Integrating the UB-Tree into a Database System Kernel

Frank Ramsak[1], Volker Markl[1], Robert Fenk[1], Martin Zirkel[2], Klaus Elhardt[3], Rudolf Bayer[1,2]

[1]*Bayerisches Forschungszentrum für Wissensbasierte Systeme Orleansstraße 34, D- 81667 München, Germany*

[2]*Institut für Informatik TU München Orleansstraße 34, D-81667 München, Germany*

[3]*TransAction Software GmbH Gustav-Heinemann-Ring 109, D-81739 München, Germany*

{frank.ramsak, robert.fenk, volker.markl}@forwiss.de, {zirkel, bayer}@in.tum.de, klaus.elhardt@transaction.de

Abstract

Multidimensional access methods have shown high potential for significant performance improvements in various application domains. However, only few approaches have made their way into commercial products. In commercial database management systems (DBMSs) the B-Tree is still the prevalent indexing technique. Integrating new indexing methods into existing database kernels is in general a very complex and costly task. Exceptions exist, as our experience of integrating the UB-Tree into TransBase, a commercial DBMS, shows. The UB-Tree is a very promising multidimensional index, which has shown its superiority over traditional access methods in different scenarios, especially in OLAP applications. In this paper we discuss the major issues of a UB-Tree integration. As we will show, the complexity and cost of this task is reduced significantly due to the fact that the UB-Tree relies on the classical B-Tree. Even though commercial DBMSs provide interfaces for index extensions, we favor the kernel integration because of the tight coupling with the query optimizer, which allows for optimal usage of the UB-Tree in execution plans. Measurements on a real-world data warehouse show that the kernel integration leads to an additional performance improvement compared to our prototype implementation and competing index methods.

Proceedings of the 26th International Conference on Very Large Databases, Cairo, Egypt, 2000

1 Introduction

Various research approaches in the past have shown that multidimensional access methods (MAMs) have a high impact on different database application domains like data warehousing, data mining, or geographical information systems. However, despite the vast research effort MAMs have not made their way into commercial database management systems on a broad scale. This is mostly due to the fact that the integration of these complex data structures into an existing database kernel is fairly complicated. Especially concurrency and recovery issues, which are as important as performance issues for commercial systems, are major obstacles. For most MAMs new solutions to these problems, e.g., locking for R-Trees [KB95, CM98], have to be developed, as the new concepts do not allow reusing standard techniques. This makes the kernel integration of an MAM a very costly task in the range of multiple man-years. As consequence many DBMS producers have not integrated the new technology into their systems, but offer it only as add-on features. So, are MAMs just another nice research gimmick, but commercially not affordable? No, in this paper we will show that there are MAMs, which provide good performance on one side and can smoothly be integrated into a DBMS kernel on the other side. A category of MAMs is based on the combination of one-dimensional index structures and space-filling curves. One prominent example is the UB-Tree [Bay97], which combines the B-Tree and the Z-curve. Together with its sophisticated query processing algorithms it has proven its performance advantages in numerous application domains. Because the UB-Tree is based on the standard B-Tree, which is the basic index structure in almost every commercial DBMS, the task of integrating this MAM into an existing kernel becomes less complex and less costly. The kernel integration of the UB-Tree into TransBase [Tra98] (as part of an ESPRIT project funded by the

European Commission) has been accomplished within one year. TransBase is a full-scale relational database system, which conforms to the SQL-92 standard. TransBase, which handles databases up to 8 Terabyte of data, is used especially in the field of CD-ROM retrieval systems, and has an installation base of well beyond 50,000 sites worldwide.

In this paper, we will present the major issues and problems that have to be tackled and solved for a successful UB-Tree kernel integration. The paper is organized as follows: Section 2 presents the basic concepts behind the UB-Tree and Section 3 deals with the issue of the UB-Tree address representation and the implementation of the standard UB-Tree operations. Section 4 addresses the specific query operation of the UB-Tree and Section 5 tackles the important topic of required optimizer extensions to efficiently support the UB-Tree. Section 6 covers additional enhancements for the UB-Tree and Section 7 presents the performance evaluation. Section 8 summarizes related work and Section 9 concludes the paper.

2 Basic Concept of the UB-Tree

The basic idea of the UB-Tree [Bay97] is to use a space-filling curve to map a multidimensional universe to one-dimensional space. Using the Z-Curve (Figure 2-1a) for preserving multidimensional clustering as good as possible it is a variant of the zkd-B-Tree [OM84].

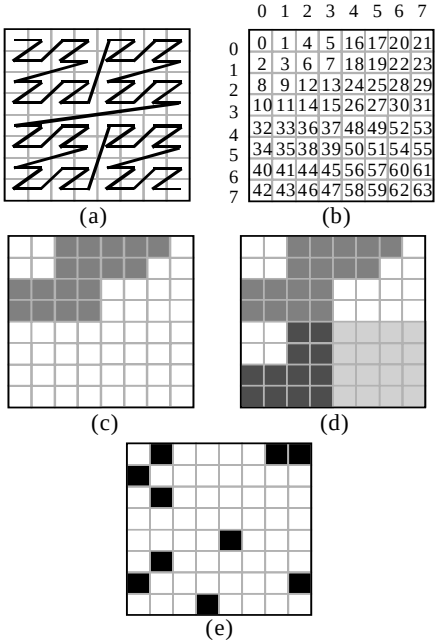


Figure 2-1 Z-Addresses and Z-Regions

A *Z-Address* α = Z(x) is the ordinal number of the key attributes of a tuple x on the Z-Curve, which can be efficiently computed by bit-interleaving (see Section 3.1). A standard B-Tree is used to index the tuples taking the Z-Address of the tuples as keys.

