Processing Operations with Restrictions in RDBMS without External Sorting: The Tetris Algorithm

Volker Markl[*] *Martin Zirkel*[+] *Rudolf Bayer*[*+]

[*]Bayerisches Forschungszentrum
für Wissensbasierte Systeme
Orleanstr.34, 81667 München, Germany
volker.markl@forwiss.de

[+]Institut für Informatik
Technische Universität München
Orleanstr.34, 81667 München, Germany
{zirkel,bayer}@in.tum.de

http://mistral.informatik.tu-muenchen.de

Abstract

Most operations of the relational algebra or SQL require a sorted stream of tuples for efficient processing. Therefore, processing complex relational queries relies on efficient access to a table in some sort order. In principle, indexes could be used, but they are superior to a full table scan only, if the result set is sufficiently restricted in the index attribute. In this paper we present the Tetris algorithm, which utilizes restrictions to process a table in sort order of any attribute without the need of external sorting. The algorithm relies on the space partitioning of a multidimensional access method. A sweep line technique is used to read data in sort order of any attribute, while accessing each disk page of a table only once. Results are produced earlier than with traditional sorting techniques, allowing better response times for interactive applications and pipelined processing of the result set. We describe a prototype implementation of the Tetris algorithm using UB-Trees on top of Oracle 8, define a cost model and present performance measurements for some queries of the TPC-D benchmark.

1 Introduction

Complex business applications like SAP R/3, statistical databases, data warehousing and data mining have created a strong demand for efficiently processing complex queries on huge relational databases. An important issue is to select the best access paths and processing techniques for any query. Especially for OLAP environments specialized access methods have been proposed [Inf97, OQ97]. Often several indexes are created on one table in order to speed up query processing [GHR+97].

The sort operation is useful for efficient implementation of almost every operation of the relational algebra. Merge sort algorithms are often superior to indexes, since a full table scan utilizes pre-fetching techniques to substantially reduce the number of expensive random page accesses. Our performance measurements on several commercial DBMS indicate that for reading an entire relation a full table scan is ten times faster than an index scan.

We focus on using multidimensional indexes to organize any table of a relational database. When considering a tuple as a point in multidimensional space, processing complex queries by a multidimensional organization of a relational table is often superior to one-dimensional clustering. However, current commercial DBMS do either not support multidimensional indexes at all or only use them in the context of geo-spatial applications.

The goal of our paper is to introduce the Tetris algorithm, a query processing technique for sort operations with multi-attribute restrictions, and thereby to show the potential of integrating multidimensional indexes as first class indexes into the kernel of a database system.

Multidimensional access methods are well researched in the field of spatial databases. [GG97] and [Sam90] provide excellent surveys of almost all of these methods. Multidimensional indexes are used to utilize spatial restrictions (e.g., range restrictions, intersection, overlap) and to efficiently compute spatial joins [Rot91, Gün93].

The Tetris algorithm generalizes the range query algorithm of a multidimensional index to efficiently process sort operations in combination with restrictions. The basic idea is to use the partial sort order imposed by a multidimensional partitioning in order to process a relation in some total sort order. Essentially a plane sweep [PS85] over a query space defined by restrictions on a relation partitioned by a multidimensional index is performed. The direction of the sweep is determined by the sort attribute. Only disk pages overlapping the query space are accessed. With sufficient, but modest, cache memory each relevant disk page is accessed only once.

Most work on applying multidimensional indexes to RDBMS discusses restrictions by range queries. [HNK+90] and [Bay97b] investigate joins and sorted processing of data organized by a multidimensional index. We extend this approach to restrictions by introducing a single operator, which combines sorting and restrictions in multiple attributes. Any multidimensional access method like Grid-Files, R-Trees, UB-Trees [Bay96, Bay97a] or hB-Trees can be used to organize the relation. However, it is desirable to use an index which creates a disjoint partitioning of the multidimensional space. To illustrate our approach we use an implementation of the UB-Tree, because the UB-Tree is easily implemented above any RDBMS by utilizing the B*-Tree of this RDBMS.

With the Tetris algorithm a multidimensional index can reduce resource requirements for virtually any operation of the relational algebra. Compared to the native access methods of Oracle 8 our prototype implementation of the Tetris algorithm shows significant speedups for queries of the TPC-D benchmark. In addition temporary storage requirements for the sorting process are reduced by two orders of magnitude and first results of a sort operation are produced several hundred times faster.

