]. This paper contains an evaluation of old and new data structures for utilization in a MMDBS. They discovered that the search algorithms for ordinary B-tree were too slow when compared with AVL-trees and sorted heaps. Since the latter showed bad insertion performance, they invented a hybrid data structure, called the T-tree. A T-tree is a binary search tree in which each node contains a small, limited sized sorted heap. Moreover, each node represents a single, dynamically changing interval in the key space. The smallest and largest elements in the nodes are used to speed-up searching the T-tree node with the desired elements.

In this paper we too focus on the data structures for maintaining dynamic access paths to memory-resident relations. To obtain maximal benefit from the large main-memory we assume that the database as a whole fits in the available space, because previous analysis show that the performance of a main-memory data structure quickly deteriorates when only a portion is mapped into a large buffer pool. However, we do not assume that memory is for free. Therefore, we are interested in data structures and algorithms with good time/space trade-offs.

Our experiments partly disproves the observations made by Lehman and Carey by presenting a data structure and an algorithm with excellent search performance and which incurs minimal space overhead. Moreover, the algorithms for modification can be balanced to obtain the best time/space trade-off for a given volatility rate. The technique can also be applied to traditional database management systems where a performance improvement can be achieved with minimal programming effort.

From a different perspective, we can re-phrase our observations as follows. Chosing a data structure and accompanying algorithms in a MMDBS using ordinary complexity measures is just the first step in their design. It should be complemented by detailed studies revealing the scale of the constant factor hidden behind the complexity measure. Our experimental results show that an 5-times processing improvement with minimal space overhead can be obtained for 'similar' data structures by focusing on this aspect alone.

The rest of this paper is organized as follows. In section 2 we illustrate the relevant issues in designing main-memory data structures for indexed tuple access. In section 3 we describe virtual trees and describe a programming technique to improve their search performance. Section 4 discusses insertion cost of virtual trees and provide a theoretical upperbound on the critical cost factor. In section 5 we show that join algorithms developed for the new data structure can be made fast as well.

2. Access paths in a main-memory database system

In this section, we introduce the performance parameters for access paths in a MMDBS. Following we focus on dynamic indices that support range queries and joins efficiently, i.e. algorithms with logarithmic behaviour. Besides, the creation of a static index in a MMDBS is relatively easy. It can be realized in two phases; collecting the pointer-set and sorting them on the indexed field. Alternatively, the algorithms described below for maintaining of a dynamic index can be used instead.

The basis for fast tuple access in both a traditional DBMS and a MMDBS is through the design of (dynamic) indices. When the relations are memory resident, it is sufficient to store pointers to tuples in the index rather than copies of the keys with their tuple-identifiers. This has several advantages. First, the size of the index elements is often smaller than the size of the indexed object. Second, using pointers eliminates the need for expensive tuple layout interpretation. The pointers can be set directly to the indexed field. Third, saving the results of queries is less costly, because one need only keep a list of pointers to qualifying tuples.

Several traditional data structures have been investigated for suitability in a MMDBS. The aforementioned paper of Lehman and Carey analyses the behaviour of B-trees, sorted heaps, AVL-trees and various hash-based structures. Each index structure was tested for all aspects of index use: creation, search, scan, range queries and query mixes.

Their overall conclusion is that, for selection, their T-tree provides excellent overall performance for range queries, and that modified hash is the best index structure for unordered data. Moreover, they claim that the choice of which algorithm to apply is simplified by the more definite ordering of preference: a hash lookup is always faster than a tree lookup; a tree lookup is always faster than a sequential scan; and a tree-merge join is nearly always preferred when an index exists.

A few comments on this study are in place here. First, a drawback of their approach is that the performance figures show only compound behaviour. This complicates the analysis of the insertion routines and limits the performance prediction of other 'normal' working environments. For example, the index maintenance analysis is mixed with search queries.

Second, the analysis of the join algorithms uses the traditional approach to take join selectivity as a dominant performance factor. In our opinion, join selectivity within a main-memory system is of less importance, because it primarily determines the amount of work that should be done to construct the resulting tuples. It is of less importance for the joining method itself, because the basic costs there are determined by the cardinalities of the operands. Notice that in a traditional system, join selectivity affects the occupancy of the disk buffers with portions of the result and this in turn may affect the speed of the joining method.

Third, the paper lacks an indication of several dominant performance factors in designing a MMDBS index. Besides space overhead associated with pointer structures and poor space utilization in most hashing approaches, the following factors are important as well.

Object granularity

To ease memory management and simplify recovery, tuples are best grouped into memory partitions, such as pages and memory segments. To avoid internal (memory) fragmentation and expensive garbage collection, dynamic index structures (which remain in use for a long time) should also be based on the page/segment architecture of the underlying operating system. For example, the total size of a T-tree node can be set to the page size of the memory management unit.

