T U M

I N S T I T U T   F Ü R   I N F O R M A T I K

The UB-Tree: Performance of Multidimensional Range Queries

Rudolf Bayer, Volker Markl

TUM-I9814
Juni 98

T E C H N I S C H E   U N I V E R S I T Ä T   M Ü N C H E N

The UB-Tree: Performance of Multidimensional Range Queries [∇]

Rudolf Bayer
bayer@informatik.tu-muenchen.de
TU München
Orleansstr. 34
81667 München
Germany

Volker Markl
marklv@forwiss.tu-muenchen.de
FORWISS München
Orleansstr. 34
81667 München
Germany

Abstract

We investigate the usability and performance of the UB-Tree (universal B-Tree) for multidimensional data, as they arise in all relational databases and in particular in data-warehousing and data-mining applications. The UB-Tree is balanced and has all the guaranteed performance characteristics of B-Trees, i.e., it requires linear space for storage and logarithmic time for the basic operations of insertion, retrieval and deletion. Therefore it can efficiently support OLTP. In addition the UB-Tree preserves clustering of objects with respect to Cartesian distance. Therefore, it shows its main strengths for multidimensional data. It has very high potential for parallel processing. A single UB-Tree can replace a large number of secondary indexes and join indexes including foreign column join indexes (FCJ). For updates this means that only one UB-Tree must be managed instead of several secondary indexes. This reduces runtime and storage requirements substantially. For retrieval the UB-Tree has *multiplicative* complexity with respect to the relative size of the ranges for range queries, resulting in a dramatic performance improvement over multiple secondary indexes which have *additive* range query complexity. Furthermore, using the Tetris-Algorithm the UB-Tree enables reading data in any arbitrary sort order without the necessity of external sorting. Thus data need to be read only once to perform most of the operations of the relational algebra, such as ordering, grouping, aggregation, projection and joining. Therefore, the UB-Tree can support OLAP very efficiently. It is useful for geometric databases, data-warehousing and data-mining applications, but even more for databases in general, where multiple secondary indexes on one relation or FCJ-indexes to join several relations are widespread, which can all be replaced by a single UB-Tree index. Therefore, the difficult index selection problem [GHRU97] largely disappears and the UB-Tree offers the potential to integrate OLAP with OLTP in the same processing environment.

1 Introduction

In commercial relational DBMS a variety of indexing techniques are used today: classical B-Trees on one or several primary key attributes [BM72], secondary B-Trees, bitmaps [OQ97], Star Indexes [Red97] and FCJ indexes on foreign columns of pre-computed joins. [Inf97] is a good and up to date survey on these methods. In this paper we investigate the usability and the performance of the UB-Tree [Bay96, Bay97a] for complex applications (like datamining and OLAP) requiring complex multidimensional range queries on relational data. In such data we consider a tuple as a point in multidimensional space (in this paper we consider only point objects. See [Bay96] for treating extended objects). The combination of multidimensional range queries with more advanced operations (joins and aggregations) are presently being investigated, the results will be reported in a forthcoming paper.

We created large databases above 1 GB to compare the performance of the UB-Tree with those indexes that prevail in commercial databases, i.e. clustered B*-indexes over multiple attribute primary keys (compound index) and several secondary B*-indexes, which require non-clustered access to the data. [GHRU97] discusses the extremely difficult index selection

[∇] This work is carried out in the research and development project MISTRAL at FORWISS and is financed by SAP, Teijin and NEC. Additional information about the project can be found under URL http://mistral.informatik.tu-muenchen.de.

problem, which becomes exponentially simpler using UB-Trees instead of classical indexes. UB-Trees also facilitate range-max and range-sum queries as described in [HAMS97].

In order to obtain objective and comparable results we implemented UB-Trees as middleware on top of the SQL interface of a commercial DBMS (TransBase) and compared the UB-Tree against compound and secondary indexes (which are implemented in the kernel) of the same underlying DBMS. Presently we are porting to ORACLE and DB2. First measurements on ORACLE show that the performance results are qualitatively the same as those reported on TransBase in this paper. By using the middleware approach we loose some performance of the UB-Tree, which we estimate roughly as a factor around 2.

The present implementation of the UB-Tree uses Z-ordering in combination with an underlying B*-Tree or Prefix-B-Tree. It integrates the structure of the B*-Tree carefully with a relaxed Z-ordering (or any other space filling curve) tiling the data space down to the level where one tile - technically called **region** - corresponds precisely to one leaf of the UB-Tree. Concentrating on the region concept simplifies discovery (e.g. sorted reading from UB-Trees [Bay97b], the Tetris method [MB98] for joining, aggregation and grouping), explanation and understanding of algorithms considerably.

Splitting and merging leaves corresponds exactly to recursive splitting and merging of regions. Z-addresses of tuples need to be computed only to a precision which suffices to determine the proper region for a tuple. Also those regions (for storing data) are used directly to construct the minimal cover for the query box to guide the search for range queries. This guarantees that we have to retrieve from the disk exactly the minimal number of pages. A number of multi-dimensional data structures have been proposed in the past, e.g. Grid-Files [NHS84], R-Trees [Gut84], Z-ordering in combination with arbitrary search methods [OM84], hB-Trees [LS90], see [GG97] for a survey.

It is not our goal to compare the theoretical properties of the UB-Tree and those data structures, but to investigate the usability of the UB-Tree in combination with commercial database systems and its application to OLAP and OLTP.

The overall performance of the UB-Tree is very encouraging and in most cases far superior both to compound and to secondary indexes. In particular, a single UB-Tree can replace many secondary indexes. This reduces the number and storage requirements of indexes substantially and simplifies their management.

2 Concept of the UB-Tree

2.1 Addresses, Areas and Regions

We iteratively define an **area** A as a special subspace of a d-dimensional cube as follows: Split the cube with respect to every dimension in the middle, resulting in 2^d subcubes numbered in some arbitrary but fixed order (for our implementation and this paper we used Z-ordering) from 1 to 2^d. An area A_1 of level 1 consists of the first i_1 closed subcubes. i_1 determines A1 uniquely. We call i_1 the address of A_1 and write A_1 = $area(i_1)$. The empty area has the address ε.