2 Query Processing

An important task of query processing in relational DBMS is to efficiently implement algorithms for the basic operations of the relational algebra. [Gra93] gives a concise survey of query processing. If the selection condition specifies a range in a single attribute, a clustering index greatly speeds up query processing. Conjunctive selection conditions are efficiently processed by composite indexes, intersection of record pointers or multidimensional indexes. The join operation is usually implemented by nested loop algorithms, join indexes, sort-merge algorithms or hash algorithms. [ME92] surveys join processing in relational databases. Projection, union, intersection and set difference are efficiently implemented by processing a relation in some sort order and then either use an index scan or a merge-sort. Efficient sort operations and the use of restrictions to limit result sets are crucial to many query processing algorithms. Very often queries combine several operations of the relational algebra like join and restriction. A multidimensional organization of a table in combination with the Tetris algorithm can utilize range restrictions in multiple attributes while processing a table in some sort order.

3 The Tetris Algorithm

In accordance with [HNK+90] we use the notions of wave, cluster and processing range to explain our generalized approach for processing sort queries with restrictions. In order to describe the Tetris algorithm we define a formal model for partitioned relations (cf. also Figure 3-1).

Let R be a relation having d attributes $A_1,..., A_d$ of domains $\Omega_1,..., \Omega_d$ composed of tuples $x = (x_1,..., x_d)$. Let $<_i$ be a total order on Ω_i and λ_i resp. υ_i the minimum resp. maximum value of Ω_i.

$$\Omega = \Omega_1 \times ... \times \Omega_d = [\lambda_1, \upsilon_1] \times ... \times [\lambda_d, \upsilon_d]$$

is the *base space* of the relation R. R is a finite subset of Ω, i.e., $R \subseteq \Omega$. R is partitioned into $P_R = \{p_1, ..., p_k\}$, a finite set of pages. Each page p stores a limited number of tuples.

A *region* ρ_i is a subspace of Ω. In contrast to [HNK+90] we neither require a region to be rectangular nor connected.

A *page p corresponds to a region* ρ ($p \leftrightarrow \rho$), if all tuples stored on p are located in the region ρ, i.e.,

$$p \leftrightarrow \rho \Leftrightarrow (x \in p \Leftrightarrow x \in \rho \cap p)$$

A *region partitioning* of Ω for a partitioned relation $P_R = \{p_1, p_2, ..., p_k\}$ is a set of regions $\Theta = \{\rho_1, ..., \rho_k\}$ with

$$\bigcup_{i=1}^{k} \rho_i = \Omega \text{ and } \forall_{j,i=1,...,k \text{ and } j \neq i}\, \rho_i \cap \rho_j = \varnothing \text{ and } \forall_{i=1,...,k}\, p_i \leftrightarrow \rho_i$$

A *query space* is some subspace of a relation defined by restrictions. Mostly restrictions define a *query box*, i.e., an iso-oriented multidimensional interval (hyper rectangle) between two points $y = (y_1, ..., y_d)$ and $z = (z_1, ..., z_d)$, which restricts each attribute A_j to the range $[y_j, z_j]$:

$$Q = [[y, z]] = [y_1, z_1] \times ... \times [y_j, z_j] \times ... \times [y_d, z_d]$$

A *processing range* for A_j is a linear interval $[a_j, b_j]$.

A *cluster* is a query box, which restricts one attribute A_j to a processing range $[a_j, b_j]$:

$$C_{[a_j,b_j]} = [\lambda_1, \upsilon_1] \times ... \times [a_j, b_j] \times ... \times [\lambda_d, \upsilon_d]$$

A *region wave* of a processing range $[a_j, b_j]$ of attribute A_j of a query space Q for a region partitioning Θ is the set of regions intersecting the cluster $C_{[a_j,b_j]}$ and Q with respect to space intersection:

$$W^r_{[a_j,b_j],Q}(\Theta) = \{\rho \in \Theta \mid \rho \cap Q \cap C_{[a_j,b_j]} \neq \varnothing\}$$

A *page wave* is the set of pages corresponding to the regions in the region wave $W^r_{[a_j,b_j],Q}(\Theta)$:

$$W_{[a_j,b_j],Q}(P_R) = \{p \in P_R \mid p \leftrightarrow \rho \text{ and } \rho \in W^r_{[a_j,b_j],Q}(\Theta)\}$$