Algorithmic complexity

The data structures in a MMDBS should not only be based on their perceived time/space complexity measures. Because, in general, a weak algorithm (using its complexity measure) may outperform the stronger one for the small range of problem sizes dealt with. Moreover, the constant factor hidden in the complexity measure is also a relevant factor for MMDBS design.

Machine instruction speed

In designing algorithms for a MMDBS one should not ignore the differing costs of machine instructions. It is often more effective to look-up a value in an encoding table than to evaluate the function that produces it. Moreover, it is often profitable to replace expensive arithmetic operations by cheaper onces; either automatically using an optimizing compiler or explicitly under programmer control.

Bounded problems

A drawback of most studies in data structures is their focus on algorithms in which the size of the problem to be solved is a parameter. For example, most implementations of a binary tree do not limit the tree size upfront. We will show in the next section that designing algorithms for a bounded problem is cost-effective. This aspect can be exploited in a MMDBS, because the number of problem sizes we have to deal with is limited too.

Code versus data trade-off

An important factor for algorithm design in a MMDBS is that data space is as expensive as code space. This means that we can develop algorithms which save data space at the expense of code space (and vica versa). Below we show that moderate code expansion may significantly improve the performance of a search algorithm.

In the subsequent sections we will show, by experimentation, that the effect of these factors should not be ignored when designing a dynamic index structure.

3. Virtual trees

In the previous section we have seen that a good data structure for supporting range queries should support logarithmic search, does not incur too much overhead in search administration, takes into account the cost of the machine instructions, and works with reasonable large (constant sized) objects. The *virtual tree* is an attempt to fulfill all these requirements.

The structure of a virtual tree can best be introduced through an example. Therefore, consider a balanced binary-search tree of 15 elements. This tree can be mapped to an array as shown in Figure 1. Moreover, any tree with less than 15 elements can be mapped into this tree as well; provided you guarantee that each node remains accessible from the root. An example partial filling is shown with the shaded nodes. Looking at a binary search tree this way makes all pointers implicit and makes space overhead solely dependent on the size of the tree. In addition, searching in this data structure has become a bounded problem, at most 3 comparisons are needed to locate any element.

Definition A *virtual tree* of order k is a balanced-binary search tree of 2^k-1 nodes represented within a buffer of size 2^k-1.

In the subsequent section we will study the search and maintenance algorithms for (partially filled) virtual trees in more detail.

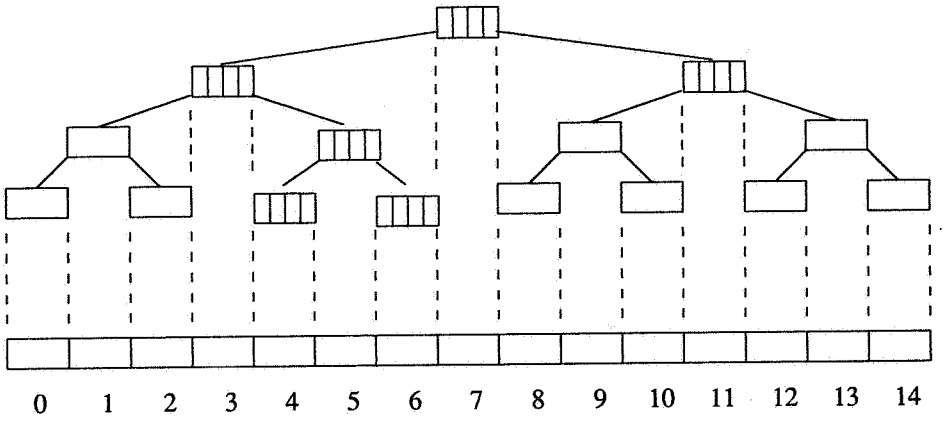

Figure 1: array of size 15 with its virtual tree.

To use the virtual tree as a dynamic index we store references to tuples (or attributes) in the virtual tree. Under the assumption that tuples do not migrate we can use memory addresses to the indexed fields directly. For the purpose of this section we assume that the tree has been partially filled with references. Empty slots are recognized by the NIL reference.

