Interval Processing with the UB-Tree [*]

Robert Fenk[*] Volker Markl[+] Rudolf Bayer[*]

[*]Bavarian Research Center for Knowledge Based Systems
Orleansstrasse 34, 81667 Munich, Germany
fenk@forwiss.de
bayer@in.tum.de

[+]IBM Almaden Research Center
K55/B1, 650 Harry Road
San Jose, CA 95120-6099, USA
marklv@us.ibm.com

Abstract

Advanced data warehouses and web databases have set the demand for processing large sets of time ranges, quality classes, fuzzy data, personalized data and extended objects. Since, all of these data types can be mapped to intervals, interval indexing can dramatically speed up or even be an enabling technology for these new applications.

We introduce a method for managing intervals by indexing the dual space with the UB-Tree. We show that our method is an effective and efficient solution, benefitting from all good characteristics of the UB-Tree, i.e., concurrency control, worst case guarantees for insertion, deletion and update as well as efficient query processing. Our technique can easily be integrated into an RDBMS engine providing the UB-Tree as access method. We also show that our technique is superior and more flexible to previously suggested techniques.

Keywords: *interval management, parameter space, UB-Tree, intersection query*

1 Introduction

The management and processing of intervals is a special case of extended object handling with growing demand in various application areas.

Applications requiring interval matching and management include:

Temporal Databases [SOL94]; Quality Classes, Personalization and Fuzzy Logic/Matching where intervals can be utilized to describe the problem; Spatial Data where a spatial object can be approximated by a bounding box or set of intervals on a space filling curve. [FR89, BKK99, KMPS01]

For point data there are only a few well defined query types, e.g., *point query* and *range query*, but for intervals there are plenty different query types, e.g., the 13 Allen Relations [AH85] for temporal data.

Following [GG98] the basic query types we consider are: **Exact Match Query (EMQ):** Checks if the data base contains an interval which exactly matches the query interval; **Intersection Query (IQ):** Find all intervals which have at least one point in common with the query interval; **Point Query (PQ):** Find all intervals containing a certain query point; **Containment Query (CQ):** Find all intervals enclosed by the query interval; **Enclosure Query (EQ):** Find all intervals enclosing the query interval.

The IQ plays a crucial role in the calculation of most other relations as an efficient filter for a candidate set. But, depending on the application field additional query types might be necessary, e.g., the *Extent Match Query*, which finds all intervals with a given extent, *Adjacency Query* (meets, precedes), which finds all neighbours of a given interval, and the *Nearest Neighbour Query*, which finds the nearest object to a given object. While some of them (e.g., meets, precedes, before, etc.) can be mapped to intersection queries and post filtering, others (e.g., *Extent Match Query*) require different algorithms or indexing techniques in order to run efficiently.

Standard relational database management systems (RDBMS) so far do not support interval objects or spatial objects efficiently. Enhancing RDBMS by new indexing methods usually implies restrictions in concurrency control and recovery. Having a separate index structure which has

[*]Copyright 2002 IEEE. Published in the Proceedings of IDEAS 2002 in Edmonton, Canada. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA.

not been integrated into the DBMS kernel results in applications which are more complex to develop and maintain. So the only economically reasonable solution to tackle interval indexing is to intelligently exploit the techniques provided by a DBMS. Recently new indexing techniques like the RI-Tree [KPS00, KMPS01] address this problem.

The contribution of our paper is to show the feasibility of the the parameter space approach when indexing it with the UB-Tree, allowing for a flexible and highly-performant interval access method. We show that the data distribution of the dual space is handled well by the UB-Tree, that the typical queries on intervals map to multidimensional range queries on the UB-Tree and compare the performance of our approach to prior art in interval indexing.

In order to implement and use interval indexing with a commercial RDBMS, we rely on existing indexing methods already provided by the DBMS. For that reason we compare the parameter space approach indexed by the UB-Tree, a multidimensional access method (MAM) provided by TransBase Hypercube DBSM, [TAS00] with the RI-Tree which also can be implemented on top of an existing B-Tree, e.g., the B-Tree provided by TransBase. We do not discuss the integration of the UB-Tree, since this has been already done in [RMF$^+$00].

The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 introduces the concepts and techniques of the parameter space and Section 4 describes how the UB-Tree can be used to index the parameter space. Section 5 describes the RI-Tree in more detail as this is the interval indexing technique we are comparing with. In Section 6 we present our performance analysis. Section 7 describes how to integrate our interval processing method into a DBMS featuring a B-Tree. Finally we conclude our paper with Section 8.

2 Related Work

A huge number of different techniques for interval management has been proposed. [KS91, TCG$^+$93, Boz98, MTT00] give surveys on them. We only consider secondary storage index structures, since main memory data structures usually can not be mapped directly to existing DBMS technology, e.g., the external Segment-Tree [BG94] is a nontrivial mapping of the Segment-Tree.

Further more we want to focus on indexing structures which can be used by exploiting the techniques of commercial RDBMSs, e.g., indexes like the B-Tree or UB-Tree. Therefore, we do not consider [SOL94, GLOT96, KS91] which require indexing techniques not available in any commercial DBMS.

[Ore90] utilizes an approach quite similar to ours, also featuring a Z-curve based access method and the parameter space transformation. However, he focuses on spatial objects and spatial joins, but not on intervals.