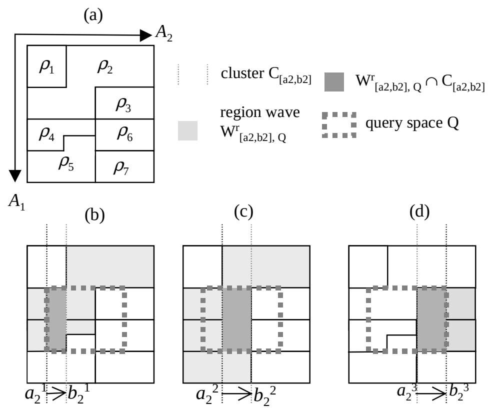

(a)
A_2
A_1
ρ_1
ρ_2
ρ_3
ρ_4
ρ_5
ρ_6
ρ_7
cluster $C_{[a2,b2]}$
$W^r_{[a2,b2], Q} \cap C_{[a2,b2]}$
region wave $W^r_{[a2,b2], Q}$
query space Q
(b)
(c)
(d)
$a_2^1 \rightarrow b_2^1$
$a_2^2 \rightarrow b_2^2$
$a_2^3 \rightarrow b_2^3$

Figure 3-1: Region partitioning, processing range, cluster, query space and region wave

A *slice* of a processing range $[a_j, b_j]$ is the set of tuples, which satisfy the restrictions of the query box and actually have a value of attribute A_j in this range:

$$S_{[a_j,b_j],Q}(R) = \{x \in R \mid x \in Q \cap C_{[a_j,b_j]}\}$$

While processing range, cluster and region wave denote subsets in one- resp. multidimensional space, page wave denotes a set of pages and slice denotes a set of tuples.

3.1 Basic Idea

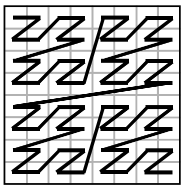
The Tetris algorithm [MB98] to process sort operations on attribute A_j of a partitioned relation P_R with restrictions defined by a query box $Q = [[y, z]]$ works as follows: for each iteration i the smallest possible page wave $W_{[a_j^i, b_j^i],Q}(P_R)$ is read into a main memory cache. For the first iteration the lower bound of the processing range a_j^1 is defined by y_j, the lower bound of the query box in attribute A_j. For the ith iteration the calculation method for b_j^i depends on the partitioning scheme, for UB-Trees we will describe an algorithm with constant I/O-complexity in Section 3.4. After completely reading the ith page wave the cached slice $S_{[a_j^i, b_j^i],Q}(R)$ is sorted internally according to A_j and transferred to the caller. The lower bound of the next processing range is defined by $a_j^{i+1} = b_j^i + 1$. Figure 3-1(b, c, d) shows iterations $i = 1, 2, 3$ of the Tetris algorithm sorting the gray-dotted query box on A_2 for the region partitioning of Figure 3-1a.

3.2 Choosing a suitable data structure

Using any variant of R-Trees [Gut84, BKK96] may result in a sub-optimal performance of the Tetris algorithm. R-Trees may subdivide the universe into overlapping tiles, which may result in multiple accesses to one disk page. Grid-Files [NHS84], hB-Trees [LS90] or space filling curves in combination with one-dimensional access methods [OM84, Jag90] provide a disjoint partitioning of the multidimensional space. Because of its easy implementation, we use the UB-Tree [Bay96] for our prototype implementation. However, the principal statements of this paper also hold for Grid-Files, hB-Trees or any other space filling curves mapped to one-dimensional access methods.

3.3 The UB-Tree

The UB-Tree [Bay96] uses a space filling curve to partition a multidimensional universe while preserving multidimensional clustering. Using the Lebesgue-curve (Figure 3-2a) it is a variant of the zkd-B-Tree [OM84].



(a)

	0	1	2	3	4	5	6	7
0	0	1	4	5	16	17	20	21
1	2	3	6	7	18	19	22	23
2	8	9	12	13	24	25	28	29
3	10	11	14	15	26	27	30	31
4	32	33	36	37	48	49	52	53
5	34	35	38	39	50	51	54	55
6	40	41	44	45	56	57	60	61
7	42	43	46	47	58	59	62	63

(b)

Figure 3-2: Z-addresses

To define the UB-Tree partitioning scheme we need the notion of Z-addresses and Z-intervals. We assume that each attribute value x_j of attribute A_j of a d-dimensional tuple $x = (x_1,...,x_d)$ consists of s bits[1] and we denote the binary representation of attribute value x_j by $x_{j,s-1}x_{j,s-2}...x_{j,0}$.