The general search algorithm for virtual trees is shown Figure 2. This algorithm returns a reference to the tree element with the require value or NIL. The search parameters are, besides the value of interest, the root of the virtual tree and the initial stepsize. The latter tells how many elements we should move to the left or right after each comparison. It is divided in each phase to limit the search to tree elements. Testing a leaf node is kept out of the loop.

```
long *BSAsrch(val,w, stepsize)
register long val, **w, stepsize;
{
        while(stepsize>0){
                if( *w== NIL) return  NIL;
                if( val<**w) w-= stepsize; else
                if( val>**w) w+= stepsize; else return *w;
                stepsize= stepsize/2;
        }
        /* test leaf node */
        if( *w== NIL) return  NIL;
        if( val== **w) return *w;
        return NIL;
}
```

Figure 2 A general search algorithm for virtual trees

From a memory management point of view it is often advisable to use only a few different buffer sizes throughout a program. For example, chosing the buffer size identical to the page size used in the memory manager will reduce the number of page faults. More importantly, it will avoid memory fragmentation and simplifies memory re-organization.

If we take this information into account during program development, we can develop a more efficient search algorithm for virtual trees. The fixed buffer size means that in a *k*-order virtual tree only k steps are needed to locate an element. Moreover, the step-size at each point is precisely known. Thus, if we explicitly write out the code for each level in the tree we safe the boundary checks and avoid the expensive task to determine the location of the left/right son. The corresponding algorithm, which heavily uses the macro facility of C, is shown in Figure 3.

```
#define Test(X)         if(*w == 0) return 0; \
                        if( val<**w) w -= X; else \
                        if( val>**w) w += X; else return *w;

long *BSAsrch15(val,w)
register long val, **w;
{
        w +=7;
        Test(4); Test(2); Test(1);
        if( *w == 0) return 0;
        if( val == **w) return *w;
        return 0;
}
```

Figure 3 Search algorithm for 15-node virtual tree

This technique is inspired by loop-unrolling optimization in optimizing compilers. However, no compiler will ever be able to apply this technique automatically to the algorithm shown in Figure 2, because it would require both a complete symbolic evaluation and knowledge about the possible initial values of *stepsize*. Note that, code-expansion is logarithmic in the size of the tree and since each level requires only a few instructions we can ignore the code blow-up for most reasonably sized trees.

The point of interest is how the virtual tree behaves compared with other algorithms used in Main-Memory Database Systems. Therefore, we have also implemented several candidate algorithms and obtained experimental performance figures through exhaustive simulation. Figure 4 gives an overview of the search performance. The bottom of the picture denotes the order of the (virtual) trees (3-255 tree elements). The line marked *linear* denotes the time when a sorted heap is searched in a linear fashion. The *calc* line reflects the performance of successive calculation of the sub-root. The *2-tree* corresponds with a binary search tree implemented with pointers. It is the performance of a general algorithm which involves pointer chasing. The performance of the virtual tree implementations are denoted by *v-tree* and *V-tree*, which correspond with the general solution and code-expanded solution, respectively. The performance figures shown reflect an implementation of the search algorithms in C under BSD 4.3 with optimized compilation on a 6 MIPS machine (Harris).

