Variable UB-Trees: An efficient way to accelerate OLAP queries[1]

Volker Markl* Michael G. Bauer+ Rudolf Bayer*+

*Bayerisches Forschungszentrum
für Wissensbasierte Systeme

Orleansstraße 34

81667 München
Germany

volker.markl@forwiss.de

+Institut für Informatik
Technische Universität München

Orleansstraße 34

81667 München
Germany

{bauermi,bayer}@in.tum.de

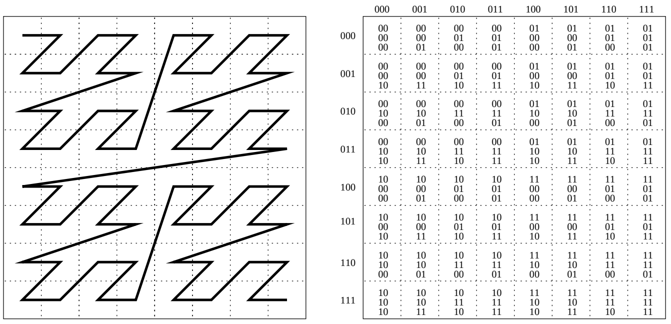
http://mistral.in.tum.de

1 Introduction

Pre-computation, clustering and indexing are common techniques to speed up query processing. Pre-computation results in the best query response time at the expense of load performance and secondary storage space. For data warehousing (DW) applications, pre-computation is mostly discussed for aggregation operations [CD97]. Indexing is used to efficiently process a query if the result set defined by the query restrictions is fairly small. Favoring retrieval response time over update response time allows to build several indexes on one table or data cube of a data warehouse. Bitmap indexes are widely discussed as an improvement over B-Trees for DW applications, since they efficiently evaluate queries with multi-attribute restrictions. However, the overall result set still must be relatively small. This is a major drawback of bitmap indexes, since usually a relatively large part of a cube must be accessed in order to calculate aggregated measures. Clustering places data that is likely to be accessed together physically close to each other. The goal of clustering is to limit the number of disk accesses required to process a query by increasing the likelihood that query results have already been cached. Clustering has been well researched in the field of access methods. B-Trees, for instance, provide one-dimensional clustering. Multidimensional clustering has been discussed in the field of multidimensional access methods. See [GG97] and [Sam90] for excellent surveys of almost all of these methods. A large fragment of queries are ad-hoc queries, where pre-computation cannot be anticipated due to the sheer number of possible aggregate paths. However, one requirement of DW is to efficiently deal with ad-hoc queries. Pre-computation also leads to a view maintenance problem. The Transaction Processing Council has taken this fact into account by creating two new benchmarks from the TPC-D benchmark, namely the reporting benchmark TPC-R and the ad-hoc benchmark TPC-H. Pre-aggregation is largely limited for the ad-hoc benchmark. The focus of our paper is on accelerating ad-hoc queries. In earlier papers [MRB99], [MZB99] we have proposed to accelerate ad-hoc OLAP queries by multidimensional access methods in combination with multidimensional hierarchical clustering (MHC) and the Tetris algorithm which utilizes restrictions to process a table in sort order of any attribute without external sorting. Experiments with the TPC-D benchmark as well as tests with a 5 GB real-world data warehouse from a SAP customer showed that the UB-Tree is a very suitable multidimensional access method to be combined with MHC and the Tetris algorithm. In this

---

[1] Proceedings 2. GI-Workshop "Data Mining und Data Warehousing als Grundlage moderner entscheidungsunterstützender Systeme" (DMDW99), LWA99 Sammelband, ISBN 3-929757-26-5, pp. 79-88, Univ. Magdeburg, Germany, September 1999



	000	001	010	011	100	101	110	111
000	00 00 00	00 00 01	00 01 00	00 01 01	01 00 00	01 00 01	01 01 00	01 01 01
001	00 00 10	00 00 11	00 01 10	00 01 11	01 00 10	01 00 11	01 01 10	01 01 11
010	00 10 00	00 10 01	00 11 00	00 11 01	01 10 00	01 10 01	01 11 00	01 11 01
011	00 10 10	00 10 11	00 11 10	00 11 11	01 10 10	01 10 11	01 11 10	01 11 11
100	10 00 00	10 00 01	10 01 00	10 01 01	11 00 00	11 00 01	11 01 00	11 01 01
101	10 00 10	10 00 11	10 01 10	10 01 11	11 00 10	11 00 11	11 01 10	11 01 11
110	10 10 00	10 10 01	10 11 00	10 11 01	11 10 00	11 10 01	11 11 00	11 11 01
111	10 10 10	10 10 11	10 11 10	10 11 11	11 10 10	11 10 11	11 11 10	11 11 11

(a) (b)

Figure 1: Z-addresses

paper we propose the variable UB-Tree (VUB-Tree) as a further optimization for special data distributions. It is a generalization of the UB-Tree that takes the distribution of the data in each dimension explicitly into account, thus leading to a better performance for certain data distributions where otherwise the dimensions are not treated symmetrically for range restrictions.