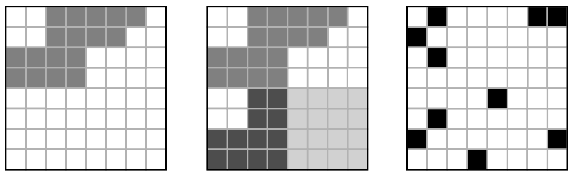
[1] Our implementation uses different lengths for the binary representation of attribute values. We just use identical lengths for an easy illustration.

A *Z-address* $\alpha = Z(x)$ is the ordinal number of a tuple x on the Z-curve and is calculated by interleaving the bits of the attribute values:

$$Z(x) = \sum_{i=0}^{s-1} \sum_{j=1}^{d} x_{j,i} \cdot 2^{i \cdot d + j - 1}$$

For an 8×8 universe, i.e., $s = 3$ and $d = 2$, Figure 3-2b shows the corresponding Z-addresses.

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. Figure 3-3a shows the Z-region [4: 20] and Figure 3-3b shows a partitioning with five Z-regions [0 : 3], [4 : 20], [21 : 35], [36 : 47] and [48 : 63].



(a) (b) (c)

Figure 3-3: Z-regions

The UB-Tree partitions the multidimensional space into Z-regions, each of which is mapped onto one disk page. At insertion time a full Z-region $[\alpha : \beta]$ is split into two Z-regions by introducing a new Z-address γ with $\alpha < \gamma < \beta$. γ is chosen so that the first half (in Z-order) of the tuples stored on Z-region $[\alpha : \beta]$ is distributed to $[\alpha : \gamma]$ and the second half is stored on $[\gamma+1 : \beta]$. Assuming a page capacity of 2 points Figure 3-3c shows ten points which created the partitioning of Figure 3-3b.

The UB-Tree requires logarithmic time for the basic operations of insertion, point retrieval and deletion.

3.4 The Tetris Algorithm for UB-Trees

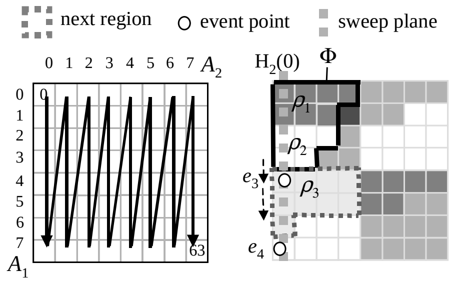
The Tetris algorithm for UB-Trees to sort a query box $[[y, z]]$ on A_j requires an order for retrieving the Z-regions and their corresponding pages from secondary storage. From the formula for $Z(x)$ we observe that for an attribute length of s bits and $j \in \{1,...,d\}$ the bits $\alpha_{(s-1)\cdot(j-1)} ... \alpha_{2\cdot(j-1)}\alpha_{j-1}$ of a Z-address α form the value x_j of attribute A_j. To sort data with respect to A_j we create the Tetris order from Z-addresses: for a Z-address $\alpha = Z(x_1, ..., x_d)$ we extract attribute $x_j = \mathit{extract}(\alpha, j)$ and concatenate ($\circ$) it to the Z-address $\mathit{reduce}(\alpha, j) = Z(x_1, ..., x_{j-1}, x_{j+1}, x_d)$ of the $(d-1)$-dimensional tuple $(x_1, ..., x_{j-1}, x_{j+1}, x_d)$, i.e.,

$$T_j(x) = \text{extract}(Z(x), j) \circ \text{reduce}(Z(x), j)$$

The ordinal numbers $T_j(x)$ create a total ordering on A_j from partial order of the ordinal numbers $Z(x)$. Figure 3-4a shows the Tetris ordering $T_2(x)$ and the corresponding ordinal numbers. $H_j(v) = \{Z(x) | x_j = v\}$ defines the Z-addresses of a *hyper-plane* on a data space.

Iteration i of the Tetris algorithm retrieves Z-regions and their corresponding pages in Tetris order to complete the region wave to define the processing range $[a_j^i, b_j^i]$. In the following let Φ be the space already retrieved by the

Tetris algorithm. Φ is constructed iteratively by adding the next Z-region to the already retrieved space.