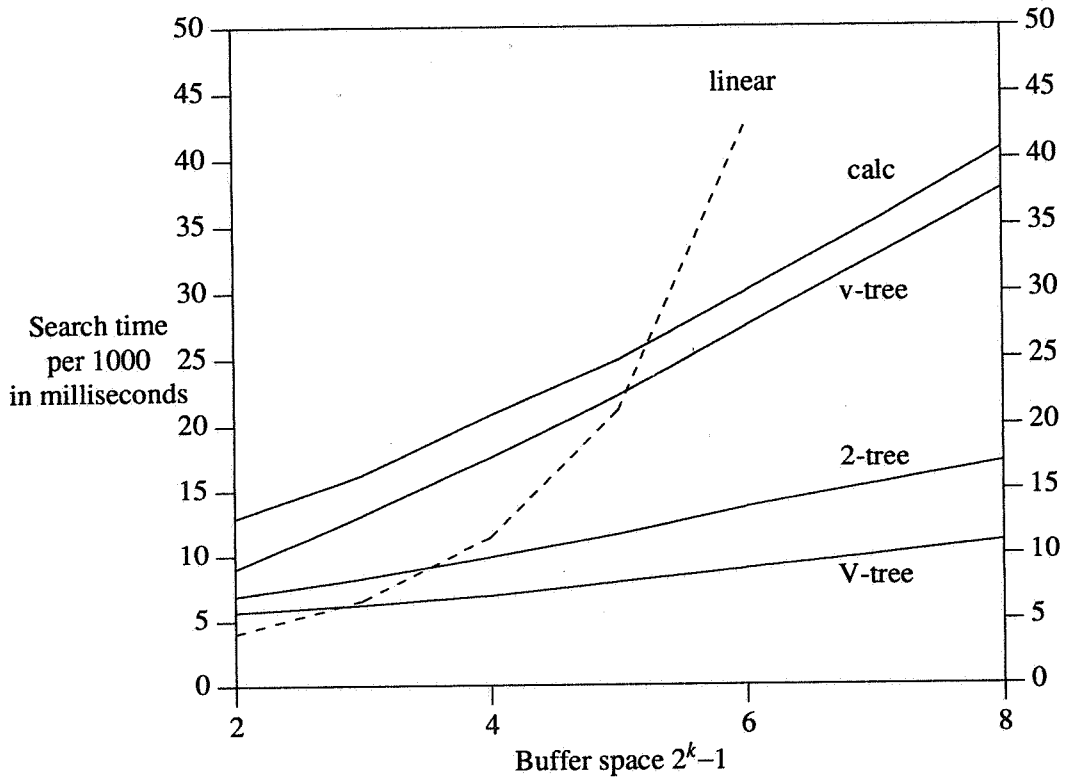

Figure 4 Search performance for search algorithm

Comparing our results with the information provided by Lehman & Carey we can observe the following. Their observation that calculated binary search in a sorted heap is expensive, is correct. However, the reasons are that expensive instructions are needed to locate the next comparison element and that the algorithm should continuously take into account the size of the tree. In other words, the boundary condition (the top of the heap) changes at run-time. Notice, that this also applies to their organization of T-tree nodes.

More importantly, our implementation of a virtual tree shows that the search performance can be better than the binary search tree if you freeze the size of the problem at programming time and use (logarithmic) code expansion. It results in a 70% improvement over the sorted heap with calculated search and 35% over the binary search tree. It also shows that in our case the specific solution is three times faster than the general solution.

In general, both techniques can not be used to improve the other algorithms. For example, fixing the buffer size is not sufficient to enable code-expansion in the 2-tree case, because you should then also guarantee that the tree won't exceed depth k. However, a few general improvements remain possible. First, notice that the programming language C does not provide a means to describe a test with three-way branch explicitly. This means that in half of the cases we perform a superfluous test. Replacing the routines with assembler code to explore this situation will result in a minor improvement all cases. Second, all implementations show a large initial cost to activate the search routine. This can easily be overcome by placing the search routine inline.

4. Insertion into a virtual tree

Insertion into virtual trees is only slightly more complicated than, for example, sorted heaps. For insertion the buffer is initialized with NIL references. Following the first element is placed in the middle. Subsequent inserts perform the search operation described above to find a duplicate or an empty slot in the tree. This process works for the first k insertions. Thereafter, it may happen that we bounce upon an already occupied element. This means that part of the buffer should be shifted to make room for the new element.

For example, consider the situation shown in Figure 5.top. Assume that the element 18 is being inserted. Then at the end of our search we find position 7 as a candidate for holding 18. This means that either we shift elements 5-7 one place to the left (Figure 5.middle) or we shift elements 7 one place to the right. (Figure 5.bottom) Notice that in the latter case we should propagate element 24 up the virtual path so as to guarantee that each node remains accessible from the root node. Since the latter is bounded by the level of the tree we can again use code expansion.