The fundamental innovation of UB-Trees is the concept of *Z-Regions* to create a disjunctive partitioning of the multidimensional space. This allows for very efficient processing of multidimensional range queries (see Section 4). A Z-Region $[\alpha : \beta]$ is the space covered by an interval on the Z-Curve and is defined by two Z-Addresses α and β. We call β the *region address* of $[\alpha : \beta]$. Each Z-Region maps exactly onto one page on secondary storage, i.e., to one leaf page of the B-Tree.

For an 2 dimensional universe of size 8×8, Figure 2-1b shows the corresponding Z-addresses. Figure 2-1c shows the Z-region [4: 20] and Figure 2-1d shows a partitioning with five Z-regions [0 : 3],[4 : 20], [21 : 35], [36 : 47] and [48 : 63]. Assuming a page capacity of 2 points, Figure 2-1e shows ten points, which create the partitioning of Figure 2-1d.

The details of the UB-Tree algorithms are described in the following sections.

3 UB-Tree Address Representation and Standard Operations

For the rest of the paper we will refer to the function computing the Z-Addresses as *UBKEY* and to the keys as *Z-values*. It is important to note that the UB-Tree algorithms can be implemented and integrated without fundamental changes to the query processing of the database kernel. They do not require special tuple handling or other significant modifications, as the following sections show.

3.1 Address Representation and Z-value Computation

An important question for the implementation of the UB-Tree inside the database kernel is how to represent the Z-values. All algorithms for the UB-Tree basically rely on Z-values in the format of variable length bitstrings (trailing zeros are omitted to reduce storage requirements). The operations on Z-values manipulate single bits and copy parts of the bitstring. The UBKEY function can be efficiently implemented, as it requires only reading the specified index attributes bitwise and writing the bits at the corresponding positions in the resulting Z-value. As input the UBKEY function requires a bitstring representation of the attribute values. The natural order $\leq$ of the attribute values in the original domain A has to correspond to the bit-lexicographical order $\leq_{bitstr}$ on bitstrings, i.e., $a_i \leq a_j \Leftrightarrow bitstr(a_i) \leq_{\text{bitstr}} bitstr(a_j)$, where $bitstr : A \rightarrow \{b \mid b \in [0,1]^*\}$ generates the corresponding bitstring. For example, in case of unsigned integers and strings $bitstr := identity$ while for signed integers the *bitstr* function has to take care of the sign bit.

To compute the Z-value of a tuple, we interleave the bits of the bitstring representation of the key attributes (see Figure 3-1).

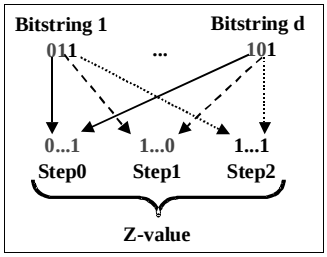


Figure 3-1 Calculation of Z-values by bit-interleaving the transformed attributes

For the following illustration we assume that all bitstrings have the same length *steplength*; *t*[i] returns the i^{th} attribute of tuple *t*; *bitstr*[i] returns the i^{th} bit of the bitstring *bitstr* (the highest order bit has number 0). We partition a Z-value into steps. A *step* consists of all bits from the input bitstrings, which have the same bit ordinal, i.e., step 0 contains the highest order bits from all input values (see Figure 3-1). We require the number of dimensions *d* (we assume that the key attributes are the first *d* attributes of the tuple) of the used UB-Tree. Figure 3-2 shows the pseudo code for the basic bit-interleaving algorithm.

```
Z-value UBKEY(Tuple t){
int i,s;
int bp;          //the bit position in the Z-value
Z-value addr;  //the result Z-value
Bitstring bs[dimno]; // bitstring representation of the attribute values
//Transformation of the key attributes
for (i=0; i < d; i++) {
// transformation of the attribute to a bitstring depends on the attribute type
     bs[i] = TransformAttribute(t[i]);
}
//Bit-interleaving – Calculation of the Z-value
bp=0; //starting with the first bit of the Z-value
//looping first over dimensions then over steps realizes the bit-interleaving
for (s=0;s < steplength; s++) {
     for(i=0; i < d; i++) {
          // the bp^th bit of the Z-value is
          // set to the s^th bit of the i^th bitstring
          addr[bp]=bs[i][s];
          bp++; //advance to next bit of Z-value
          }}
return addr;}
```

Figure 3-2: Basic UBKEY Function

It is also possible to use other transformation functions for generating the bitstring for an attribute value, allowing a more powerful semantics, e.g., soundex codes or case insensitive search on strings. As consequence, the *bitstr* function has to be adapted for the individual data types supported by the UB-Tree. If possible, it is useful to normalize attributes to unsigned integers starting with 0 as one step of the transformation, as this allows for a much better space partitioning with shorter Z-values. One example for normalization for complex data types is MHC [MRB99]. Other possibilities for normalization include hashing or more complex non-linear methods. Note that complex normalization may lead to significant performance overhead, which may then not be neglected any more. Our standard normalization techniques just require a few microseconds of CPU cycles and therefore do not affect the address calculation performance.

3.2 Insertion, Deletion, Update

The basic algorithms of the UB-Tree are handled by the underlying B-Tree. To perform an insertion, deletion or update one just has to compute the Z-value corresponding to the tuple. The underlying B-Tree uses that Z-value to determine the page where the tuple is stored and processes the operation as usual. As consequence, the same performance guarantees as for the basic operations on B-Trees can also be given for the basic operations of the UB-Tree. Figure 3-3 shows the code for insertion, which is similar to deletion and update.

```
Status UBTree_insertTuple(Tuple t)
{
return BTree_insertTuple(UBKEY(t),t); //call to B-Tree standard insertion algorithm, inserting tuple t with key UBKEY(t)
}
```

Figure 3-3: Insertion Function

3.3 Page Splitting

The split algorithm of the underlying B-Tree handles the page splitting in UB-Trees. Only the calculation of the page separator has to be modified, as the page split (i.e., region split for UB-Trees) strategy is crucial for the range query performance of the UB-Tree by influencing the space partitioning. This modification adds no complexity to the split costs as it is done in O(*n*) bit operations, where *n* is the length of the Z-value in bits, and the worst case page utilization of 50% is still guaranteed [Mar99].