(a) (b)

Figure 3-4: Tetris ordering

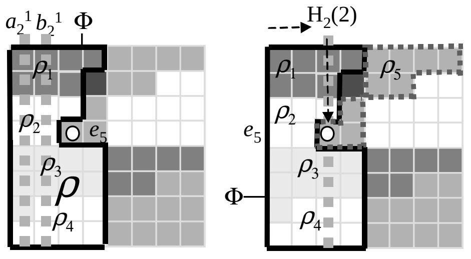
For the lower limit a_j^i of the current processing range the next Z-region to be retrieved is calculated by determining the *next event point* on the sweep plane $H_j(a_j^i)$ in Tetris order that is not part of the retrieved space Φ (T_j^{-1} denotes the inverse function of T_j):

$$e = (e_1, e_2 ..., e_d) = \text{eventpoint}(\Phi, Q) =$$
$$T_j^{-1}(\text{MIN}\{ T_j(x) \mid (x \in Q) \text{ and } (x \notin \Phi)\})$$

After complete traversal of the hyperplane $H_j(a_j^i)$, i.e., $H_j(a_j^i) \subseteq \Phi \cap Q$, and retrieval of the corresponding Z-regions $W^r_{[a_j^i, a_j^i],Q}(\Theta)$, the next position of the sweep plane a_j^{i+1} and the least upper bound $b_j^i = a_j^{i+1} - 1$ of the current processing range is determined by the first *event point* not intersecting the sweep plane:

$$\text{if } H_j(a_j^i) \subseteq \Phi \cap Q \text{ then } a_j^{i+1} = e_j$$
$$\text{where } e = \text{eventpoint}(\Phi, Q)$$

For sorting the entire universe in A_2 Figure 3-4b shows the flow of operation of the Tetris algorithm for the sweep plane $H_2(0)$ after ρ_1 and ρ_2 have been retrieved. The *event point* e_3 yields ρ_3, the next region to be retrieved. The *event point* e_4 computed upon retrieval of ρ_3 does not move the sweep plane, since the entire hyper-plane has not been retrieved yet, i.e., $H_2(0) \not\subseteq \Phi \cap Q$.



(a) (b)

Figure 3-5: Processing a relation in Tetris ordering

Figure 3-5a shows the complete *region wave* after the region wave $\{\rho_1, \rho_2, \rho_3, \rho_4\}$ has been retrieved. The next *event point* e_5 according to the retrieved space Φ defines the sweep plane $H_2(a_2^2)=H_2(2)$ in Figure 3-5b. The upper bound b_2^1 (=1) of the processing range of iteration 1 then is computed by decrementing a_2^2 (=2).

Two iterations of an actual run of the Tetris algorithm created by a visualization tool are illustrated in Figure 3-6. The thick bordered query box is sorted from bottom to top according to the vertical dimension. The part of each Z-region from which tuples are cached is shaded in this figure. The visualization also gives a hint why we named this algorithm *Tetris algorithm*: The processing order of the regions reminds us of the Tetris computer game.

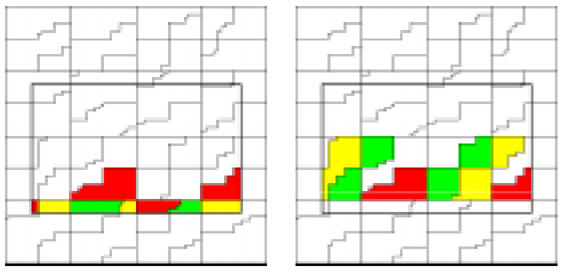


Figure 3-6: Sorted processing with Tetris

A large volume of a cached region does not mean that a lot of cache memory is needed: The maximum number of tuples in each Z-region and thus maximum cache size for each cached Z-region is limited by the capacity of a page.

```
/* sort the query box according to attribute j and produce
   a sorted stream of tuples */
void tetris(Table *t, QueryBox *Q,int j,
            ResultBuffer *resultBuffer)
{
 Z_address     z_eventPoint, z_end, ad_tuple;
 Region        r;
 Tuple         tup;
 Proc          procRange_a, procRange_b;
 RegionSet     phi;
 TupleCache    tetrisCache;
 z_address(&(Q->low),&z_eventPoint);
 z_address(&(Q->high),&z_end);
 procRange_a = extract(&z_eventPoint, j);
 do{
 /* retrieve the Z-region containing the event
    point and cache all tuples intersecting Q */
   retrieveRegion(t, &z_eventPoint,&r);
   insertIntoCache(&r,Q,&tetrisCache);
   /* update phi and calculate the next even point  */
   insertIntoRetrievedRegionSet(&r,&phi);
   nextEventPoint(&phi,q,&z_eventPoint);
 /* if a new slice is completed, sort the cache and
    return all tuples intersecting the  processing  range */
   if(procRange_a < extract(&z_eventPoint,j)){
     procRange_b = pre(extract(&z_eventPoint,j));
     sortCache(&tetrisCache,j);
     getNextTupleFromCache(&tetrisCache,&tup);
     while((!emptyCache(&tetrisCache)) &&
      (extract(z_address(&tup,&ad_tuple),j)<=
            procRange_b))
     {
        insertToResultBuffer(resultBuffer,&tup);
        removeFromCache(&tetrisCache,&tup);
        getNextTupFromCache(&tetrisCache,&tup);
     }
     procRange_a = extract(&z_eventPoint,j);
   }
 }while(T(&z_eventPoint,j)<=T(&z_end,j));
}
```