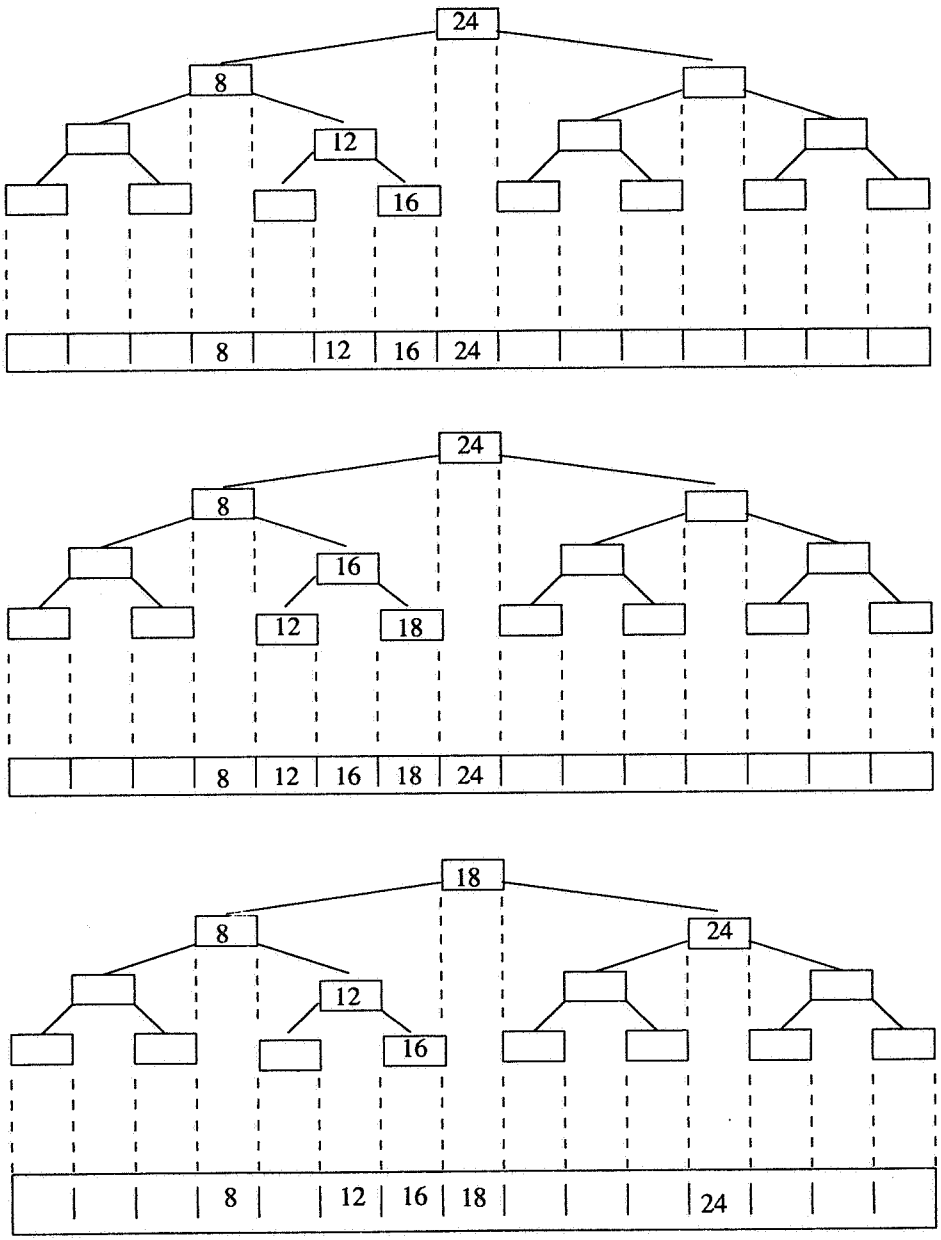

Figure 5. Insertion into a virtual tree

Since we are forced to shift elements when the buffer becomes filled, we can not expect a reduced complexity. Thus, insertion in virtual trees remains an $O(k^2)$ process. But, because we spread the gaps over all elements in the tree we get a significant improvement.

The critical remaining issue then is which way to shift and how. Several options can be considered. First we can look in either direction of the point of insertion for an empty slot. This technique is used in the algorithm shown in Figure 6. The closest one is used by shifting a portion into its direction. Observe that searching an empty slot only requires inspection of the leaf nodes, because a non-empty leaf implies a non-empty parent. By definition all non-leafs occupy even slots in our array.

```
#define test()   if(*w==0){ *w=val;return;}
#define leaf()   test(); if( **w== *val) return;
#define level(X)        test(); if(*val<**w) w-=X; else if(*val>**w) w+=X;else return;

#define replace()       if(*w==0){ *w= *hl; *hl=0; return;}
#define pushup(X)       replace(); if(w<hl) w+= X; else w-= X;

long *inslay7(val,base)
long *base[];
register long *val;
{
        register long **w;
        register long **hf, **hl;

        /* locate element starting at root*/
        w=base+3;
        level(2); level(1); leaf();

        /* search the hole under the assumption that one exists */
        hf=w-2;hl=w+2;
        for(;;){
                if( hf>=base && *hl==0 ) {hl=hf; break;}  else hf-= 2;
                if( hl<base+7 && *hl==0 ) break;  else hl+= 2;
        }

        /* hole is left of hl */
        if(w<hl){
                /* right shift phase */
                for(hf=hl;hf>w;hf--) *hf= *(hf-1);
                if(**w>*val) *w=val; else *(w+1)=val;
        } else {
                /* left shift */
                for(hf=hl;hf<w;hf++) *hf= *(hf+1);
                if(**w<*val) *w=val; else *(w-1)=val;
        }
        /* push last element upward */
        w= base+3;
        pushup(2); pushup(1);
}
```

Figure 6 Insertion algorithm for virtual trees

Alternatively, under the assumption of a uniform key distribution we can take a lazy attitude and use an oracle to determine the shift direction. For example, when the most significant bit of the key value is set then we shift right, otherwise left. Unfortunately, this mechanism does not lead to an acceptable performance in practice. This can be seen as follows. In half of the cases the oracle provides the wrong decision, which means that after we have shifted a portion of the tree and encountered the buffer boundary, we should undo the effect by shifting the portion

back into its original place and continue shifting in the proper direction. Since shifting becomes expensive in the end only, the errors made by the oracle are also noticable.

