The Tetris-Algorithm for Sorted Reading from UB-Trees[∇]

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

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

1 Concept of the UB-Tree

1.1 Addresses, Areas and Regions

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

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

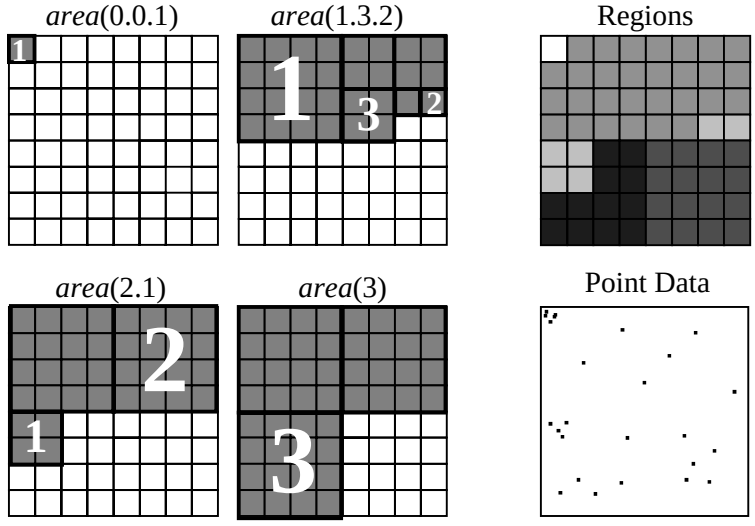

Figure 1: Areas and Regions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

We assume that we have a universe U of values. For simplicity we assume that U has $v = 2^r$ values per dimension which are numbered $0,1,2,..., 2^r-1$. In this paper arbitrarily shaped spaces are simply considered as a subset of a suitable cube-shaped universe. It is also possible to drop this assumption and tailor the UB-Tree to the universe. This approach is described in [MB97c].

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

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

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

The UB-Tree gives logarithmic worst case guarantees for update and retrieval. Due to its multidimensional clustering it shows superior behavior over current indexing techniques for updates, point queries and range queries. In addition the point query performance of the UB-Tree is identical to that of classical B-Trees with concatenated key attributes.

2 Impacts on the relational Algebra – the Tetris Algorithm

Tables organized by a UB-Tree can be read in any sort order in $O(n)$ disk accesses where n is the number of pages of the table or the minimal number of regions covering a query box [Bay97a]. This is made possible by a modification of the range query algorithm and a caching technique, the so called ”Tetris-Algorithm” [MB97b]. This algorithm performs a sweep over a query box of the UB-Tree with respect to the lexicographic order of the specified sorting dimensions (in the spirit of the well known sweep line algorithms [PS85]). The Tetris-Algorithm works similar to the range-query algorithm. The only difference is that the calculation of the next intersecting region does not return a region according to Z-ordering, but according to the specified sort order: Initially the algorithm calculates the first region that is overlapped by the query-box, retrieves it and caches it in main memory. Then it continues to read and cache the next regions with respect to the sort order, until a complete thinnest possible slice of the query box has been read. Then the cached tuples of this slice are sorted in main memory, returned in sort order to the caller and removed from cache. The algorithm proceeds reading the next slice, until all regions which intersect the query box have been retrieved and output.

The algorithm to calculate the next intersecting region with respect to the sorting dimension merely requires one B-Tree-Search. Therefore only *n* disk accesses to data pages need to be performed to sort a query box overlapped by *n* regions according to any of the *d!* sort order definable over *d* attributes. Thus each page only needs to be accessed once in order to produce a sorted output in any dimension.

```
ξ = alpha(ql); ω = alpha(qh)
repeat
  search [α:β] in the UB-Tree, so that α <a ξ ≤a β
  store all tuples from page([α:β]) in the cache
  if a new slice in the sorting dimension is completed
     s = endpoint of the slice in the sorting dimension
     sort all cached tuples
     output all cached tuples where x_sort ≤ s
     remove all cached tuples where x_sort ≤ s from cache
  ξ = address of the next point intersecting the querybox
  with respect to the sorting dimension
until ξ >a ω
```

Algorithm 1: Tetris-Algorithm for a query box (ql:qh)

The Tetris-Algorithm is illustrated in figure 2. For simplification of the presentation we assume that the query box is the complete universe. We sort the data in the vertical dimension, i.e. from left to right. Regions from which tuples are cached are shaded in this figure. The slice currently being operated on has white borders. The algorithm starts by retrieving the region in the very left corner (2a). Successive regions are retrieved and cached (2b) until a vertical slice is completed (2c). The tuples of this slice are then sorted in main memory. All tuples up to the end coordinate of the slice in the sorting dimension are output in sort order and removed from cache. The regions of this slice can not be removed from cache completely at this point since each region still might have some tuples that have not been output yet. However, not the entire regions are cached. Caching is only necessary for those tuples that have not been output yet. Figure 2d shows the cached regions after reading the next region, i.e., after completion of the second slice. After completion of the next slice (2e) two regions of the first and second slice have been handled completely. These regions can be removed from cache completely. The Tetris-Algorithm continues processing in this way until the last stripe of the query box has been read (2f), i.e., the complete universe has been read in sort order.

