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 Universal B-Tree for multidimensional Indexing

Rudolf Bayer

TUM-I9637
November 96

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

TUM-INFO-11-96-I9637-350/1.-FI

Druck: Fakultät für Mathematik und
Institut für Informatik der
Technischen Universität München

The Universal B-Tree for multidimensional Indexing[1]

Rudolf Bayer
Institut für Informatik
Lehrstuhl Datenbanksysteme, Wissensbasen
Technische Universität München
bayer@informatik.tu-muenchen.de

September, 1996

[1]Patent pending: Deutsches Patentamt Nr. 196 35 429.3

Abstract

Today almost all database systems use B-trees as their main access method. One of the main drawbacks of the classical B-tree is, however, that it works well only for one-dimensional data.

In this paper we present a new access structure, called UB-tree (for universal B-tree) for multidimensional data. 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 INSERT FIND DELETE. In addition the UB-tree has the fundamental property, that it preserves clustering of objects w.r. to Cartesian distance. Therefore, the UB-tree shows its main strengths for multidimensional data. It has very high potential for parallel processing. With the new method, a single UB-tree can replace an arbitrary number of secondary indexes. 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 queries and in particular range queries the UB-tree has *multiplicative complexity* instead of the *additive complexity* of multiple secondary indexes. This results in dramatic performance improvements over secondary indexes.

The UB-tree is obviously useful for geometric databases, datawarehousing and datamining applications, but even more for databases in general, where multiple secondary indexes are widespread, which can all be replaced by a single UB-tree index.

1 Introduction

In this paper the UB-tree (universal B-tree) access structure is described, a method to organize the objects populating an n-dimensional space (called the "universe") in such a way, that they can be stored on, managed on, retrieved from and deleted from peripheral storage very efficiently.

Methods known so far to handle this problem [1], [2], [3] have deficiencies either w. r. to performance guarantees or w.r. to the handling of dynamic datasets.

So far methods with good performance guarantees were only known for 1-dimensional (or linear) spaces. The best known and the most widely used methods - particularly for database systems - are several variants of the B- Tree coinvented by the author.

Performance guarantees for UB-trees are given for all basic operations required to manage diskstorage for multidimensional data. Also guarantees can be given for the utilization of space and the regaining of unused space on disk storage.

Since the performance guarantees for processing time are logarithmic in the number of objects in the dataspace, the method is particularly suitable and robust for very large applications. It has excellent performance over an extremly large range of dataspace sizes, in other words, it scales very well to very large problems as they arise e.g. in applications of geographical databases as well as datamining and datawarehousing applications. A further important advantage of the new method is, that it can be implemented on top of any database system or even on top of just an index-structure, like e.g. a classical B-tree implementation, by a preprocessing technique. The basic property making this possible is the fact, that multidimensional space is mapped to a linear ordered space in such a way, that the multidimensional clustering is preserved by this mapping and is reflected in the one-dimensional clustering of the pages of disk-storage-architecture. Therefore it is easy and cost effective to extend existing database systems to take advantage of the UB-tree data organization and access method.

2 Basic Concepts

We define the formal concept of an **Area** as follows:

Def. 2.1 Partition a cube C of dimension n into 2^n subcubes numbered

$sc(i)$ for $i = 1, 2, \ldots 2^n$

by partitioning C w.r. to each dimension in the middle. Then recursively define:

$Area^C(k) := \cup_{i=1}^{k} sc(i)$ for $k = 0, 1, \ldots 2^n - 1$
$Area^C(k.j) := Area^C(k) \cup Area^{sc(k+1)}(j)$
where $Area^{sc(k+1)}(j)$ is a nonempty Area of $sc(k+1)$, i.e. $j \ \geq \ 1$

Def. 2.2: An **address** α is a sequence
$i_1.i_2 \ldots i_l$ where
$i_j \,\epsilon\, 0, 1, \ldots 2^n - 1$ for $j \,\epsilon\, 0, 1, 2, \ldots l - 1$
$i_l \,\epsilon\, 1, 2, \ldots 2^n - 1$

Lemma 2.1: Addresses are ordered lexicographically, we denote this order by $<\cdot$, $\leq\cdot$, $\doteq, \ldots$
Areas are ordered by set inclusion denoted by $\subset, \subseteq, \ldots$
Let α, β be addresses, then
$Area(\alpha) \subset Area(\beta) \Leftrightarrow \alpha <\cdot\ \beta$
$Area(\alpha) = Area(\beta) \Leftrightarrow \alpha = \beta$

Lemma 2.2: The orders on areas and addresses are isomorphic with the following correspondences

$<\cdot$ corresponds to $\subset$

$\leq\cdot$ $\quad\quad\quad\quad\subseteq$

$\doteq$ $\quad\quad\quad\quad =$

Following are four examples of Areas and their addresses;

In the following figure the shaded Area A has address 0.3, we write $alpha(A) = 0.3$.

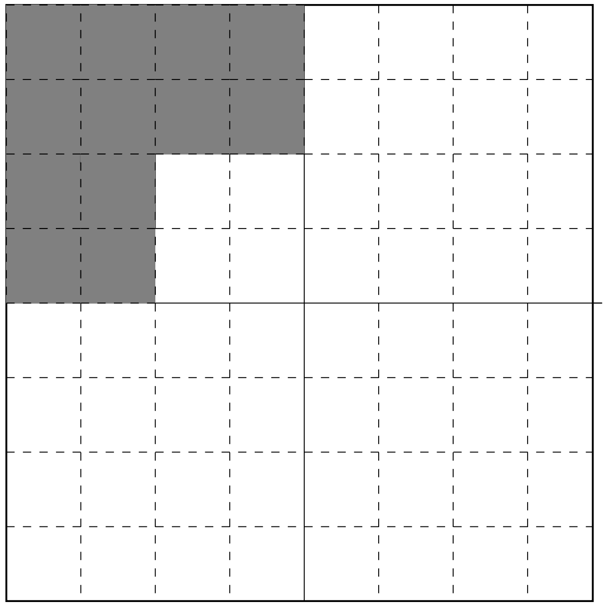

Figure 1:

