# UB-Trees and UB-Cache[1]
## *A new Processing Paradigm for Database Systems*

R. Bayer, Institut für Informatik und
Bayerisches Forschungszentrum für Wissensbasierte Systeme (FORWISS),
TU München, 15.3.1997

## Abstract

In this paper we describe a special caching technique, called UB-Cache, which is tailored to work with data organized as a UB-Tree [6], [7], a novel multidimensional datastructure. The UB-Cache makes it possible to read data from disk in arbitrary sort order according to those attributes that are used in the UB-Tree. This property can be used to speed up all operations of relational algebra substantially. We assume that the reader is familiar with the UB-Tree as described in [6] or [7].

## CR-Classification and Keywords

**F.2.2** Sorting and searching, **H.2.2** multidimensional access method, UB-Tree, multidimensional index, **H.2.3** relational algebra, query optimization, **H.2.4** query processing, caching, UB-Cache**, H.2.8** datamining.

## 1. UB-Tree and Sort-Orders

Let F be a file of records with attributes (A,B,...C,...,D) which is organized as a UB-Tree in which the multidimensional index uses the attributes (A,B,...,C). We call the attributes (A,B,...,C) also *UB-attributes.*

Example: A = Name
B = PLZ
C = Faxnr

*With the aid of the UB-Cache the data in F can now be read from the peripheral store in such a way, as if they were sorted according to an arbitrary selection and sequence of attributes out of (A,B,...,C).*

For three attributes we get for example the following possibilities:

| | | |
|---|---|---|
| ABC | BAC | CAB |
| ACB | BCA | CBA |
| AB | BA | CA |
| AC | BC | CB |
| A | B | C |

i.e. a total of 15 different sort orders.

In general we get for a d-dimensional UB-Tree

$$d! + d*(d-1)*...*2 + ...+ d*(d-1)*...*i + ... + d*(d-1) + d$$

possible sort orders.

For d=3 this results in

$$3! + 3*2 + 3 = 15$$

[1] Patent pending: Deutsches Patentamt Nr. 197 09 041.9

Therefore we can read or process the file F e.g. in the sort
(Name, PLZ, Faxnr) or also in the order
(PLZ, Name) , where now Faxnr appears in arbitrary sequence.

## 2. Jump-Regions

In order to describe the UB-Cache in more detail, we need the concept of a *Jump-Region.* In [6] and [7] we described, how space is recursively subdivided into subcubes and how these subcubes are combined into regions. After k-fold partitioning of the dataspace of relative size 1 into sucubes, we get subcubes of the relative length 1/n for each dimension, where $n=2^k$ and k>=0. Now we consider a subcube of sidelength 1/n. Such a subcube is covered by the regions of a UB-Tree in a very special way: There is a first region, which intersects the subcube, but which need not be completely contained in this subcube. We call such a region a *Jump-Region,* since it *jumps* from the outside *into* the subcube. Then the subcube is filled by further regions up to a last region which may not be completely contained in the subcube and *jumps outside*. We call this region also a Jump-Region. In order to differentiate more precisely we could use the terms *In-Region* and *Out-Region*. It is important to note, that on a certain level of partitioning there are at most two Jump-Regions per subcube, one In-Region and one Out-Region, independent of the level of partitioning, at which the subcube arises. Whether a given region is a Jump-Region or not depends on the level of partitioning and therefore on the size of the considered subcube.

## 3. Partitioning into Slices

For an introduction we consider a two-dimensional dataspace F with attributes A,B and assume for simplicity, that
attribute A assumes integer values between 0 and a given number a and that,
attribute B assumes integer values between 0 and a given number b

If we partition our dataspace k-times with $n=2^k$,
then the first row contains only points (A,B) with 0 <= A < a/n and
the first column contains only points (A,B) with 0 <= B < b/n.
In general the i-th row contains only points (A,B) with (i-1)*a/n <= A < i*a/n and
the j-th column contains only points (A,B) with (j-1)*b/n <= B < j*b/n.
In this way the whole dataspace is partitioned into n rows of the relative width 1/n and simultaneously into n columns of the same width.
In the d-dimensional case we call these subspaces *Slices w.r. to* attribute A (A-slice) resp. B (B-slice) instead of rows and columns.

If we now want to sort the data in the given space according to attribute A, one proceeds as follows:

**for** j := 1 **to** n **do**
sort the data in the j-th row (A-slice) according to A-values and
output these data in sort order

This rowwise processing works, since in the sort order according to A all data in the first row come before all the data in the second row and in general all data in the j-th row come before the data in the j+1st row.