To enlarge an area, we iteratively add an area with address $i_2 \in \{0,1,...,2^d-1\}$ of the next subcube with number i_1+1. The address of this enlarged area A_2 is $i_1.i_2$, which is lexicographically larger than the address i_1 of area A1. Next we may enlarge A_2 by adding an area of the brother subcube i_2+1 of i_2, etc. The left part of figure 1 shows four areas *area*(0.0.1), *area*(1.3.2), *area*(2.1) and *area*(3) of a two-dimensional universe. The shaded subcubes of the two-dimensional universe belong to the corresponding area.

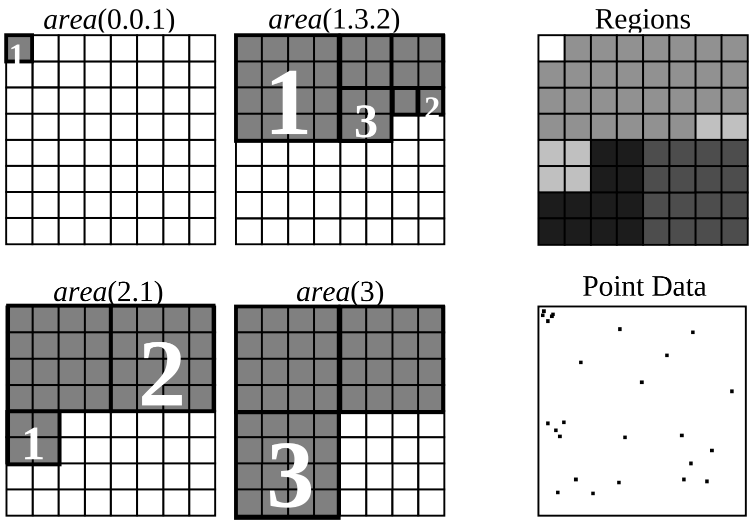


Figure 1: Areas and Regions

In the following we suppress trailing zeros of addresses and denote addresses by $\alpha, \beta, \gamma, \ldots$

We call i_j the j^{th} step of address $\alpha = i_1.i_2. \ldots .i_k$. We call k the length of the address α.

Note that the volume of a subcube decreases exponentially with its step number. We therefore obtain a fine partitioning of the multidimensional space with relatively short addresses.

Lemma: The lexicographic order of addresses (denoted by $\preccurlyeq$) and set containment of areas in space (denoted by $\subseteq$) are isomorphic: $area(\alpha) \subseteq area(\beta) \Leftrightarrow \alpha \preccurlyeq \beta$

Definition: A **region** is the difference between two areas: If $\alpha \prec \beta$ then we define the region between α and β as: $[\alpha : \beta] := area(\beta) \setminus area(\alpha)$, where "\" means "set difference". Note that regions are disjoint and therefore partition – or tile - the universe.

The areas in figure 1 are used to create five regions: [ε : 0.0.1], [0.0.1 : 1.3.2], [1.3.2 : 2.1], [2.1 : 3], [3 : 4]. Each region is shaded with a different gray.

Definition: A **page** is a fixed size byte container to store the objects or object identifiers in a region between two successive areas. We write *page*($[\alpha : \beta]$) for the page corresponding to the region$[\alpha : \beta]$. By *count*($[\alpha : \beta]$) we denote the number of objects located in $[\alpha : \beta]$.

Definition: A **tuple (or pixel)** is a smallest possible subcube at the limit of the resolution, but the resolution may be chosen as fine as desired. The **address of a tuple** is identical to the address of the area defined by including the tuple as the last and smallest subcube contained in this area. In the following we use the terms **attribute of a tuple**, **dimension** and **relation column** synonymously.

Lemma: A one-to-one map between Cartesian coordinates $(x_1, x_2, \ldots, x_d)$ of a d-dimensional tuple and its address α is implicitly defined by the above addressing scheme. We use the following notations for these maps:

$alpha\ (x_1, x_2, \ldots, x_d) = \alpha$ and $cart\ (\alpha) = (x_1, x_2, \ldots, x_d)$

Since the two maps are inverses of each other we get:

$cart(alpha(x_1, x_2, ..., x_d)) = (x_1, x_2, ..., x_d)$ and $alpha(cart(\alpha)) = \alpha$

If we have a set of areas we can order them according to their addresses. Since a region is the difference between two successive areas in this ordered set this also implies an order on the regions and therefore on the corresponding pages.

We assume that we have a universe *U* of values. For simplicity we assume that *U* has $v = 2^r$ values per dimension which are numbered $0,1,2,..., 2^r-1$. In this paper arbitrarily shaped spaces are simply considered as a subset of a suitable cube-shaped universe. It is also possible to drop this assumption and tailor the UB-Tree to the universe. This approach is described in [MB97b].

Since addresses are linearly ordered by $\prec$, they can be treated as the keys of any variant of a B-tree. New point- objects lie in a unique region. The identifiers of new objects are stored (inserted) into the page of their region.

Definition: A **UB-Tree** is any variant of a B-Tree, in which the keys are addresses of regions ordered by $\prec$. The leaf pages hold objects in regions or their object identifiers.

The five regions in figure 1 build a UB-Tree for the point data displayed in the lower right corner of figure 1. Although the regions differ in size (volume), each region stores about the same number of points because of the storage utilization guarantees of UB-Trees. Both the upper left corner and the lower right quarter of the universe contain five points, although the size (volume) of the region covering the lower right quarter of the universe is 16 times larger.

For performance comparisons we also need the notion of compound indexes and multiple secondary indexes: A **compound index** (also called *concatenated index* in the literature) is a primary B-Tree index built over all index dimensions. The index key of a compound index is the concatenation of the attributes in some order. By **multiple secondary indexes** (also called *inverted file* in the literature) we mean that a secondary B-Tree (with Bitmap or RID representation in the leaves) is built upon every dimension. The index key of the secondary index for dimension *i* is the i^{th} attribute of the relation.