From now on we omit the period in the notation for addresses.
In the figure 2 the shaded Area B has address 132 and we write $alpha(B) = 132$

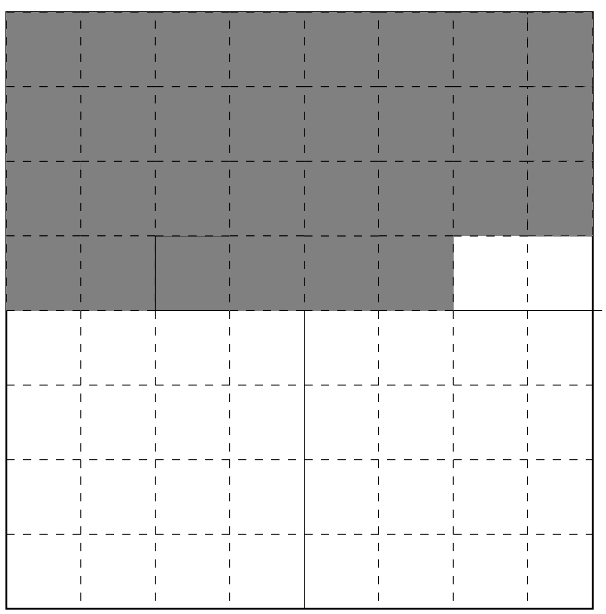

Figure 2:

In the following figure 3 the shaded Area C has address $alpha(C) = 2331$.
The following figure 4 shows the 3 - dimensional Area E with address 541.

To understand the definition of areas intuitively, imagine that areas are built up successively by adding subcubes in the order of subcube numbering or by adding smaller subcubes within the next

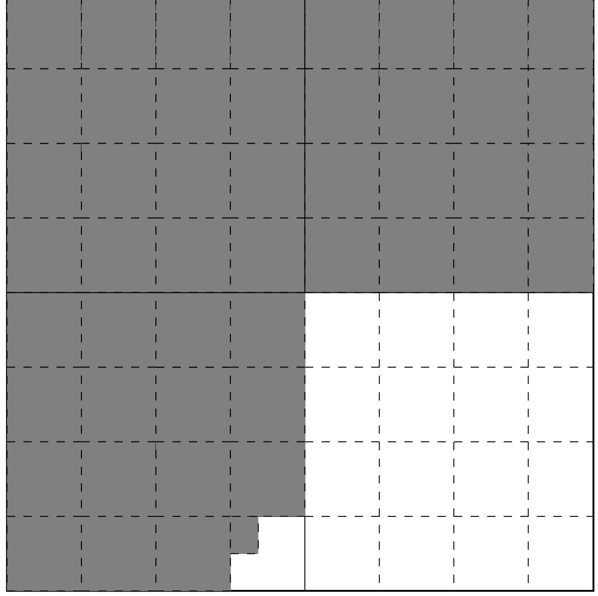

Figure 3:

subcube etc. An area is defined uniquely by the last subcube included in it. The address of this subcube is identical to the address of this area.

Def. 2.3: A **region** is the difference of two areas:
<u>if</u> $\alpha <\cdot \beta$ <u>then</u>
$region(\alpha, \beta) = Area(\beta) - Area(\alpha)$
where $-$ means "set difference"

Def. 2.4: A **page** is a fixed size byte container, e.g. a main-memory page or a disk-page. Pages fit into main-memory slots and also into disk-slots.

Note: We will arrange and maintain areas in such a way, that the objects (or better the identifiers of objects) in a region between two successive areas fit into a page. The content of a page p can then be specified by two addresses:

$content(P) =$ the set of objects in $region(\alpha, \beta)$.

Def. 2.5: the **address of a pixel** is identical to the address of the area defined by including the pixel as the last and smallest subcube contained in this area. A pixel is the smallest possible subcube and the limit of the resolution, but the resolution may be chosen as fine as desired.

Lemma 2.3: There is a one-to-one map between Cartesian coordinates $(x_1.x_2, \ldots x_n)$ of a n-dimensional pixel and its address α implicitely defined by the above addressing scheme. We use the following notations for these maps:
$alpha(x_1, x_2, \ldots x_n) = \alpha$
$cart(\alpha) = (x_1, x_2, \ldots x_n)$
Since the two maps are inverses of each other we get
$cart(alpha(x_1, x_2, \ldots x_n)) = (x_1, x_2, \ldots x_n)$
$alpha(cart(\alpha)) = \alpha$
$cart(\alpha).i$ is the i^{th}-coordinate of $cart(\alpha)$.