## 4. UB-Buffer

Now, in order to read the data in the j-th row, we consider the j-th row as a query-box FZ[j] and read those regions, which intersect FZ[j], from the peripheral store into a special section of the working store of the computer - we call this section *UB-Buffer*. Now those data are sorted internally (e.g. using Heapsort or Quicksort). The data from FZ[j] can be processed in sorted order immediately, the data from Jump-Regions, which do not lie in FZ[j] are buffered in UB-Buffer for a later time.

## 5. The General Case

We now illustrate the general d-dimensional case using a three-dimensional dataspace with attributes A,B,C as example. The attributes A,B,C can assume values between 0 and a, resp. 0 and b, resp. 0 and c.

If we want to sort the data w.r. to attribute A, we partition the whole dataspace by successive hyperplanes of the form:

$$A = j/n*a$$

into slices. Then the i-th slice contains exactly the data with the property for attribute A:

$$(i-1)*a/n <= A < i*a/n.$$

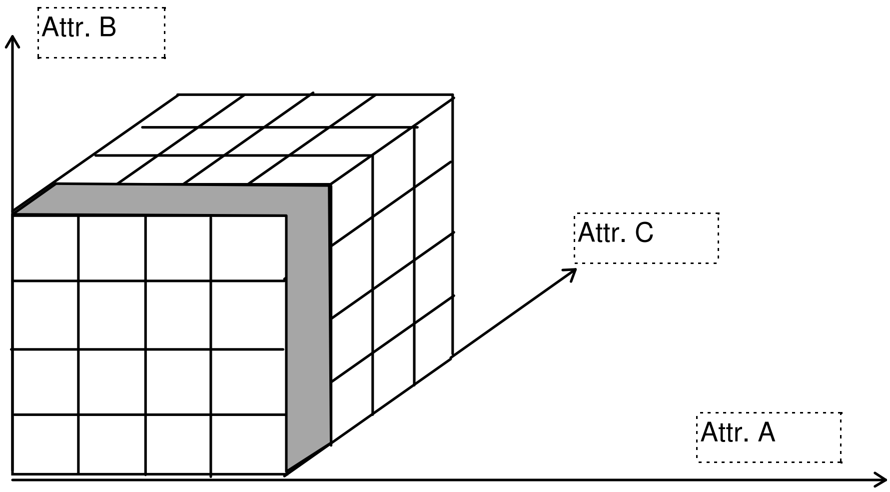
If on the other hand we want to sort the data w.r. to attribute C, then we partition the whole dataspace again by successive hyperplanes of the form:

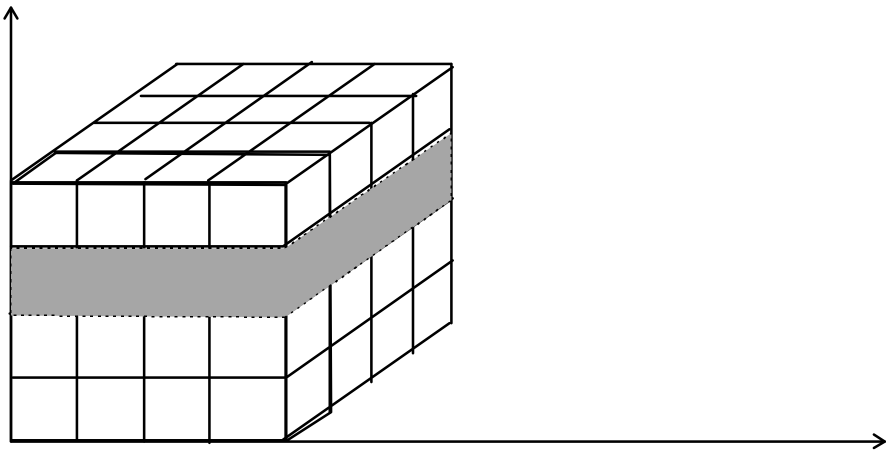
$$C = j/n*c$$

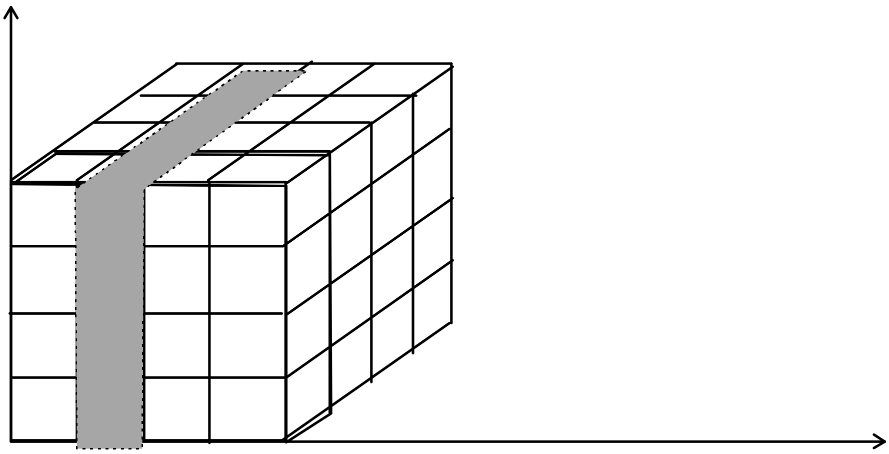
into slices of width 1/n. We call such slices also C-slices and process the data again slice after slice. The i-th slice then contains exactly those data with the property for attribute C:

$$(i-1)*c/n <= C < i*c/n.$$

The following figures show a 3-dimensional cube after two divisions (partitioning level 2). Shaded are the first slice w.r. to C, the third slice w.r. to B and the second slice w.r. to A.







One slice of width 1/n then contains approximately (if we assume uniform distribution of the data in our space) N/n dataobjects, if the whole space contains N dataobjects.

## 6. UB-Cache

The UB-Cache has the purpose to process the dataspace „slice by slice", to store those data of jump-regions, which lie outside a slice, for a later time, and to control the datatransports between disk and mainstore in an optimal way.

The UB-Cache method consists of two essential components:

- *UB-Buffer*, i.e. a section of the main store in order to store slices - and parts of jump-regions for processing at a later time.
- *UB-Control*, which manages the datatransport between peripheral storage and UB-Buffer in an optimal way and performs the processing of the data (e.g. internal sorting or join operation) of the data in the UB-Buffer.

UB-Buffer is needed for two tasks:

1. Read in all regions, which intersect a slice.

2. Buffer those parts of jump-regions which were read in step 1, but which do not lie in the slice that is presently being processed, for a later time when the proper slice is processed.

UB-Control performs the following:
1. Determine the optimal size of slices, see chapter 8.
2. Manage the datatransports from the peripheral store into UB-Buffer.
3. Internal (in UB-Buffer) sorting of the data in those regions that were read in. For this one uses standard sort techniques for internal sorting like Heapsort or Quicksort.
4. Output of the data in the presently processed slice in the desired sort order.
5. Management of those parts of jump-regions which are still needed for the processing of later slices.

We now present the algorithm in a high level pseudo language:

## 7. Algorithm for *UB-Control*

We describe the algorithm for the 2-dimensional case, in order to process the dataset F in sort order: Let the dataset F be partitioned k-times, i.e. there are $n=2^k$ slices from 1 to n, there are $n^{d-1}$ subcubes per slice, and therefore at most $2*n^{d-1}$ jump regions (see chapter 2). We denote the i-th slice as FS[i].

UB-Buffer B := empty;
**co** let B be organized as a heap **oc**
**for** i := 1 **to** n **do**
    **begin co** process the i-th slice **oc**
        read new regions which intersect FS[i] and
        insert tuples into heap B;
        output tuples in FS[i] from B in sort order
    **end;**
    **co** after FS[n] has been read in, heap B becomes empty again **oc**

## 8. Optimal Partitioning of Slices for F

If F is partitioned into slices of breadth 1/n, the necessary size of the UB-buffer in order to process F in sort order is obtained for the 2-dimensional case as follows, where |P| is the size of a region resp. of a disk block:

$$B(F,n) = |F|/n + 2*n*|P|$$

|F|/n is needed for the slice itself, 2*n*|P| for the jump regions.
This buffer size becomes minimal, if the derivative w.r. to n becomes zero:

$$B'(F,n) = -1*|F|/n^2 + 2*|P| = 0$$

with the optimal size $n_{opt}$ for n:

$$n_{opt} = \text{squareroot}\ (|F|/(2*|P|)).$$

Now choose k approximatively in such a way that $n=2^k$ is as close as possible to $n_{opt}$ in order to obtain a nearoptimal size for UB-Buffer.