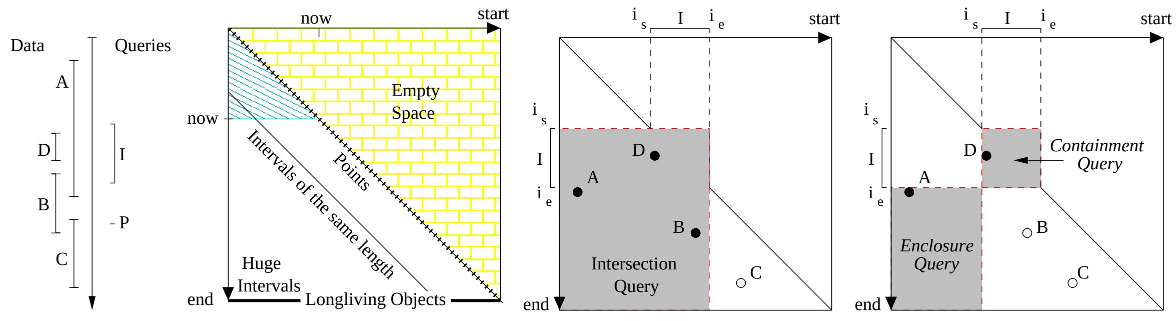
The use of regular B$^+$-Trees for indexing valid time intervals was suggested in [GLOT96]. Here, the intervals are mapped to two-dimensional points with the same mapping function used for the TP-index [SOL94]. These two-dimensional points are mapped back to one-dimensional points (not intervals) by defining a total order among them using either horizontal, vertical, or diagonal sweep lines. B$^+$-trees are used to index these points after the final transformation. Temporal queries also go through these transformations. In this scheme, some specific temporal queries transform into range search queries for the B$^+$-trees, and can be efficiently evaluated. However, because of the transformations, many queries require multiple range search operations, and cannot be handled efficiently.

The SR-tree (Segment R-tree) [KS91] is a variant of the R-tree to index segments. Unlike the R-tree, the SR-tree keeps data also in the internal nodes. Any segment that spans any of the children of a node is kept in that node and is checked every time that node is visited in a search query. The SR-tree is a dynamic index, i.e., it allows deletions and insertions at any time. However, insertion and deletion algorithms may cause a high degree of overlap between the nodes. One should also mention that, although insertion and deletion times are logarithmically bound, they are relatively more expensive compared to index structures such as the B$^+$-/UB-Tree. The same drawbacks hold also for the R-Tree and the R*-Tree. Moreover, no integrated R-Tree offers the excellent concurrency and maintenance properties of the B-/UB-Tree. Furthermore all the R-Tree variants integrated in commercial DBSMSs are secondary indexes and consequently they cannot perform as good as clustering indexes like the B-Tree or the UB-Tree.

The Relational Interval Tree (RI-Tree) [KPS00, KMS01, KMPS01] utilizes a virtual binary tree to partition the data space and group intervals to nodes of the binary tree. The node values are used as an additional attribute of the relation storing the intervals. Standard DBMS indexes (e.g., B-Trees) are used to index the relation. The intersection query maps to a SQL statement and can be processed efficiently by the DBMS. Due to its design, the integration into a DBMS or application is quite easy.

3 The Parameter Space

Simple geometric shapes can be considered as points in higher dimensional space called the *parameter space* [Ore90, GG98]. This doubles the number of dimensions and therefore it is also common to use the term *dual space* for this approach.



(a) Data and Queries

(b) Properties

(c) IQ

(d) ECQ and CQ

Figure 1. Native Space and Mapping of Intervals and Queries to Parameter Space

3.1 Data Transformation

The required transformation of an interval interprets the beginning and end of the interval as the coordinates of a point in two dimensional space. This transformation is called *endpoint transformation* and results in a parameter space with two dimensions of equal domain.

$$\begin{array}{ccc} \text{[start,end]} & \xrightarrow[\text{Transformation}]{\text{End Point}} & \text{(start,end)} \\ \text{1D Interval} & & \text{2D Point} \end{array}$$

Fig. 1 shows the *endpoint transformation* of the intervals A, B, C and D. The horizontal axis denotes the start point of the intervals, the vertical axis denotes the end point.

This transformation results in the following properties of the parameter space, which are depicted in Fig. 1(b):

- the space above the main diagonal is empty, i.e., due to $s \leq e$ no points are mapped to this space
- points are placed on the main diagonal, i.e., $s = e$
- intervals of the same length are mapped to a diagonal, i.e., $e = s + \text{length}$
- the bigger a interval gets the nearer it is to (min, max)
- intervals within a given range are mapped to a rectangular subspace, i.e., they can be clearly separated from the rest. e.g., when indexing time intervals we can get old data (before now) by the query box $[(min, min), (now, now)]$. See also CQ in the next section.
- temporal databases: long-living objects are mapped to $(start, max)$, e.g., time intervals where the end is not known

3.2 Query Transformation

We define the universe of intervals to be $U = \{[u_s, u_e] | min \leq u_s \leq u_e \leq max\}$. Fig. 1(a) on page 3 shows a set of intervals $\{A, B, C, D\}$, a query point P p and a query interval I $[i_s, i_e]$.

Mapping of the basic queries to parameter space results in 2d query boxes and can be formally defined as follows:

Exact Match Query: EMQ($[i_s, i_e]$) = $\{[r_s, r_e] | r_s = i_s \wedge r_e = i_e\}$ is a point query in parameter space; **Intersection Query:** IQ($[i_s, i_e]$) = $\{[r_s, r_e] | (r_s \leq i_e) \wedge (i_s \leq r_e)\}$ can be mapped to a query box $[(min, i_s), (i_e, max)]$ by adding the lower and upper bounds of the domain. Fig. 1(c); **Point Query:** PQ(p) = $\{[r_s, r_e] | r_s \leq p \leq r_e\}$ is actually a special case of IQ and results in the query box $[(min, p), (p, max)]$. **Containment Query:** CQ($[i_s, i_e]$)] = $\{[r_s, r_e] | (i_s \leq r_s \leq i_e) \wedge (i_s \leq r_e \leq i_e)\}$ maps to the query box $[(i_s, i_s), (i_e, i_e)]$. Fig. 1(d); **Enclosure Query:** EQ($[i_s, i_e]$) = $\{[r_s, r_e] | (r_s \leq i_s \leq r_e) \wedge (r_s \leq i_e \leq r_e)\}$ can be simplified to $\{[r_s, r_e] | (r_s \leq i_s) \wedge (i_e \leq r_e)\}$ due to $r_s \leq r_e$ and $i_s \leq i_e$ and extended like IQ resulting in the query box $[(min, i_e), (i_s, max)]$. Fig. 1(d).