2.2 Efficient Address Calculation via Bit-Interleaving

If each attribute x_i of a *d*-dimensional tuple $(x_1, x_2, ..., x_d)$ consists of 2^r values, it can be considered as a sequence of bits $x_{i,r} ... x_{i,1}$. Bit-interleaving (see also [OM84]) creates an *r*-dimensional tuple out of a *d*-dimensional tuple by re-arranging the bits of the tuple in the following way:

$interleave^{d,r}(x_{1,r} ... x_{1,1}, x_{2,r} ... x_{2,1}, ..., x_{d,r} ... x_{d,1}) = (x_{1,r}\, x_{2,r} ... x_{d,r}, x_{1,r-1} ... x_{d,r-1}, x_{1,1} ... x_{d,1})$

Thus $interleave^{3,4}(1110,1010,0111) = (110,101,111,001)$

If the result of $interleave^{d,r}$ is considered to be a binary number instead of an *r*-dimensional tuple, incrementing this number by *1* yields the UB-Address of a tuple. Thus:

$alpha(x_1, x_2, ..., x_d)= interleave^{d,r}(x_1, x_2, ..., x_d)+ 1$

Therefore $alpha(14,10,7) = alpha(1110, 1010, 0111) = interleave^{3,4}(1110,1010,0111) + 1 = (110,101,111,001) + 1 = (110,101,111,010) = 6.5.7.2$

The inverse function to $interleave^{d,r}$ can be computed in the same efficient way. We call this function $inv\text{-}interleave^{d,r}$ and define: $cart(\alpha) = inv\text{-}interleave^{d,r}(\alpha - 1)$

Only a slight modification of the interleave operation is necessary to support a universe where the domain of each dimension does not consist of the same number of bits *r*. In this case the number of bits is not identical for each step of an address. If d_i denotes the number of dimensions with a domain that is expressed by *i* or more than *i* bits, then step *i* of that address consists of d_i bits. The values d_i are identical for every point (and thus address) in one multi-dimensional universe. Using bit-interleaving for non-uniformly distributed data is also possible by a slight modification of the algorithm. For details see [MB97b].

The algorithm of bit-interleaving has the CPU-complexity of $O(d*r)$, where *r* denotes the length of each attribute in bits. The same holds for *inv-interleave*. Switching a tuple between Cartesian representation and address representation can therefore be performed very efficiently. Our current non-optimized implementation performs such a switching within 500µs for a 6-dimensional integer tuple on a SUN ULTRA SPARC Workstation with 167 MHz, where 2000 region addresses can be calculated in 1s of CPU time.

3 Update Operations

3.1 Insert Procedure

A point P to be inserted into the universe *U* is specified by its Cartesian coordinates $(x_1,x_2, ..., x_d)$ with address $\xi = alpha(x_1,x_2, ..., x_d)$. P belongs to the unique region $[\alpha : \gamma]$ satisfying $\alpha \prec \xi \preccurlyeq \gamma$. Note that ξ must be computed only to a precision which is sufficient to determine the proper region. P is inserted into the leaf-page corresponding to that region, which is found by a point query. Since pages can store only a maximum number *M* of Ids or objects, pages may overflow and are split like in B-trees. $[\alpha : \gamma]$ is split by introducing a new area with address β such that $\alpha \prec \beta \prec \gamma$. The region $[\alpha :\gamma]$ is partitioned by β into $[\alpha : \beta]$ and $[\beta : \gamma]$. The objects in *page*($[\alpha :\gamma]$) are distributed onto *page*($[\alpha : \beta]$) and *page*($[\beta :\gamma]$) accordingly. β is constructed by increasing $area(\alpha)$ as follows: Add to area α subcubes from $[\alpha :\gamma]$ in increasing order until the number of the objects in $[\alpha : \beta]$ is between $½ M - \varepsilon$ and $½ M + \varepsilon$. If the next subcube in this process contains too many objects, it is recursively subdivided until the condition can be met. The parameter ε is used to get shorter split addresses, which are favorable for the UB-Tree performance especially of the range query algorithm. Our measurements indicate that an ε of 5% is already very effective.

```
$\xi$ = alpha(P)
find [$\alpha$ :$\gamma$ ] in the UB-Tree, so that $\alpha \prec \xi \preccurlyeq \gamma$
retrieve page([$\alpha$ :$\gamma$ ])
insert P into page([$\alpha$ :$\gamma$ ])
if count([$\alpha$ :$\gamma$ ]) $\geq M$
  choose $\beta \in$ [$\alpha$ :$\gamma$ ], so that $\frac{1}{2}M - \varepsilon \leq$ count([$\alpha$ :$\beta$ ]) $\leq \frac{1}{2} M + \varepsilon$
  split page([$\alpha$ :$\gamma$ ]) into page([$\alpha$ :$\beta$ ]) and page([$\beta$ :$\gamma$ ])
```

Algorithm 1: Insertion Algorithm for Point P

Lemma: If a cube has a resolution of *pix* pixels in each dimension, then addresses have a length of at most $\lceil log_2(pix) \rceil$ steps. If we have a universe *U* with $pix = 2^r$ pixels in each of the d dimensions, the number of bits necessary to store the address is $r * d$.

Example: Taking a square bounding a map of Bavaria with a side of 512 km, then addresses of length 16 (=32 Bits) yield a resolution of 8 meters per pixel.

Our performance measurements indicate that the insert performance of a UB-Tree is similar to that of a compound B-Tree. The additional overhead for the UB-Tree address calculation is negligible. For a 6-dimensional integer tuple it uses less than 1% of the total insertion time. The UB-Tree insert is about $(h-1)/h * d$ times faster than multiple B-Trees, where h denotes the height of the UB-Tree. The factor $(h-1)/h$ in the above formula is due to the fact that for a given database the UB-Tree in the average is one level higher than each of the secondary indexes.