**Computational Example for the 2-dimensional Case:**
$|F| = 10^9$ Bytes, $|P| = 10^3$ Bytes
$n_{opt} = \text{squareroot}\ (10^9/(2*10^3)) = 707$

Choosing k=10 and therefore n = 1024 results in B(F,1024) = 3 MB
Choosing k= 9 and therefore n = 512, results in B(F,512) = 3 MB
The exact optimum would be B(F,707) = $10^9/707 + 2*707*10^3$ = 2.82 MB.
This also shows that B(F,n) has a very *flat* minimum and is not very sensitive w.r. to the exact choice of n.

## 9. General Case of the Cache-Size

In the general d-dimensional case the necessary size of the UB-Buffer in order to process F in sort order is obtained as follows:

$$B(F,n,d) = |F|/n + 2*n^{d-1}*|P|$$

|F|/n is needed for the slice itself, $2*n^{d-1}*|P|$ for the jump regions, since a slice has $n^{d-1}$ subcubes.

This buffer size becomes minimal, if the derivative w.r. to n becomes zero:

$$B'(F,n,d) = -1*|F|/n^2 + 2*(d-1)*n^{d-2}*|P| = 0$$

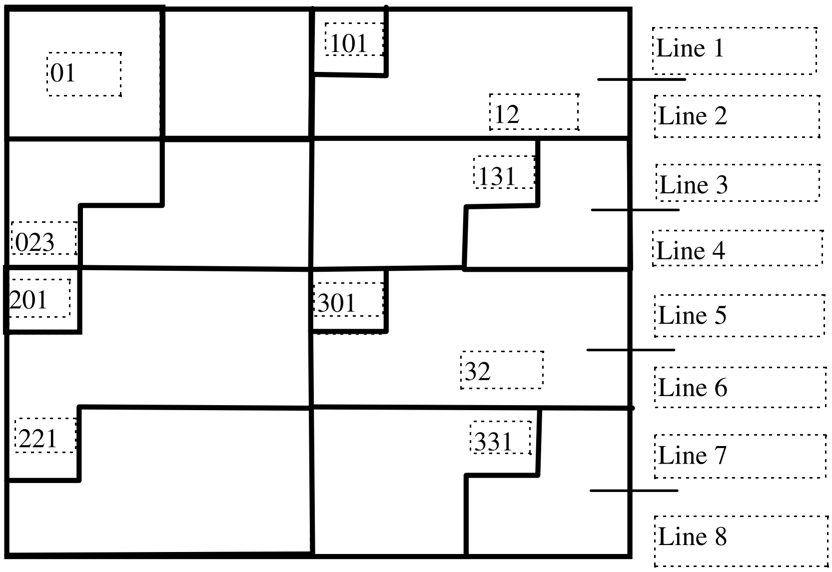
with the optimal size $n_{opt}$ for n:

$$n_{opt} = \text{d-th root } (|F|/(2*(d-1)*|P|)).$$

**Example:** The 2-dimensional cube (square) in the following figure consists of 8 lines after 3 partitionings. For every line we have the following jump regions, whose parts reaching outside of the line must be buffered in UB-Buffer for later processing. The regions are denoted by the higher of the two bounding addresses:

| | | | | | |
|---|---|---|---|---|---|
| Line 1: | 01 | 023 | 101 | 12 | |
| Line 2: | | 023 | 101 | | |
| Line 3: | | 023 | 101 | 131 | 201 |
| Line 4: | | | | | 201 |
| Line 5: | 221 | 301 | 32 | | |
| Line 6: | 221 | (301) | | | |
| Line 7: | | 301 | 331 | infinite | |
| Line 8: | no further jump regions | | | | |

The example shows the jump regions per line and the contents of UB-Buffer. It also shows how pessimistic our estimate of the buffer size is for typical cases, according to which we could have up to 16 jump regions, whereas here we get only 4 jump regions. If one reads the dataset according to columns one gets 6 jump regions. Also interesting is line 6, in which only jump region 221 is new, but the region in parentheses 301 is still in UB-Buffer, since it must be processed later in line 7. In subcube 2 region(023, 101) is the starting *jump-in-region*, then this subcube is first filled by the regions (101,12) and (12,131), and then the *jump-out-region* (131,201) follows. The smaller subcubes 01, 12, 32 do not have *jump-out-regions*, since the subcube boundaries coincide with the region boundaries. This means that the estimate of two jump-regions per subcube is quite pessimistic in many cases. We are working on a more precise analysis and estimate.




**Other Sort Orders**
If we want to process F in another sort order, we read the region in the 2-dimensional case by columns instead of by lines. In the general d-dimensional we get special subspaces instead of lines and columns, which we call slices. Slices are not separated by lines but by hyperplanes. These hyperplanes can be equidistant or they can be adapted to the density of the population of the objects in our universe.