The goal of region splitting is to create rectangular regions whenever possible to reduce the number of regions overlapped by a range query. This can be achieved by choosing the *shortest* Z-value (i.e., the Z-value that has as many trailing zero bits as possible) between the two middle tuples *s* and *t* as new separator, instead of *s*, *t* or another Z-value in the middle of the page. Figure 3-4 shows the split algorithm of UB-Trees.

```
Status UBTree_splitPage(Page P, Tuple s, Tuple t) {
Z-value sep;
sep=UBTree_calculateSeparator(UBKEY(s),UBKEY(t));
//Calculating the best separator between two Z-values, i.e., the shortest Z-value between UBKEY(s) and UBKEY(t)
BTree_splitPage(P,s,t,sep);  // Btree_splitPage is the standard algorithm of the Btree: It splits page P between the two tuples s and t with sep as the separator between the two generated pages
}
```

Figure 3-4 Page Split algorithm of UB-Trees

The advantage of this split strategy is twofold: first, better range query performance on average; second, shorter separators lead to a more compact index part of the UB-Tree, a phenomenon also exploited in Prefix-B-Trees [BU77].

4 Range Query Processing

Processing a multidimensional range query, a UB-Tree retrieves all Z-regions, which are properly intersected by the query box [Mar99]. Due to the mapping of the multidimensional space to Z-values, this results in a set of intervals on the B-Tree storing the Z-values (

Figure 4-1).

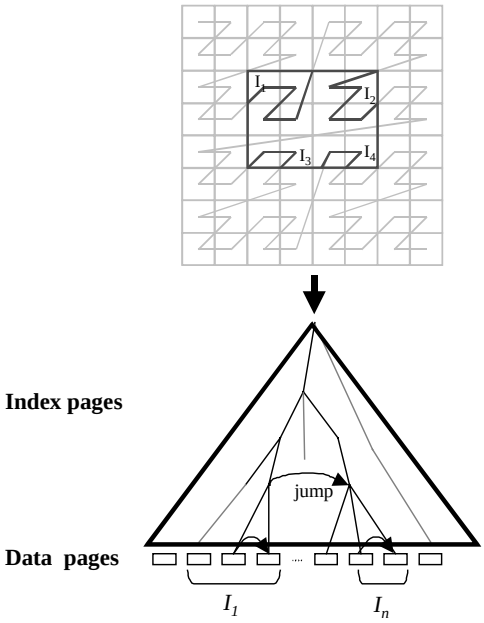


Figure 4-1: How query boxes map to a set of Z-value intervals

A B-Tree answers sets of interval restrictions on its key efficiently by traversing the corresponding intervals of pages directly on the leaf node level. For large B-Trees, i.e., when the index part cannot be completely cached, the processing is improved in TransBase as it supports jumps in the index part of the B-Tree, instead of starting at the root for each interval. Relying on this standard technique, the main task of the UB-Tree range query algorithm is to efficiently calculate the set of one-dimensional intervals of Z-values from one multidimensional interval.

4.1 UB-Tree Range Query Algorithm

One possibility of processing UB-Tree range queries is to compute all Z-value intervals (cf.

Figure 4-1) for a query box in advance before accessing the B-Tree. Due to the nature of the Z-curve this may lead to a large set of intervals, many of which may be located in the same Z-Region. This naive approach will either result in multiple accesses to the same B-Tree page or will require calculation effort to identify all intervals belonging to one page. The resulting processing overhead is avoided by constructing the intervals on the fly, processing the query box page-by-page. As a consequence we use the page-by-page approach for the kernel integration. Iteratively constructing the intervals on the fly also has the advantage of providing the first result tuples earlier, which is good for pipelining. However it requires post filtering of the retrieved pages.

The iterative range query algorithm (see Figure 4-2) for the UB-Tree works as follows. Let the multidimensional range restriction be specified by a query box *Q* with a starting corner and an ending corner, which are given by the two tuples *ql* resp. *qh* (UBKEY(*ql*) < UBKEY(*qh*)). First the algorithm computes the Z-values for *ql* and *qh*, then the region containing *ql* is located[1]. Let *P*, *Q* be two adjacent pages in the UB-Tree and the Z-value *sep* be the separator between the two pages with $\forall t \in P, \forall s \in Q : UBKEY(t) \le sep < UBKEY(s)$. We then call *sep* the *end* or *region address* of the region corresponding to page *P*. The range query algorithm then iteratively determines all the regions intersected by the query box. This is achieved by calculating the Z-value for the next intersection point of the Z-curve with the query box based on the currently processed region/page (see next section).

```
Status UBTree_Range_Query(Tuple ql, Tuple qh) {
Z-value start = UBKEY(ql);
Z-value end = UBKEY(qh);
Z-value cur = start;
While (1) { //continue as long we are in the query box
     cur = getRegionSeparator(cur); // getting the address of the region containing cur
     FilterTuples(GetPage(cur), ql, qh); //post-filtering of the tuples in the region
     if (cur >= end) break; //stop once we covered the whole query box
     cur = getNextZvalue(&cur, start, end); //calculation of next region
}}
```

Figure 4-2: Range Query Algorithm

All these steps are performed in *O(n)* bit operations where *n* is the length of the Z-value [Mar99]. The separator computation and fetching of the page can be performed by one B-Tree access. After filtering out all matching tuples of a fetched page, they can be piped for further processing by the DBMS.