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

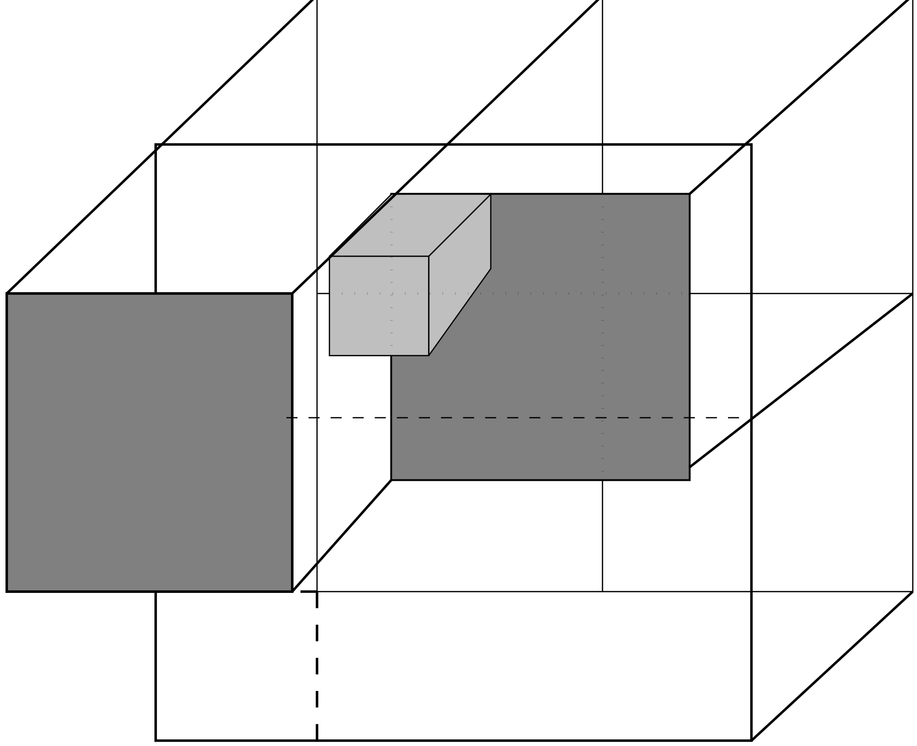

Figure 4:

We assume that we have a universe U of pixels. For simplicity we assume that U has $pix = 2^r$ pixels per dimension wich are numbered $0, 1, 2, \ldots 2^r - 1$. Arbitrarily sized spaces are simply considered as a subset of a suitable cube-shaped universe.

Since addresses are linearly ordered, by $<\cdot$, they can be treated as the keys of any variant of a B-tree. New point-type objects lie exactly between two area addresses and therefore in a unique region. The identifiers (Ids) of new objects are entered into their proper region, i.e. stored (inserted) into the page of their region.

3 The UB-Tree Access Structure

The UB-tree is a variant of a B-tree, in which the keys are addresses of regions ordered by $<\cdot$. Figure 5 illustrates this data-structure:

Id_1^m are the identifiers of objects in region (γ_0, γ_1) and Id_i^n are the identifiers of objects in region $(\gamma_{i-1}\gamma_i)$. Instead of storing identifiers Id_i^m in leaf-pages the objects could be stored there directly.

3.1 Insertion Algorithm

Points: Points to be inserted into the universe U are specified by their Cartesian coordinates $(x_1, x_2, \ldots x_n)$ and their addres $\gamma = alpha(x_1, x_2, \ldots x_n)$. They belong to the unique region(β, δ) with the condition $\beta <\cdot\ \gamma \leq\cdot\ \delta$. They are inserted into the leaf-page corresponding to that region, which is found by a point query, see section 4.

Extended Objects: An extended object, which is not just a single point, may intersect several regions. In that case, the object is inserted into each region which it intersects properly. More specifically, if object O intersects region (α, β) and region (γ, δ) the identifier $Id(O)$ is inserted into both pages page (α, β) and page (γ, δ) . This may cause either or both pages to overflow and to trigger region-splits and corresponding page-splits.