## 10. UB-Cache and Relational Databases

In particular, in order to process the operations of relational algebra efficiently, it is required that at least one of the operands involved in the operation be sorted according to certain attributes.

For databases which are organized in a conventional way (without UB-Tree and without UB-Cache), relations must be sorted very frequently. For this purpose the data must be transported between mainstore and peripheral store (hard disk) several times.

As we just explained, data which are organized as a UB-Tree can be read and processed with the aid of the UB-Cache method in such a way *as if they were sorted on the disk*. The UB-Cache technique therefore avoids the sort operations that were necessary so far. In this way the data transports between mainstore and disk mentioned above for sort purposes can be avoided. Therefore the operations of relational algebra can be speeded up substantially.

For this reason the UB-Cache method can speed up all relational database systems substantially. Some operations of relational algebra are also used in a similar form in other database systems, e.g. in object oriented database systems. These operations can be speeded up in a similar way with the UB-Cache technique in combination with the UB-Tree data organization.

To give a specific example let us consider the join operation between 2 relations R and S. With present technology this requires the following processing steps:

1. read relation R from disk
2. presort portions of R
3. write portions of R to disk
4. read portions of R again from disk
5. sort R by merging the presorted portions
6. write R back to disk
7. if S is not yet sorted according to the join attribute, the steps 1 to 6 must be performed for S analogously. The steps 1 to 7 are just preparations for the real join operation:
8. read S by portions in sort order from disk
9. read R by portions in sort order from disk and join tuples with the suitable join partners from S
10. write the final join result to disk

The steps 4,5,6 are performed at least once, in some sort methods even several times. Therefore altogether R is read from disk at least three times (in steps 1,4,6) and written to disk at least three times (in steps 3,6,10, since the join result contains the data from R and S). Exactly like R also S is read from and written to disk at least three times. The join method we just described is known in the literature as *sort-merge-join*.

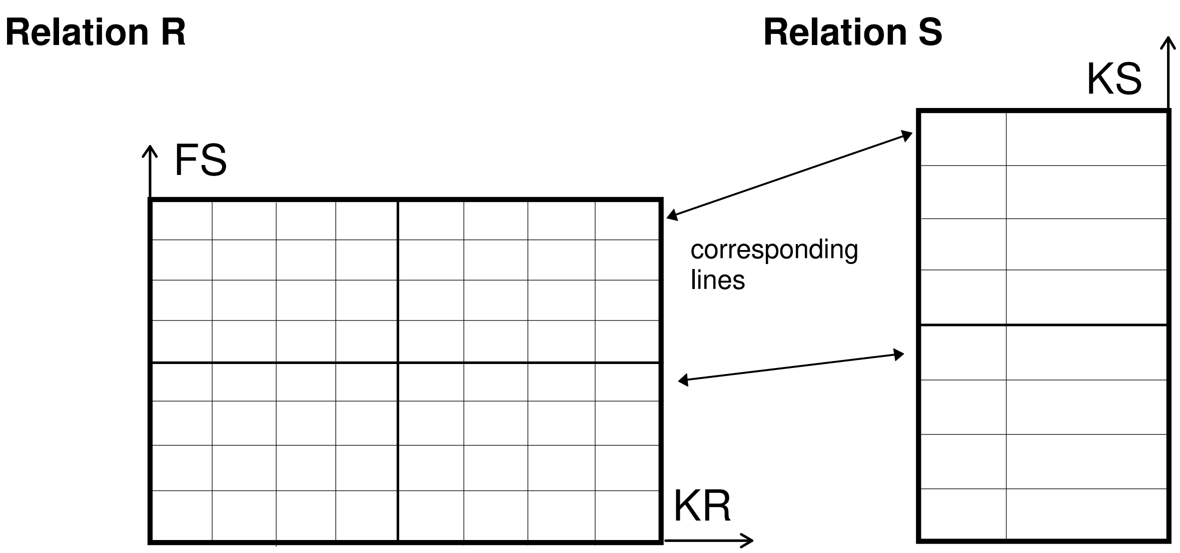
If relation R is organized in a suitable way as a UB-Tree, then R can be read with the aid of the UB-Cache method in such a way, as if R were sorted on the disk. The same holds for S. Therefore steps 1 to 7 in the join described above are no longer needed and the complete join operation is speeded up at least by a factor of 3.