4.2 Calculating the next intersection point

Calculating the next intersection point is the crucial part of the UB-Tree range query algorithm. In the following we will show that this step only requires bit operations on Z-values and no I/O or B-Tree search is necessary. Starting point for getNextZvalue is the region address *cur* of the current region. The task now is to find the next intersection point of the Z-Curve with the query box *Q*. This intersection point *nisp* is the minimal Z-value larger than the current region address, which is inside Q, i.e., $nisp = \min\left(\left\{UBKEY(x) \middle| UBKEY(x) > cur \wedge (x \in Q)\right\}\right)$

[1] Note: *ql* and *qh* do not have to be tuples existing in the database – it is only important to which regions they are mapped according to the space partitioning.

Figure 4-3 illustrates two examples of the next intersection point calculation for the dotted query box *Q*.

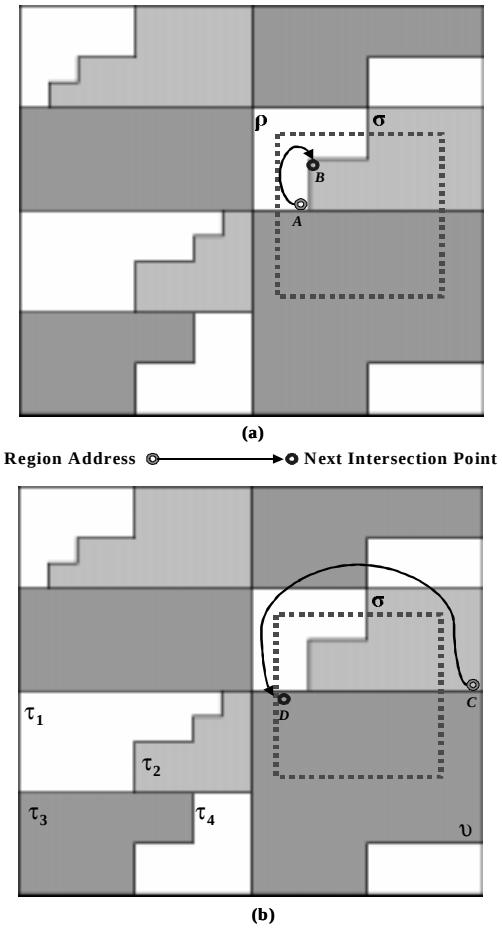


Figure 4-3 Region Address and Next Intersection Points

In Figure 4-3a, the region address *cur=A* of region ρ yields *B* of region σ as the next intersection point with *Q*. Consequently the range query algorithm will continue with processing region σ. In this case, *B* is the direct successor of *A* on the Z-Curve, i.e., *B=A*+1. In Figure 4-3b we are looking for the next intersection point for the region address *cur=C* of region σ. The call to getNextZvalue yields the Z-value *D* causing to process region υ next, skipping the four regions τ_1, τ_2, τ_3, and τ_4.

Informally, the algorithm determines for the successor of a region address in which dimensions it is not contained in the query box. From this information it identifies the bits that have to be modified to generate the correct next intersection point.

For the detailed description of the algorithm we introduce the following functions: BP(*i,s*) returns for each bit position/step of a dimension *i* to which bit position in the resulting Z-value it corresponds; given a bit position *bp* in a Z-value the functions DIM(*bp*) and STEP(*bp*) will return the corresponding dimension resp. step.

The first step is to increment the region address by one, i.e., *nisp=cur*+1. We then test if *nisp* is in the query box by bitwise comparing with the Z-values of *ql* and *qh* – we do not have to transform *nisp* back to Cartesian coordinates. During the comparison we also determine additional information for each dimension *i*:

$$flag[i] = \begin{cases} -1 & \text{if } nisp \text{ has fallen below the minimum of } qb \text{ in dimension } i \\ 0 & \text{if } nisp \text{ is in } qb \text{ in dimension } i \\ 1 & \text{if } nisp \text{ has exceeded the maximum of } qb \text{ in dimension } i \end{cases}$$

$outStep[i]$ | the step in dimension i where qb has been left; ∞ if $nisp$ in qb in dimension i

$saveMin[i]$ | the step in dimension i where the minimum has been exceeded

$saveMax[i]$ | the step in dimension i where $nisp$ has fallen below the maximum

If *nisp* is in *qb* then we have already found our next intersection point. If not, we have to determine the bits we have to set to 1 and set to 0 to get the correct *nisp*. Let be $outStep = \min(outstep[i])$, and *d* the corresponding dimension. We have to distinguish two cases: first, if *flag*[*d*]=-1 then we have found the bit *changeBP=BP*(*d*,*outstep*) that we can safely set to 1 such that *nisp* > *cur*. Second, if *flag*[*d*]=1 then we have to find a lower bit position in *nisp* we can set to 1, because the bit specified by *outstep* has to be set to 0. In both cases the bits following the changed bit have to be adapted accordingly (see Figure 4-4).

```
BP changeBP=BP(d,outstep); //we start with the minimal bit position that has to be changed
int i;
if (flag[d] == 1) // we cannot set this bit to 1, therefore we have to find a lower bit position we can safely set
then {
       changeBP=max({bp|bp<changeBP and bp >= saveMax[Dim(bp)] and Val(nisp,bp)=0}); //maximal bitposition that is save to set to 1
       saveMin[DIM(changeBP)]=STEP(changeBP);
       flag[DIM(changeBP)]=0;
       }
// now we can change the rest of the Z-value
for(i=0;i<dimno;i++) { //for each dimension we determine how to change the bits
if(flag[i]>=0) // we have not fallen below the minimum in this dimension
then {
     if(changeBP > BP(i,saveMin[i]))
     then "set all bits of dim with bit positions > changeBP to 0"
     else "set all bits of dim with bit positions > changeBP to the minimum of the query box in this dim"
}
else { // if we have fallen below the min in this dimension the lowest possible value is the min itself
     "set the bits to the minimum of the query box in this dim"
}}
```

Figure 4-4 Pseudo code for parts of getNextZvalue

5 Query Engine Extensions

One of the major benefits of a kernel integration of the UB-Tree is the tight integration with the query engine. In the following we will discuss the most important issues that enable the query engine to use the UB-Tree in the most efficient way.

