Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

Omega-storage: A Self Organizing Multi-Attribute Storage Technique for Very Large Main Memories

J. S. Karlsson, M. L. Kersten

Information Systems (INS)

INS-R9910 September 30, 1999

Report INS-R9910
ISSN 1386-3681

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Ω-storage: A Self Organizing Multi-Attribute Storage Technique for Very Large Main Memories

Jonas S. Karlsson, Martin L. Kersten
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
jonas@cwi.nl, mk@cwi.nl

ABSTRACT
Main memory is continuously improving both in price and capacity. With this comes new storage problems as well as new directions of usage. Just before the millennium, several main memory database systems are becoming commercially available. The hot areas include boosting the performance of web-enabled systems, such as search-engines, and auctioning systems.

We present a novel data storage structure – the Ω-*storage* structure, a high performance data structure, allowing automatically indexed storage of *very* large amounts of multi-attribute data. The experiments show excellent performance for point retrieval, and highly efficient pruning for *pattern searches*. It provides the balanced storage previously achieved by random kd-trees, but avoids their increased pattern match search times, by an effective assignment bits of attributes. Moreover, it avoids the sensitivity of the kd-tree to insert orders.

1991 ACM Computing Classification System: D.4.2 : Storage Management – Main Memory E.1 : Data Structures – Trees, E.2 : Data Storage Representations – Contiguous representations, E.2 : Data Storage Representations – Composite structures, E.5 : Files – Organization/structure F.2.2 : Pattern Matching H.2.2 : Physical Design – Access Methods
Keywords and Phrases: Main-memory multi-attribute data storage, self organizing, pattern matching, based-on-bits
Note: Partially funded by the HPCN/IMPACT project.

1. Introduction

Many unconventional database applications require support for multi-attribute indices to achieve acceptable performance. Decision Support Systems allow users to analyze and use large amounts of data online. Queries may use several attributes simultaneously. Using a main-memory multi-attribute index can greatly speedup interactive analysis for the online user.

The *holy grail* in database systems is a data structure that supports multi-attribute indexed storage, that has minimal insert overhead, and yields highly accelerated searches over very large amounts of online data.

We can observe from the abundant literature that most multidimensional data structures fail one way or another, either for a high number of attributes [WSB98], or when the data is not evenly distributed [NHS84]. Most schemes are static in their partitioning, assuming total randomization, which lead to multi-dimensional hashing of different kinds. Other schemes use adaptive and dynamic partitioning schemes, often resulting in the cost of large main-memory overhead instead.

The Ω-storage technique proposed here is a novel design. It is an *automatic* and *adaptive* indexed storage technique. It requires no tuning or programmer/application selection of indices. Indexing is only performed for data when beneficial in terms of balanced storage under inserts, keeping the indexing overhead low. The Ω-structure is optimized for high performance record retrieval and searches, while allowing *incompletely specified searches*, i.e., searches where only a subset of attributes' values

are known, also known as *pattern searches*. The Ω-tree is a dynamic tree data structure that copes well with varying data distributions. Point inserts and retrievals are completed in logarithmic time.

In contrast to most multi-dimensional hashing schemas, the Ω-tree exploits the data skew. It ignores bits which have no use for indexing, providing highly efficient and adaptable incremental reorganizations. Moreover, data inserts in sorted order on one or several attributes hardly affects the shape of the resulting tree. Experiments in Section 4 ascertains this by comparing with a kd-tree which experience high skew.

2. Related Work

In this section we give a short overview of different partitioning methods. This area of research started out optimizing the usage of narrow resources, such as main memory, by reducing disc accesses, and limiting CPU usage. Now, during the 90's the scenario has evolved to the needs of new applications areas focusing on high availability and high-performance accesses. This is achieved by index structures that use main memory (sometimes distributed) to automatically manage highly dynamic datasets and which can adopt itself to different distributions, avoiding the deficits of earlier indexing methods' worst case behaviors.