Figure 3-7: Tetris algorithm for sorting Q on A_j

Retrieving a Z-region by the above algorithm merely requires one point query in the UB-Tree (i.e., one B*-Tree search) and inexpensive bit operations. Since almost all levels of a B*-Tree are cached during the normal operation of a DBMS, in average only one page access is necessary to retrieve a Z-region. To sort a query box with the Tetris algorithm only the Z-regions overlapping the query box are retrieved and each Z-region is only accessed once to sort a query box with respect to one attribute. The C implementation of the main routine of the Tetris algorithm is shown in Figure 3-7.

3.5 Correctness of the Tetris Algorithm

Using the formal model of Section 3 the correctness of the Tetris algorithm can easily be proven. We just state the idea of the proof here: The Tetris order defines an ordering on the multidimensional space that is identical to the requested sort order. The Tetris algorithm is correct, since Z-regions are accessed in Tetris order and no Z-region intersecting the query box is skipped.

4 Performance Analysis

For sorting a relation in combination with restrictions in some attributes we define cost functions for response times and intermediate temporary storage. Our analysis considers the Tetris algorithm, an index organized table (IOT, clustering B*-Tree) and a full table scan (FTS) with a merge sort algorithm.

4.1 The Cost Model

In accordance with [HR96] we use a cost model that takes random pages accesses and page transfers into account. Let t_π be the (average case or worst case) positioning time and t_τ be the transfer time of a hard disk. We assume that the prefetching strategy of the file system reads a physical cluster of C consecutive pages from disk with one random access into the read-ahead cache. This takes time $t_\pi + t_\tau \cdot C$. Reading k pages in consecutive order therefore takes

$$c_{\text{scan}}(k) = \lceil k/C \rceil \cdot t_\pi + \max(k, C) \cdot t_\tau$$

4.2 Cost Functions

Using the cost model of Section 4.1 we calculate the cost of sorting a relation of P pages restricted by a multidimensional interval Q = [[y, z]] with a selectivity of s_j in attribute A_j using a main memory of M pages and a merge degree of m for the merge-sort algorithm. For UB-Trees we assume a d-dimensional organization of the relation. We divide the sort process in a retrieval phase (which retrieves the data to create initial runs for the merge-sort) and a sort phase (which actually performs the merge-sort). Using a full table scan (FTS) for the retrieval phase results in the formula for $c_{\text{fts-sort}}$. Using an index organized table (IOT) with a composite B*-Tree in lexicographic order A_1, ..., A_d allows to use the index for restriction in A_1 at the expense of having a random access for each page access of the retrieval phase. If $M > p \cdot \Pi s_i$, sorting takes place in main memory. The merge sort factor of $c_{\text{fts-sort}}$ and $c_{\text{iot-sort}}$ then is reduced to zero. For $c_{\text{iot-sort}}$ the merge sort factor is zero also, if the sorting attribute is A_1.

$$\boxed{\begin{aligned}
c_{\text{fts}} &= \left(t_\pi \cdot \left(\frac{1}{C} \right) + t_\tau \right) \cdot P \\
c_{\text{iot}} &= s_1 \cdot P \cdot (t_\pi + t_\tau) \\
P_{\text{sort}} &= 2 \cdot \left(P \cdot \prod_{i=1}^{d} s_i \right) \cdot \log_m \left(\frac{p}{M} \cdot \prod_{i=1}^{d} s_i \right) \\
c_{\text{sort}} &= \left(t_p \cdot \left(\frac{1}{C} \right) + t_t \right) \cdot P_{\text{sort}} \\
c_{\text{fts-sort}} &= c_{\text{fts}} + c_{\text{sort}} \\
c_{\text{iot-sort}} &= c_{\text{fts}} + c_{\text{sort}} \\
c_{\text{tetris}} &= (t_\pi + t_\tau) \cdot \prod_{j=1}^{d} n_j(d, P, y_j, z_j)
\end{aligned}}$$