5.1 General Extensions

Supporting a new index method usually requires extension of schema information and DDL of the DBMS. For the UB-Tree the database catalog does not have to store additional information except the transformation function for each key attribute. The rest is also required by other index structures, for example, the index attributes, the number of dimensions, the domain of each attribute, etc.

A new storage clause (CLUSTERING UBTREE ON {<set of attributes>}) in the DDL statement for creating a table specifies the creation of a UB-Tree index. Additionally, specifying the actual domains for the attributes indexed by the UB-Tree by check constraints allows for optimal multidimensional clustering. Figure 5-1 shows the creation of a three-dimensional UB-Tree with two additional attributes.

```
CREATE TABLE fact(
date DATETIME [YY:DD] CHECK(date BETWEEN '1997-
1-1' AND '2020-12-31',
region INTEGER CHECK(region BETWEEN 0 AND 2047),
product INTEGER CHECK(product BETWEEN 0 AND
999999),
price NUMERIC (12,2),
quantity NUMERIC(8,2),
PRIMARY KEY (product, date, region),
CLUSTERING UBTREE ON {product, date, region})
```

Figure 5-1 Create statement for a UB-Tree

With the kernel integration the UB-Tree query functionality is hidden by the standard SQL interface, i.e., no extension of the DML is required. The extensions of the query engine, especially of the optimizer, will take care of the appropriate usage of the new index, e.g., processing a multidimensional range query on the UB-Tree if possible.

5.2 Handling Multidimensional Range Queries

One of the big advantages of the UB-Tree is the efficient processing of multidimensional range queries. State-of-the-art query engines usually only extract the most selective predicate from the query to pass it down to the selected index. In case of the UB-Tree the query engine has to take care of generating a suitable query box or a set of query boxes from the query predicate. This requires passing down a more complex structure instead of a single range to the underlying index module. We will specify an optimizer rule that allows for generating a query box from the query. We introduce the physical operator $RQ(R, qb)$ that represents a range query specified by the query box *qb* on a relation *R*. Let *A* be the set of attributes of *R* with attribute $A_i \in A$ having the domain $[\min_i ; \max_i]$. The query box *qb* specifies for each attribute the restricted range, i.e., $qb = \begin{pmatrix} \dots & \dots \\ ql_i & qh_i \\ \dots & \dots \end{pmatrix}$ for $A_i \in [ql_i ; qh_i]$[2]. For each $A_i \in A, S(A_i, qb)$ specifies the selectivity of attribute A_i in *qb*.

Each restriction on a table specified by the predicate Ψ in the WHERE clause is represented by the logical operator $\sigma_\Psi(R)$. This is transformed to $\sigma_\Psi(R) = \sigma_\Xi(\sigma_P(R))$ with P corresponding to the set of multidimensional query boxes on *R*, i.e., $P = \rho_1 \vee \dots \vee \rho_m$ where $\rho_i = \bigcap_{j=1}^{n} A_j \in [a_j^i ; b_j^i]$, and Ξ corresponding to the restrictions, which cannot be mapped to multidimensional query boxes, e.g., $A_i < A_j$. We can therefore write: $\sigma_\Psi(R) = \sigma_\Xi(\bigcup_i \sigma_{\rho_i}(R))$.

Example: The SQL statement

SELECT count(*)
FROM R
WHERE A_1 BETWEEN 0 AND 10 AND A_2 IN [3,12,31] AND A_3 BETWEEN 3 AND 9

leads to the predicate describing 3 query boxes:

$$P = (A_1 \in [0;10] \wedge A_2 \in [3,3] \wedge A_3 \in [3;9]) \vee (A_1 \in [0;10] \wedge A_2 \in [12,12] \wedge A_3 \in [3;9]) \vee (A_1 \in [0;10] \wedge A_2 \in [31,31] \wedge A_3 \in [3;9])$$

As the example shows, the identification of query boxes from arbitrary predicates is a complex problem by its own the query optimizer needed to be enhanced for. The TransBase optimizer already recognized a special subset of multidimensional intervals, namely those, which can be directly processed by B-Trees intervals. This greatly facilitated the extension of the optimizer to general multidimensional intervals.

Let $P \subseteq A$ be the set of attributes of *R*, which are specified in ρ_i. We can specify the following rule to create the corresponding query box:

REPLACE $\sigma_{\rho_i}(R)$ BY $RQ(R, qb)$

with $ql_i = \begin{cases} a_i & \text{if } A_i \in P \\ \min_i & \text{otherwise} \end{cases}$ and $qh_i = \begin{cases} b_i & \text{if } A_i \in P \\ \max_i & \text{otherwise} \end{cases}$

[2] Note: In this paper we only deal with closed intervals ($\geq, \leq$) for restrictions. Other restrictions ($>, <$), which cannot be mapped to closed intervals (e.g., for non-discrete domains), are handled with appropriate post filtering.

Given such a query box, the optimizer then can decide which access method to use to answer the query.

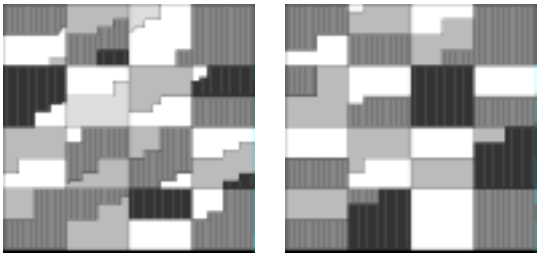
6 UB-Tree Enhancements

This section deals with some enhancements of the UB-Tree and its algorithms, which have not been integrated into the TransBase kernel yet. These improvements are not necessary for a successful integration of the UB-Tree, but they provide further performance optimizations.

