Scalable distributed data structures for database management
Karlsson, S.J.

Citation for published version (APA):

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 4

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

This chapter is based on the publication Ω-storage: A Self Organizing Multi-Attribute Storage Technique for Very Large Main Memories presented at the Australian Database Conference, Canberra, Australia, January 2000. Here we present the concept of the Ω-storage, and identify the design space on how to build the Ω-trees. It is a combination of several choices. First, the attribute bits are preferred in an order beneficial for range-queries. Second, bits which would give a badly balanced tree are avoided. Third, the algorithm attempts enable each attribute to have the widest influence on the pruning by a clever queuing of attributes to use.

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.

**Keywords:** Main-memory multi-attribute data storage, self organizing, pattern matching, based-on-bits

### 4.1 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 speed up 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 4.4 ascertains this by comparing with a kd-tree which experiences high skew.

### 4.2 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 disk accesses, and limiting CPU usage. Now, during the 90's the scenario has evolved to the needs of new application 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 (bit-string 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-trees [Ore82], and others [Oto88] [HSW88] [Tam81].

However, it is common in 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 $\Omega$-storage combines these methods in a way explained in the next section.
4.3 The $\Omega$-storage

We now explore the design space of the $\Omega$-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 4.3.4. We show, in Section 4.3.1, how the records are stored in buckets clustered on attributes, and, in Section 4.3.3, 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.

4.3.1 Buckets and Branch nodes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Domain</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age</td>
<td>[0..120]</td>
<td>7</td>
</tr>
<tr>
<td>City</td>
<td>[0..10]</td>
<td>4</td>
</tr>
<tr>
<td>X</td>
<td>[0..7]</td>
<td>3</td>
</tr>
<tr>
<td>Y</td>
<td>[0..49]</td>
<td>6</td>
</tr>
<tr>
<td>Sex</td>
<td>{male,female}</td>
<td>1</td>
</tr>
</tbody>
</table>

First Bucket

<table>
<thead>
<tr>
<th>Count=10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue=(X,Y,City,Age,Sex)</td>
</tr>
<tr>
<td>L=0</td>
</tr>
<tr>
<td>053 09 2 38 0</td>
</tr>
<tr>
<td>027 10 4 22 1</td>
</tr>
<tr>
<td>092 04 6 16 0</td>
</tr>
<tr>
<td>047 00 1 42 0</td>
</tr>
<tr>
<td>033 02 7 04 1</td>
</tr>
<tr>
<td>044 10 0 08 0</td>
</tr>
<tr>
<td>098 10 5 14 0</td>
</tr>
<tr>
<td>081 03 4 33 0</td>
</tr>
</tbody>
</table>

Figure 4.1: A bucket of an $\Omega$-tree and its attributes.

The $\Omega$-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 $\langle\text{attribute, bit-number}\rangle$. 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 minimal $\Omega$-tree consisting of a single bucket is shown in Figure 4.3.1. Figure 4.3.1 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.
4.3. THE Ω-STORAGE

4.3.2 An Example

In Figure 4.2 we show a more elaborate Ω-tree. The branch-nodes have a set of split-points. A split-point poses 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.

In some cases, a bit will not be used, as can be seen in the uppermost right node in Figure 4.2. There there is a node split using (City, 1), bit (City, 2) has been "skipped" over. The reason is that the bit was of no use for splitting for all records the bit have the same value. However, when searching this is not known, therefore there is an uncertainty about which domain the sub-tree a specific value belongs to. This creates a complex
active interval for the resulting buckets. We have depicted the domain of the sub-tree for the city attribute as \texttt{city="0X0X"} and \texttt{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 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 $\in \{2,3,6,7\}$.

### 4.3.3 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.

### 4.3.4 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 4.4.5. More significant bits are preferred.

We use a new split strategy called $\Omega$-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 $\Omega$-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-marshall strategy.

### Splitting Algorithm

The pseudo-code in Figure 4.3 describes the details of how a bucket is split in the Ω-marshall 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.

```plaintext
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 (acceptable count(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 Records[i].a to Bucket[move[i]] od od.
```

Figure 4.3: Bucket Split Algorithm

### 4.4 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 \( \Omega \)-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 cached. 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 \( \Omega \)-bucket. Then we discuss the insert performance.

### 4.4.1 Bucket Size vs Pruning

Although the \( \Omega \)-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 \( \Omega \)-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.

<table>
<thead>
<tr>
<th>Bucket Size</th>
<th>Max</th>
<th>Min</th>
<th>Avg</th>
<th>Std Dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>17</td>
<td>15</td>
<td>16.5</td>
<td>0.82</td>
</tr>
<tr>
<td>100</td>
<td>18</td>
<td>16</td>
<td>16.7</td>
<td>0.82</td>
</tr>
<tr>
<td>200</td>
<td>21</td>
<td>15</td>
<td>16.9</td>
<td>1.51</td>
</tr>
<tr>
<td>1000</td>
<td>25</td>
<td>18</td>
<td>20.1</td>
<td>2.23</td>
</tr>
<tr>
<td>2000</td>
<td>38</td>
<td>19</td>
<td>25.5</td>
<td>5.40</td>
</tr>
<tr>
<td>3000</td>
<td>44</td>
<td>25</td>
<td>34.2</td>
<td>6.72</td>
</tr>
<tr>
<td>10,000</td>
<td>130</td>
<td>44</td>
<td>77.5</td>
<td>32.63</td>
</tr>
</tbody>
</table>

Table 4.1: Statistics on point searches, varying bucket limits, times in [\( \mu \)s].