Figure 4-1: Cost functions

c_{tetris} for uniformly distributed data partitioned by a d-dimensional UB-Tree and a query box [[y, z]] is the product of the number of Z-regions intersecting the multidimensional interval [[y, z]] in each dimension [Mar99]. For each attribute A_j the cost function c_{tetris} requires the values y_j and z_j to be normalized to [0, 1]. The formula

$$\begin{aligned} n_j(d, P, y_j, z_j) = n(y_j, z_j, l_j(d, P)) + ((n(y_j, z_j, l_j(d, P) + 1) \\ - n(y_j, z_j, l_j(d, P))) \cdot p_j(d, P) \end{aligned}$$

to calculate the number of Z-regions intersected by the restriction [y_j, z_j] in attribute A_j of a d-dimensional UB-Tree requires the following auxiliary functions:

- lower bound of recursive splits in A_j:

$$l_{j\downarrow}(d, P) = \left\lfloor \frac{\lfloor \log_2 P \rfloor}{d} \right\rfloor$$

- actual number of completed recursive splits in A_j:

$$l_j(d, P) = \begin{cases} l_{j\downarrow}(d, P) + 1 & \text{, if } \lfloor \log_2 P \rfloor \bmod d \le j \\ l_{j\downarrow}(d, P) & \text{, otherwise} \end{cases}$$

- probability of an incomplete split in A_j:

$$p_j(d, P) = \begin{cases} \dfrac{P}{2^{\lfloor \log_2 P \rfloor}} - 1 & \text{, if } j = (\lfloor \log_2 P \rfloor \bmod d) + 1 \\ 0 & \text{, otherwise} \end{cases}$$

- number of Z-regions for l_j completed splits in A_j:

$$n(y_j, z_j, l_j) = \begin{cases} 2^{l_j} - \lfloor y_j 2^{l_j} \rfloor & \text{if } z_j = 1 \wedge y_j \ne 1 \\ \lfloor z_j 2^{l_j} \rfloor - \lfloor y_j 2^{l_j} \rfloor + 1 & \text{otherwise} \end{cases}$$

Our measurements have shown that this rather complicated cost function describes the actual behavior of the UB-Tree very accurately [Mar99].

4.3 Response Times

Current operating systems usually prefetch C = 16 pages with one random access. For our cost analysis for sorting attribute A_2 with restrictions on attribute A_1 we assume t_π = 10 ms and t_τ = 1 ms, a main memory cache of 32 MB and a merge degree of m = 2.

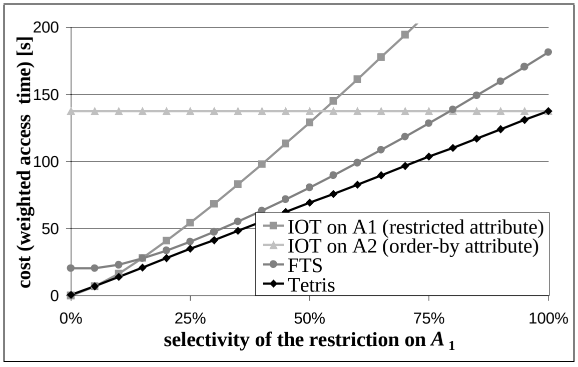


Figure 4-2: Sorting on A_2 with a restriction in A_1

Varying the selectivity of the restriction in A_1 for a table size of 125k pages (about 1GB for a page size of 8kB) shows that the Tetris algorithm is superior to both an IOT on A_1, an IOT on A_2 and a FTS (Figure 4-2). The figure also shows that an IOT on A_1 can only compete with a FTS if A_1 is restricted sufficiently. An IOT on A_2 is only competitive if A_1 is hardly restricted. Varying the table size for a selectivity of s_1 = 20 % in A_1 confirms these observations (Figure 4-3).