6.1 Optimizing Space Partitioning for Range Queries

The UB-Tree range query performance is enhanced by a modification of the page splitting algorithm leading to a better space partitioning. The so-called *ε-Split* chooses the split point in an interval of ε% of the page capacity around the middle of the page and not directly in the middle.

That is, it chooses the optimal split point according to the space partitioning in a certain range of tuples around the middle of the page, but not only between the two tuples in the middle. The cost for the optimal splitting is the reduced page utilization guarantee of $50\% - \varepsilon\%$. The enhanced split algorithm requires the two tuples, which are $\varepsilon\%$ away to the left and to the right from the middle of the page, whereas the standard version takes just the two tuples to the left and right of the page middle. Figure 6-1 shows the same two-dimensional UB-tree with standard split point calculation and with *ε-Split* ($\varepsilon = 14\%$): the effect is clearly visible.



(a) w/o ε-Split (b) with ε-Split

Figure 6-1: Influence of Epsilon-Split on the Space Partitioning

The optimal choice of ε presents a tradeoff between storage utilization and quality of space partitioning: the higher ε the better the space partitioning and as consequence the range query performance but the lower the worst-case storage guarantee. However, empirical results (Figure 6-2 shows the results of 6-dimensional range queries with varying volume and location on a 6D UB-Tree for growing ε) have shown that already a small ε (around 5%) leads to significant improvement of the space partitioning.

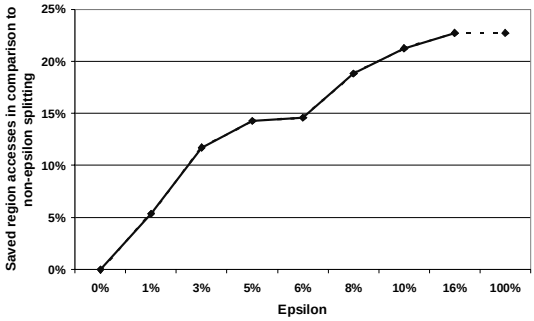


Figure 6-2 Influence of Epsilon Splitting on Range Query Performance

For $\varepsilon > 15\%$ no significant improvements in the space partitioning are observed. It is important to note that the *ε-Split* has the same complexity as the regular split algorithm, and that for the individual UB-Tree ε is a performance tuning parameter.

6.2 Dealing with multiple query boxes

Often complex queries do not only lead to one query box, but to a set of query boxes on the multidimensional space. Processing the set of query boxes sequentially with the UB-Tree range query algorithm may lead to unnecessary page accesses in the cases where query boxes overlap the same regions. As consequence, performance problems arise if the already accessed page cannot be cached over the total processing of the query box set. The algorithm presented in [FMB99] solves this problem by handling all query boxes simultaneously. It guarantees that each region/page is accessed only once, leading to significant performance improvement over the standard UB-Tree range query algorithm in case of query boxes overlapping many pages together.

6.3 Reducing Post Filtering of the Range Query Algorithm

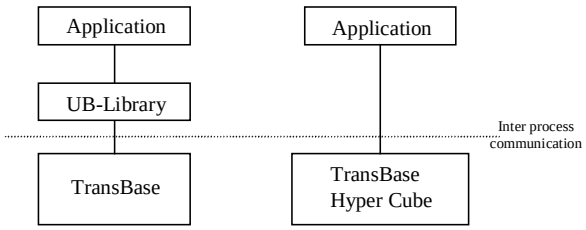
Post filtering is only necessary for regions not fully contained within the query box. We provide an algorithm similar to getNextZvalue and with the same complexity, which determines for a region if it is completely overlapped by a query box or not. This is an important performance optimization for large query boxes as most regions will be completely inside of the query box.

7 Performance Evaluation

In this section we provide some measurement results that show the performance gains that are achieved by the kernel integration of the UB-Tree into the TransBase database system (called TransBase Hyper Cube, Figure 7-1b). We are comparing with our prototype implementation of the UB-Tree (called UB/API, Figure 7-1a) and the native TransBase B-Tree. Our previous comparisons of the UB-Tree with competing index methods of TransBase and other RDBMSs [MZB99, MRB99], based on the UB/API, yielded significant

performance improvements. The speed-up achieved by the kernel integration applies directly to the results of our previous comparisons, yielding a speed-up of several orders of magnitude of UB-Trees compared to the methods used in these papers.

Our prototype implementation of the UB-Tree is realized as an application on top of an existing DBMS using the standard SQL interface (currently TransBase, Oracle, Informix, SQL Server 7, and DB2 are supported). The leaf pages of the UB-Tree are stored as single tuples in a relation and the index part is mapped to the B-Tree index of the underlying system.



(a) UB/API (b) TransBase Hyper Cube

Figure 7-1: Implementation as UB/API versus TransBase Hyper Cube

The measurements were conducted on a real world data warehouse provided by one of our project partners, a market research company. The measurements consist of 604 real world reports on a three dimensional (time, products, segments) star-schema with a total of around 4GB of data. The reports represent typical analysis of product hitlists in a given period, market share, market trends, and the like, which result in three-dimensional range queries on the fact table.

We created three instances of the fact table: two indexed by UB-Trees (integrated and external), and one indexed by a compound B-Tree on the three dimension keys. From the reports we only measured the fact table access, as the further processing is identical for all methods. Table 7-1 shows the measurement results over all 604 reports.

On average, the integrated UB-Tree is about 2-3 times faster than UB/API (see Table 7-1). As mentioned above, this speed up applies directly to the performance advantage over both clustering compound B-Trees as well as index intersection of multiple secondary B-Trees or bitmap indexes.