2 Variable UB-Trees

2.1 The UB-Tree

The UB-Tree [Bay96] uses a space filling curve to create a partitioning of a multidimensional universe while preserving multidimensional clustering. Using the Lebesgue-curve (Z-curve, Figure 1a) it is a variant of the zkd-B-Tree [OM84].

To define the UB-Tree partitioning scheme we need the notion of Z-addresses and Z-intervals. We assume that each attribute value x_i of attribute A_i of a d-dimensional tuple $x = (x_1, \ldots, x_d)$ consists of s bits and we denote the binary representation of attribute value x_i by $x_{i,s-1}x_{i,s-2}\ldots x_{i,0}$.

A *Z-address* $\alpha = Z(x)$ is the ordinal number of a tuple x on the Z-curve and is calculated by interleaving the bits of the attribute values:

$$Z(x) = \sum_{j=0}^{s-1}\sum_{i=1}^{d} x_{i,j} \cdot 2^{j \cdot d + i - 1}$$

For an 8x8 universe, i.e., $s = 3$ and $d = 2$, Figure 1b shows the corresponding Z-addresses in binary represenation. The binary numbers should be read from left to right and top to bottom within a subsquare.

A *Z-region* $[\alpha : \beta]$ is the space covered by an interval on the Z-curve and is defined by two Z-addresses α and β. Figure 2b shows the Z-region [4:20] and Figure 2c shows a partitioning with five Z-regions [0:3], [4:20], [21:35], [36:47] and [48:63].

The UB-Tree utilizes a B^*-Tree to partition the multidimensional space into Z-regions, each of which is mapped onto one disk page. At insertion time a completely filled Z-region $[\alpha : \beta]$ is split into two Z-regions by introducing a new Z-address γ with $\alpha \leq \gamma < \beta$. γ is chosen such that the first half (in Z-order) of the tuples stored on Z-region $[\alpha : \beta]$ is distributed to $[\alpha : \gamma]$ and the second half is stored on $]\gamma : \beta]$. Thus a worst case storage utilization of 50% is guaranteed. There is some freedom of choice for the Z-region split. For optimal query performance the split algorithm for UB-Trees tries to maintain rectangular regions and minimize fringes whenever possible. Assuming a page capacity of 2 points Figure 2a shows ten points whose insertion into an empty UB-Tree created the partitioning of Figure 2c.

The UB-Tree requires logarithmic time (in the cardinality of a relation R) for the basic operations of insertion, point retrieval and deletion.

2.2 Address Transformation

The above mentioned partitioning works very well for uniformly distributed data. For other data distributions the techniques of UB-Trees may be en-

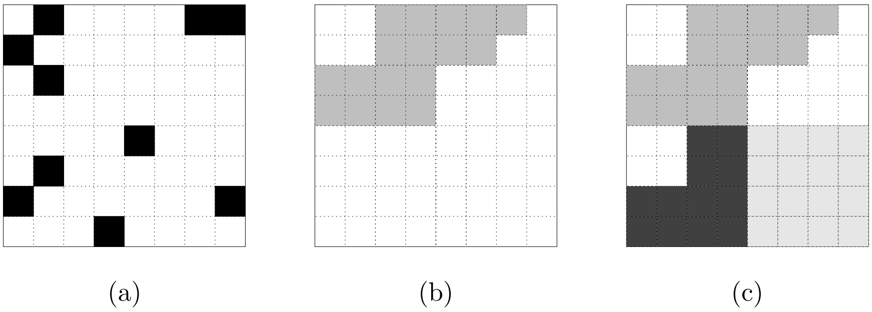


Figure 2: Z-regions

hanced to provide an even better performance. In a case where the actual domains of certain dimensions have a very narrow range of values (e.g. the day, month or year of a date is mapped to a 32-bit integer value), it can happen that pages are split mainly in those dimensions that have a broad range of values in their attributes [LS90]. Of course this also causes splits in the unevenly populated dimension but this is usually not sufficient. A query that is restricted in such an unevenly populated dimension touches a lot of regions because it does not restrict the query volume enough.