For static data sets, one can employ a *choice-vector* which defines what bits from what attributes to use. Furthermore the bits can be chosen in such a way that recurring queries run fast. This is shown in the *multi-attribute* hashing structure proposed in *Towards Optimal Storage Design — Multi-attribute Hashing* [Har94]. Two strategies are investigated for the selection of the bits, one method gives each attribute *equal* chance of being used, the other gives the *minimal* bit allocation, also referred to as the *optimal* allocation.

Many multi-dimensional storage structures are based on the idea of mapping several dimensions into a one dimension and then exploit the highly investigated field of one dimensional data structures. An effective scheme is to use multi-dimensional (order preserving) hash structures. A pseudokey (bitstring of fixed length) is constructed by interleaving bits from the different attributes. During the insertions of data into the storage structure, an increasing number of bits are used to organize, access to data. Different strategies include MOLPHE [KS86], PLOP-hashing [KS88], quad-trees [Sam89], kd-tries [Ore82], and others [Oto88] [HSW88] [Tam81].

However, it is common for these statically defined hashing schemes that while some bits occur to be "random" others are totally useless for indexing and leads to unbalanced structures.

The prominent tree-based structure for multi-attribute searching is the kd-tree [Ben75]. It is a binary tree. The discriminator in internal nodes, was originally limited by strict cycling through the attributes, attribute $i\%k$ at level i in the tree. Later, the *optimized kd-tree* [FBF77] was introduced, storing the records in buckets, and choosing the attribute with the largest spread in values as discriminator, using the mean value for partitioning. kd-tree were then introduced as a general search accelerator for searching multi-key records by suggesting means for storing data on secondary storage devices [Ben79].

Rivest PhD Thesis [Riv74] analyzes, among other structures, a kd-tree style structure using a binary bit from the data as discriminator. The performance of Tries [Riv74] are also analyzed which parallels the analysis the kd-trees. Here the discriminator of a node is chosen so that it has not been used higher up in the tree.

This overview demonstrates that both static and dynamic methods can supply only a partial solution to the problem space. The Ω-storage combines these methods in a way explained in the next section.

3. The Ω-storage

We now explore the design space of the Ω-structure using a dynamic tree structure to efficiently prune the search space. The tree uses *actual* bit values from the attributes to organize the tree during the *split* of a leaf-node into several new nodes. A number of split-strategies are discussed in Section 3.4. We show, in Section 3.1, how the records are stored in *buckets* clustered on attributes, and, in Section 3.3, we show how the structure is searched efficiently.

For simplicity it is assumed that all attributes are discrete and of limited cardinality. Furthermore,

for simplicity we assume the domains of an attribute to be compacted to the range $[0..2^N - 1]$, where N bits are needed to store the data.

3.1 Buckets and Branch nodes

Attribute	Domain	Bits
Age	[0..120]	7
City	[0..10]	4
X	[0..7]	3
Y	[0..49]	6
Sex	{male,female}	1

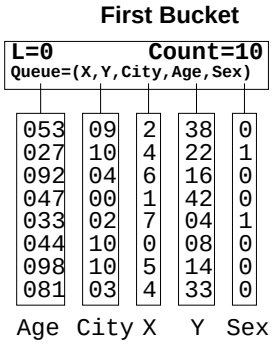

Figure 1: A bucket of an Ω-tree and its attributes.

The Ω-tree consists of two components, the *leaf*-nodes, which store the data, and the *branch*-nodes, which organize the access structure. A branch is defined by the *split-points* of a node. A split-point is a tuple `<attribute, bit-number>`. In general there can be $2^{\#\text{split-points}}$ branches. The *leaf*-nodes (*buckets*) contain vertically partitioned records, i.e., using one array per attribute. Vertical partitioning has been shown to give supreme performance in Monet [BWK98] [BK95]. A mimimal *Omega*-tree consisting of a single bucket is shown in Figure 3.1. Figure 3.1 show the stored attributes. The domain column shows the discrete value span for the domain, the column "Bits" shows the number of bits required to store the normalized domain. The *branch*-nodes, lead to *branches* at lower nodes or to leaf-nodes. The characteristics of the branches is decided at split time.