More exotic algorithms have been designed that aimed at postponing the shifts as long as possible. For example, a k-order virtual tree can be considered as two (k-1)-order virtual trees plus a single node. Then one of these trees can be used as a general overflow area for the other. When a node requires a shift in the primary tree then it is moved into the overflow tree. If, however, a shift is required in the overflow tree then both trees are globally re-organized to push as many nodes as possible in the primary tree. Unfortunately, experience shows that extensions in this direction do not pay off either.

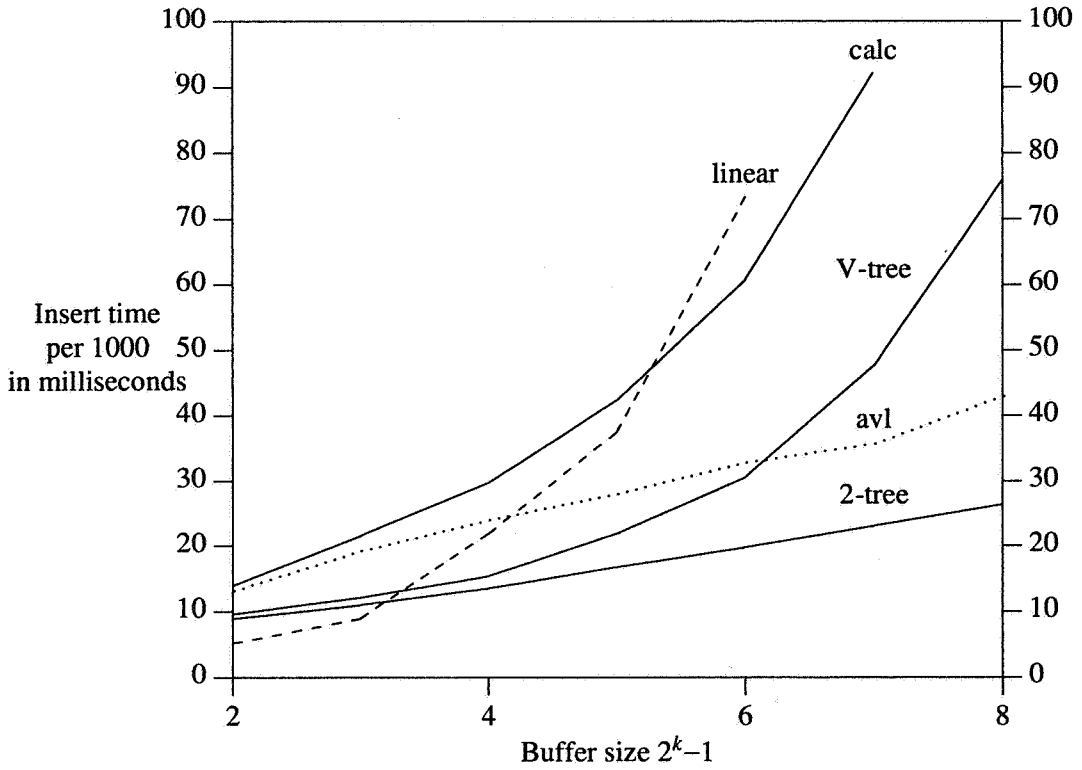

Figure 7 Performance of the insert operator

From the performance figures we may conclude that the layout of the tree elements over the buffer indeed reduces the shifting cost. But still the actual behaviour strongly depends on the insertion order. If the keys are submitted in-order then shifting won't occur and the virtual tree is equally fast as the binary search tree. The worst that can happen to a virtual tree is that during each insert half of the nodes should be shifted. Fortunately, this can not occur, because the nodes are evenly spread as much as possible. This means that the worst-case performance, the *calc* won't be reached.

The measurement data also shows that keeping the tree balanced incurs a constant cost. Yet, for small buffer sizes (<64 elements) the V-tree remains faster.

The performance of the insert routines can be improved in several ways. First, specialized machine instructions can be used to improve the insertion speed. One such operator is a block move operator available on several processors. This technique is also used in the main-memory prototype of Office-By-Example. Alternatively, the duplicated comparison can be removed from the program using a three-way branch encoded in assembler.

An alternative approach is to take a more liberal attitude towards space utilization, because the shift costs strongly depend on the filling of the buffer. If less then k elements are being inserted no shift occurs at all. Contrary, if the buffer is nearly full then on the average half of

the available elements should be shifted. Therefore, it seems warranted to introduce a high-water mark which ensures that the buffer is filled to a certain level. The effect of this technique on insertion cost is shown in Figure 8. It shows the insertion cost into a buffer of 127 elements with a fixed high-water mark ranging from 3 to 95%. We have also included the corresponding insertion cost for binary trees. As you can see the break-even point occurs at about 75%.