All the necessary transformations are simple and result in iso-oriented query boxes. This does not hold for all transformation methods proposed for parameter space, e.g., the *midpoint transformation* [GG98] has the problem that all typical interval queries result in query boxes, which are not iso-oriented and cannot be handled by standard MAMs. Therefore, we do not consider this transformation method.

Despite the conceptual elegance, the properties of the parameter space approach and especially the following ones have been regarded as problematic for indexing [GG98]:

1. The parameter space doubles the number of dimensions, i.e., instead of indexing one object of type interval, one has to index a two dimensional space

2. The data distribution in parameter space is highly skewed

Although the parameter space doubles the number of dimensions the curse of dimensionality, the problem of indexing high dimensional spaces, can be neglected in the case of intervals as two dimensions are handled well by all MAMs.

The data distribution in parameter space is highly skewed, i.e., start and end of intervals are highly correlated (i.e., $i_s \leq i_e$) and usually the length of the intervals in a data set is similar. Therefore, all data is below the main diagonal and usually distributed along some diagonals (intervals of similar length). Consequently the used MAM has to handle skewed data distributions well. For example, the the GRID-file cannot guarantee a good space utilization with this kind of data distribution and the R-tree and its variants cannot guarantee good query performance due to the overlap of bounding regions.

Due to these problems the parameter space approach has not been considered further in the past [GG98]. However, as we will show in the next section, the UB-Tree handles the skewed data distribution in the parameter space well, both with respect to storage utilization and query performance.

4 Interval Handling with the UB-Tree

In this section we shortly describe how the UB-Tree, a MAM for point data, can be combined with the parameter space approach to index intervals. We start with a short introduction to the UB-Tree and then focus on the indexing and querying of the parameter space.

4.1 The UB-Tree

The UB-Tree [Bay97, Mar99] is a clustering index for multidimensional point data, which inherits all good properties of the B-Tree [BM72]. Logarithmic performance guarantees are given for the basic operations of insertion, deletion and point query, and a page utilization of 50% is guaranteed. The UB-Tree clusters data according to a space filling curve, namely the Z-curve [OM84] and introduces the new idea of partitioning the data space into disjoint Z-regions, which map to disk pages. The Z-regions are then indexed by a B-Tree using last included Z-address as key, which is the ordinal of a point on the Z-curve.

These Z-regions in conjunction with a sophisticated algorithm for multidimensional range queries [BM97] and the Tetris algorithm [MZB99] for sorted reading of multidimensional ranges offer excellent properties [Mar99] for multidimensional applications like data warehousing, archiving systems, temporal data management, etc. Integrating the UB-Tree into a RDBMS providing a clustering B-Tree is simple [RMF$^+$00].

The first commercial RDBMS with an integrated UB-Tree is TransBase HyperCube [TAS00], which has also been used for the measurements in this paper.

4.2 Indexing the Parameter Space

As described in section Section 3 we use the end point transformation to map an interval to a point in two dimensional space and then indexing the points by a UB-Tree. As previously mentioned the crucial question is, how well the UB-Tree can handle the highly skewed data distribution?

As the UB-Tree inherits the good properties of B-Trees it guarantees a 50% page utilization and usually it will be around 81% [Ks83, BY89]. Its Z-region partitioning adapts to the data distribution, i.e., densely populated areas are finer partitioned while the dead space above the main diagonal is covered by a few big Z-regions. Data structures like the GRID file would partition also the dead space, because of the fixed partitioning. Due to its disjoint Z-partitioning, point and range queries are always limited to just one path searches in the B-Tree index, which is not true for R-Trees due to the overlap problem of R-Trees.

The UB-Tree adapts well to the non-uniform data distribution in parameter space. The Z-regions of the UB-Tree narrowly adapt to the data distribution without degenerating. The empty part of the parameter space is covered by a few huge Z-Regions, but those are not empty as they occupy some populated part of the universe with a small spot. The Z-regions cluster the intervals according to their start and end. Therefore, it is likely that intervals within a Z-region have a similar position and length which is a benefit for query processing, since queries likely want to get these intervals together.

The correlation of the data, i.e., $start \leq end$, is no problem, since the query mapping fit to this. Also previous investigations [RMF$^+$01] have shown that the UB-Tree handles skewed data distributions well.

Additionally the UB-Tree is a dynamic index structure and supports updates with logarithmic performance guarantees.

4.3 Query Handling

As we have seen in Section 3.2 all basic query types (EQ, IQ, CQ, ECQ) map to multidimensional range queries on the parameter space. The same holds for all the 13 Allen relations [AH85], they also map to range queries, since they apply only constant constraints to the attributes.

As the UB-Tree is designed to perform multidimensional range queries it is able to handle these query types efficiently. Further more we need just one algorithm to deal with all these query type and there is no further need for specialized algorithms dealing with a specific query type.

The Z-region partitioning, which tries to maintain rectangular Z-regions with just a few fringes, fits perfectly to the query profile created by the interval queries. Therefore, the UB-Tree mainly loads those data pages from disk which contribute to the result.

5 The RI-Tree

The conceptual structure of the RI-Tree is based on a virtual binary tree of height h which acts as a backbone over the range $[1, 2^h - 1]$ of potential interval bounds. Traversals are performed purely arithmetically by starting at the root value 2^{h-1} and proceeding in positive or negative steps of decreasing length 2^{h-i} , thus reaching any desired value of the data space in $O(h)$ time. This backbone structure is not materialized, and only the root value 2^h is stored persistently in a meta data tuple. For the relational storage of intervals, the nodes of the tree are used as artificial key values: Each interval is assigned to a *fork node*, i.e., the first intersected node when descending the tree from the root node down to the interval location.

An instance of the RI-Tree consists of two relational indexes which in an extensible indexing environment are at best managed as index-organized tables. These indexes then obey the relational schema *lowerIndex* $(node, start)$ and *upperIndex* $(node, end)$ and store the artificial *fork node* value, the *start* and *end* of the interval. Additionally, one can add an identifier or other attributes to the relation.