3.2 An Example

In Figure 2 we show a more elaborate Ω-tree. The branch-nodes have a set of split-points. A split-point pose limitations on the subtrees that are given by following the branches. Each bucket's domain is completely and uniquely specified by it's path from the root node. For a record to be stored in a bucket it has to fulfill the conditions summarized in the box of the bucket.

The "root-node" — the leftmost node — splits the tree into two branches. There is only one split-tuple in that node, namely `(City, 3)`, which indicates that the tree has two branches split using the 3rd bit from the attribute `City`. Bits are numbered 0 from the Least Significant Bit (LSB) to the Most Significant Bit (MSB). Since the split uses the highest bit of `city` (bit 3), it divides its domain into two intervals. One being `city`<8, and one being `city` >= 8. At the next level, splits have been decided independently in the two sub-trees. The `city`<8 branch splits on age, `(Age, 6)`, giving the two branches `age`<64 and `age`>=64. The lower sub-tree, `city`>=8, was split using *two* attributes bits, again `(Age,6)` but with `(Sex,0)`, giving 4 branches. The identity (additional restrictions) of the branches can be seen in the figure. In this tree, both nodes of the second level are split again on the `age` attribute using the 5th bit (value = $2^5 = 32$), further dividing the intervals.

For attributes where not all bits are used there is an uncertainty about which domain the sub-tree they belongs to. In the uppermost right node in Figure 2, we can see a node split using `(City, 1)`. Notice that the bit `(City, 2)` has been left out, since it was of no use for splitting: for all records the bit have the same value. This creates a complex active interval for the resulting buckets. We have depicted the domain of the sub-tree for the `city` attribute as `city="0X0X"` and `city="0X1X"`. The "`X`:s" can still be used in a further split in this sub-tree. When searching an explicit value using only `city`, still only one branch needs to be visited. An interval search on `city` both branches may have

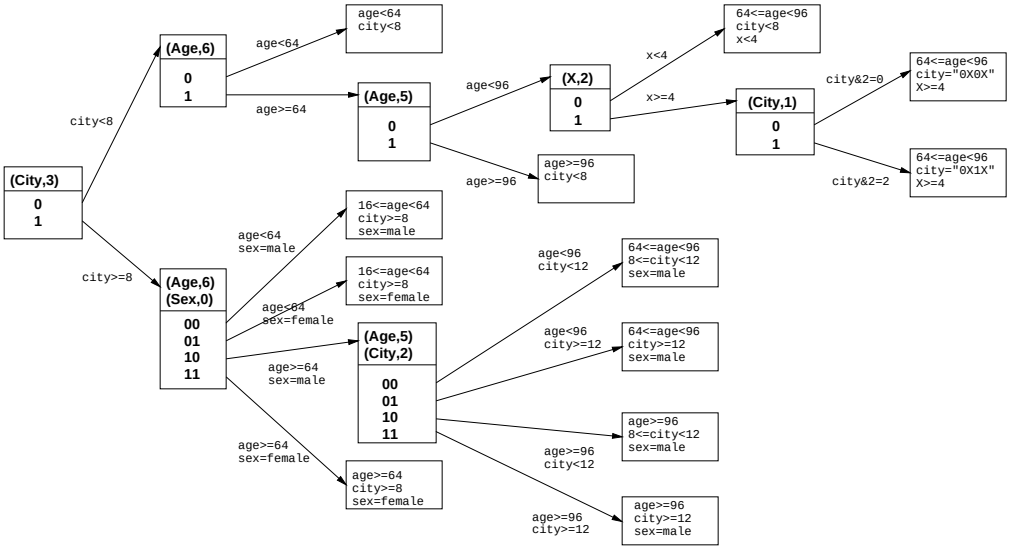

Figure 2: A "typical" Ω-marshaled tree.

to be visited. However, studying the domain set of the two buckets we find that the first stores `city` $\in$ {`0,1,4,5`} and the second `city` $\in$ {`2,3,6,7`}.

3.3 Point Searching

To locate the position in the tree for a given record we start navigating the tree at the root node. By examining the branches at the current node we can decide which branch to follow reaching a new node. The process is repeated, eventually leading up in the appropriate bucket. A branch is chosen if the nodes split-points' values on the branch agrees with the same bits in the record.