	Speedup Factor UB/API / HCI	**Speedup Factor Compound/HCI**
Average	2,8	11,5
1. Quartil	2,3	8,5
Max	3,0	5,0
Min	3,0	2,3
3. Quartil	2,8	13,2

Table 7-1 Summary of Report Results (average response times in seconds)

The significant performance improvement of the kernel integration can be explained by the design of UB/API. First, each fetch of a page requires one SQL statement; this leads to heavy client/server communication together with the DBMS overhead of processing the query, which sums up to about 30% of the total processing time of a range query. Second, we had to implement some database functionality, like page and tuple handling, post filtering etc., which causes additional overhead. Putting all together, the above mentioned speedup factor fits our expectation.

8 Related Work

Integrating new index methods into a database kernel is often regarded as a too costly task. On the other side, database vendors have recognized the need for more flexible, powerful indexing methods, often tailored to specific application domains. As consequence, most vendors have extended their standard B-Trees and provide interfaces that allow the users to include their own functions for the index key computation. Some systems even allow the user to implement their own index structures in external modules. We will point out the deficiencies of these approaches in this section.

8.1 Function-based B-Trees

For standard B-Trees the key of the index consists of a subset of the attributes of the underlying table *T*. A more general idea is to use a function to compute the key values for the tuples $t \in T$. We will refer to this type of B-Trees as *function-based B-Trees* or short B^F-Trees.

Example: A standard B-Tree is a special instance of the B^F-Tree where *F* is the projection of the key attributes from the tuple. B-Trees storing SOUNDEX codes or case-insensitive keys are other well-known examples.

B^F-Trees were motivated by the need to support indexing on user-defined types in object-relational systems. Commercial implementations are provided for example by function-based indexes in Oracle8i [Ora99], functional indexes in Informix [Inf99], the high level indexing framework of IBM DB2 [CCF+99], or as indexes on computed columns in MS SQL Server 2000 [MS00]. However, B^F-Trees do not allow for the integration of new query algorithms, like the UB-Tree range query algorithm. Therefore, implementing the UB-Tree as a B^F-Tree with UBKEY as *F* will not lead to the expected performance.

8.2 Extended Index Interfaces

Some commercial database management systems provide even more enhanced indexing interfaces, which allow for implementation of arbitrary index structures by the user in external modules (e.g., Extensible Indexing API by Oracle [ORA99], Informix Datablade API [Inf99]). Analogous to the GiST framework (see next section), the user has to provide a set of functions/operators that are used by the database server to access the index. The index

itself can be either stored inside the database (e.g., as an IOT in Oracle) or in external files. The problem of these index interfaces is threefold: performance of the index, optimizer support, and locking and recovery. The performance problem of extended index interfaces has two aspects: first, only non-clustered indexes are supported. Index structures, whose performance is achieved by appropriate clustering, like the UB-Tree that clusters according to multiple dimensions, can therefore not be implemented via these interfaces. In addition, as the DBMS internal modules cannot be used, efficient page and tuple handling has to be implemented. This leads to significant coding effort for the index implementation. The coupling of the external index with the query optimizer is achieved by providing cost functions for the index operations. However, to our knowledge, there is no way to add rules to guide the optimizer with heuristics, which is very important to achieve optimal query plans. Another significant drawback of these 'add-on' approaches is the handling of locking and recovery. The external indexes are not tightly coupled with the DMBS locking and recovery services. As consequence, the index implementation has to take care of recovery issues itself [RSS+99], and the lock granularity is often the complete index itself.
Taking all these aspects into account, in case of the UB-Tree the kernel integration is much more favorable than an implementation as an external index.

8.3 GiST – Framework

The General Search Tree (GiST) approach of [HNP95] provides a single framework for any tree-based index structure. The GiST framework provides the basic functionality for trees, e.g., insertion, deletion, splitting, search, etc. The individual semantics of the index are provided by the user with a key class, which implements six key functions the basic functions rely on. As consequence, the user has only to change a small part of the code to implement various index methods (e.g., B^+-Trees, R-Trees). In general, the UB-Tree fits perfectly into the GiST framework, but efficient implementation would require more user control at two points: search algorithm and splitting. The major drawback of the first GiST approach is the fixed query functionality – the user cannot adapt the search algorithm to the specific indexing technique, which in many application scenarios will lead to significant performance problems. The UB-Tree range query algorithm is one example of such a search algorithm. The extension of [Aok98], which gives the user the control of the tree traversal during search, should suffice for an efficient range query implementation. With respect to splitting, the ε-Split (see Section 6.1) cannot be implemented as long as the Penalty function of GiST only takes two tuples (entries) of a page into account, but not a set of tuples. Putting all together, if the extended GiST framework is available, the benefits described in [Kor99] apply to the UB-Tree integration as well.

9 Summary and Conclusion

Multidimensional access methods are not widely supported by commercial database management systems despite their performance impacts in various application domains. This is mostly due to the fact that a kernel integration of these sophisticated data structures is considered to be a very costly and complex task. In this paper we have shown that this is not the case for the UB-Tree, as it heavily relies on the well-known B-Tree, reducing the complexity of the additional algorithms to a minimum. The UB-Tree was integrated within one year, and by now TransBase Hyper Cube is a commercially available product. Figure 9-1 shows the changes of the individual database kernel modules required by the UB-Tree integration.