As any interval is represented by exactly one entry for each the lower and the upper bound, $O(n/b)$ disk blocks of size b suffice to store n intervals. For inserting or deleting intervals, the node values are determined arithmetically, and updating the indexes requires $O(\log bn)$ I/O operations per interval.

For processing intersection queries we collect the nodes of the binary tree which may contain intervals intersecting the query interval $[i_s, i_e]$. Theses nodes fall into three classes:

Those nodes which are left to i_s and where we have to test $i_s \leq r_e$ (*leftnodes*), those nodes which are to the right of i_e and where we have to test $r_s \leq i_e$ (*rightnodes*) and those nodes included by the query interval, i.e., $i_s \leq r_s \leq r_e \leq i_e$ (*contained*).

The *leftnodes* and *rightnode* are stored in transient tables and a SQL query is executed that joins these with the base table while checking the predicates.

The RI-tree can be implemented by (procedural) SQL or within the application and does not assume any lower level interfaces. In particular, the built-in index structures of a DBMS are used as they are, and no intrusive augmentations or modifications of the database kernel are required.

6 Performance Analysis

For our measurements we use generated data and queries. The data type of start and end is an integer with the domain $[0, 2^2 0 - 1]$ and to each tuple we add an additional payload field of 200 bytes, resulting in a tuple size of 208 bytes. With a page size of 2kB this make 9 tuples per page. The page size is small compared to modern DBMSs, but it allows to measure results qualitatively similar those obtained by measuring with larger data sets.

We use a Sun Enterprise Server 450 with two Sparc-Ultra4 248 MHz CPUs and 512MB RAM. The secondary storage medium is an external 90 GB RAID system. The database system is TransBase HyperCube, which offers a product strength implementation of the UB-Tree.

In order to get an insight view and understanding of the measured indexing techniques we use four different data sets and two different query sets.

6.1 Data Sets

In order to reflect different applications scenarios we use four data distributions. The start position is always uniform distributed on the interval domain, while the length was varied. We used data set sizes of 1000, 10000, 100000 and 1 million tuples.

usul: uniformly distributed start and length

usul100k: uniformly distributed start and uniform distributed length within the range $[0, 100000]$

usel: uniformly distributed start and exponentially distributed length according to the exponential distribution function $\lambda e^{-\theta x}$ where $\lambda = 0.000476837$ and $\theta = 5.24288$

usel100k: uniformly distributed start and exponentially distributed length within the range $[0, 100000]$ according to the exponential distribution function $\lambda e^{-\theta x}$ where $\lambda = 0.002765655$ and $\theta = 367.0016$

The uniform distribution on start and end is used as it allows easier understanding of effects due to the availability of a cost model for UB-Trees. The exponential distribution of the length reflects most real world applications where short intervals are more likely to occur than long intervals. Further more, in real world applications there is usually a upper bound for the interval length[1] and therefore we use also variants of the data distributions that restrict the interval length to a given range.

[1]When storing transaction time intervals, they do not last forever since transactions will be aborted after a timeout period.

6.2 Query Sets

We focus on intersection queries, since exact match queries are trivial, i.e., they are efficiently handled by a point query. The results for intersection queries also hold for the containment and enclosure query, as those are a subset of the intersection query.

We use two different query sets:

window scan: a set of 5000 queries sorted according to the start point and with a length of 300. This results in a query set covering the whole data space, while two consecutive query intervals have an overlap of 50%.

random: 1000 random query intervals, where the intervals have been taken from the **usel** data set.

The *window scan* query set is used to locate performance dependencies of the index structures which are based on the location of the query interval. On the other hand it gives the index structures/DBMS a chance to profit from caching and clustering, due to the overlap of query intervals.

The *random* query set is used to see how the index structures perform without caching. It shows how good the index structures handle ad-hoc queries, which deteriorate caching.

6.3 Index Structures

For our comparison we have used the composite key index (B-Tree) and UB-Tree provided by TransBase and variants of the RI-Tree [KPS00, KMPS01] as a reference candidate for indexing techniques designed for intervals. Multiple secondary indexes are neglected as they do not provide clustering and their performance degenerates with bigger result sets due to random accesses on the base table, which can been observed in test measurements.

COMP: one composite key index on $(start, end)$

RIS: RI-Tree with two secondary indexes, one on $(node, start)$ and the other on $(node, end)$

RIP: RI-Tree with primary composite key index on $(node, start)$ and a secondary index on $(node, end)$

UBPS: UB-Tree on $start, end$

The RI-Tree was chosen, since it provides the same practically important properties as our approach: it is easy to implement/integrate, uses standard DBMS methods and provides scalability, update-ability, concurrency control, space efficiency, etc. Further more it has been proven to be superior to the Window-List [Ram97], Tile Index (T-Index) [Ora00] and IST-technique [GLOT96] and so we can transfer our performance results to these indexing techniques.

Actually, we first implemented the RI-Tree as it was presented in [KPS00] with two secondary indexes and "transient" tables, but the performance was worse than that of multiple secondary indexes. So in order to cluster the data we replaced the secondary index on $(node, start)$ by a clustering index on $(node, start)$ [KMS01]. Still the performance was not as good as expected. Actually, the problem was that TransBase does not support transient tables and the join of *leftnodes* and *rightnode* with the base table is not handled as efficiently as our solution, which was to embed the *leftnodes* and *rightnode* list as a `IN` clause in the SQL statement. The maximum length of the list is $\log_2(max - min) - 1$ and for the data used in our measurements it was 19. However, the average length was just 8 and using an array search on such a short array is more CPU efficient than the operations caused by a join.

6.4 Qualitative Comparison of Index Structures

As we have seen before the UB-Tree handles all the discussed query types with its range query algorithm. This is superior compared to other techniques that support just a few or one query type, e.g., the RI-Tree as presented in [KPS00] handles just intersection queries. In order to handle other query types one has to use a combination of intersection query and possibly expensive post filtering or develop specific algorithms.