Similarly to kd-tree the search/insert complexity is logarithmic [Ben75].

Incompletely specified searches are performed similarly, but may, enter several branches and buckets.

3.4 Splitting Strategy

During the growth of the tree, buckets will become overloaded, i.e., reaching their storage capacity causing them to split. A split is performed by partitioning the content of the current bucket into some new buckets and replacing the current bucket by a branch node. The partitioning is defined by a split-point. Which split-point is chosen depends on the *split-strategy* employed. More explicitly a split strategy defines; the attributes to consider and their order, and in which order the bits of the attributes are preferred, and which bit value distributions are *acceptable*. A bit is *acceptable* when the count of 1:s over the records in the bucket is in the percentage range $[50\% - A, 50\% + A]$, where A is a structure parameter, further investigated in Section 4.5. More significant bits are preferred.

We use a new split strategy called Ω-marshal, which fulfills a number of goals. First, all attributes should be given a chance of being used in the decision split-points in the tree structure. Secondly, it aims to use attributes in split-points on the whole width of the tree, to guarantee efficient pruning. Third, bits are used for easy splitting and organization of the tree. Fourth, bits are to be preferred in such an order that range-queries would benefit. And, finally, bits are chosen by a local split operation only if they are acceptable.

Alternative strategies are used in: *randomized* kd-tree [DECM98], and kd-trie [Ore82], and Ω-pseudo [KK99]. These strategies have been found to have their limitations. This is further discussed in [KK99] where a metrics is developed to shed light on their inner workings. Based on these experiences we have designed the Ω-marshal strategy.

Splitting Algorithm The pseudo-code in Figure 3 describes the details of how a bucket is split in the Ω-marshal structure. If a bucket after an insert reached its LIMIT it is split and replaced by a new internal node. The new node contains branches to newly created buckets. For efficient splitting, using our vertically partitioned storage schema, we first determine the *split-point*. The split-point is determined by searching the attributes from the queue in the bucket. The first attribute with an acceptable bit is chosen.

When a split-point has been found, it is used to create a splitvector that holds the destination bucket for every attribute. Both the search for a split-point and creating the splitvector requires sequential accesses only. Then in `Split`, the attributes are moved sequentially to the new buckets. The buckets are then assigned a queue where the used attribute has moved to the end, enabling a cycling through the attributes used in the queue.

```
proc Determine splitpoint(array Records, array Attributes) ≡
  for ∀a ∈ Attributes do
      for ∀r ∈ Records do
          update count(r.a) od
      for ∀bit ∈ 31..0 do
          if (acceptablecount(bit))
            return < a, bit > fi od od.

proc Calculate Splitvector(array Records, a) ≡
  array move[1..LIMIT]
  for ∀i ∈ 1..LIMIT do
      move[i] := (Records[i].a.bit) od
  return move.

proc Split(array Records, array Attributes, array move) ≡
  array Bucket[0..1];
  for ∀a ∈ Attributes do
      for ∀i ∈ 1..LIMIT do
          add Record[i].a to Bucket[move[i]] od od.
```

Figure 3: Bucket Split Algorithm

4. Performance evaluation and Tuning

In this section we analyze and benchmark the Ω-storage structure. We confirm that single records inserts and searches conform to $O(\log(n))$, and that the time for partially specified record searches decreases exponentially with the amount of attributes specified. Comparison is performed against the kd-tree showing excellent search performance, and highly improved stability in single record search times, enabled by the better balanced tree. The number of internal nodes used by the Ω-structure is 14K compared to the 24K of the kd-tree.

For our experiments we use an SGI Origin 2000 currently equipped with 24 CPUs and a total of 48 GBytes RAM. A 64-bit process can transparently, from the programming point of view, access all its RAM. However, there are extra costs. Each CPU has "local" access to 2 GBytes RAM and "remote" memory is cashed. The operating system may move processes from one CPU (and memory) to another if it decides that this would be beneficial for the process, because a large number of remote memory accesses can be eliminated by executing the process at an other CPU where that memory is local.