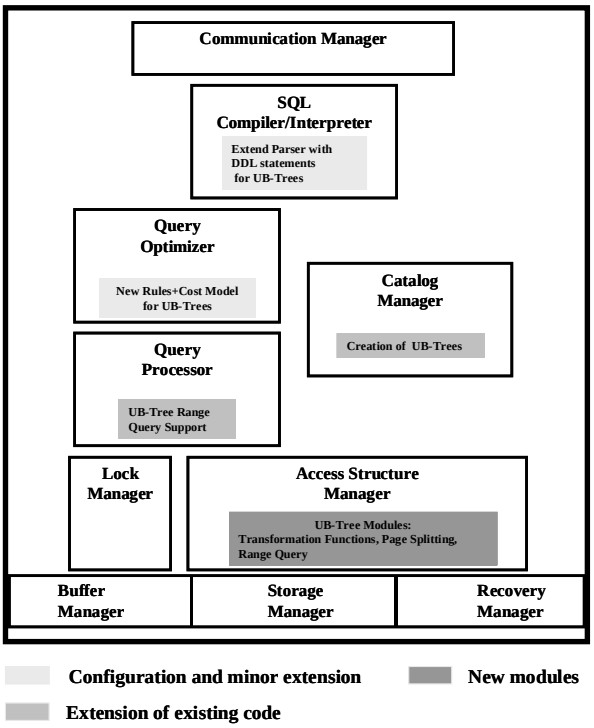


Figure 9-1 Affected Modules of TransBase

The shaded boxes mark the modifications in the single modules, where the darker shading signals the more complex modifications. The performance gains show that the effort of the integration is worth it: with up to a factor of 3 faster than the prototype implementation on top of a DBMS, the kernel integrated UB-Tree provides significantly better performance than traditional access methods in various application domains. The big advantage of the kernel integration in comparison with other approaches is the tight coupling with the query optimizer. This allows for optimal usage of the UB-Tree.
As we do not rely on any TransBase specific features for the UB-Tree algorithms, we expect the same integration effort for other database systems as well as long they provide clustering B-Trees on computed keys.

An issue not addressed in this paper is the question, when to use the UB-Tree for a specific application scenario and which UB-Tree organization (e.g., number of dimensions) is optimal. This problem is addressed in what we call *physical data modeling* and we are in the progress of identifying design rules for optimal indexing strategies with UB-Trees.

Summarizing our experiences, UB-Trees smoothly integrate into the indexing engine and extend the B-Tree concept in order to handle multiple dimensions symmetrically. They are extremely useful for both clustering tables as well as covering secondary indexes (secondary indexes that contain all attributes required by a given query) to speed up multidimensional range queries. Extending the optimizer allows for good combination with single attribute access methods in physical data modeling and query processing, which will lead to efficient and flexible index schemes.

Acknowledgements

We thank our project partners the European Commission, Teijin Systems Technology, and Microsoft Research for funding this research work. We also thank our master student Stephan Merkel for his effort in doing the performance measurements reported in this paper. In addition we thank Goetz Graefe for his constructive comments.

References

[Aok98] P. M. Aoki. *Generalizing "Search" in Generalized Search Trees.* Proc. of ICDE 1998.

[Bay97] R. Bayer. *The universal B-Tree for multidimensional Indexing: General Concepts.* World-Wide Computing and Its Applications '97 (WWCA '97). Tsukuba, Japan, 10-11, Lecture Notes on Computer Science, Springer Verlag, March, 1997.

[BSS+99] R. Bliujute, S. Salentis, G. Slivinskas, and C.S. Jensen. *Developing a DataBlade for a New Index*. Proc. of ICDE 1999.

[BU77] R. Bayer and K. Unterauer. *Prefix B-Trees*. ACM TODS 2(1), 1977, pp. 11-26.

[CCF+99] W. Chen, J.-H. Chow, Y.C. Fuh, J. Grandbois, M. Jou, N. Mattos, B. Tran, and Y. Wang. *High Level Indexing of User-Defined Types.* Proc. of 25th VLDB, Edinburgh, Scotland, 1999.

[CM98] K. Chakrabarti and S. Mehrotra. *Dynamic Granular Locking Approach to Phantom Protection in R-Trees*. Proc. of ICDE, 1998.

[FMB99] R. Fenk, V. Markl, and R. Bayer. *Improving Multidimensional Range Queries of non rectangular Volumes specified by a Query Box Set*. Proc. of International Symposium on Database, Web and Cooperative Systems (DWACOS), Baden-Baden, Germany, 1999

[HNP95] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. *Generalized Search Trees for Database Systems*. Proc. of 21st VLDB, Zurich, Switzerland, 1995.

[HR96] E.P. Harris, and K. Ramamohanarao. *Join algorithm costs revisited*. VLDB Journal, 5, 1996

[Inf99] Informix Software Incorporation. *Informix Dynamic Server with Universal Data Option Version 9.1.X Documentation*. 1999.

[LKC99] J.-H. Lee, D.-H. Kim, C.-W. Chung. *Multi-dimensional Selectivity Estimation Using Compressed Histogram Information*. Proc. of SIGMOD99, Philadelphia, U.S.A., 1999

[KB95] M. Kornacker and D. Banks. *High-Concurrency Locking in R-Trees*. Proc. of 21st VLDB, Zürich, Switzerland, 1995.

[Kor99] M. Kornacker. *High-Performance Extensible Indexing*. Proc. of the 25th VLDB, Edinburgh, Scotland, 1999.

[Mar99] V. Markl. *MISTRAL: Processing Relational Queries using a Multidimensional Access Technique*. Ph.D. Thesis, Technische Universität München, 1999.

[MRB99] V. Markl, F. Ramsak, and R. Bayer. *Improving OLAP Performance by Multidimensional Hierarchical Clustering*. Proc. of IDEAS'99, Montreal, Canada, 1999.

[MS00] Microsoft Coperation. *SQL Server 2000 Books Online*. 2000.

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

[OM84] J. A. Orenstein and T.H. Merret. *A Class of Data Structures for Associate Searching*. Proc. of ACM SIGMOD-PODS Conf., Portland, Oregon, 1984, pp. 294-305.

[Ora99] Oracle Corporation. *Oracle 8i Server, Release 8.1.5 Documentation*. 1999.

[PI97] V. Poosala, and Y.E. Ioannidis. *Selectivity Estimation Without the Attribute Value Independence Assumption*. Proc of the 23th VLDB, 1997

[Tra98] TransAction Software GmbH. *TransBase Documentation*. 1998.

[WKW94] K.Y. Whang, S.W. Kim, G. Wiederhold. *Dynamic Maintenance of Data Distribution for Selectivity Estimation*. VLDB Journal Vol. 3, No. 1, 1994