Table 4.1 shows experiments of single-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 4.4(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 4.4(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.

### 4.4.2 Insert costs

Figure 4.4(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.
CHAPTER 4. ω-STORAGE: MULTI-ATTRIBUTE STORAGE

4.4.3 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 4.5(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 4.5(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 4.6(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 assess 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 4.6(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.

Figure 4.5: Search time using a) 1,2,3,8 attributes in Ω-tree b) details of 8 attribute.
4.4. PERFORMANCE EVALUATION AND TUNING

Figure 4.6: a) Search times b) 8, 16 attributes files c) 8 attribute file e) 16 attribute file.
4.4.4 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 with 16 attributes.

Figure 4.6(d) shows statistical values for searching a 16 attribute file. When the information of the pattern search is increased the average search times decrease exponentially. For example, when 10 out of 16 attributes are known, the average search time is approximately 10 ms, and for 12 out of 16, it is about 1 ms. The individual searches do, however, exhibit a significant variance in their search times in the order of magnitudes. The curves for the 8 attribute file, Figure 4.6(c), have similar characteristics.

Observing the minimum and maximum search times, it is clear that they are orders of magnitudes apart. Comparing the case of the 8 attributes to that of the 16 attributes file, one notices that the spread (variance) of the searches decreases. In this case of the 16 attribute file each record has the double amount of data (bits) stored in it, giving more choices for the split algorithm to choose a bit for the split. This gives a higher chance of a better load-balanced structure, by giving more options to fulfill the goals during the split.

4.4.5 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 4.7(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 4.7(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
4.5. **EXPLORATION OF THE Ω-TREE DESIGN SPACE**

Figure 4.7: 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%

providing more stable search performance.

We make two observations: First, increasing the acceptance interval, as defined in Section 4.3.4, 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 stable than that of the kd-tree. The Ω-tree avoids creating skewed trees in cases where the kd-tree would.

### 4.5 Exploration of the Ω-tree design space

This Section seeks to give some background information on the design choices that were made during the development of the Ω-storage.

#### 4.5.1 Branches

(Binary) tree structures have the disadvantage over hashing structures that they require substantial navigation effort, visiting large number of disperse memory locations. The number of navigation steps can be reduced by having more branches in each branching node, for example $2^i$ elements instead of 2 as in the binary tree. However, additional costs are associated with calculating the appropriate branch(es), i.e. a cost similar to the cost of hashing.

In our case, we tried solutions which just “compacts” the binary tree by combining several split-points into one node, giving $2^{\#\text{split-points}}$ branches. The additional cost of combining the splitpoints into one value to be used for navigation turned out to be higher or the same for the navigation avoided. For such an approach to be fruitful a large number of branches are needed.
Linear scanning is faster than complicated tree-navigation or hashing up to a certain number of elements. This is due to the added complexity which hashing and tree navigation adds. Adding more branches also reduces the flexibility of the organization of the tree as well as increases the storage overhead.

**Reducing dimensionality**

The high-dimensional curse causes problems with the building of the tree. For example, for a balanced $k$ dimensional kd-tree, at least $2^k$ leaf-nodes are required for each dimension to affect the organization of the tree, i.e. to aid the navigation.

By avoiding splits directly along the axis (dimensions) and by combining data from different dimensions in a non-linear fashion the actual dimensions can efficiently be reduced.

Hashing functions are well researched. They are used to create a characterize fixed-length bitstring from a variable or arbitrary large sized input data. This fixed-lenth bitstring/number is called a pseudokey. Hash-functions investigated in the literature focus on creating an apparent random mapping from the input data to the hashed value. Hashing provides uncomparable efficient retrieval when all the input data to the hash-function is known, since it calculated the location where the requested data can be found. One problem often faced by hashing concern storage overhead due to data skew. Another, more fundamental problem, is the inability to perform efficient pattern searches, searches where not all of the input data (for the hash) is known, ending up searching all of the stored data. This is the effect of the normally sought randomness of the hash-function.

By using a table-lookup hash-function, as in ADB hashes [Riv74] the input and output data can be related in such a way that a search using incomplete input data still limits the amount of data needed to be searched. The hash-function of ADB-hashing “compacts” a number of bits in a bitstring to smaller number of bits by introducing a “don’t care” symbol (“*”) in the mapping. It reduces the number of buckets to search when bits are missing in the pattern matching queries by allowing more bits to take part in the indexing schema. There are two drawbacks for the ADB hashes: First, there is an added complexity because of using a table for implementing the hash-functions; Second, there is a penalty for searches that use fewer attributes since more buckets then need to be searched.