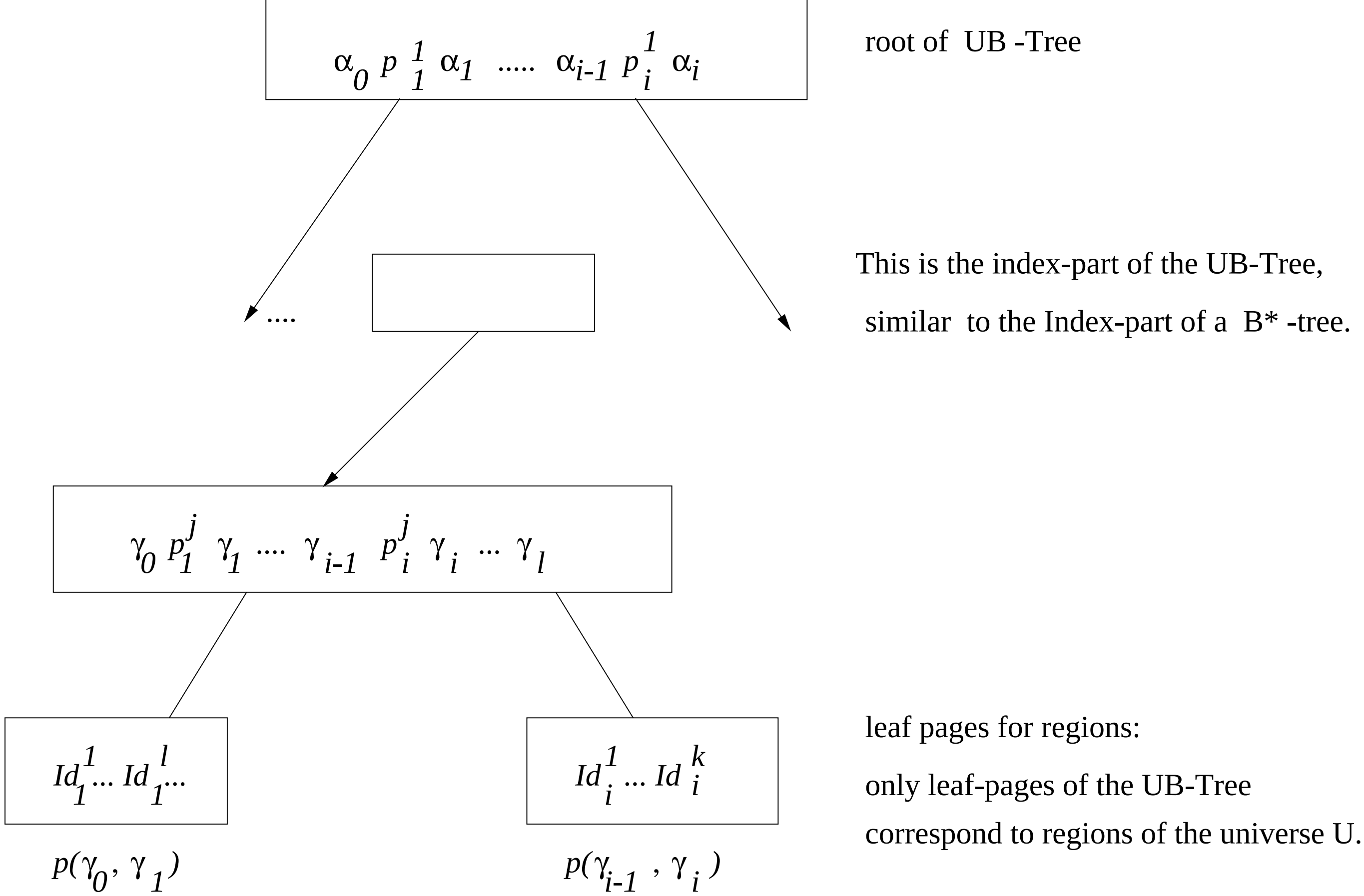

Figure 5:

Splitting Algorithm: Since pages can store only a maximum number M of Ids or objects, pages may overflow and split like in B-trees. If a region is defined by α and γ, the region is split by introducing a new area with address β such that $\alpha <\cdot\ \beta <\cdot\ \gamma$.
The region (α, γ) is partitioned by β into region (α, β) and region (β, γ) . The objects in page (α, γ) are distributed onto page (α, β) and page (β, γ) accordingly. β is constructed by increasing area α as follows: Add to area (α) subcubes from region (α, γ) in increasing order until
$\frac{1}{2}M - \epsilon \leq$ number of objects in region $(\alpha, \beta) \leq \frac{1}{2}M + \epsilon$
If the next subcube in this process contains too many objects, it is recursively subdivided until the condition can be met.

Lemma 3.1: If a cube has a resolution of pix pixels in each dimension, then addresses have a length of at most $\lceil log_2(pix) \rceil$. If we have a universe U with $pix = 2^r$ pixels in each dimension, then the addresses have a length of at most r.

Example: Taking a map of Bavaria with about 512 km length of a side, then addresses of length 16 yield a resolution of 8 meters per pixel.

Example of the Splitting process:

If $M = 5$, the shaded region (1.1, 2.3.2) in figure 6 is split into the two differently shaded regions region(1.1, 2.0.3) and region (2.0.3, 2.3.2) by the splitting address 2.0.3 as soon as the sixth object is inserted into region (1.1, 2.3.2)

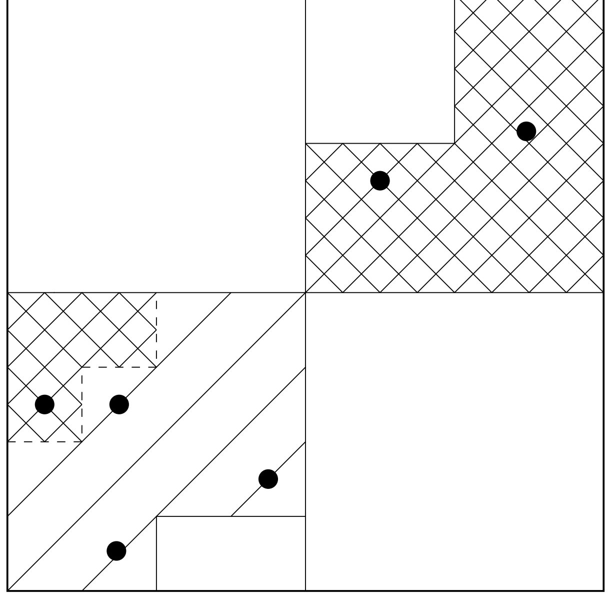

Figure 6:

3.2 Deletion- and Merging-Algorithm

When objects are deleted from a region (α, β), their corresponding Ids are removed from the page (α, β). If after this deletion page (α, β) has $< \frac{1}{2}M - \epsilon$ Elements, then page (α, β) is merged with the following page (β, γ) and the region (β, γ) disappears. If the resulting page (α, γ) overflows, it is split again "in the middle" by introducing a new area with address β' and regions: region (α, β'), region (β', γ) with the corresponding pages page (α, β'), page (β', γ) respectively. This final split of regions and pages is analogous to the underflow technique between pages of B-trees[4].

4 Query Processing

We treat two types of queries:

- point queries and
- m-dimensional interval queries, also called region queries or range queries.

We describe query processing for the general case, but illustrate it graphically only for the planar case, i.e. for 2-dimensional data.

4.1 Point Queries

Point Queries are also called "exact match queries". They are specified by the Cartesian coordinates $(y_1, y_2, \ldots y_n)$ of the point P. Usually additional information about P is of interest, e.g. temperature, height, time or monetary value. Such additional information may be stored as additional attributes with the point P, but separately from the index structure. It might also simply be added to the index structure, thereby increasing the dimensionality of the space and allowing queries on these additional attributes.

In databases this problem is usually solved by constructing a new secondary index; with UB-trees it is handled by increasing the dimension of the searchable object space.

To find P we compute its address π:

$\pi := alpha(y_1, y_2, \ldots, y_n)$

Then we find the unique region (α_{i-1}, α_i) with the property:
$\alpha <\cdot\ \pi \leq\cdot\ \alpha_i$
and fetch the page $page(\alpha_{i-1}, \alpha_i)$.
This is achieved by searching the UB-tree, using address π as the search key.

Page (α_{i-1}, α_i) must contain point P with the additional information or the identifier $Id(P)$ which is used as a reference to P.

Lemma 4.1: P can be found in $O(log_k N)$ time, where N is the number of objects in our universe U and $k = \frac{1}{2}M$.