First, we find the optimal capacity of an Ω-bucket. Then we discuss the insert performance.

4.1 Bucket Size vs Pruning

Although the Ω-tree is an automatic storage schema, there are still a few parameters of interest to tune. These parameters depends on the underlying hardware, i.e memory access costs, memory access patterns and cache-performance.

We identify two such parameters for the Ω-tree. The first parameter is the average bucket size implied by an upper LIMIT on the bucket size. This parameter is affected by the target hardwares' cache capabilities. The second parameter is the acceptable 0/1 frequency of a bit to be considered in a split.

If the bucket size is too large, a single element search exhibits time linear to the size of the bucket, and if the bucket size is too small, we will spend more time navigating the tree structure. Therefore, we choose to determine a LIMIT large enough not to influence single element search times.

Bucket Size	Max	Min	Avg	Std Dev
50	17	15	16.5	0.82
100	18	16	16.7	0.82
200	21	15	16.9	1.51
1000	25	18	20.1	2.23
2000	38	19	26.5	5.40
3000	44	25	34.2	6.72
10,000	130	44	77.5	32.63

Table 1: Statistics on point searches, varying bucket limits, times in [μs].

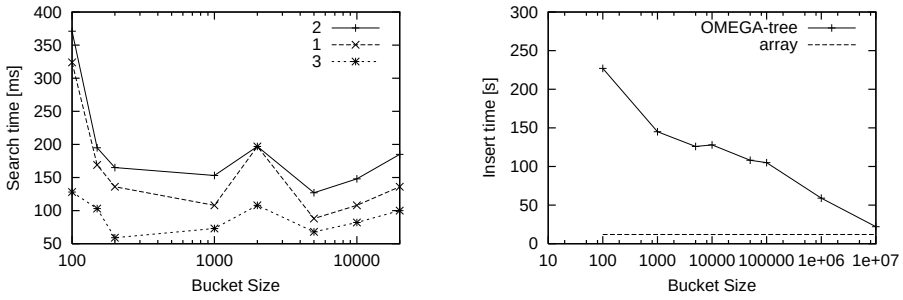

Figure 4: Varying a) search times b) insert times.

Table 1 shows experiments of singe-point searches where the upper bucket size limit is varied between 50 up to 10,000 elements per bucket. 10 M records with 8 attributes were inserted using 320 MBytes of memory just for the record storage. The search time is dominated by the navigation time. For a limit up to 1000 records the search time is stable, starting at 15 μs for bucket of size 50, going up to approximately 20 μs at a limit of 1000. Beyond the 1000 records the actual size of the bucket is reflected or dominates (over 4000) the search times.

Figure 4(a) depicts the search time for different searches using 1,2, and,3. Using less attributes then querying increases the search time, with a local minima at 200 to 1000 and around 5000 to 10000 limited bucket size. For the remaining experiments we choose the maximum bucket size to be 1000 since point searches are predictable and fast. Others searches are reasonable in search time.

In Figure 4(a) there is a peak at a LIMIT of 2000. The explanation is simple – at this point the number of buckets decreases, and with that the number of branch-nodes causing the tree height of the tree to be somewhat lower. For these queries this had the effect that the search times increased since a vital index level vanished.

4.2 Insert costs

Figure 4(b) shows the total insert times for 10M inserts in seconds for the Ω-tree using increasing bucket sizes. The overhead consists of function call costs and the cost of tree reorganizations. As a reference we show the time for inserting the record into one array each for the attributes. The quotient between the insert time of the Ω-tree and the insert into arrays roughly approximates the height of the tree, thus reflecting the number of times a value has been copied during a split.

For example inserts with $LIMIT = 1000$, uses 145 seconds, 12.1 times as long as linear storage. This insert time is related to the current tree depth, in this case approximately 14. Increasing the bucket size by a factor of 10 ($LIMIT = 10000$) just causes a slight decrease to 10.6 times as long. Still, at $LIMIT = 10,000$ the actual time per insert is only 14.5 μs. Inserts using the Ω-tree shows a final overhead of 22 seconds for an "infinite" large bucket.