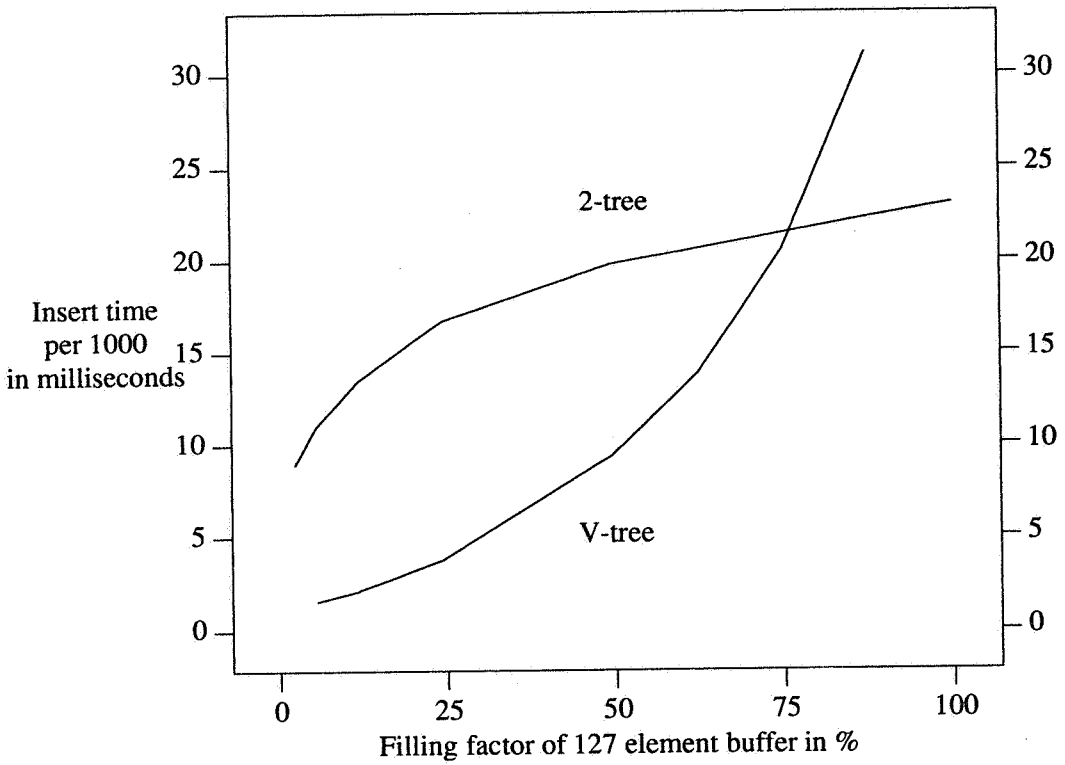

Figure 8 Performance of insertion under high-water mark regime

5. Join operations

One of the reasons for maintaining an access path is to optimize the processing of recurring joins. The general consensus is that a hash-based join algorithm works best when non of the operands is indexed. So far we have been unable to improve the processing of joins using the virtual tree as the intermediate structure. Building a hash index is often cheaper than the construction a virtual tree. In fact, for join processing it suffices to construct a static index.

We have studied the behaviour of several join methods under the assumption that virtual tree(s) exists. The performance of these algorithms are shown in Figure 9. The bottom line again shows the size of the operands (3-1023 elements). The 'tree' line illustrates the performance of a tree-based join method with two virtual tree arguments. The algorithm recursively descends the trees in search of matching tuples. The cost involved in making the result table is left out, because this activity is independent of the way the join method searches the elements of interest.

The 'merge' line shows the performance of the sort-merge join algorithm. The 'lookup' line represents a lookup join algorithm. One of the operands has a virtual tree which is inspected for all joining attributes in the second operand. From these performance figures we learn that all three algorithms have similar performance for small operand sizes. However, for large operands the merge algorithm becomes the primary choice. If the selectivity is low and the matching tuples are clustered in the tree, then a tree-based join algorithm should be considered. Unfortunately, such detailed information is rarely available upfront.