Furthermore, when indexing additional attributes one may just add them as additional dimensions to the UB-Tree. As UB-Tree clusters data symmetrically with respect to all indexed attributes range restriction on them will also be handled efficiently. When using the RI-Tree one has to add the attributes to its composite key index or use secondary indexes. However, most implementation of composite key indexes cannot handle multidimensional range queries efficiently, i.e., they only utilize the range restriction on the first indexed attribute, and secondary indexes do not perform well, since they do not cluster the data accordingly.

6.5 Results

When making performance measurements of index structures it is important to take all operations into account and not just the query performance. Theses are loading, space requirements, clustering, queries, updates, locking, archiving, etc.

We have concentrated on the following ones: Loading, space consumption, query performance and clustering, as updates, locking and archiving are handled well by the B-Tree underlying the tested indexing techniques.

count	COMP	UBPS	RIP	RIS
1000	148	150	146	171
10000	1450	1471	1402	1634
100000	14486	14681	13895	16189
1000000	143626	146788	146087	161489

Table 1. Space Requirements

6.5.1 Bulk Loading Time

Loading time scales linear for all index structures. The data was sorted according to start before loading it, since this can usually be assumed for real world data.

COMP loads fastest since it requires no further sorting of the input data. RIS follows as it only requires updates of its indexes while just appending data to the base table. The RIP index requires even more time due to the clustering primary composite key index on $(node, start)$. UBPS is faster than RIP for small **usul** data sets and follows shortly after the RIP for all other data sets.

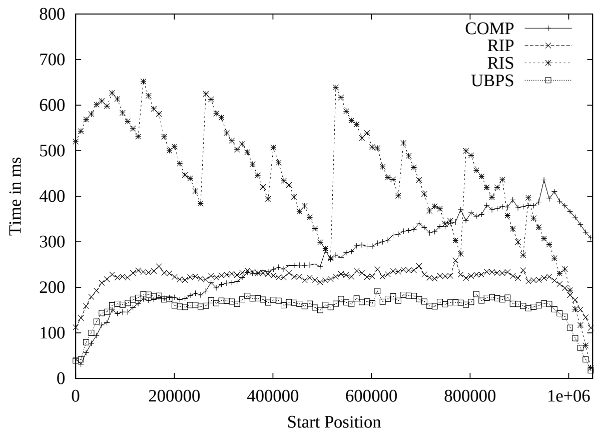
We have also performed bulk loading of unsorted data. The results of this were that UBPS was fastest, since it requires less sorting than COMP, which took 2.2 times longer in average. RIS and RIS followed with times longer by a factor of 2.32 and 2.35, due to their more expensive index maintainance and complex sorting.

6.5.2 Size of the Tables and Indexes

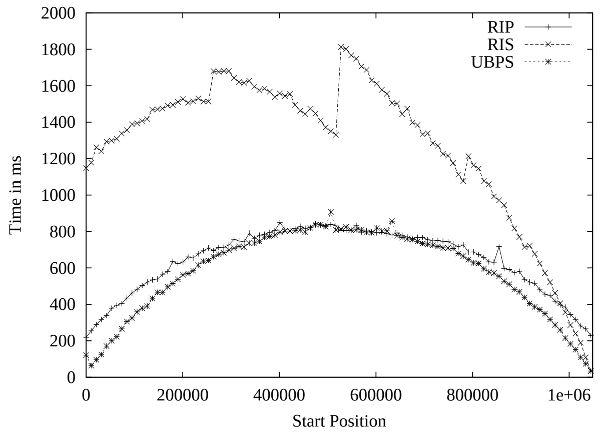
The overall size of the measured indexes is fairly the same and grows linearly with the data volume as shown in Table 1. The extra attribute *note* of the RI-Tree has only a minor influence, since four extra bytes do not contribute much to a tuple size of 208 bytes. However, it should be considered that tuples of the form $(start, end)$ would result in 50% higher space requirements, due to the extra $node$ attribute.

In general COMP and RIP requires least pages, followed by UB-Tree and RIS. UBPS requires slightly more pages compared to COMP and RIP, because compression of UB-Tree data pages has not been implemented in TransBase HyperCube yet. However, the differences can be neglected as they are minimal. The page utilization of the different techniques is usually between 77% and 90% and varies with the scaling factor and no best indexing technique can be recognized.

At the maximum scaling factor of 1 million tuples the DB size was around 290 MB in average and UBPS requires 2% more pages than COMP while RIP/RIS require around 1% more pages than COMP in average.



(a) Time **usul100k**



(b) Time **usel**

Figure 2. Measurement: homogen query

6.5.3 Query Performance

We use a data base size of 10000 tuples for the measurements. Test measurements with bigger data sets have shown the similar qualitative results as those presented here. First we want to consider the **window scan** query set.

Fig. 2(a) shows the times for measuring the **usul100k** data set. All the measurements plots are sampled to fewer measurement points in order to be easier to distinguish.

RIS is not a clustering index and we clearly see peaks which mark the nodes of the binary tree. When the scan window reaches a new node it causes random page accesses to disk pages containing tuples of that node. After that these pages remain in cache and therefore the performance gets better until the query window reaches the next node.

COMP reflects the fact that such an index is only able to exploit the range restriction on the first attribute *start*, but not the range restriction on the second attribute *end*. Due to the mapping of the IQ, an IQ at the beginning of the domain has a selective restriction on *start* and no restriction on *end*. As the query window shifts towards the end of the domain the restriction on *start* decreases while the restriction on *end* increases. Finally, only *end* will be restricted.

As we have seen in Fig. 2(a), the times for COMP and RIS are not only related to the result set size but they also show a high relationship between the position of the query window and the response time. We refer to this as *positional dependency*. RIP and UBPS show a better performance and the response time reflects no positional dependencies, but it is linear to the result set sizes (plot omitted). As there are less intervals at the beginning and end of the interval domain they are faster there.

The rations between COMP/UBPS, RIS/UBPS and RIP/UBPS. In average RIP requires 46% more time that UBPS and in the worst case RIP required nearly 8 times longer than UBPS. In general one can say, UB-Tree outperforms RIP for small result sets.