The deletion, merging and underflow methods of UB-Trees are similar to B-Trees [BM72]. For details see [Bay96].

4 Queries

4.1 Point-Queries

Point Queries are also called "exact match queries". They are specified by the Cartesian coordinates $(y_1, y_2, ..., y_d)$ of the point *P*. In OLAP these co-ordinates are usually called dimensions or dimension attributes. Usually additional information about *P* is of interest, e.g. temperature, height, time or monetary value. Such additional information (called measures in OLAP) may be stored as additional attributes with the point *P*. It might also simply be added to the index structure, thereby increasing the dimensionality of the space and allowing queries on these additional attributes. This problem is usually solved by constructing a new secondary index; with UB-trees it can be handled by increasing the dimensionality of the searchable object space.

To find *P* we compute its address $\xi := alpha(y_1, y_2, ..., y_d)$ with sufficient precision to find the unique region [α :β] with the property $\alpha \prec \xi \preccurlyeq \beta$ and fetch *page*([α :β]).This is achieved by searching the UB-tree, using address ξ as the search key. *page*([α :β]) must contain point *P* with the additional information or the identifier *Id(P)* which is used as a reference to *P*.

P can be found in $O(log_k N)$ time, where *N* is the number of objects in our universe *U* and $k = \frac{1}{2}M$, since UB-trees are balanced and searched exactly like the variant of B-tree used as the underlying data structure for the UB-tree. Thus the point query performance of a UB-Tree is similar to that of a compound B-Tree index. The additional address calculation overhead is negligible. With multiple secondary indexes finding a point may be more expensive, since several or even all indexes need to be queried to find *Id(P)* for non-unique attributes.

4.2 The Range Query Algorithm

Range queries are a fundamental problem for all database systems. A range query is specified by an interval for each dimension. No specification for a dimension formally means the interval (-∞, +∞). The query is the Cartesian product of the intervals for all dimensions, called the **query box** *Q* with the lower and upper bounds *ql* and *qh*. The answer to *Q* is the set of point-objects in *Q*. In the following we will call this set of objects **result set of Q**.

To answer a range query, only those regions, which properly intersect the query box, must be fetched from the database and thus from the disk. Initially the range query algorithm calculates and retrieves the first region that is overlapped by the query-box. Then the next intersecting region is calculated and retrieved. This is repeated until a minimal cover for the query box has been constructed, i.e., the region that contains the ending point of the query box has been retrieved.

```
$\xi$ = alpha(ql); $\omega$ = alpha(qh)
repeat
  find [$\alpha$ :$\beta$ ] in the UB-Tree, so that $\alpha \prec \xi \preccurlyeq \beta$
  output all points x from [$\alpha$ :$\beta$ ] where x $\in$  [[ql, qh]]
  $\xi$ = address of the first point intersecting the querybox
  with $\xi \succ \beta$
until $\xi \succ \omega$
```

Algorithm 2: Range Query Algorithm for a Query Box [[ql, qh]]

The algorithm for retrieving the next intersecting region merely requires one B-Tree search and *O(d*r)* CPU operations. Therefore only *n* disk accesses to data pages need to be performed to retrieve the data within a query box overlapped by *n* regions. [Bay96] gives an algorithm exponential in the number of dimensions for calculating the address of the next intersecting region. We have developed a linear version of this algorithm that is solely based on UB-addresses and does not require any transformation into Cartesian co-ordinates. With this algorithm the calculation of the next intersecting region takes 26 µs for 6 dimensions and 76 µs for 31 dimensions on a SUN ULTRA SPARC with 167 MHz, where 10000 to 40000 intersecting regions can be calculated in one second.

The number of regions intersecting a query box *Q* is related to the **selectivity of *Q***, i.e. to the number *q* of objects properly intersecting *Q* compared to the number of objects in the universe. If *Q* covers a highly populated part of the universe, then it contains a large number of objects and many regions are needed to store those objects. On the other hand, if *Q* covers a sparsely populated part of *U*, then only few regions are needed to store all the objects in *Q*. Therefore, the number of regions fetched from the disk is closely related to the number of objects in *Q*. Except for cases, where the query box degenerates to a hyperplane of the universe, the number of regions, which we must fetch from the database, is for sufficiently large databases proportional to the volume of *Q* and therefore proportional to the size of the answer to the range query.

Figure 2 shows the retrieved regions of two query boxes in the same UB-Tree. Here the data is distributed non-uniformly. The query box 2a has a result set of 617 points and overlaps 27 regions. Although query box 2b has the same volume as query box 2a, it only covers a sparsely populated part of the universe and thus only 78 points in 3 regions are retrieved by the range query.