3 Applications of the Tetris-Algorithm

3.1 Range Queries

The Tetris-Algorithm may be used to replace the standard range query algorithm for UB-Trees described in [Bay96] and [MB97a]. In this case the Tetris-Algorithm does not need to cache tuples until the completion of a slice in a certain dimension. Instead, tuples may be processed immediately after a data page has been retrieved.

3.2 Sorted Reading of Relations

The main goal of the Tetris-Algorithm is to speed up sorted reading of relations. If the first sort-attribute is a UB-Tree attribute, the Tetris algorithm reading slices with respect to this attribute can be used to read the relation in sort order. Sorted reading is required for the efficient processing of most of the operations of the relational algebra, such as projection, ordering and joining of relations. See [Bay97a] for a description of algorithms using UB-Trees for these operations. The necessary cache size for sorted reading with the Tetris-Algorithm for uniformly distributed data is approximated by $p^{(d-1-r)/d}$, whereby *p* is the number of pages, *d* is the dimensionality of the UB-Tree and *r* is the number of restricted dimensions.

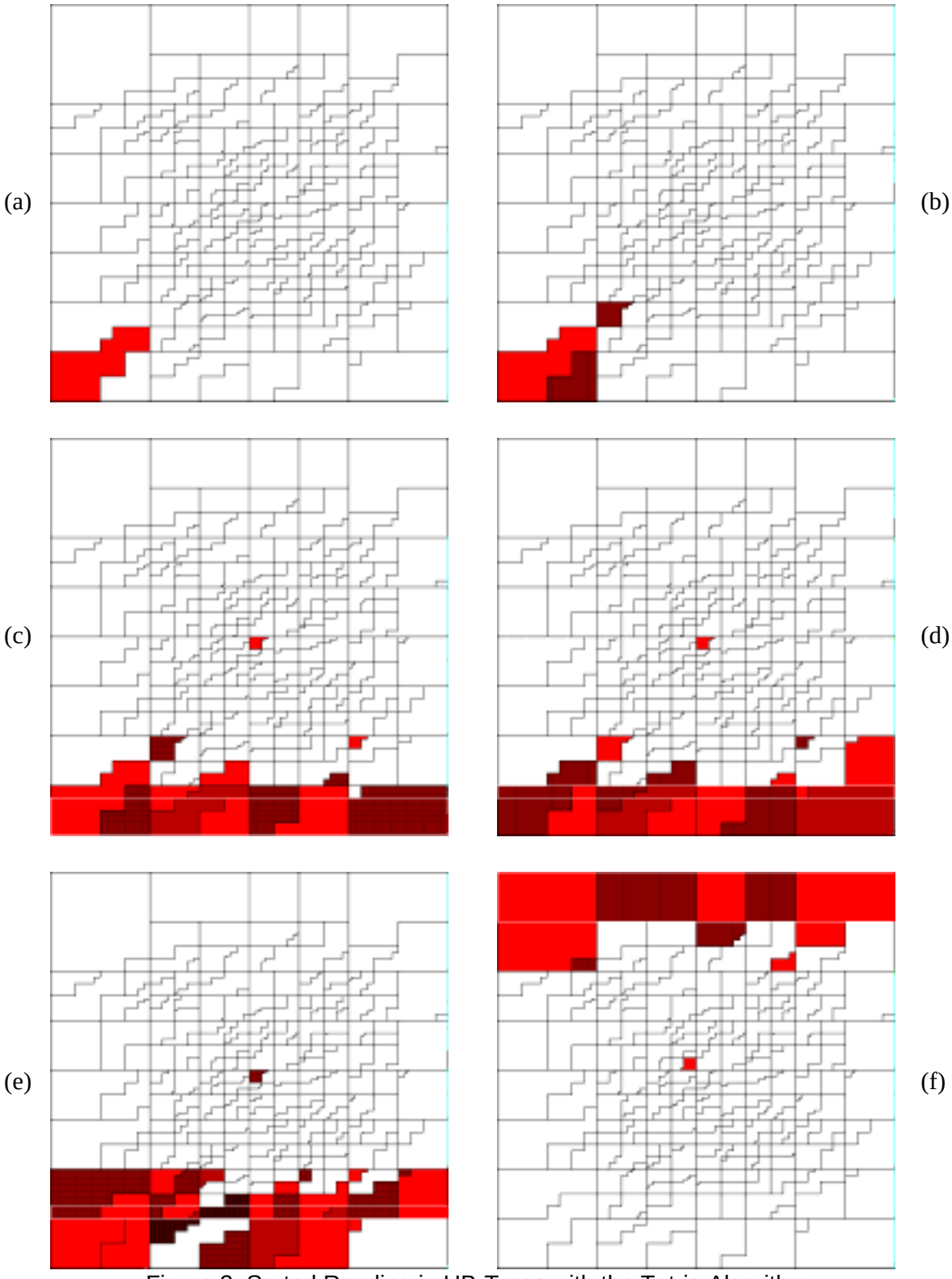

Figure 2: Sorted Reading in UB-Trees with the Tetris-Algorithm

3.3 Grouping and Aggregation

The ability to read data in sort order with respect to any dimension of the UB-Tree also enables efficient grouping and aggregation without the need of externally sorting a relation. The basic algorithm is identical to algorithm 1. A lower cache size compared to sorted reading is required, if several dimensions are specified for the grouping and if the data does not need to be sorted. In this case a cache size of $p^{(d-g-r)/d}$ pages is necessary to perform the grouping. p, d and r are defined as in the previous section and g is the number of attributes specified for grouping.

4 Speeding up the Relational Algebra

UB-Trees and the Tetris-Algorithm can tremendously speed up the operations of the relational algebra. For example, the SQL-query

```
SELECT avg(f) FROM t WHERE a= x1 and b = x2 GROUP BY c, d, e ORDER BY c
```

against the 6-dimensional UB-Tree (a,b,c,d,e,f) consisting of $p = 10^6$ pages requires a cache size of $10^{6*((6-1-2)/6)} = 10^3$ pages to perform the operation. Only $10^{6-2} = 10^4$ pages need to be retrieved from disk to perform the operation. Traditional techniques would require a merge sort of 10^6 pages. With a cache size of 10^3 pages this means $10^6 * \log_{1000} 10^6 = 2 * 10^6$ disk accesses for reading, and the same number of disk accesses for writing the data during the merge sort algorithm. The restriction of the WHERE-clause can in general not be used. This is only possible if the first attribute of the primary index were specified in the WHERE-clause. All in all we get $4 * 10^6$ disk accesses for the merge sort and 10^4 disk accesses for the Tetris-algorithm. If one disk access takes 10ms, the UB-Tree answers the query in 10 seconds. In contrast, the standard technique takes more than 1 hour to answer the same query.