The measurements with the **usel100k** data set had similar results, due to this we omit them in this paper. In general the response times have been shorter as the result set sizes are smaller, because there are more shorter intervals in comparison to the data sets which do not restrict the length of the intervals. Due to this there are also more intervals assigned to lower nodes of the binary tree of RIS/RIP and RIS shows more peaks, but not so big ones. This holds also for RIP, while UBPS maintains its behavior.

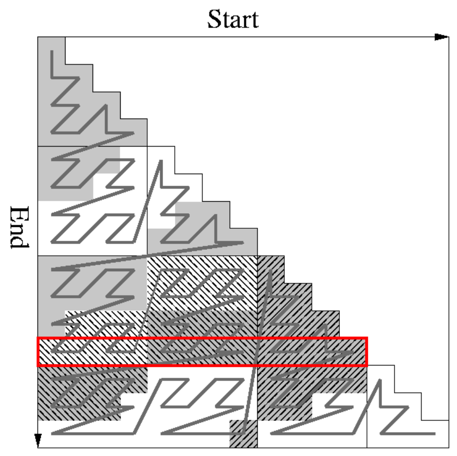
Fig. 2(b) shows the measurement for RIP and UBPS with the **usel** data set. The results are similar as before, however the difference between RIP and UBPS becomes less. In this measurement RIP requires 30% more time in average and RIS was three times slower than UBPS. Again RIS shows the positional dependency. As before these results are also qualitatively observed fro the **usul** data set.

As the performance of COMP depends on the position of the query interval we do not consider it further, but we focus on RIP and UBPS which perform linear to the result set size. Further more we also show for completeness RIS.

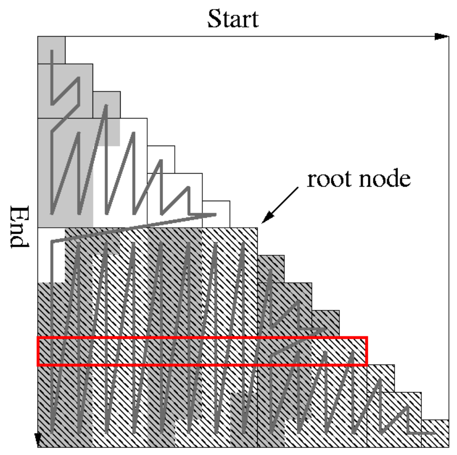
The **random** query set has performed as follows. With small result set sizes UBPS is up to five times faster than RIP and 8 times faster than RIS. RIS shows again the unpredictable varying response times due to missing clustering. With growing result set size RIS and RIP become better and finally RIP is even 5% faster than UB-Tree, which comes from the better page utilization of the composite key index which is used for RIP. In average UBPS performs just 2% better than RIP.

6.6 Clustering

In order to get a better understanding on the differences between UBPS and RIP it is crucial to investigate their clustering and how it differs.



(a) UBPS



(b) RIP

Figure 3. Clustering and Page Partitioning

Fig. 3 shows the region partitioning (and also disk page clustering) for the UB-Tree and the RI-Tree with a clustering composite index on (node,start,end)[2]. The domain for this picture is $[0, 14]$ and we assume a uniform distribution on *start* and length of the intervals. A *end-within* query (all intervals that end at a given range) is also depicted and

[2]Taking *end* not into account for the clustering of the RI-Tree would add random jumps to the space filling curve of the RI-Tree, since there would not be an order on *end*.

the pages which have to be loaded are filled with a striped pattern. In this case the RI-Tree has to load twice the data pages (6) of the UB-Tree (3).

The UB-Tree clusters the data symmetrically, i.e., it treats *start* and *end* of an interval equally. This results in clustering intervals with respect to their position and length at the same time. The RI-Tree with a primary index on $(node, start, end)$ clusters according node and then to the composite key order resulting in a stripe like partitioning. With growing data volume the UB-Tree would adapt its regions by making them smaller and more rectangular like. The RI-Tree will get even more strip like and a query like presented here will cut it like a puff-pasty, while the UB-Tree provides a good approximation for the query.

Therefore, the *meets*, *left-overlaps*, *left-covers* relations [AH85] are not handled well by a RI-Tree with the primary index on $(node, start)$ resp. *met by*, *right-overlaps*, *left-covered* are not handled well by a RI-Tree with the primary index on $(node, end)$. The reason for this is that those queries restrict just the *start* resp. *end* cannot be processed well by a clustering index on the not restricted attribute. Using the secondary index results in random page accesses and depending on the DBMS even multiple page accesses. Therefore, an index scan on the clustering index is usually more efficient, but it results in a complete scan of all affected nodes. Compared to this, the UB-Tree Z-region space partitioning allows for reading a good approximation of the subspace which contains intervals contributing to the result. With more data the UB-Tree will become even better compared to the RI-Tree.

However, a query restricting just start can be processed more efficiently by a RI-Tree on $(node, start)$, but also the UB-Tree can perform these queries and the RI-Tree is not able to outperform the UB-Tree by orders of magnitude.

6.6.1 Measurements

We use a **usul** data set with 100000 tuples for this measurement and move a restriction on *end* with length 10000 in steps of 10591 from the start of the domain to the end of it. Caching was enabled for this measurement. We also included the plots for multiple secondary indexes on a table without a specific sort order.

For the RI-Tree we used a modified version of the IQ query algorithm collecting just those nodes that might contribute tuples to the result, i.e., it works like the intersection query for intervals which are points, but is has to traverses only one path. This was done by traversing the virtual binary tree and collecting all those nodes with intervals which may contain the *start* resp. *end* point.