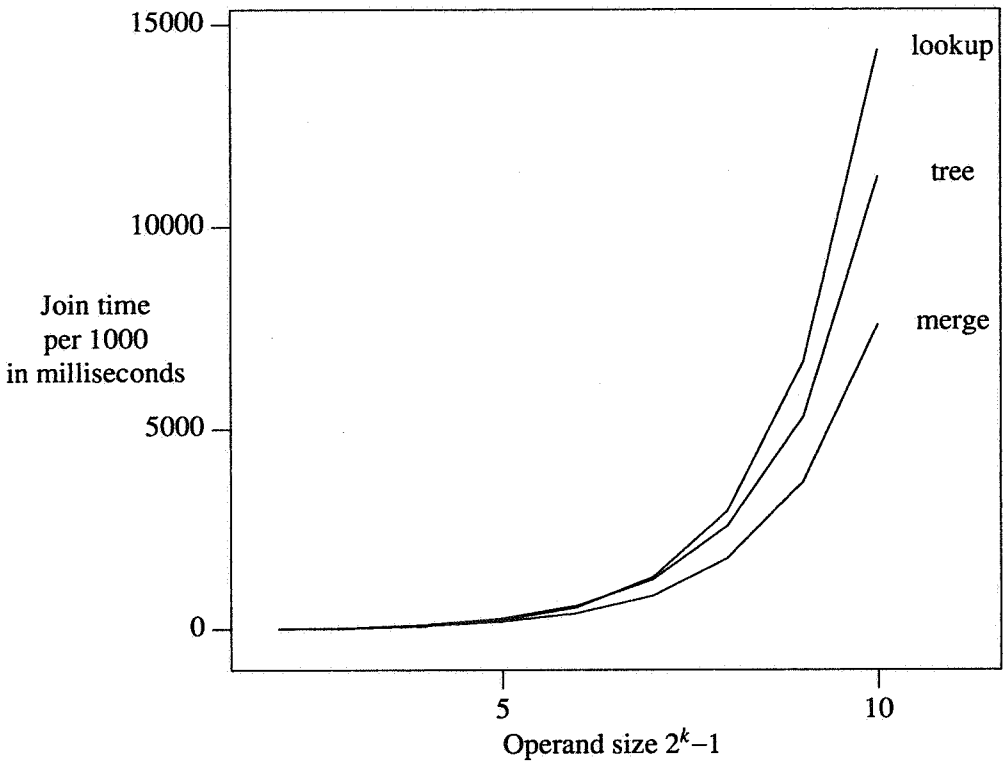

Figure 9 The performance of the join algorithms

6. Summary and Conclusions

In this paper we have introduced an alternative representation of binary search-trees and showed that with judicious use of code-expansion algorithms with good performance can be obtained. Moreover, the techniques can also be applied to existing disk-based systems where the nodes in B-trees (and T-trees) can be configured as virtual trees as well. In many cases this would result in an immediate system improvement with minimal impact on the DBMS software. In one case [Kersten81], applying virtual trees would have resulted in an overall system improvement of about 4 percent for search intensive queries.

In this paper we have refrained from an average-case analysis, because it turns out to be a complex mathematical problem. However, the experimental results once again show that the complexity measure alone is too crude a measure to predict the actual behaviour of algorithms. In many cases we should seriously take into account the constant factor hidden in this measure.

References

[Bitton86]
Bitton, D. and Turbyfill, C., "Performance Evaluation of Main Memory Database Systems", Tech. Report TR 86-731, Jan. 1986.

[DeWitt84]
DeWitt, D., R.Katz, Olken, F., Shapiro, L., Stonebraker, M., and Wood, D., "Implementation Techniques for Main Memory Database Systems," *Proc. ACM SIGMOD* , pp.1-8, June 1984.

[Garcia-Molina83]
Garcia-Molina, H., Lipton, R.J., and Honeyman, P., "A Massive Memory Database Machine", Tech. report 314, September 1983.

[Kersten81]
Kersten, M.L. and Wasserman, A.I., "The Design of the PLAIN Database Handler," *Software, Practice & Experience*, vol. 11, pp.175-186, 1981.

[Kersten87]
Kersten, M.L., Apers, P.M.G., Houtsma, M.A.W., Kuyk, E.J.A. van, and Weg, R.L.W. van de, *A Distributed Main-Memory Database Machine; Research Issues and a Preliminary Architecture*. Proc. 5th Int Workshop on Database Machines, Oct 1987.

[Lehman86]
Lehman, T.J. and Carey, M.J., "Query Processing in Main Memory Database Systems," *Proc. ACM SIGMOD Conference*, pp.239-250, May 1986.

[Leland87]
Leland, M.D.P. and Roome, W.D., "The Silicon Database Machine : Rationale, Design, and Results," *Proc. 5-th Int. Workshop on Database Machines*, pp.454-467, Oct 1987.

[Leland85]
Leland, M.P.L. and Roome, W.D., "The Silicon Database Machine," *Proc. 4-th Int. Workshop on Database Machines*, pp.169-189, March 1985.