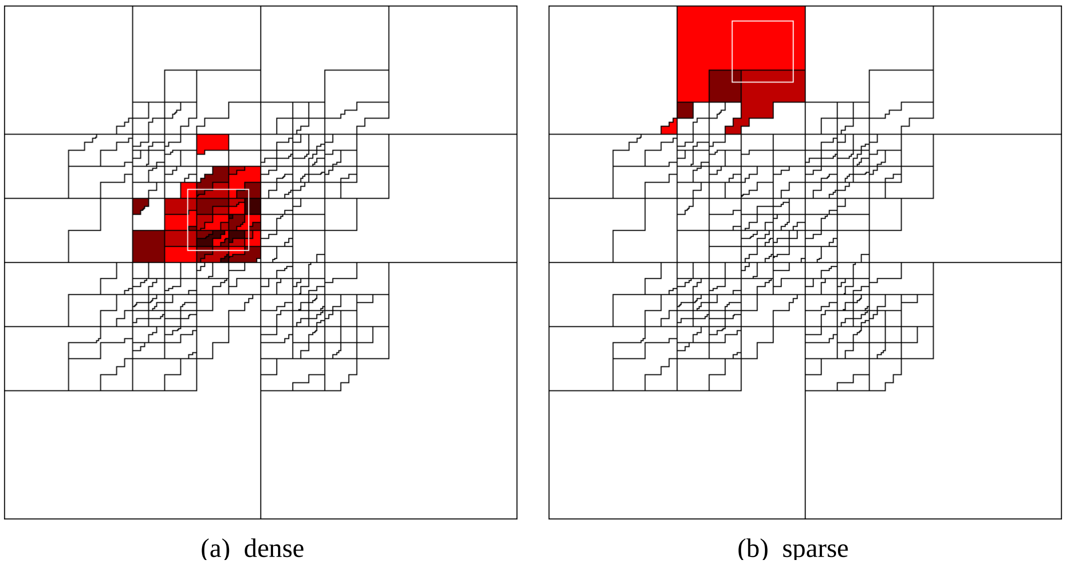


Figure 2: Query Boxes in Sparse and Highly Populated Parts of the Universe

Answering a range query over a database, which is organized as a UB-tree, requires time proportional to the size of the answer to the query. This is a rather surprising result and atypical for query processing in databases, where processing time is often related to the complexity of a query and the size of the database, but not to the size of the answer.

4.3 Performance of UB-Tree Range Queries

Currently the most widely used technique in commercial relational DBMS to handle multidimensional data is the use of a secondary index for each dimension. Compared to that, the UB-Tree has the following advantages:

- Only one single index structure has to be managed and updated upon insertion and deletion of objects in contrast to a total of d indexes.
- Opposed to the *additive* behavior of multiple secondary indexes, the UB-Tree has *multiplicative* behavior: Assume that the data universe contains N objects and $p_i\%$ of the values lie in the query interval of Q with respect to dimension i. Then a total of $N * p_i\%$ of the data must be fetched via the secondary index for dimension i. This adds up to fetching from the disk $\Sigma^d_{i=1} N * p_i\%$ of the data or at least object identifiers and computing intersections between these sets. With a UB-tree the amount of data to be fetched is proportional to the size of the query box Q, i.e., $N * \Pi^d_{i=1} p_i\%$. A precise analysis of the number of pages retrieved by a range query is given in [MB97a].

Thus the performance of multiple secondary indexes deteriorates with the number of dimensions, whereas the performance of the UB-tree improves with the number of dimensions.

Alternatively, a single compound B-Tree index may be used for answering a range query. A compound B-Tree can only use the restriction in the first dimension in order to reduce the number of pages that need to be retrieved. Thus $N * p_1\%$ of the data are retrieved. Since the compound index is a primary index, the data can be retrieved by reading large clusters. This may result in an advantage over the random access of multiple secondary indexes if the result set is large. In contrast to both UB-Trees and multiple secondary indexes, compound indexes do not show **symmetrical behavior** with respect to the relative size of the restricted dimensions, since the first dimension is extremely favored.

Both multiple secondary indexes and compound primary indexes differ very much from an ideal index that would retrieve only the pages contributing to the result set without any overhead. The UB-Tree gets very close to an ideal index since it retrieves only the pages for regions intersecting the query box.

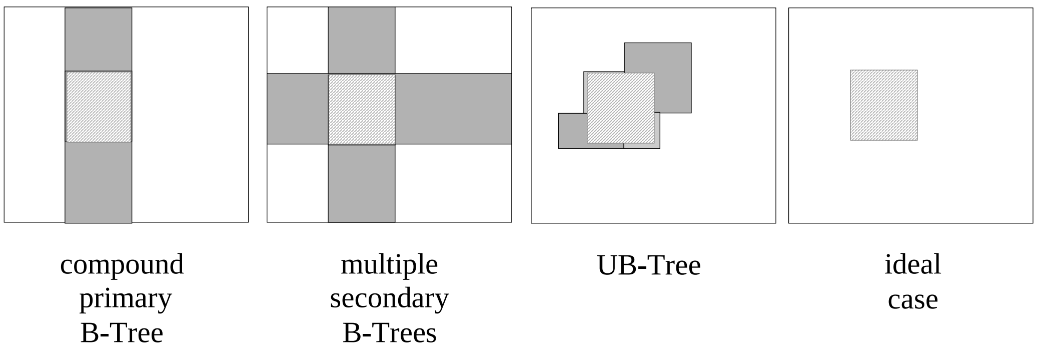


Figure 3: Theoretical Range Query Behavior for the Striped Query Box

This is illustrated in figure 3, where the retrieved part of the universe is shaded.

Figure 4 and 5 show two performance measurements of multidimensional range queries against a 6-dimensional test database consisting of 10 million tuples and about 250 000 pages (= regions). The tests were performed using the commercial DBMS TransBase on a 167 MHz SUN ULTRA SPARC 2 on a hard disk with an average positioning time of 8ms. In order to get comparable results, caching was eliminated. If caching were allowed between queries, the UB-Tree would even gain a performance advantage.

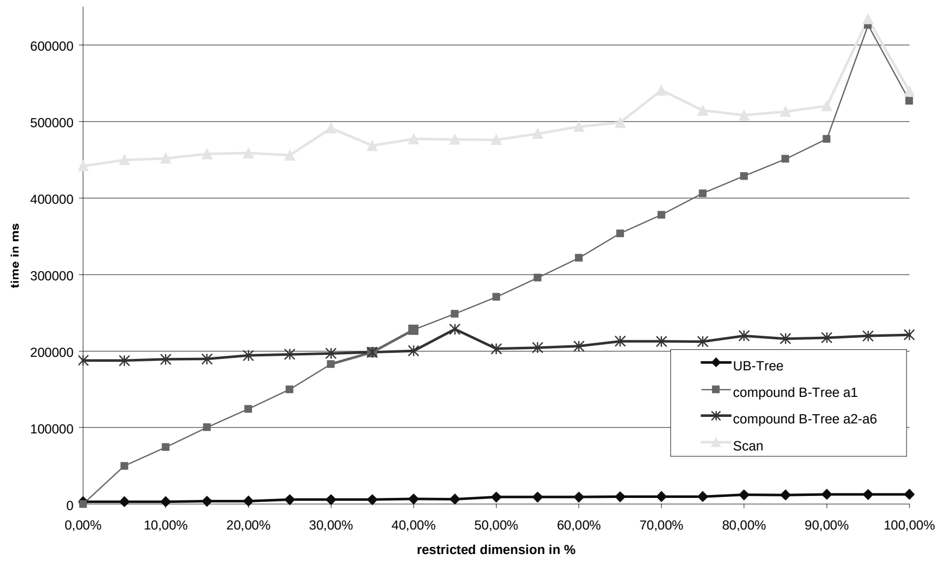


Figure 4: Linearly Growing Query Box Volume with a 35% Restriction in 5 Dimensions

Figure 4 illustrates a measurement series for a query box with a selectivity of 35% in each of five dimensions. The sixth dimension is varied from 1% to 100%. This causes a linearly growing result set.