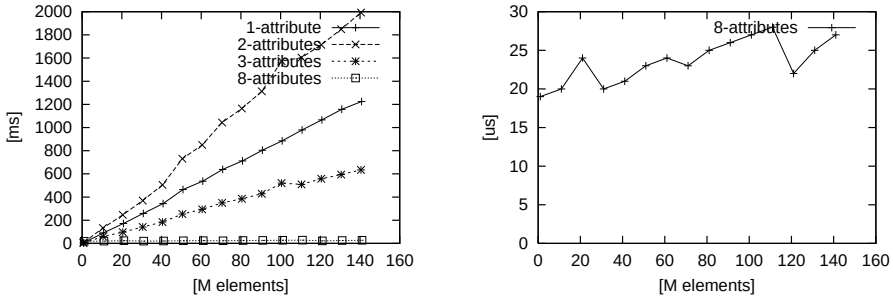

Figure 5: Search time using a) 1,2,3,8 attributes in Ω-tree b) details of 8 attribute.

4.3 Search cost for a growing data set

First we ascertain that a point search is a highly efficient and fast operation. As can be seen in Figure 5(b), the cost starts at 19 μs for 1 million tuples to increase to 27 μs for 140 times more data!

When querying larger data sets, as shown in Figure 5(a), query times are significantly higher. For these specific queries the query time as well as the result set size increase linearly with the file size. The 3 attributes query rises to just above 400 ms, 2 attributes gives a search time of 2000 ms, nearly the performance of linear scanning. Whereas 1 attribute is somewhat more efficient. The reason here is that the result of the 2-attribute query is a subset of the results of the 1-attribute query and that there is no index available for the second attribute, thus the same amount of data is scanned but requiring two attributes testing.

In Figure 6(a) we compare the Ω-tree pattern search performance with linear scanning. Point search, in this case 8 attribute search, gives the highest improvement, with a search time negligible compared to scanning. The other queries improve the search time by a factor of 3 to 10 times over scanning. The improvement ultimately depends on the search pattern specified and the result size.

To access the average performance of different queries, we observe all partial match queries (256 for 8 attributes), using a subset of the attributes from a specific record. The resulting plot is shown in Figure 6(c). The average performance quickly improves when more attributes are present. Decreasing from seconds to micro seconds for point queries. The best performance – 14 μs - is achieved when all attributes (8) are specified in the query. Using 7 attributes gives results in the range from 20 μs up to 299 μs, with an average search time of 50 μs.

4.4 Influence of Number of Attributes

To investigate how Ω-tree performs for a higher number of attributes we build files with a varying number of attributes. We have already shown the performance for 8 attributes, and will now quickly compare it will 16 attributes.

Figure 6(d) shows statistical values for a 16 attribute file. The average search times are drastically (exponentially) reduced when more information is available in pattern search, however, the search times varies orders of magnitudes depending on the values of which attributes are present in the query The curves have similar character as for the 8 attribute file, Figure 6(c). The minimum and maximum observed values are orders of magnitudes lower respectively higher. One difference in this experiment