Proof: UB-trees are balanced and searched exactly like the variant of B-tree used as the underlying data structure for the UB-tree.

4.2 Range Queries

Range Queries are a fundamental problem for all database systems. The query is specified by an interval for each dimension. No specificatin for a dimension formally means the interval $(-\infty, +\infty)$.

The query is the Cartesian product of the intervals for all dimensions, called the **query box** q. The answer to the query q is the set of point-objects in q or the set of extended objects intersecting q.

We denote the query interval w. r. to the i^{th} dimension by $[ql_i : qh_i]$ (for low and high value), of course $ql_i \leq qh_i$

We describe the search algorithm for the n-dimensional case but illustrate it only for the 2-dimensional case.

The query box q for 2 dimensions is shown in Fig. 7

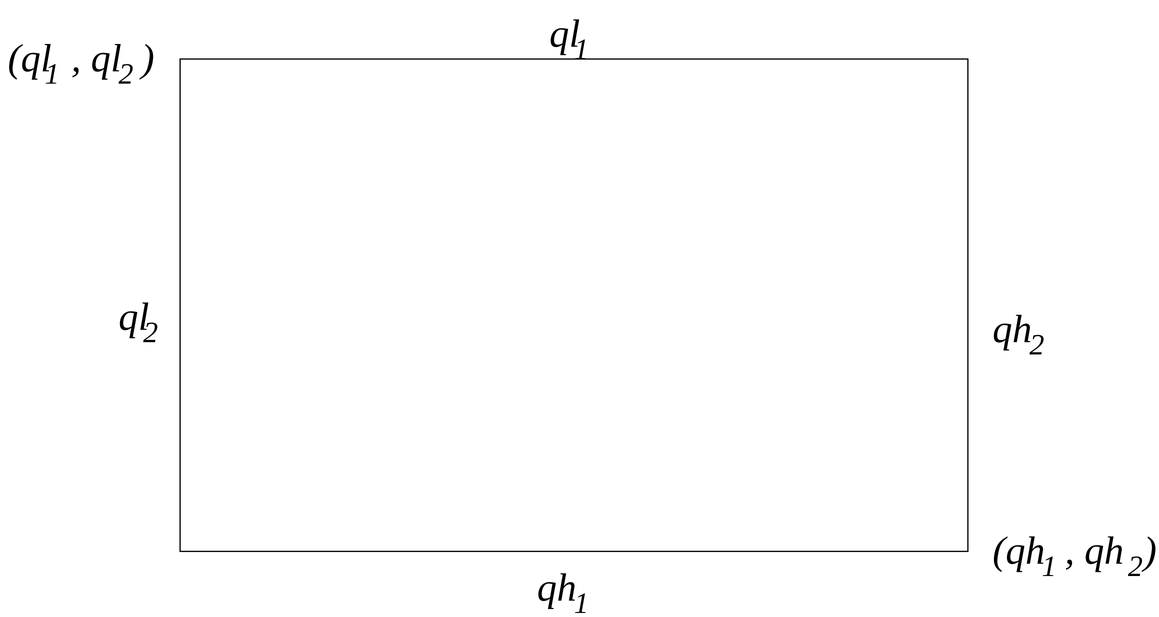

Figure 7:

The smallest point of q has Cartesian coordinates $(ql_1, ql_2, \ldots, ql_n)$ and lies in the well defined region $reg(\alpha_{j-1}, \alpha_i)$ with the property
$\alpha_{j-1} <\cdot\ \lambda\ \leq\cdot\ \alpha_j$
where $\lambda = alpha(ql_1, ql_2, \ldots, ql_n)$

A search in the UB-tree yields page (α_{j-1}, α_j) which contains all objects (or their identifiers) properly intersecting region $reg(\alpha_{j-1}, \alpha_j)$. We fetch these objects and check their intersection with q. Region $reg(\alpha_{j-1}, \alpha_j)$ itself consists of a sequence of subcubes, which are ordered by their addresses. Let β be the address of the last subcube of $reg(\alpha_{j-1}, \alpha_j)$ which intersects q. Let β have the form $\beta'.l$.

After processing all of $reg(\alpha_{j-1}, \alpha_j)$ we must find the next region intersecting q. To do this, consider the situation of subcube β w.r. to q and w.r. to father (β), i.l. the next larger subcube of which β is an partition:

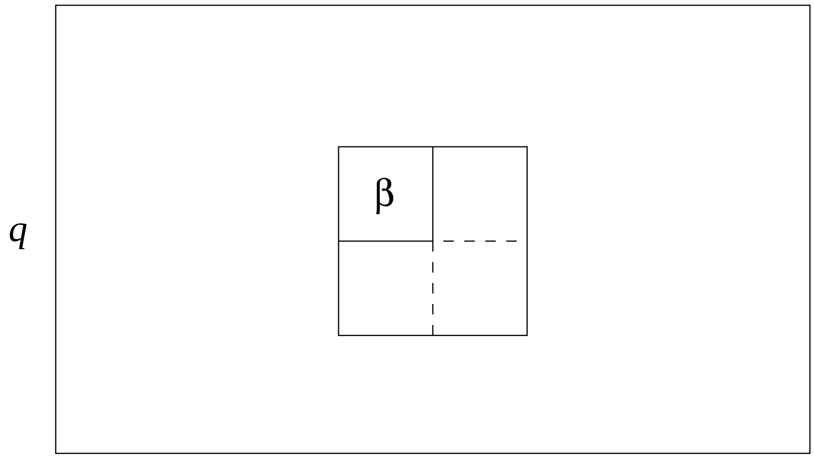

Figure 8:

We call all subcubes of level s of a cube at level $s-1$ *brothers*, those with a smaller or a larger address than β *younger* or *older* brothers resp.

The next objects, which intersect q and were not found yet, must intersect an older brother of β (thus exhausting father β) or an ancestor of β.