Since the compound primary B-Tree was built on the concatenation of the attributes a_1,a_2,a_3,a_4,a_5,a_6 in this order, only varying a_1 and a_2 show different performance results. The results obtained if a_3, a_4, a_5 or a_6 were varied would be identical to those for a_2. The compound B-Tree with varying a_1 retrieves 1% to 100% of the database, while the compound B-Tree with varying a_2 constantly retrieves 35% of the database (corresponding to the 35% restriction of a_1), which consumes 35% of the time of the relation scan. The slight linear time increase of the compound index in the latter case stems from result size increases with growing a_2. Because of that the necessary result set processing takes more CPU time, the same effect is seen as a linear increase in the scan time.

The UB-Tree takes advantage of the restriction in every dimension and of the multi-dimensional clustering of the index itself and of the data. Therefore it increases linearly on a much smaller scale than a compound B-Tree.

Multiple B-Trees can not take advantage of any clustering and need to perform an expensive intersection operation. Finally, the tuples of the result set must be fetched randomly from the disk. Because of the size of the database, pages that have been retrieved once can not remain in cache. Thus the disk pages have to be accessed several times, which results in a time behavior that is more than ten times higher than that of the relation scan. For this reason multiple secondary indexes are not included in figure 4. Looking at our figures, multiple secondary B-Trees seem to be of little or even no use for multi-dimensional range queries. Table 1 gives the exact times for a restriction of 20% and 40% in one attribute, while the selectivity of all other attributes is restricted to 35%.

Restriction	**UB-Tree**	**Compound B-Tree a_1**	**Compound B-Tree a_2**	**Multiple B-Trees**	**Relation Scan**
20%	3,9 s	124,2 s	194,2 s	1890,3 s	458,9 s
40%	6,6 s	228,1 s	200,4 s	2120,9 s	477,3 s

Table 1: Linearly Growing Query Box Volume with a 35% restriction in 5 Dimensions

Figure 5 shows a measurement series where the selectivity of the query box is the same for every dimension. This selectivity is varied for every dimension from 1% to 100%. The result set of this query grows polynomially with the 6th power. All indexes show the expected polynomial behavior. The polynomial behavior caused by the increasing I/O times is amplified by the polynomially growing result set size, since the tuples of the result set also need to be transferred to the application program. This is also the reason for the polynomial behavior of the relation scan. The multiple secondary indexes grow at a very high rate, resulting in the worst performance of all indexes compared. They are already worse than the relation scan at a 5% restriction in every dimension. The compound index increases at a much smaller rate. However, it is worse than the UB-Tree. It only uses the restriction on the first dimension, while the UB-Tree is able to use the restrictions in every dimension. The compound index overtakes the UB-Tree at a point where the relation scan is already preferable to both of the indexes. Table 2 gives the exact numbers for a restriction of 20% and 40% in all attributes.

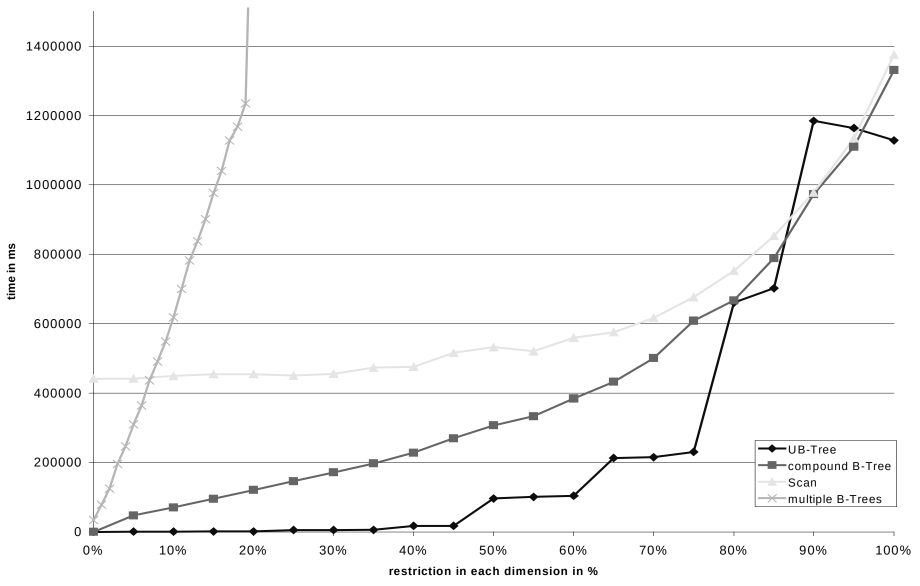
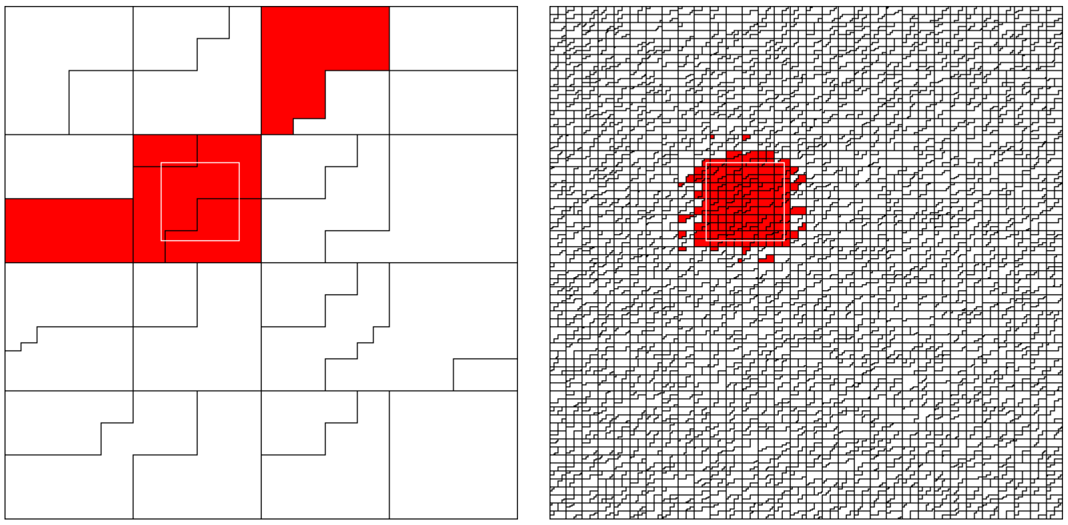