Example: Imagine three attributes x, y, z with the possible domain in the range of [0,7]. In bit representation they should have the following actual domains:
$W_x = \{000, 001, 010, 011, 100, 101, 110, 111\}$,
$W_y = \{000, 010, 011\}, W_z = \{000, 001\}$. The possible UB-addresses $x_1y_1z_1x_2y_2z_2x_3y_3z_3$ all have a structure like $x_1 00\ x_2 y_2 0\ x_3 y_3 z_3$. The consequence of this structure is that splits in those bits which are set to 0 for all possible addresses will not have any effect on the space partitioning but instead lead to a strongly layered partitioning.

Differences are especially visible if queries are answered which have the characteristics of hyperplanes. Hyperplanes are an extreme case of range queries which are restricted in only one attribute. If a query is restricted to an interval in an unevenly populated actual domain then the consequences are not very different to the extreme case. In the above example all regions are affected if a query is only restricted to the actual domain W_z. If the interval is restricted to only one value from W_z then still all of the pages have to be read if splits have not yet affected the last bit of W_z. The picture changes if queries are made which are restricted in the evenly populated domains. The number of retrieved regions is reduced a lot in such a case.

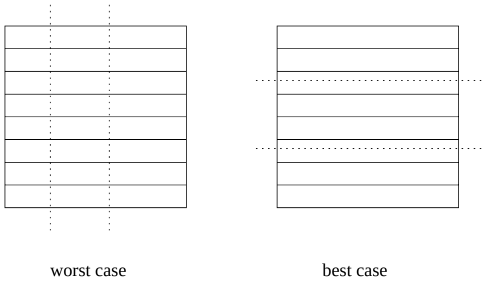


Figure 3: Cutting through a puff pastry

This effect is called *puff pastry effect* as the scenario can be informally seen as if a knife (the query) is cutting through a puff pastry (the UB-Tree) (Fig. 3).

Due to this it is necessary to transform the data for the address calculation. The actual data which is stored in the database has to remain the same of course.

2.3 Split Point Trees with Quantiles

For uniformly distributed data the partitioning of the universe is optimal if the universe is split recursively in the middle. This partitioning leads to split hierarchies which can be represented in a tree. This tree is called a *split point tree.* For arbitrary data distributions a partitioning which recursively divides the space with respect to the data distribu-

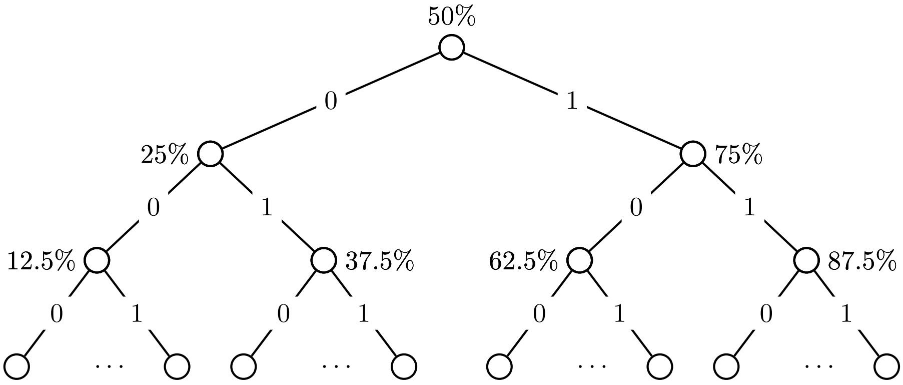


Figure 4: Universal split point tree

tion can lead to a much better partitioning. The consequence of this is that the classic partitioning is only a special case of a more general partitioning relative to the data distribution (for uniformly distributed data a split in the middle of the universe is the same as a split in the middle of the data distribution). Figure 4 shows the first levels of the general split point tree.

Due to the nature of the split hierarchies, the split point tree represents the data distribution of one underlying attribute. This makes it necessary to analyze the data distribution for every attribute to which a split point tree should be attached. It is fairly easy to calculate a split point tree if the data distribution and certain parameters (e.g., μ and σ for Gaussian distributed values (Fig. 5)) are known. The node values are the α-quantiles of the distribution, i.e., the values of the domain where the cumulated data distribution exceeds α-percent. As we want to split the data recursively in the middle of the data distribution we choose the α-quantiles with $\alpha = \frac{2k+1}{2^{treelevel}}$ (k denotes the nodenumber from left to right starting with 0 for every level) for the node values of the split point tree (Fig. 4). A similar transformation using quantiles is also used in combination with hashing techniques [KS87].

In cases where the data distribution is unknown it is necessary to do an intensive analysis of the data to calculate the desired α-quantiles. Algorithms for the calculation of α-quantiles have been published in the literature [JC85], [Raa87].