Most operations of relational algebra (not only the join operation, but the join is the most important and performance critical of all of them) require that the operands involved must be read in sort order. Therefore, using conventional algorithms, the operands must be sorted first before really starting the operation. On the other hand, these sort processes are no longer needed if one uses the UB-Tree data organization in combination with the UB-Cache method, and the operations can be speeded up substantially.

We now describe the join algorithm for the 2-dimensional case according to the following figure. This case arises always, if e.g. a relation contains a primary key and a foreign key. A relation would be organized as a d-dimensional UB-Tree if it contains a primary key and d-1 foreign keys. In the case of compound keys the compound key itself or selected single attributes of the key can be used as UB-attributes of the UB-Tree. There are interesting and important connections and dependencies between DB-schema, structure of keys, foreign keys and the optimal usage of UB-Trees. We are presently investigating this important and fascinating problem area. In the following example FS denotes a foreign key. KS and KR stand for primary keys of S resp. R.

## 11. Join-Algorithm with UB-Cache Support

1. UB-Control optimizes the buffer size and determines automatically $n_{opt}$. For our example we assume that $n_{opt} = n = 8$.
2. A relation which is organized as a UB-Tree is partitioned into portions of size 1/n where $n=2^k$, so in the example k=3. In the 2-dimensional case we call these portions *lines.* They are numbered from 1 to n.
3. After k partitionings of the space a line contains n subcubes, in our example therefore 8 squares.
4. Now for the join operation a line of S is completely read into UB-Buffer (this corresponds to step 8 in the sort-merge-join described before). If S is organized as a B-Tree or suitably sorted, exactly one line can be read from the peripheral store into the UB-Buffer. (If S is organized as a UB-Tree, the line is read in sort order, portions of jump regions reaching beyond the line are buffered in UB-Buffer until they are needed at a later time.)
5. Now relation R is processed line by line, in order to join the tuples in the line with he matching tuples from corresponding lines of S (corresponding to step 9 in the sort-merge-join described before). For this purpose those regions are read from the disk which cover the n subcubes of the line. Since regions correspond exactly to disk blocks, regions are always transported completely from the disk to mainstore in a single read or write operation. We get the following two cases:
   - Those regions which are completely contained in the line can immediately be processed completely, i.e. their tuples can be matched with their join partners from S, they do not even have to be buffered (in contrast to the general sort algorithm).
   - Those regions which reach beyond the line into later lines - these regions we called *jump regions* - contain two types of data: such data which lie in the line presently being processed (*line internal data)* and such data which lie outside the line (*line external data).* Line internal data can be processed immediately and can be joined with their partners from the corresponding line from S. Line external data must be buffered in UB-Buffer until later, when the proper line of R is processed.



Relation R after 3 partitionings with $2^3=8$ subcubes per line and column. Relation S is partitioned into 8 lines.

WE present the join-algorithm as a pseudo program:

```
UB-Buffer B := empty;
for i:=1 to n do
begin  read i-th slice  Si from  S into mainstore;
       compute join-partners of tuples in Si with tuples in regions in B and remove
       finished regions from B;
       co after reading the last slice of S buffer B becomes empty oc;
       for k := 1 to n do
           begin read regions in sequence which intersect k-th subcube in  i-th
           slice of R and compute in mainstore the join result with the tuples in Si;
           buffer those regions of R in B, which jump to a higher slice of R
           end
end
```

Here it is important to know the maximal number of jump regions which could arise in order to estimate the storage required to buffer jump regions of R or at least their slice external data in UB-Buffer. For simplicity we assume that jump regions are buffered completely. By differentiating between slice internal and slice external data this storage requirement of the UB-Buffer for R could be reduced to about one half, since in the join algorithm only line external data would have to be buffered. In addition some jump regions jump into other subcubes of the same line, therefore they do not need to be buffered for later lines.

The necessary size of the UB-Buffer for the join algorithm is therefore obtained as follows:

**Case 1:** Let S be sorted suitably, let R be organized as a UB-Tree:

- store complete slice of S: |S|/n Bytes
- store jump regions of line of R: 2*n*|P| Bytes
- Total storage need: B1(S,n) = |S|/n + 2*n*|P| Bytes

The optimum is obtained for B1'(S,n) = $-|S|/n^2 + 2*|P| = 0$
and therefore $n_{opt}$ = root (|S|/(2*|P|)).

**Case 2:** Both relations R and S are not suitably sorted, but organized as UB-Trees:

- store complete slice of S and jump regions of this slice:
  |S|/n + 2*n*|P| Bytes
- store jump regions of slice of R, at most: 2*n*|P| Bytes
- Total storage need: B2(S,n) = |S|/n + 4*n*|P| Bytes

As an optimal upper bound we obtain B2'(S,n) = $-|S|/n^2 + 4*|P| = 0$
and therefore $n_{opt}$ = root (|S|/(4*|P|)).