We will not explore these methods any further in this PhD, but we notice that these methods closely resembles operations in the field of coding theory. Furthermore, we do not here consider any structures which do not use bits for organizing the search spaces. For those structures it has been proposed to use mathematical tools such as fractal-coding.
4.5. EXPLORATION OF THE Ω-TREE DESIGN SPACE

4.5.2 Dynamic hash-function

Normally, a hash-functions output is predictable since it only depends on the function itself and the input data. Relaxing this constraint one can build a mapping which is knowledgeable about the data that the function organizes. In effect letting the current structure act as additional input to a newly constructed hash-function. The hash-function's behavior is updated at times to allow for balancing storage and a growing data set. Typically, the hash-function would state how many significant bits the output contains for each output. The hash value produced would vary, at different times over the existence of the storage structure, reflecting the movement of the data item inside the structure.

LH [Lit94] has a dynamic hash-function. In the case of LH, only the amount of data stored (or the “load”) controls which hash-function is used. DH [Lar78] and hQTT* [Kar98], on the other hand, uses a different number of bits from a statically generated pseudokey. The number of bits used depends on the local position in the structure and the local load. When a part of the structure is overloaded more bits are used for organizing the data in this part. In effect DH generates a tree structure. The structure itself can be said to be contained in the mapping which defines the hash function. In these structures, however, a bit’s value is purely determined from the item itself. Only the number of bits used varies in different parts of the structure.

Contrary, to having the bits purely defined by the input data, the input data and the current state of the structure can be used to choose how the output should be chosen. This allows for larger flexibility in organizing the structure.

For example, a binary tree can be seen as a dynamic hash-function that generates bit by bit when traversing the tree. When a leaf node overflows, it is split in two leaves causing a branching node to be inserted into its place. Such a node essentially poses a decision of right/left, i.e. 0/1. The hash-function “grows” with the input data. At each moment an item in the tree has a unique hash value associated with it. When the tree grows by a split of a leaf-node another bit is added to the output of the hash-functions for a group of items. In practice, such a hash-function or mapping is implemented by a tree structure.

4.5.3 Explored variants of Ω-trees

The current Ω-tree is essentially a binary tree with a special function for selecting how a split is to be made. During our early exploration of the Ω design space the following methods were considered.

- Ω-random: Random choose any non-used bit which is acceptable at bucket split time. This structure is similar to the randomized kd-trees [DECM98].
CHAPTER 4. $\omega$-STORAGE: MULTI-ATTRIBUTE STORAGE

- $\omega$-stiff: k-d tree style splitting schema where the “best bit” from a list of cycled attribute is chosen, resembles kd tries [Ore82].

- $\omega$-pseudo: “Cleverly” interleave bits of differently sized domains to create a fair and easy split of buckets, by limiting the choices, and promoting attributes equally, independent on their domain sizes.

- $\omega$-marshal: A relaxed schema with the advantages of $\omega$-stiff differing in that more significant bits are preferred. If no acceptable bit is found another attribute is chosen. The attribute skipped at one level has a higher priority to be chosen at further splits in the subsequent subtree.

We summarize the experience for our structures below.

$\Omega$-random achieves a well-balanced storage, however the pattern search is more costly because no guarantees are given for the number of nodes that prune an attribute.

The $\omega$-stiff has some of the advantages of the k-d-tree by cycling through the attributes. This creates “barriers” on every level for one attribute that efficiently prunes half of the branches in a search using that attribute. The disadvantage is that the predetermined attribute may be of no use for a split, causing internal nodes in the tree to be just “dummies”. The tree generated is highly unbalanced, however, it achieves slightly better pruning, thus more efficient search for well-behaved data distributions.

$\omega$-pseudo, being a combination of $\Omega$-random and $\omega$-stiff, tries to be more relaxed avoiding the problems of the unbalanced trees of $\omega$-stiff by allowing a certain amount of randomness at each level. It is designed to mimic the bit-interleaving of the multi-dimensional hashing structures, while being more flexible. The drawback is that highly random attributes are favored above others, and it does not achieve as efficient pruning as the structures with “barriers” (k-d-tree, $\omega$-stiff).

The $\Omega$-marshall, as described in this chapter, efficiently stores data, as well as allows for a stable efficient retrieval compared to the kd-trees.

4.5.4 Implementation Notes

The language C was used for the implementation of the $\Omega$-structures. Optimization were made to avoid many function calls by allowing the compiler to inline functions. The search function operates by a performing a recursive call at each level.

Since, the depth of a large tree can be 10-20 levels, the costs of recursion calls were of concern. We instead the use of an iterative solution by the means of our own stack. It turned out that this was not an important issue, the overhead of building your own stack and iterating using this instead led to no substantial improvements in speed, only more complicated code.
4.6. CONCLUSIONS

The most important gains in speed were made by streamlining bulk operations, such as scanning, splitting and copying data from a bucket to a new bucket. These operations are performed regularly so substantial time is spent there in inserting and searching the data.

4.6 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.