Figure 5: Polynomially Growing Query Box Volume

Restriction in each dimension	UB-Tree	Compound B-Tree	Multiple B-Trees	Relation Scan
20%	0,9 s	120,7 s	1235,1 s	449,8 s
40%	17,5 s	228,2 s	2753,3 s	475,9 s

Table 2: Polynomially Growing Query Box Volume

Our measurements indicate that the range query performance of UB-Trees is more symmetrical than that of a compound B-Tree. It also shows a better absolute performance than multiple B-Trees when a sufficient number of attributes is specified. In our 6-dimensional test database this is already true for 2 or 3 dimensions. The UB-Tree range query performance is on the average several orders of magnitude faster than compound B-Trees and multiple secondary B-Trees. We measured an increase in speed of several thousands compared to secondary indexes and – depending on the restriction – between two and one-hundred compared to a compound index. Performing an index scan over the whole relation with a UB-Tree results in a performance similar to a scan over a clustered primary compound B-Tree. Our study shows that the relative performance of the UB-Tree increases with growing database sizes and thus results in a good scalability. This is illustrated in figure 6, where the regions that are retrieved for the same range query are shaded for a UB-Tree with 1000 tuples (=25 regions) and 50000 tuples (=2500 regions). The figure shows that the query box is approximated more closely by the region partitioning as the database increases.



(a) 1000 tuples (b) 50 000 tuples

Figure 6: Range Queries and Growing Database Sizes

5 Impacts on the relational Algebra – the Tetris Algorithm

Tables organized by a UB-Tree can be read in any sort order in *O(n)* disk accesses where *n* is the number of pages of the table or the minimal number of regions covering a query box [Bay97a]. This is made possible by a modification of the range query algorithm and a caching technique, the so called ”Tetris-Algorithm” [MB98]. This algorithm performs a sweep over a query box of the UB-Tree with respect to the lexicographic order of the specified sorting dimensions (in the spirit of the well known sweep line algorithms [PS85]). The Tetris-Algorithm works similar to the range-query algorithm. The only difference is that the calculation of the next intersecting region does not return the next region according to Z-ordering, but according to the specified sort order: Initially the algorithm calculates the first region that is overlapped by the query-box, retrieves it and caches it in main memory. Then it continues to read and cache the next regions with respect to the sort order, until a complete thinnest possible slice of the query box has been read. Then the cached tuples of this slice are sorted in main memory, returned in sort order to the caller and removed from cache. The algorithm proceeds reading the next slice, until all regions which intersect the query box have been retrieved and output.

Figure 7 shows two vertical slices during a sorted reading in the horizontal dimension. For simplicity, the query box in the figure is assumed to be the whole universe. The cached regions are shaded, the slices are emphasized by white borders.

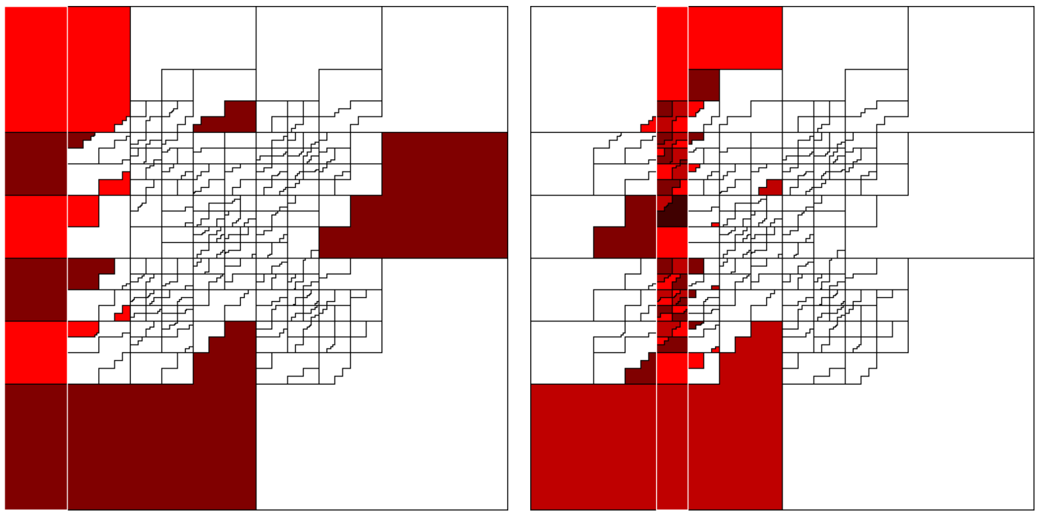


Figure 7: Two slices of a UB-Tree during sorted reading with the Tetris-Algorithm

Therefore only *n* disk accesses to data pages need to be performed to sort a query box overlapped by *n* regions according to any of the *factorial*(*d*) sort orders definable over *d* attributes. Thus each page only needs to be accessed once in order to produce a sorted output in any combination of dimensions.