We hypothesize that this upper bound is sharp, more precise analysis is in progress. In the general d-dimensional case the buffer requirement can be estimated in analogy to chapter 9.

***Fundamental Observation: In both cases the size of the required UB-Buffer is independent of the size of R. Therefore one would interchange the role of R and S if R is the smaller relation.***

## 12. Computational Examples

We do the computation for some examples which are realistic for computer architectures of today:

**Example 1:** This example is typical for relations as they arise in databases on workstations of today:

|S| = 100 MB (Megabyte) = $10^8$ Bytes; |P| = 1 KB (Kilobyte);
$n_{opt}$ = root ((100 * $10^6$ )/ (2 * $10^3$ )) = 224
The required buffer space is: B($n_{opt}$ ,S) = B(224, $10^8$) = 895 KB

**Example 2:** This example is at the upper performance limit of database systems of present workstations:

|S| = 1 GB (Gigabyte) = $10^9$ Bytes; |P| = 1 KB (Kilobyte)
$n_{opt}$ = root ((1 * $10^9$ )/ (2 * $10^3$ )) = 707
The required buffer space is: B($n_{opt}$ ,S) = B(707, $10^9$) = 2.84 MB

**Example 3:** This is an example for very large datasets, as they arise today on host computers or even supercomputers, but not on workstations:

|S| = 100 GB = $10^{11}$ Bytes; |P| = 1 KB (Kilobyte)
$n_{opt}$ = root (($10^{11}$ )/ (2 * $10^3$ )) = 7071
The required buffer space is: B($n_{opt}$ ,S) = B(7071, $10^{11}$) = 28.3 MB

The storage requirement for the Buffer B is surprisingly small. Workstations today have typically hard disks of 2 to 4 gigabytes, therefore single relations of size 1 GB (example 2) are rare. On the other hand main stores of workstations today are in the area of 16 to 64 MB, so that a buffer of 3 MB is easily available. Main stores of hosts and super computers today are 256 MB upward, therefore the estimated UB-Buffer of 30 MB is easily available.

**Comment on the Size of R:** It may surprise that the buffer size B(n,S) depends only on the size of S, but not on the size of R. Even this buffer size is only needed, if R has at least a certain minimal size, as the following argument shows: If a partitioning of R into n slices really causes 2*n jump regions per slice, then in the 2-dimensional case R must have at least $n^2$ regions, therefore it must have about $n^2$*|P| bytes. In the preceding examples this is about 50 MB resp. 500 MB resp. 50 GB. If R should be smaller, less than the computed buffer space will be needed, on the other hand R may become arbitrarily large without requiring more than the analyzed buffer space.

## 13. Further UB-Tree/Cache Algorithms for the Relational Algebra

**Notation:** Let R be a relation with attributes (A, B, ... C, D, ... ,E). We denote a UB-Tree for the relation R, which is built using the multidimensional attribute (A,B,...,C) as *UB-Index (R, (A,B,...,C))*
The attributes A,B,...,C are called *UB-Attributes* of R.

### 13.1 Projection $\pi$

We only sketch the algorithm for the projection $\pi_{A,D}$ (R) :
The simplest method reads R in sort order according to (A,D) and projects to the attributes A,D. Then duplicates are easily recognized and eliminated. Buffer requirement is like for sorted reading, but reduced by the space for attributes which are projected away.

**Lemma:** The projection of a relation R, which is organized as a UB-Tree, to a set of attributes containing at least one UB-attribute can be computed in linear time including duplicate elimination, if at least the following buffer space is available for the UB-Cache:

$B(n_{opt},R) = 2^* \text{ root } (2^*|R|^*|P|)$ Bytes in the 2-dimensional case resp.

$B(R,n_{opt},d) = |R|/n + 2^*n_{opt}^{d-1*}|P|$ Bytes in the d-dimensional case with

$n_{opt} = \text{d-th root } (|R|/(2^*(d-1)^*|P|))$

### 13.2 *Set Union union*

The main problem in computing the union of two relations R and S is again the elimination of duplicates. If both relations are sorted in the same way then the duplicate elimination is trivial: Both relations are read simultaneously and the duplicates are easily recognized and taken into the final result only once.