Proper data modeling is still necessary, since a query without selection and one sort attribute against a 6 dimensional UB-Tree would achieve no reduction of the disk pages and result in a cache size of $p^{5/6}$. In this case it could make sense to use a UB-Tree of a lower dimensionality in order to reduce cache sizes. Thus the choice of index attributes is a critical data modeling question that depends on the database schema and the query profile. We are currently investigating data modeling with UB-Trees in our Research Group.

Generally speaking, the Tetris-Algorithm allows joining, grouping, aggregation, projection and any other operation where sorted reading of a relation (or parts of it) is involved in *O(n)* disk accesses, whereby *n* is the number of pages needed to store the data. An additional selection may be used to reduce the necessary disk accesses, if the restricted attributes are also part of the UB-Tree. A further advantage of the Tetris-Algorithm is that it needs no disk space to perform the operation. Only a main memory cache is required which in general is smaller than the main memory cache required for a good performance of the merge sort algorithm.

5 Conclusions and Future Work

The Tetris algorithm for database relations organized by a UB-Tree can speed up any query, where sorting, grouping and multi-dimensional restrictions are involved. Currently we are implementing the algorithm and will evaluate its performance for both OLTP and OLAP applications. We also investigate data modeling with the presence of multi-dimensional indexes in order to get a new methodology for schema evolution. Practical test beds for the UB-Tree and the Tetris algorithm will be the queries of the TPC-D benchmark as well as typical queries and schemata of our project partners.

References

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

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

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

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

[MB97b] Markl, V.; Bayer, R.: The Tetris-Algorithm for multidimensional Sorted Reading from UB-Trees, Internal Report, FORWISS München, 1997

[MB97c] Markl, V.; Bayer, R.: Variable UB-Trees for the efficient Indexing of arbitrarily distributed multidimensional Data, Internal Report, FORWISS München, 1997

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