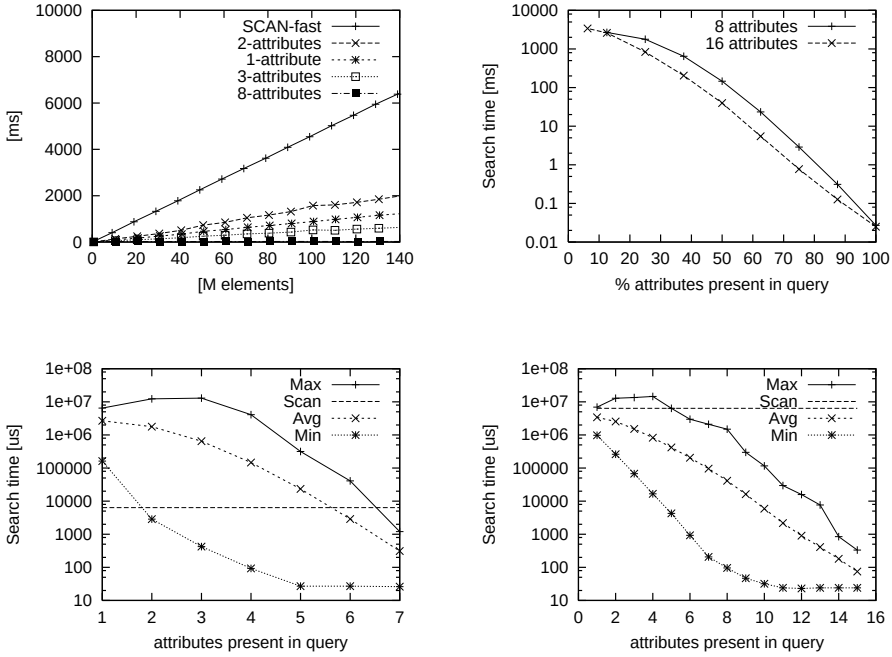

Figure 6: a) Search times b) 8, 16 attributes files c) 8 attribute file e) 16 attribute file.

is that the area of the search time bounded by maximum and minimum search time is smaller than the case of a 8 attribute file.

4.5 Comparison with kd-tree

We now compare the performance of the Ω-tree with the performance of the kd-tree, using an 8-attribute file, searching 10M records.

Figure 7(a) shows the performance of the kd-tree and Ω-tree 10% relative to the high performing Ω-tree 20%. For the Ω-tree 10% the overhead is reasonably stable around 10-20%, but for the kd-tree it starts at around 30% going up to 10 times as much for searching records using 8 attributes. The reason for this is the skewed kd-tree which for most of the data in this experiment got very deep thereby causing large overhead to accessing individual records at the leaves.

Figure 7(b) shows the standard deviation observed for the kd-tree and Ω-tree. For less specified pattern searches the standard deviation is slightly higher using the kd-tree than for the Ω-tree. For highly specified patterns the deviation on the kd-tree does not improve compared to the Ω-tree. The Ω-tree comes near a perfectly stable search time with a very low variance of 2.

This is due to the inability of the kd-tree to handle different data distributions and insert orders, the kd-tree might be very skewed giving very fast access to some records that are near to the root but performing poorly when searching for records that are stored further down in the tree. The Ω-trees on the contrary exploits these skewed distributions and insert order providing more stable search performance.

We make two observations: First, increasing the acceptance interval, as defined in Section 3.4, from $A = 10\%$ to $A = 20\%$ gives a slight but stable performance increase. The only noticeable drawback is that it increases the amount of buckets needed to store the data from around 14K to 15K with the depth still around 14 levels. This is to be compared with the 24K buckets generated by the kd-tree and its depth of 25 levels. The second observation is that the performance of the Ω-tree is much more

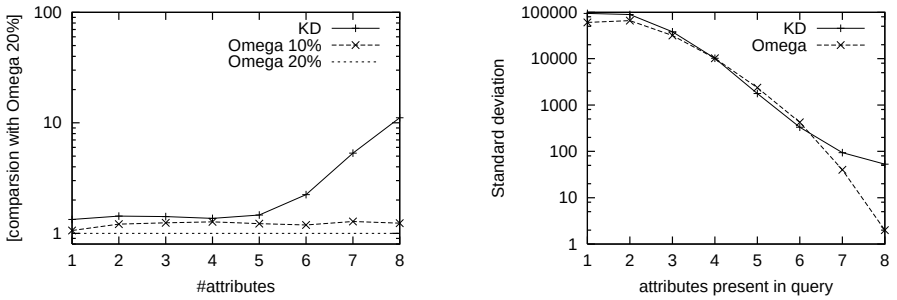

Figure 7: 8 attribute file, standard deviation for patterns search a) The kd-tree and Ω-tree 10% compared with the Ω-20% b) KD compared with Ω-tree with acceptance limit of 10% and 20%

stable than that of the kd-tree. The Ω-tree avoids creating skewed trees in cases where the kd-tree would.