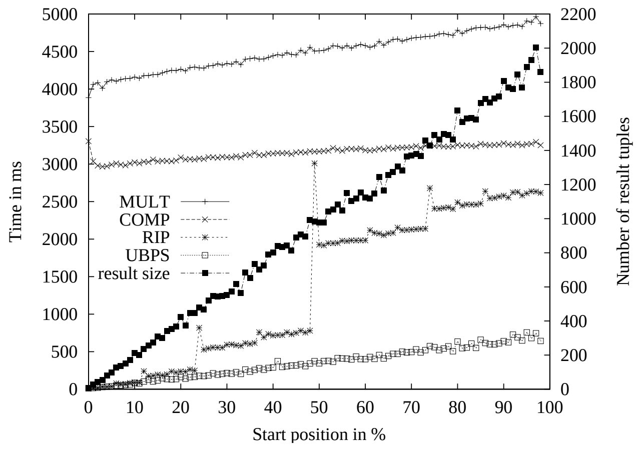
Fig. 4(a) shows the results for a query restricting just the end position. By shifting the restriction to the end of the domain the number of result tuples grows as shown in Fig. 4(a), since there are more longer intervals. UBPS times reflects the linear dependency to the result set sizes. RIP starts similar to UBPS, but the further the restriction moves the more nodes containing more tuples have to be scanned for results. Again we see the phenomenon of peaks we have only seen for RIS before. Whenever, the restriction shifts over a new node the data base has to fetch all the intervals corresponding to that node. The first two levels of the binary tree are clearly visible at 50% for the first level (root node) and 25% and 75% for the second level. At 50% there is the highest peak, since suddenly the root node has to be processed, which was not cached so far. Additionally also pages from the nodes before the root node are affected by this query. COMP on (*start*,*end*) has to perform an index scan since its implementation does not support a range restriction on end only. Its slightly increase comes from the growing amount of tuples which have to be transfered to the application. Fig. 4(a) depicts the linear relation between response times and number of loaded pages. MULT performs really bad. It loads fewer pages than COMP, but it has to fetch them by random accesses and therefore it is not able to exploit prefetching as COMP does, further more it has to load more pages than UBPS or RIP due to the lack of clustering. Speaking in averages, MULT is nearly 13 time slower than UBPS, COMP is 9 times slower than UBPS, where RIP is 4 times slower.

The results for a query restricting just *start* in the same way as *end* was restricted before are depicted in Fig. 4(b). As expected COMP performs best, since its clustering is perfect for this restriction. It performs linear to the result set size. RIP is only a bit slower, but again with peaks at the places where the query window covers new nodes. UBPS follows again performing linear to there result set size. MULT is again bad for the same reasons as before. Speaking in averages, MULT is nearly 12 time slower than UBPS, while COMP takes 67% and RIP 75% of the time UBPS required.

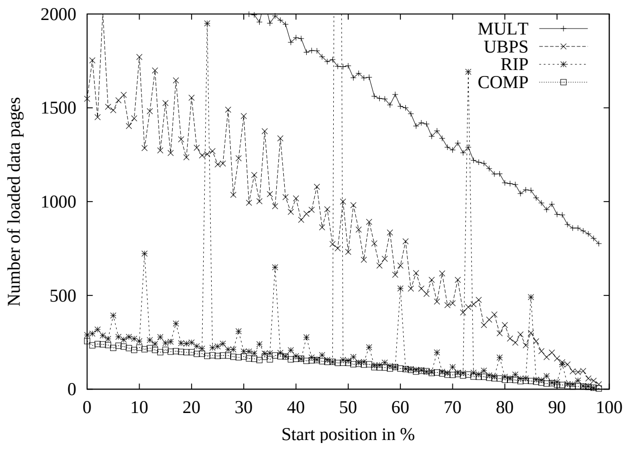
Finally speaking RIP requiring 400% of UBPSs time for *ends in* queries and 75% for *starts in* queries, therefore UBPS is better in the overall average.

7 DBMS integration

The parameter space technique can be implemented by embedding it into an application or building a middle ware using a underlying RDBMS with integrated UB-Tree. The UB-Tree adds efficient query processing by indexing the parameter space. It would also be possible to integrate it into a DBMS providing a procedural query language without modifying the database kernel. However, a DBMS integration is best, as this reduces programming overhead on the application side, allows for easy use and minimizes DBMS maintenance while maximizing the performance.



(a) *Ends Within* query: times



(b) *Starts Within* query: logical Pages I/Os

Figure 4. *Ends Within* **and** *Starts Within* **query**

Both, the UB-Tree and the necessary parameter space transformations are straight forward approaches and they are easy to implement and robust to be used in large-scale applications. Additionally they can be integrated with minimum impact to existing parts of a RDBMS, as long as it supports a clustering B-Tree.

Only minor changes to the existing code of a RDBMS are necessary, i.e., there has to be a new data type for intervals in order to allow for the recognition of the required parameter space transformations. With this we need just a new kernel module handling all the transformations of DDL statements, insert/update/delete statements and interval queries.

The DDL statement for creating a interval relation "`CREATE TABLE t(i INTERVAL(INTEGER), ...)`" is transformed to "`CREATE TABLE t(istart INTEGER, iend INTEGER, ...)`".

With this the query "`SELCT * FROM t WHERE i INTERSECTS` $[i_s, i_e]$" maps to the SQL statement "`SELECT * FROM t WHERE istart BETWEEN` $mininteger$ `AND` i_e `AND iend BETWEEN` i_s `AND` $maxinteger$".

8 Summary and Outlook

We have presented a hybrid method to manage and query intervals efficiently. It transforms intervals to a two dimensional space (parameter space) and indexes that space with a UB-Tree. Queries on intervals are transformed to query boxes in parameter space which are handled well by the UB-Tree range query algorithm. The required transformations of intervals and queries to parameter space are simple, efficient and independent of other DBMS functionality.

Further more, this technique is dynamic, i.e., it allows updates and deletes, it is superior compared to prior techniques with respect to query performance as well as the supported query types. Finally, it allows for indexing additional dimensions symmetrically and thus also provides efficient access to these.

In our further research we will investigate the usability and performance of the described approach with real world data from data warehouses and spatial objects approximated by a space filling curve.

In addition, the parameter space technique can be generalized to index bounding boxes of higher dimensional objects, thus enabling indexing spatial objects (e.g., GIS data) in two/three dimensional space with the UB-Tree. For two/three dimensional objects we expect good performance, since earlier measurements [Mar99] have proven that the performance of the UB-Tree is also excellent for the number of dimensions (4 for 2d objects and 6 for 3d objects) required here.