Generally speaking, the Tetris-Algorithm allows joining, grouping, aggregation, projection and any other operation where sorted reading a relation (or parts of it) is involved in *O(n)* disk accesses. An additional selection may be used to reduce the necessary disk accesses, if the restricted attributes are also part of the UB-Tree. A further advantage of the Tetris-Algorithm is that it needs no disk space to perform the operation. Only a main memory cache is required which in general is quite small compared to the main memory cache required for a good performance of the standard merge sort algorithm. For details see [Bay97b] and [MB98].

We are currently doing performance measurements with the Tetris-Algorithm which will be reported in a forthcoming paper.

6 Conclusions and Future Work

We have shown the usability of the UB-Tree for insertion, point-queries and range-queries. The behavior of a UB-Tree primary index is superior to the classical indexing methods used in present DBMS for both OLAP and OLTP applications. Additional secondary B-Trees are useful to further speed up “hyperplane queries”, i.e., queries, where only one attribute is restricted. In a data warehouse one UB-Tree may be used to replace several traditional star indexes ([Inf97] and [Red97]), since it shows the desired symmetrical behavior for multidimensional range queries. The response time of the UB-Tree for multidimensional range queries does not depend on any order of the restricted dimensions, but only on the number of dimensions which have been restricted. Instead of organizing the foreign keys of the fact table in a star schema as *factorial(d)* compound indexes or bitmap indexes, a single UB-Tree algorithm may be used to efficiently perform star joins using the Tetris-Algorithm.

We are currently doing performance measurements on ORACLE and are porting to DB2. We further investigate data modeling with the presence of multidimensional indexes in order to find out, when and how to use UB-Trees to support a database schema with a certain query profile.

We are also implementing the Tetris-Algorithm and will use it to efficiently support the operations of the relational algebra such as selection, sorting, grouping with aggregation and projecton. With that implementation we will investigate practical data warehousing scenarios of our project partners.

Future work also includes the development of a cost based query-optimization technique based on an already developed cost model for UB-Trees [MB97a] and the investigation of variable UB-Trees to improve the support for attributes with non-uniform data distributions [MB97b]. We will also investigate the application of our technique to high dimensional data spaces as they occur for instance in the field of image processing, where images are described by a list of features that may be considered to be a point in a high dimensional space.

References

[Bay96] Bayer, R.: The universal B-Tree for multidimensional Indexing. Technical Report TUM-I9637, Institut für Informatik, TU München, 1996

[Bay97a] Bayer, R.: The universal B-Tree for multidimensional Indexing: General Concepts. - In: World-Wide Computing and Its Applications '97 (WWCA '97), Tsukuba, Japan, 10-11, Lecture Notes on Computer Science, Springer Verlag, March, 1997.

[Bay97b] Bayer, R.: UB-Trees and UB-Cache – A new Procesing Paradigm for Database Systems. Technical Report TUM-I9722, Institut für Informatik, TU München, 1997

[BM72] Bayer, R.; McCreight, E.: Organization and Maintainance of large ordered Indexes. Acta Informatica 1, 1972, pp. 173 - 189

[Gut84] Guttman: A dynamic Index Structure for spatial Searching. ACM-SIGMOD Intl. Conference on Management of Data, 1984, pp. 47-57

[GG97] Gaede, V.; Günther, O.: Multidimensional Access Methods, Humboldt Universität, Berlin, 1997, http://www.wiwi.hu-berlin.de/~gaede/survey.rev.ps.Z

[GHRU97] Gupta, H.; Harinarayan, V.; Rajaraman, A.; Ullman, D. J.: Index Selection for OLAP. Intl. Conf. on Data Engineering, 1997

[HAMS97] Ho, C.T.; Agrawal, R.; Megiddo, N.; Srikant, R.: Range Queries in OLAP Data Cubes. ACM SIGMOD Intl. Conf. On Management of Data, Tucson, 1997, Arizona, pp. 73-88

[Inf97] Informix Software Inc.: A New Generation of Decision Support Indexing for Enterprisewide Data Warehouses, http://www.informix.com/informix/corpinfo/zines/whitpprs/wpxps.pdf, 1997

[LS90] Lomet; Salzberg: The hB-Tree: A Multiattribute Indexing Method with good guaranteed Performance. ACM TODS, 15(4), 1990, pp. 625 – 658

[MB97a] Markl, V.; Bayer, R.: A Cost Model for multidimensional Queries in Relational Database Systems, Internal Report, FORWISS München, 1997

[MB97b] Markl, V.; Bayer, R.: Variable UB-Trees for the efficient Indexing of arbitrarily distributed multi-dimensional Data, Internal Report, FORWISS München, 1997

[MB98] Markl, V.; Bayer, R.: The Tetris-Algorithm for Sorted Reading from UB-Trees, In: "Grundlagen von Datenbanken", 10th GI Workshop, Konstanz, 1998

[NHS84] Nievergelt, J.; Hinterberger, H. ; Sevcik, K. C.: The Grid-File. ACM TODS, 9(1), March 1984, pp. 38-71

[OQ97] O´Neill, P.; Quass, D.: Improved Query Performance with Variant Indexes, ACM SIGMOD Intl. Conf. On Management of Data, Tucson, 1997, Arizona, pp. 38-49

[OM84] Orenstein, J. A.; Merret, T. H.: A Class of Data Structures for Associate Searching. Proc. ACM SIGMOD Int. Conf. On Management of Data, Portland, Oregon, 1984, pp. 294-305

[PS85] Preparata, F.P.; Shamos, M. I.: Computational Geometry: An Introduction. Springer-Verlag, New York, 1985

[Red97] Redbrick Systems: Star Schema processing for Complex Queries. http://www.redbrick.com/rbs-g/whitepapers/starJoin.pdf, 1997