A split point tree which is attached to an attribute can be used after its creation to transform the values into a new representation which meets the characteristics of a uniform distribution. By annotating the edges of the split point trees with bits (0 for left edges, 1 for right edges) a transformation $f : D \rightarrow B$ (D denotes the domain values, B the resulting bitstrings) can be performed which transforms each value into a new bitstring. This is done by traversing the tree and deciding at every tree level if the attribute value is smaller or equal/larger than the node value. According to this, bits in the bitstring are set.

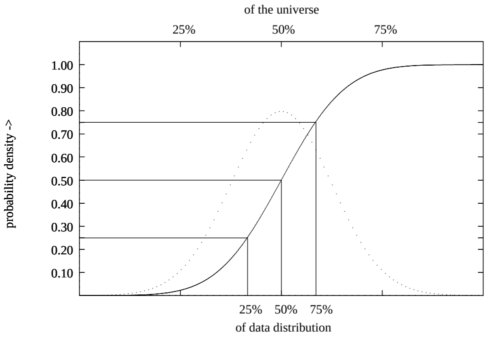


Figure 5: Determining α-quantiles

The transformation f (from now on referred to as **vub_trans**) is defined by the following algorithm (in pseudo code).

```
fct vub_trans = (tree t, attr a) bitstring r

 node = t[root];
 i = 0;
 while (i < height(t))
   if a >= node
     setBit(r, i);
     node = rightSon(node);
   else
```

```
    clearBit(r, i);
    node = leftSon(node);
  i = i + 1;

return r;
```

The created bitstring replaces the original attribute value in the calculation of the *Z-address* (of course the original value is stored in the database).

If the split point tree is not of sufficient height, the transformation creates the same bitstrings for different input values. This can cause duplicates for the address calculation and can be prevented by using either a tree of a sufficient height or by using an interpolation technique which calculates node values of a tree on the fly during the transformation by simply splitting the intervals on the last tree level in the middle. This technique though assumes that the values are uniformly distributed after the last tree level, which is not neccessarily the case.

To retransform a value from a bitstring, the tree has to be traversed with the bitstring as the input.

```
fct vub_reTrans = (tree t, bitstring r) attr a

 node =t[root];
 a = minval(a);
 i = 0;
 while(i < height(t))
   if bitSet(r, i)
     a = node;
     node = rightSon(node);
   else
     node = leftSon(node);
   i = i + 1;

 return a;
```

3 Performance of VUB-Trees

In [Mar99] a very accurate cost model for the UB-Tree has been developed. With this cost model it is possible to make assumptions about the approximate number of pages that will be retrieved by a query. This cost model can be adapted to variable UB-Trees. For the sake of completeness the cost model from [Mar99] is presented first.

3.1 Cost model for classic UB-Trees

A table with p pages has $l(p) = \log_2 p$ hierarchical split levels. The number of completed split levels $l_\downarrow(p)$ is therefore

$$l_\downarrow = l_\downarrow(p) = \lfloor l(p) \rfloor = \lfloor \log_2 p \rfloor$$

At split level i the idealized uniform partitioning divides the multidimensional space with respect to the i^{th} left bit of the UB-address. Therefore the number of splits with respect to one attribute j is at least $l_{j\downarrow} = l_\downarrow(p) \div d = \lfloor \frac{l_\downarrow}{d} \rfloor$. Another additional complete split exists if $l_\downarrow \bmod d > d - j$. Please note that in the current implementation and in this paper the leftmost bits of the UB-address are assigned to the rightmost dimensions. Also note that the numbering of dimensions starts with 1. With this consideration in mind the number l_j of completed splits in dimension j is:

$$l_j(d,p) = \begin{cases} l_{j\downarrow} + 1 & \text{, if } l_\downarrow \bmod d > d - j \\ l_{j\downarrow} & \text{, otherwise} \end{cases}$$

If $l_{k\downarrow} \neq l_k$ for some attribute k, there is another incomplete split level for attribute $j = d - (l_\downarrow \bmod d)$ at split level i. The complete split levels produce only $2^{l_\downarrow}$ pages, thus $p - 2^{l_\downarrow}$ pages are missing to reach the given number of pages, i.e., the table size. These regions are created from the $2^{l_\downarrow}$ pages by splitting these pages with respect to attribute j. Therefore the probability of an additional split in an attribute j is:

$$p_j(d,p) = \begin{cases} \frac{p-2^{l_\downarrow}}{2^{l_\downarrow}} = \frac{p}{2^{l_\downarrow}} - 1, \text{ if } j = d - (l_\downarrow \bmod d) \\ 0 \quad \text{, otherwise} \end{cases}$$