In the following figure we illustrate some situations for the 2-dimensional case and a subcube with address β of the form $\beta'.2$.

If no older brother of β intersects q (i.e. the last in Figure 9), then $father(\beta)$ cannot contain any objects, that were not yet found, but intersect q. Therefore we must check, whether the older brothers of father (β) intersect q (this exhausts the grandfather (β)), then the older brothers of grandfather (β), etc., This process must eventually cover all of q, at the latest, when the whole universe has been checked, and all objects intersecting q are found.

For the performance analysis observe that if subcube (β) is at level s, we may go to the next higher level at most s times. At each level we must check at most $2^n - 1$ older brothers for intersection with q . In addition, $s \leq ld(pix)$, and to switch to the father and check its older brothers for intersection with q involves only address calculations, but no I/O. Therefore, this method to find the next subcube intersecting q is extremely fast and can be ignored in the overall performance analysis.

After finding according to this method the first subcube w intersecting q, we compute the Cartesian coordinates of the smallest point of intersection (the solid small squares in Figure 9), which works as follows: Let w have the low and high coordinates xl_i and xh_i; w.r. to dimension i.

The condition that $w \bigcap q$ is empty is:

$\exists\, i : xh_i < ql_i$ **or** $xl_i > qh_i$

The condition that w intersects q is the negation of this formula:

not $\exists\, i : xh_i < ql_i$ **or** $xl_i > qh_i$

which is equivalent to:

$\forall\, i : xh_i \geq ql_i$ **and** $xl_i \leq qh_i$

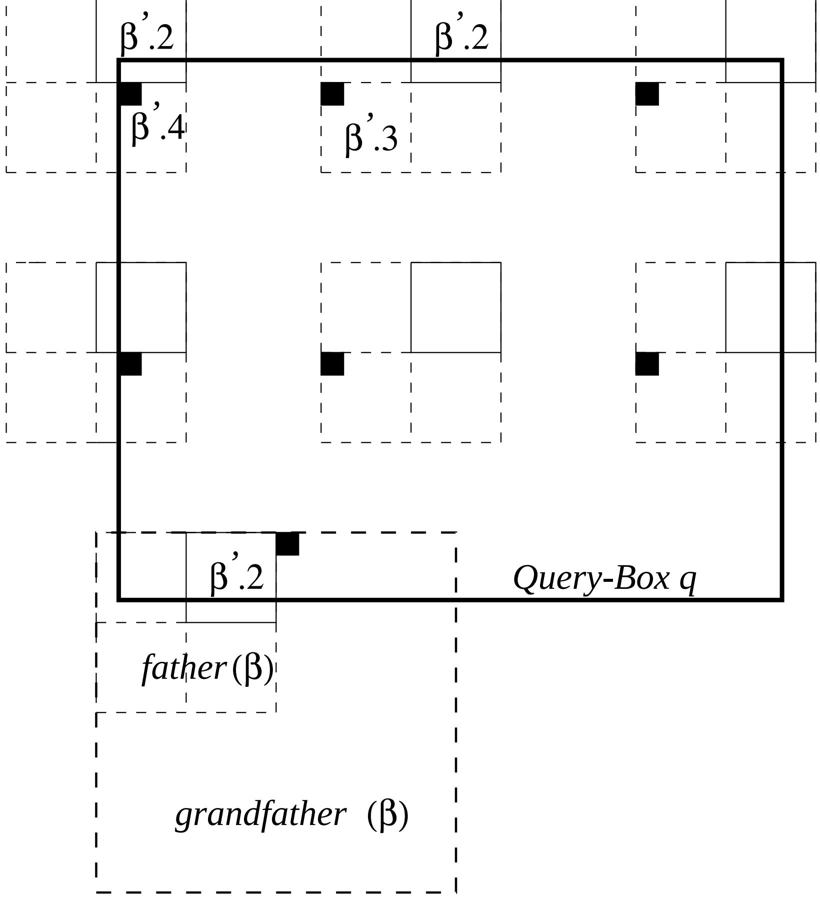

Figure 9:

Then the coordinates of the smallest point sp of intersection with q are for the i^{th} dimension:
if $xl_i > qh_i$ **then** $sp_i := xl_i$ **else** $sp_i := ql_i$

The intersection point sp then has the Cartesian coordinates $sp = (sp_1, sp_2, \ldots, sp_n)$ and its address is :
$\sigma := alpha\,(sp_1, sp_2, \ldots, sp_n)$

Note that up until now the determination of σ did not require any disk-accesses.

Now we have to find the unique region, in which the point sp lies. This is a point-query in the UB-tree with search key σ exactly as described above, requiring at most $O(log_k N)$ disk accesses and time.

This means for the performance analysis:

To answer range queries we have to do real work - i.e. to perform I/O - only for those regions which properly intersect q. For each such region the cost in $O(log_k N)$, i.e. a total cost of $r * O(log_k N)$, if r regions intersect q.

It can be shown, that the number r of regions intersecting q is in the average
$r \leq 2 * Q/(\frac{1}{2} * M) = 4 * Q/M$
where q contains Q objects.

Lemma 4.2:

Answering a range query q costs $r * O(log_k N)$ disk-accesses and time, if r regions intersect q .

In addition for point object data
$r \leq 4 * Q/M$

where q contains Q point objects.