If both relations R and S are organized as UB-Trees with at least one common UB-attribute, then both relations can be read slice by slice according to this common attribute. Duplicates can only be contained in corresponding slices, therefore they are easily recognized and eliminated. Therefore ***union*** is computed in linear time

Space requirement for the UB-Buffer is the same as for the join operation.

### 13.3 Intersection of Relations *intersect*

The algorithm has the same structure as ***union***, also storage requirement is the same.

### 13.4 Difference of Relations *minus*

To compute the difference of two relations one also works slice by slice if the two relations have at least one common UB-attribute. The algorithm has linear time complexity, if we have as much buffer space as for ***union*** and ***intersect***.

### 13.5 Division of two Relations *divide*

Let the division be R(A,...,B,C,...,D) ***divide*** S(C,...,D).
The division produces a result relation with the structure T(A,...,B). Let at least one of the attributes A,...,B be a UB-attribute of R, without loss of generality let this be the attribute A. Then R can be considered as sorted according to the attribute A and can be processed slice by slice. According to the definition of ***divide*** the result T is defined as follows:

T(A,...,B) = {r´ | for all tuples s in S exists a tuple r in R : r´s = r }

These r´ are easily found by processing R according to slices of A. For this we need a buffer of size $B(n_{opt},R) = 2^* \text{ root } (2^*|S|^*|P|)$

In addition all of S must fit into the buffer - then the division is computable in linear time - or S must be read once completely for every slice of R. Sorting of R, however, is - in contrast to the conventional algorithm - never needed.

## 14. UB-Tree Algorithms for Aggregations

Aggregate functions are a central component of database systems and play an important role in applications like datamining. The most important aggregate functions are:

- ***count*** to count the tuples of a group
- ***sum*** to add the attribute values of the tuples of a group
- ***max*** to compute the maximum of the values of the tuples of a group
- ***min*** to compute the minimum of the values of the tuples of a group
- ***average*** to compute the average of the values of the tuples of a group

But in principle there are many other aggregate functions. In general aggregate functions for a relation R(A,...,B,C,...,D,E,...,F) are computed as follows:

1. All tuples of R which have the same value for the attributes A,...,B are collected into the same group. Therefore the attributes A,...,B are also called *grouping-attributes.*
2. For the attributes C,...,D aggregations for every group of tuples are computed, e.g. in order to compute the maximum of the C-values and the average of the D-values. Therefore the attributes C,...,D are also called *aggregation-attributes*
3. The attributes E,...,F are removed by projection.
4. The result consists of one tuple per group with the values A,...B, $agg_1(C),...,agg_k(D)$, where $agg_1(C)$ and $agg_k(D)$ are the aggregation values.

Such aggregations are the consequences of SQL-statements like in the example:

**select** A,...,B, $agg_1(C),...,agg_k(D)$
  **from** R
    **group by** A,...,B

In conventional algorithms the relation R must first be sorted according to the grouping attributes, in order to have the tuples of a group positioned next to each other and to compute the aggregations efficiently.

We propose a new algorithm which organizes the relation R as a UB-Tree in such a way that at least one of the grouping attributes - assume this is attribute A - appears as a UB-attribute. Then R is partitioned according to A and read slice by slice from the peripheral store. Tuples whose A value is in the slice can be grouped immediately in the main store and the corresponding aggregations can be computed easily.

Tuples from jump regions whose A-value lies in a later slice must be buffered until it is the turn for this slice to be processed. The needed buffer space is as large as for the projection operation.

If this buffer space is available also aggregate functions can be computed in linear I/O-time, i.e. by reading the relation R only once.

**Summary:** *In this paper we have shown that all operations of relational algebra including the extension of aggregate functions can be computed in linear I/O-time if the relations involved are organized in a suitable way as UB-Trees and if sufficient buffer space for the UB-Cache technique is available. For today's computer architectures and for typical practical applications this is usually the case.*

***An important additional property is that the needed buffer space in main store grows with an exponent smaller than 1, i.e. sublinear in the relation size. This has the practical consequence that the algorithms presented here should also be suitable very well for future computer architectures and applications.***

## Literature

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

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

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

[4] Bayer, McCreight: *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

[6] Bayer: *The Universal B-Tree for multidimensional Indexing.* Technical Report Nr. I9639, see also URL: http://www.leo.org/pub/comp/doc/techreports/tum/informatik/report/1996 /TUM-I9639.ps.gz

[7] Bayer: *The Universal B-Tree for multidimensional Indexing: Basic Concepts.* WWAC´97 Conference, March 10-11,1997, Tsukuba, Japan, pp. B-3-2-1 to B-3-2-12