In the following we assume that the boundaries of the query interval in each dimension are normalized to the interval $[0,1]$. If we have s completed split levels in dimension j, then the number of slices of the multidimensional space that are overlapped by the query box $[a,b]$ in dimension j can be determined by counting the slices that are contained in the interval $[0,b_j]$ but not in the interval $[0,a_j]$. Since the slice containing a_j is also a slice overlapped by the query box, we have to add it to the above difference to get the correct number of slices. If the number of slices for the interval $[0,c]$ is calculated as $\lfloor c2^s \rfloor$ a non-existing slice $2^s + 1$ was added for $c_j = 1$ by the above formula. We must correct this error for the case $a_j < 1$ and $b_j = 1$. This is done by decrementing the number of slices by one. For $a_j = b_j = 1$ the subtraction removes the error by itself, therefore no correction is necessary here.

The number of slices $n(a_j, b_j, s)$ in dimension j overlapped by the query interval $[a_j, b_j]$ for the s completed splits in dimension j can be calculated by the following formula:

$$n(a_j, b_j, s) = \begin{cases} 2^s - \lfloor a_j 2^s \rfloor, \text{ if } b_j = 1 \wedge a_j \neq 1 \\ \lfloor b_j 2^s \rfloor - \lfloor a_j 2^s \rfloor + 1, \text{ otherwise} \end{cases}$$

If the probability of an incomplete split is taken into account, the number of slices overlapped by a query range in a certain dimension can be derived from the value for the completed splits. By subtraction we calculate how many slices would be overlapped additionally, if another completed split existed. For each of these splits the probability of its existence is $p_j(d, p)$. The average number of additional splits may then be calculated by a simple multiplication. We get the number of slices in dimension j overlapped by the range $[a_j, b_j]$ as follows:

$$\begin{aligned} n_j(d, p, a_j, b_j) \quad = \quad & n(a_j, b_j, l_j(d, p)) + \\ & + (n(a_j, b_j, l_j(d, p) + 1) - \\ & - n(a_j, b_j, l_j(d, p))) \cdot p_j(d, p) \end{aligned}$$

The key attributes are independent and the query box is iso-oriented with respect to each dimension. Therefore the number of pages u is calculated by multiplying the number of slices in each dimension:

$$u(d, p, a, b) = \prod_{j=1}^{d} n_j(d, p, a_j, b_j)$$

3.2 Cost model for variable UB-Trees

The assumption in 3.1 is that of an idealized uniform partitioning of the multidimensional space, where the attributes are uniformly distributed and independent. With variable UB-Trees this assumption holds because of the transformation with split point trees. Due to the way the address calculation is modified for variable UB-Trees also the cost model requires adaption.

A closer look at the cost model reveals that the boundaries of the query box can no longer be used in this form, because the underlying space partitioning is no longer the same. This would result in a query box which would touch regions where no tuples answering the query are located. It is therefore necessary to transform the boundaries of the query box in the same way as the other tuples that were inserted into the database. It is then possible to get a new function to calculate the costs by simply replacing the query boundaries a and b by a^t and b^t which denote the new transformed boundaries.

To calculate the new boundaries a^t and b^t of the query box one can make use of the bitstring into which a and b are transformed by `vub_trans()`. This function sets bits according to the position of the value relative to the data distribution. As every bit corresponds to exactly one node (which represents exactly one percentage) in the split point tree a^t can be calculated as

$$a^t = \frac{\texttt{vub\_trans(a)}}{2^{bitlength(\texttt{vub\_trans(a)})}}$$

The function *bitlength*() is equivalent to the length of the path from the root to the leaf in the split point tree.

This leads to a new function to calculate the number of slices for dimension j which are overlapped by the query interval for the completed splits in dimension j.

$$n(a_j^t, b_j^t, s) = \begin{cases} 2^s - \lfloor a_j^t 2^s \rfloor, \text{ if } b_j^t = 1 \wedge a_j^t \neq 1 \\ \lfloor b_j^t 2^s \rfloor - \lfloor a_j^t 2^s \rfloor + 1, \text{ otherwise} \end{cases}$$

Therefore:

$$\begin{aligned} n_j(d, p, a_j^t, b_j^t) \quad = \quad & n(a_j^t, b_j^t, l_j(d, p)) + \\ & + (n(a_j^t, b_j^t, l_j(d, p) + 1) - \\ & - n(a_j^t, b_j^t, l_j(d, p))) \cdot p_j(d, p) \end{aligned}$$

And:

$$u(d, p, a^t, b^t) = \prod_{j=1}^{d} n_j(d, p, a_j^t, b_j^t)$$

4 Performance Measurements

For the measurements the database `gaussuni6d500k` (containing 500000 tuples) was used. It consists of five independent Gaussian distributed ($\mu = 8000000$, $\sigma = 1000$) and one uniformly distributed index attribute. A 100 bytes binary character string was added to each tuple to increase the number of pages in the database.