References

[AH85] J.F. Allen and P.J. Hayes. A common-sense theory of time. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, volume 9., pages 528–531, 1985.

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

[BG94] Gabriele Blankenagel and Ralf Hartmut Gting. External segment trees. In *Algorithmica*, volume 12(6), pages 498–532, 1994.

[BKK99] Christian Böhm, Gerald Klump, and Hans-Peter Kriegel. XZ-Ordering: A space-filling curve for objects with spatial extension. In *Proceedings of Advances in Spatial Databases, 6th International Symposium, SSD'99, Hong Kong, China, July 20-23, 1999*, volume 1651 of *Lecture Notes in Computer Science*, pages 75–90. Springer, 1999.

[BM72] Rudolf Bayer and E. McCreight. Organization and Maintainance of large ordered Indexes. In *Acta Informatica 1*, pages 173–189, 1972.

[BM97] Rudolf Bayer and Volker Markl. The UB-Tree: Performance of Multidimensional Range Queries. Technical Report TUM-I9814, Institut fr Informatik, TU Mnchen, 1997.

[Boz98] Toga Bozkaya. *Index Structures For Temporal And Multimedia Databases*. PhD thesis, Department of Computer Engineering and Science Case Western Reserve University, 1998.

[BY89] Ricardo A. Baeza-Yates. The Expected Behaviour of B^+-Trees. In *Acta Informatica 26(5)*, pages 439–471, 1989.

[FR89] Christos Faloutsos and Shari Roseman. Fractals for secondary key retrieval. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 29-31, 1989, Philadelphia, Pennsylvania*, pages 247–252. ACM Press, 1989.

[GG98] Volker Gaede and Oliver Gnther. Multidimensional Access Methods. In *Computing Surveys 30(2)*, pages 170–231. ACM Press, 1998.

[GLOT96] Cheng Hian Goh, Hongjun Lu, Beng Chin Ooi, and Kian-Lee Tan. Indexing temporal data using existing b+-trees. In *Data & Knowledge Engineering*, volume 18(2), pages 147–165, 1996.

[KMPS01] Hans-Peter Kriegel, Andreas Müller, Marco Pötke, and Thomas Seidl. Spatial data management for computer aided design. In *Proceedings of SIGMOD'01, Santa Barbara*. ACM Press, 2001.

[KMS01] H.-P. Kriegel, M. Ptke M., and T. Seidl. Interval Sequences: An Object-Relational Approach to Manage Spatial Data. In *Proc. 7th Int. Symp. on Spatial and Temporal Databases (SSTD'01), Redondo Beach, CA, 2001.*, 2001.

[KPS00] Hans-Peter Kriegel, Marco Pötke, and Thomas Seidl. Managing intervals efficiently in object-relational databases. In *Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 407–418. Morgan Kaufmann, 2000.

[Ks83] Klaus Kspert. Storage Utilization in B^*-Trees with a Generalized Overflow Technique. In *Acta Informatica 19*, pages 35–55, 1983.

[KS91] Curtis P. Kolovson and Michael Stonebraker. Segment indexes: Dynamic indexing techniques for multi-dimensional interval data. In James Clifford and Roger King, editors, *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, Denver, Colorado, May 29-31, 1991*, pages 138–147. ACM Press, 1991.

[Mar99] Volker Markl. *Processing Relational Queries using a Multidimensional Access Technique*. PhD thesis, DISDBIS, Band 59, Infix Verlag, 1999.

[MTT00] Y. Manolopoulos, Y. Theodoridis, and V.J. Tsotras. Chapter 4: Access methods for intervals. In *Advanced Database Indexing*, Boston, MA: Kluwer, 2000.

[MZB99] Volker Markl, Martin Zirkel, and Rudolf Bayer. Processing Operations with Restrictions in RDBMS without External Sorting: The Tetris Algorithm. In *Proceedings of the 15th International Conference on Data Engineering, 23-26 March 1999, Sydney, Australia*, pages 562–571. IEEE Computer Society, 1999.

[OM84] Jack A. Orenstein and T. H. Merrett. A Class of Data Structures for Associative Searching. In *Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, April 2-4, 1984, Waterloo, Ontario, Canada*, pages 181–190. ACM, 1984.

[Ora00] Oracle. *Oracle8i Data Cartridge Developers's Guide*. Oracle Corporation, Redwood City, CA, rel. 8.1.7 edition, 2000.

[Ore90] Jack A. Orenstein. A Comparison of Spatial Query Processing Techniques for Native and Parameter Spaces. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 343–352. ACM Press, 1990.

[Ram97] S. Ramaswamy. Efficient indexing for constraint and temporal databases. In *Proceedings of the 6th International Conference on Database Theory (ICDT)*, pages 419–431, 1997.

[RMF$^+$00] Frank Ramsak, Volker Markl, Robert Fenk, Martin Zirkel, Klaus Elhard, and Rudolf Bayer. Integrating the UB-Tree into a Database System Kernel. In *Proceedings of International Conference on Very Large Data Bases, 2000, Cairo, Egypt*, 2000.

[RMF$^+$01] F. Ramsak, V. Markl, R. Fenk, R. Bayer, and T. Ruf. Interactive ROLAP on Large Databases: A Case Study with UB-Trees. In *Proc. of IDEAS Conf. 2001, Grenoble, France*, 2001.

[SOL94] Han Shen, Beng Chin Ooi, and Hongjun Lu. The tp-index: A dynamic and efficient indexing mechanism for temporal databases. In *Proceedings of the Tenth International Conference on Data Engineering, February 14-18, 1994, Houston, Texas, USA*, pages 274–281. IEEE Computer Society, 1994.

[TAS00] TAS. *Transbase HyperCube*. TransAction Software, http://www.transaction.de, 2000.

[TCG$^+$93] A.U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Sondgrass. *Temporal Databases: Theory, Design and Implementation*. Benjamin/Cummings, Redwood City, CA, 1993.