Algorithm for range queries

After these considerations and analysis we now present the algorithm to answer range queries for an n-dimensional data-universe and an n-dimensional query box q with coordinates ql_i and qh_i for $i = 1, 2, \ldots, n$ as described above:

Initialize:
$sigma := alpha(ql_1, ql_2, \ldots, ql_n)$; $Answer$ = empty;
RegionLoop: begin co for every region which properly intersects q **oc**
find by searching in the UB-tree the region $reg(\alpha_{j-1}, \alpha_j)$
containing $sigma$, i.e. with condition $\alpha_{j-1} <\cdot sigma \leq\cdot \alpha_j$;
fetch $page(\alpha_{j-1}, \alpha_j)$;
ObjectLoop: **for** all objects o on page $page(\alpha_{j-1}, \alpha_j)$, **do**
if $o\, intersects\, q$ **then** o is part of the answer:
$Answer := Answer \cup o$
od ObjectLoop;
find the last subcube with address β of $reg(\alpha_{j-1}, \alpha_j)$
such that $Subcube(\beta)\, intersects\, q$;
if $(qh_1, qh_2, \ldots, qh_n)\, contained\, in\, Subcube(\beta)$ **then co** finished **oc goto** Exit **else**
FatherLoop: begin co let β be of the form $\beta = \beta'.i$ **oc**
$i := tail(\beta)$;
BrotherLoop: for $k := i + 1\, to\, 2^n$
do if $Subcube(\beta'.k)\, intersects\, q$ **then**
begin $sp := smallest\, intersection\, with\, q$;
$sigma := alpha(sp)$;
goto RegionLoop
co both loops will be left here,
since q is not finished yet **oc**
end BrotherLoop
od co for all larger brothers of β intersection with q is empty **oc**;
$\beta := father(\beta)$;
goto FatherLoop
end FatherLoop;
end RegionLoop;
Exit: co end of program, variable $Answer$ contains the query result **oc**

4.3 Management of general extended objects

Now let us consider general extended objects o instead of just point objects, as e.g. a lake in the following figure of a geographical map. First we surround such objects with a bounding box $bb(o)$. From now on we call such extended objects simply **objects**.

For an object o we store only an identifier $Id(o)$ with every region, which o intersects properly. o itself is in general stored outside of the UB-tree. Note the o can only intersect regions which are also intersected by $bb(o)$. This is a necessary but not a sufficient condition which we exploit in order to speed up the algorithms substantially.

We now present the abstract algorithm to insert an object o into the UB-tree:

Step 1: compute $bb(o)$. For most methods to represent o this is a simple matter.

Step 2: for all regions R which intersect $bb(o)$ do

if R intersects o then insert $Id(o)$ into R
co this may cause splits of R oc

Note To find the regions R which intersect $bb(o)$ one treats $bb(o)$ exactly like a query box q. This leads to the following detailed algorithm:

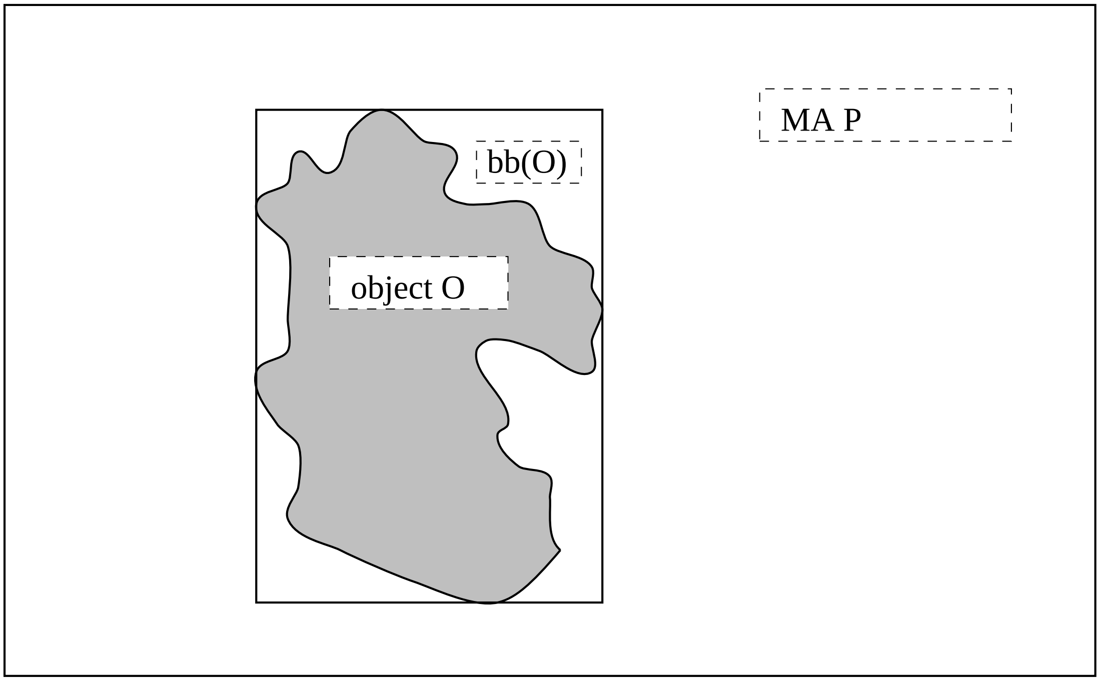

Figure 10:

Insertion algorithm for object o with bounding box $bb(o)$:

Initialize:
compute $bb(o)$;
$q := bb(o)$;
$sigma := alpha(ql_1,\ ql_2,\ \ldots,\ ql_n)$;
RegionLoop: begin co for every region intersecting q **oc**
find by search in UB-tree the region $reg(\alpha_{j-1}, \alpha_j)$
containing $sigma$ i.e. with $\alpha_{j-1} <\cdot\ sigma\ \leq\cdot\ \alpha_j$;
fetch $page(\alpha_{j-1}, \alpha_j)$;
if $o\, intersects\, R$ **then** insert $Id(o)$ into R, i.e.:
if number of objects intersecting R is $\leq M$
then store $Id(o)$ on $page(\alpha_{j-1}, \alpha_j)$
else split R and $page(\alpha_{j-1}, \alpha_j)$ as described in section 3.1;
find last subcube with Address β of $reg(\alpha_{j-1}, \alpha_j)$
such that $Subcube(\beta)\, intersects\, q$;
if $(qh_1,\ qh_2,\ \ldots,\ qh_n)$ *contained in* $Subcube(\beta)$ **then co** finished **oc**
goto Exit else
FatherLoop:
begin co let β have the form $\beta = \beta'.i$ **oc**
$i := tail\,(\beta)$;
BrotherLoop:
for $k = i + 1$ **to** 2^n

do if $Subcube(\beta'.k)\ intersects\ q$ **then**
begin
$sigma := alpha(sp)$;
goto RegionLoop
co FatherLoop and BrotherLoop will be left here,
since q is not finished yet **oc**
end
od co for all larger brothers of β
intersection with q is empty **oc**;
$\beta := father(\beta)$;
goto FatherLoop
end FatherLoop;
end RegionLoop;
Exit: co end of insertion algorithm **oc**