Dimension	1	2	3	4	5	6
Split levels	1	1	1	1	1	11

Table 1: Split levels for `gaussuni6d500k`

In a three dimensional scenario (one uniformly and two Gaussian distributed attributes) the values would be accumulated inside a cylinder in a cube, where the cube represents the universe. All measurements were performed with the prototype implementation of the UB-Tree [MP] for the database system Transbase [TAS98]. The results are the same for Oracle and DB/2 since they show conceptional advantages of variable UB-Trees which are not database specific.

We start with a closer look at the classic UB-Tree and its behaviour for the above mentioned data distribution.

The co-domain of integer values in the prototype implementation of the UB-Tree is 24 bits. This leads to an extremely strong puff pastry effect because the values of the Gaussian distributed attributes are located around μ in a range of about 8σ (approx. 2^{13}). All these values have the same prefix 01111010001_2. The consequence is that the first 11 steps of the UB-address have identical bits for 5 dimensions for all addresses in the database and that splits will only affect the last 13 bits. Only the uniformly distributed attribute can add splits in the otherwise constant 11 steps. The *n-th step* in the above context is the part of the UB-address that is created by bit-interleaving the n-th bits from every dimension.

With the help of the cost model we can expect from $p = 50670$ $l(p) = \log_2 p = \log_2 50670 \approx 15.6$ hierarchical split levels. Since 11 splits have to occur in the uniformly distributed attribute only 4 to 5 splits are left over. These splits are divided among the other five attributes which results in at most one split in the other 5 dimensions (Table 1).

This leads to a strongly layered space partitioning. If the size of the database were increased by several orders of magnitude then also more splits in the Gaussian distributed attributes would occur and the effect would be lessend. The number of splits though is logarithmic in the number of pages in the database so this would lead to exponentially larger databases which get easily beyond reasonable sizes even in data warehouse environments.

The first split tries to partition the universe exactly in the middle (which is $2^{23} = 8388608$) in each dimension. This does not lead to a partitioning of the attributes in the Gaussian distributed dimensions, but only in the uniformly distributed attribute. Even the next splits in the Gaussian distributed attributes at 25% (= 4194304) and 75% (=12582912) do not lead to any improvement of the situation because the distribution is slightly shifted from the center of the dimension and has a very low σ. The same split in the uniformly distributed attribute again leads to a partitioning. Informally speaking the universe has to be truncated to the actual domain so that splits can occur which really partition the space.

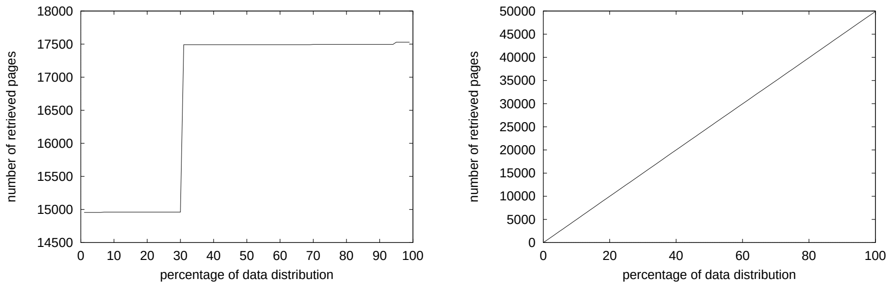
Figure 6 shows two $c\%$-measures for `gaussuni6d500k`. In an original $c\%$-measure $n-1$ dimensions are set to $c\%$ while the remaining dimension grows from 0% to 100% of the domain. The $c\%$-measures in this series of measurements were slightly adapted for the special requirements. The restrictions in the $n-1$ dimensions are no longer set to $c\%$ of the universe, but instead to $c\%$ of the data distribution. This is necessary because the data distribution is located in a very small area of the universe. The graphs in Figure 6 show the number of pages which had to be retrieved to answer the queries.

Measurement: For Figure 6a a *35%*-measure for `gaussuni6d500k` was performed. The first Gaussian distributed index attribute was variable and grew from 0% to 100% of the data distribution in 1% steps.