5. Conclusions

We have presented the Ω-storage structure, a self organizing multi-attribute indexed storage. The performance has been assessed using generated data from the drill down benchmark [BRK98]. Compared to the kd-tree, the Ω-storage method provides a highly stable performance for single records searches over GBytes of data, while avoiding the highly skewed structures easily created by the kd-trees. This is realized by relaxing the constraints, while maintaining the intended properties of kd-trees like highly efficient pruning.

Future work, currently being investigated involves creating a scalable distributed extension of the Ω-storage. For this we plan to use the dissection splitting algorithm of hQT* [Kar98] which was previously shown to provide excellent partitioning for the distributed quad-tree structure.

References

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[Ben79] Jon L. Bentley. Multidimensional binary search trees in database applications. *IEEE Transactions on Software Engineering*, SE-5(5):333–340, July 1979.

[BK95] Peter A. Boncz and Martin L. Kersten. Monet: An Impressionist Sketch of an Advanced Database System. In *Basque International Workshop on Information Technology: Data Management Systems*, San Sebastian (Spain), July 1995. IEEE.

[BRK98] P. A. Boncz, T. Rühl, and F. Kwakkel. The Drill Down Benchmark. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 628–632, New York, NY, August 1998.

[BWK98] Peter A. Boncz, Annita N. Wilschut, and Martin L. Kersten. Flattening an object algebra to provide performance. In *IEEE 14th International Conference on Data Engineering*, pages 568–577, Orlando, FL, USA, February 1998.

[DECM98] Amalia Duch, Vladimir Estivill-Castro, and Contrado Martinez. Randomized k-dimensional binary search trees. In *Lecture Notes in Computer Science*, volume 1533, pages 199–208, Taejon, Korea, December 1998. Springer.

[FBF77] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(2):209–226, September 1977.

[Har94] Evan Philip Harris. *Towards Optimal Storage Design for Efficient Query Processing in Relational Database Systems*. Phd-thesis tech report 94/31, University of Melbourne, November/May 1994.

[HSW88] Andreas Hutflesz, Hans-Werner Six, and Peter Widmayer. Globally Order Preserving Multidimensional Linear Hashing. In *ICDE: Fourth International Conference on Data Engineering*, pages 572–579, Los Angeles, California, 1988.

[Kar98] Jonas S Karlsson. hQT*: A Scalable Distributed Data Structure for High-Performance Spatial Accesses. In Katsumi Tanaka and Shahram Ghandeharizadeh, editors, *FODO'98: The 5th International Conference on Foundations of Data Organization*, pages 37–46, Kobe, Japan, November 1998.

[KK99] Jonas S. Karlsson and Martin L. Kersten. An Exploration of the Omega-Storage Design Space. Unpublished Technical Report, CWI, The Netherlands, 1999.

[KS86] H.-P. Krigel and B. Seeger. Multidimensional Order Preserving Linear Hashing with Partial Expansions. In *International Conference on Database Theory*, pages 203–220, Rome, 1986.

[KS88] H.-P. Krigel and B. Seeger. PLOP-Hashing: A Grid File without Directory. In *ICDE: Fourth International Conference on Data Engineering*, pages 369–376, Los Angeles, California, 1988.

[NHS84] Jörg Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems (TODS)*, 9(1):38–71, March 1984.

[Ore82] Jack A. Orenstein. Multidimensional tries used for associative searching. *Information Processing Letters*, 14(4):150–157, June 1982.

[Oto88] Ekow J. Otoo. Linearizing the Directory Growth in Order Preserving Extendible Hashing. In *ICDE: Fourth International Conference on Data Engineering*, Los Angeles, California, 1988.

[Riv74] Ronald Linn Rivest. *Analysis of Associative Retrieval Algorithms*. PhD-thesis STAN-CS-74-415, Computer Science Department, Standford University, May 1974.

[Sam89] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. January 1994 edition, 1989.

[Tam81] Markku Tamminen. Order Preserving Extendible Hashing and Bucket Tries. *BIT*, 21(4):419–435, 1981.

[WSB98] Roger Weber, Hans-J. Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24th VLDB Conference*, pages 194–205, New York, USA, 1998.