Deletion To delete o we again use $bb(o)$ to find all regions which might intersect o . $Id(o)$ is stored on exactly those pages whose regions intersect o (not just $bb(o)$). $Id(o)$ is removed from those pages and the corresponding regions. This may of course result in the merging of regions and the corresponding pages, as described above in section 3.2.

Searching in dataspaces with extended objects:

To find extended objects which intersect a query box q or are completely contained in it one proceeds as follows: find all regions and retrieve the corresponding pages exactly as in usual range query. The pages contain the identifiers $Id(o)$ of all objects o which intersect those regions. For all the $Id(o)$ thus found check, whether o intersects q.

5 Performance Analysis

In this section we assume that our data universe contains N objects at the time the operations are performed. Let $k = 1/2M$. Let Q be the number of objects intersecting the querybox q. Let r be the number of regions intersecting q.

5.1 Point-Query:

The cost of a point query is stated in Lemma 4.1 and is
$O(log_k N)$

5.2 Range Query

The cost of a range query is stated in Lemma 4.2 and is
$r\ *\ O(log_k N)$
For data spaces containing point objects only this can be estimated as
$(\frac{4*Q}{M} * O(log_k N)$
or in other words: for a given universe the cost depends directly on the size of the answer.

5.3 Point Insertion

To insert an object requires a point search to locate the proper region and page. This costs
$O(log_k N)$

If splits are triggered by the insertion they are restricted to the search path in the UB-tree and do not increase the order of the cost.

5.4 Insertion of an Extended Object

We assume that the object is inserted into each region it intersects and thereby into the corresponding page. This requires finding all regions (and the corresponding pages) which the bounding *box* $bb(o)$ of the object o intersects. Let that number be r. Then the cost is
$r * O(log_k N)$
or in other words: The cost depends directly on the size of the bounding box of the inserted object.

5.5 Point-Deletion

Again this requires a point search and cost
$O(log_k N)$

5.6 Deletion of an Extended Object

The object must be deleted from the page of each region it intersects. This requires a range search with the bounding box $bb(O)$ as query and cost
$r * O(log_k N)$

6 Comparison with previous work

The limitations of R-Trees, Grid-files and dd-trees were already mentioned before. The hB-tree of [5] allows good performance but requires complex hybrid datastructures and algorithms. The most widely used technique in databases to handle multidimensional data is to use a secondary index for each dimension (relation-column) which is to be used in a multidimensional search. Compared to the UB-tree this has the following disadvantages:

- instead of maintaining a single index structure like in the case of the UB-tree a total of n indexes must be managed and updated upon insertion and deletion of objects.

- multidimensional searching with several indexes has *additive* behaviour instead of *multiplicative* behaviour for UB-trees. More precisely we mean the following: assume that $p_i\%$ of the values lie in the query interval of q w.r. to dimension i. Then via the secondary index for dimension i a total of $N * p_i\%$ of the data must be fetched. This adds up to fetching

  $$N * p_1\% + N * p_2\% + \ldots + N * p_n\% = N * (p_1\% + p_2\% + \ldots + p_n\%)$$

  of the data or at least object identifiers from the disk and computing intersections between these sets.

  With an UB-tree the amount of data to be fetched is proportional to the size of the query box q, i.e

  $$N * (p_1\% * p_2\% * \ldots * p_n\%)$$

In other words, 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.

Example:

For an example calculation let us assume that
$n = 4$ and $p_1 = 2\%, p_2 = 5\%, p_3 = 4\%, p_4 = 10\%$
Then $p_1 + p_2 + p_3 + p_n = 21\% = 21 \cdot 10^{-2}$
and $p_1 * p_2 * p_3 * p_n = 4 \cdot 10^{-6}$

If our data universe contains 10^7 objects ($N = 10^7$), which would be only a medium size database, then using 4 secondary indexes retrieves 2.1 million objects from disk whereas the UB-tree technique would retrieve only about 80 objects from dik, i.e. an improvement by a factor of about 26.250.

Further work:

We are presently implementing and avaluating the UB-tree to determine its real performance for large, complex applications. We are also investigating architectural questions

- how to incorporate the UB-tree by preprocessing techniques into applications
- how to incorporate the UB-tree access structure into the kernel, the optimizer and the interpreter of database systems.

Literature:

[1] Mehlhorn: *Multidimensional Searching and computational Geometrie.* Springer, Heidelberg 1984

[2] Nievergelt, Hinterberger, Sevcik: *The Grid File.* ACM TODS, 9, 1, March 1984

[3] Guttman: *A dynamic Index Structure for spartial Searching.* Proceedings ACM SIGMOD Intl. Conference on management of Data, 1984, pp. 47-57

[4] Bayer, Mc Creight: *Organization and Maintenance of large ordered Indexes.* Acta Informatica, 1, 1972, Springer Verlag, pp. 173-189

[5] Lomet, Salzberg: *The hB- Tree: A Multiattribute Indexing Method with Good Guaranteed Performance.* ACM TODS, 15, 4, 1990, pp. 625 - 658