In Figure 6a a huge increase of the number of retrieved pages at about 30% of the data distribution is visible. This step shows a split in this dimension because the number of pages is constant before and after the change. This proves the assumptions made above that at most one split can occur in the Gaussian distributed attributes.

Measurement: In this measurement (Fig. 6b) the variable attribute was the uniformly distributed attribute. The other Gaussian distributed attributes were restricted to 35% of the data distribution.

The results of this measurement are completely different to the previous one. At first glance no



(a) (b)

Figure 6: *35%*-measure for `gaussuni6d500k` in the first Gaussian distributed attribute (a) and in the uniformly distributed attribute (b)

splits are visible at all in Figure 6b. The theoretical considerations showed $2^{11} (= 2048)$ splits in this dimension. With such a high number it is clear that no isolated splits occur. They degenerate into a straight line if the graph is not plotted with an extremely high resolution. A straight line as shown in Figure 6b is also characteristic for a compound B-Tree. The number of pages only depends on the selectivity of the uniformly distributed attribute (as in this attribute the most splits occur). The number of retrieved pages in this query can be calculated by the formula $s_6 \cdot p$ where s_6 denotes the selectivity of the uniformly distributed attribute. This means that the UB-Tree degenerates in its behaviour into a compound B-Tree with s_6 as the first attribute.

In the rest of this section we examine the improvements which are possible with a variable UB-Tree.

The database `gaussuni6d500kvar` was created with exactly the same data as `gaussuni6d500k`. The only difference was that split point trees of depth 10 were used to transform the Gaussian distributed attributes. With the help of the cost model it is again possible to do some calculations which then can be verified by measurements. The number of pages $p = 50540$ is at the same level as with classic UB-Trees. The number is slightly different though which shows that transformed tuples are stored in a different order than with classic UB-Trees.

Dimension	1	2	3	4	5	6
Split levels	2	2	2	3	3	3

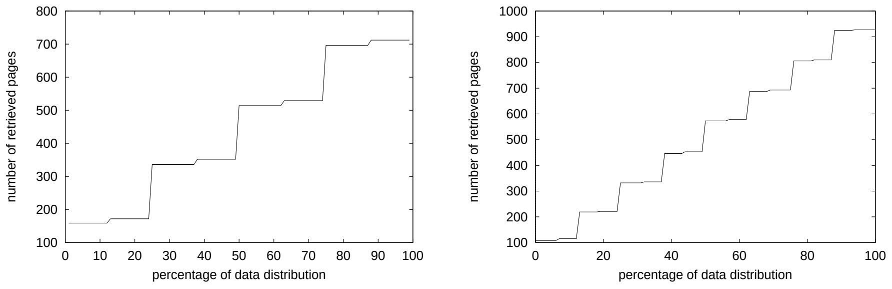
Table 2: Split levels for `gaussuni6d500kvar`

The number of pages leads again to $l(p) = \log_2 p = \log_2 50540 \approx 15.6$ expected hierarchical split levels. Since it is assumed that the transformed values come close to the uniform distribution the splits are approximately distributed in the same way over the 6 dimensions. Table 2 shows the expected numbers. At this point the order of the attributes is important again (3.1). The order of right to left is due to the fact that the address calculation in the prototype implementation starts with the rightmost dimension. In addition to Table 2 dimension 3 has an additional incomplete split.

Measurement: Figure 7a shows the results for a *35%*-measure for the database `gaussuni6d500kvar`. The first Gaussian distributed attribute grew from 0% to 100% of the data distribution. For this attribute (as well as for the other 4 Gaussian distributed attributes) a split point tree was available to transform the original attributes.

From the first column in Table 2 one can expect 2 split hierarchies in this attribute which divides the dimension into $2^2 = 4$ intervals. The graph in Figure 7a shows exactly these 4 distinctive splits.

Measurement: Figure 7b shows the results for a *35%*-measure for the database `gaussuni6d500kvar`. Again the uniformly



(a) (b)

Figure 7: *35%*-measure for `gaussuni6d500kvar` in the first Gaussian distributed attribute (a) and the uniformly distributed attribute (b)

> distributed attribute grew from 0% to 100% of the data distribution (which is identical to the co-domain in this case). This attribute was not transformed with a split point tree.

There are apparently more steps than in Figure 7a, but the number does not differ as much as for classic UB-Trees between Figure 6a and Figure 6b. Every step can be assigned to a split in each dimension. With Table 2 it is again possible to calculate the number of expected splits for these dimensions. This gives an expected value of 8 ($= 2^3$) splits for this dimension. Figure 7b shows exactly these splits.

In Figure 6a the number of retrieved pages starts at 14956 pages and grows to 17529, while the number of pages with the variable UB-Tree for the same measurement grows from only 159 pages to 712 pages. This means that with variable UB-Trees the number of pages is at the beginning smaller by a factor of 94(!) and at the end still by a factor of 24. Although the factor sharply decreases, the speedup for queries is tremendous. The elapsed time for answering the queries is directly linked to the number of retrieved pages and thus ad-hoc OLAP queries are sped up significantly.

5 Summary

The previous sections explained the theoretical background of variable UB-Trees and the necessity of their application in certain cases. The puff pastry effect of classic UB-Trees is avoided. The variable UB-Tree proved in the presented measurements that it is able to completely eliminate the puff pastry effect and to achieve the performance as it would be expected for classic UB-Trees with uniformly distributed data. The gain for ad-hoc OLAP queries is significant.

However, it is not possible to use variable UB-Trees for every data distribution. Especially important is that for data distributions which do not cause a puff pastry effect the number of retrieved pages can even be higher than for a classic UB-Tree although exactly the same data was spooled into the database. This restricts the usage of variable UB-Trees to certain data distributions. These cases have to be identified exactly. If the data distribution changes in an already existing database then the whole database has to be reorganized because the split point trees, which are required for variable UB-Trees, have to be chosen before any data is spooled into the database. This makes VUB-Trees static for one data distribution. For dynamic applications this is not always useful, but in typical DW applications this is not a problem as the databases are periodically reorganized in many cases (although this should be avoided in general). The complete data is then even available for an exhaustive analysis before it is reinserted into the database.

References

[Bay96] R. Bayer. The UB-Tree for Multidimensional Indexing. Technical report I9637, *Institut für Informatik, TU München*, November 1996.

[BM72] R. Bayer, E. McCreight. Organization and Maintainance of Large Ordered Indexes. *Acta Informatica 1*, 1972, pp. 173-189.

[BM98] R. Bayer, V. Markl. The UB-Tree: Performance of Multidimensional Range Queries. Technical report I9814, *Institut für Informatik, TU München*, 1998.

[CD97] S. Chaudhuri, U. Dayal. An Overview of Data Warehousing and OLAP Technologies. *ACM SIGMOD Record 26(1)*, March 1997.

[GG97] V. Gaede, O. Günther. Multidimensional Access Methods. *ACM Computing Surveys 30(2)*, 1997.

[JC85] R. Jain, I. Chlamtac. The P2 Algorithm for Dynamic Calculation of Quantiles and Histograms Without Storing Observations. *Comm. of ACM 28(10)*, 1985, pp. 1076-1085.

[KS87] H.-P. Kriegel, B. Seeger. Multidimensional Dynamic Quantile Hashing is Very Efficient for Non-Uniform Record Distributions. *ICDE*, 1987, pp. 10-17.

[LS90] D. Lomet, B. Salzberg. The hb-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance. *ACM TODS, 15(4)*, 1990, pp. 625-658.

[MB97] V. Markl, R. Bayer. The UB-Tree: A Cost Model for Multidimensional Queries in Relational Database Systems. Internal Report, *FORWISS München*, 1997.

[MB98] V. Markl, R. Bayer. The UB-Tree: A Multidimensional Index and its Performance on Relational Database Systems. *FORWISS München* 1998.

[MRB99] V. Markl, F. Ramsak, R. Bayer. Accelerating OLAP Queries by Multidimensional Hierarchical Clustering. *Proc. of IDEAS*, Montreal, Canada, 1999.

[MZB99] V. Markl, M. Zirkel, R. Bayer. Processing Operations with Restrictions in Relational Database Management Systems without External Sorting. *Proc. of ICDE*, Sydney, Australia, 1999.

[Mar99] V. Markl. MISTRAL-Processing Relational Queries using a Multidimensional Access Technique. Dissertation, *TU München*, 1999.

[MP] The homepage of the project MISTRAL containing everything about the UB-Tree and the prototype implementation.

`http://mistral.in.tum.de/`

[Raa87] K. Raatikainen. Simultaneous Estimation of Several Percentiles. *Simulation 49, 4*, Oct. 1987, pp. 159-164.

[OM84] J.A. Orenstein, T.H. Merret. A Class of Data Structures for Associate Searching. *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, Portland, Oregon 1984, pp. 194-305.

[Sam90] H. Samet. The Design and Analysis of Spatial Data Structures. *Addison Wesley*, 1990.

[TAS98] TransAction Software GmbH. TransBase Relational Database System Version 4.3, Manual. *Transaction Software GmbH*, München, Germany, 1998.