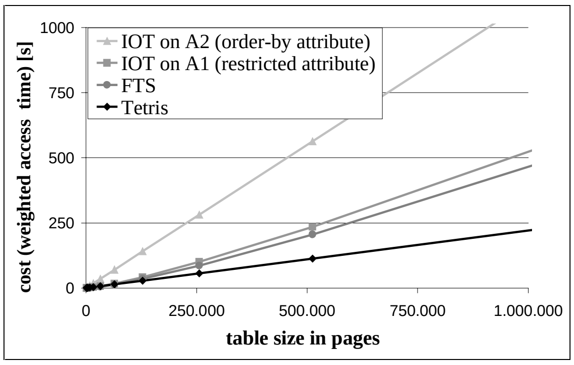


Figure 4-3: Sorting on A_2 with a restriction in A_1

4.4 Intermediate Storage and Pipelined Processing

The Tetris algorithm requires less temporary storage than FTS or IOT on a restricted attribute: While an IOT on a sorting attribute requires no additional memory, the merge-sort algorithm used by an FTS or by an IOT on a restricted attribute requires a memory size of $p \cdot \Pi s_i$ pages. To sort A_j the Tetris algorithm just caches one slice, i.e.,

$$cache_{\text{tetris}}(d, P, y, z, j) = \prod_{\substack{i=1,..,d \\ i \neq j}} n_i(d, P, y_i, z_i)$$

For two-dimensional UB-Trees the cache size is a square root function of the number of Z-regions overlapping the query box, i.e., $cache_{\text{tetris}}(P,s_1,s_2) = \sqrt{P \cdot s_1 \cdot s_2}$.

The completion of a wave also means, that the corresponding slice is available in sort order. Thus, first results are available for further processing after a time of $cache_{\text{tetris}}(d,P,x,y) \cdot (t_\pi + t_\tau)$. For an IOT on a restricted attribute and a FTS it is necessary to wait until the entire merge sort is completed. This yields a tremendous performance advantage of the Tetris algorithm for pipelined processing. Using the parameters of section 4.3, Figure 4-4 shows the intermediate storage sizes for the query of Figure 4-3 (s_1 = 20%). Qualitatively Figure 4-4 also displays the time delay, until the first result is available.

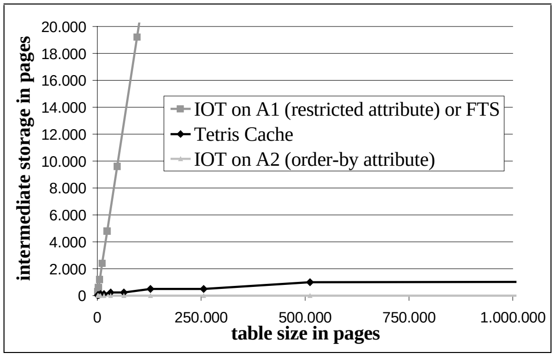


Figure 4-4: Intermediate storage sizes

4.5 Further Results

Using our cost functions we found out that for sort operations with restrictions in some attributes a multidimensional partitioning and the Tetris algorithm are superior to one-dimensional access methods, unless a strongly preferred sort order on one attribute per relation exists or the restrictions are not selective enough to make up the tenfold speed of the FTS.

5 Tetris Algorithm and Query Processing

Relational queries or SQL queries consist of restrictions, projections, ordering, grouping and aggregation and join operations. The Tetris algorithm can accelerate all of these operations. The performance results reported in this section were measured with a prototype implementation of the Tetris algorithm for UB-Trees on top of Oracle 8. We compare the performance to native query processing techniques of Oracle 8, namely access via an index organized table (IOT), which essentially stores a relation in a clustered B*-Tree, and access via a full table scan (FTS) of an entire relation. We used a SUN ULTRA SPARC II with 512 MB main memory and an array of five 4 GB hard disks with an average positioning time of 8ms and a transfer rate of 0.7ms per page. To show the performance gain of the Tetris algorithm we selected the queries Q3, Q4 and Q6 of the TPC-D benchmark [TPC97].

The performance figures reported in this paper disfavor the Tetris algorithm to some extent: The measurements were conducted with UB-Trees emulated on top of Oracle against IOTs and FTS integrated into the Oracle kernel.

5.1 Joins and Restrictions

Query Q3 of the TPC-D benchmark illustrates a complex query with restrictions and join operations involving three relations, which is efficiently processed by the Tetris algorithm.

```
SELECT L_ORDERKEY, ,O_ORDERDATE, O_SHIPPRIORITY,
 SUM(L_EXTENDEDPRICE*(1- L_DISCOUNT)) AS REVENUE
FROM CUSTOMER, ORDER, LINEITEM
WHERE
   C_MKTSEGMENT = 'FOOD' AND
   C_CUSTKEY = O_CUSTKEY AND
   L_ORDERKEY = O_ORDERKEY AND
   O_ORDERDATE < DATE 1.5.98 AND
   L_SHIPDATE > DATE 1.6.98
GROUP BY L_ORDERKEY, O_ORDERDATE, O_SHIPPRIORITY
ORDER BY REVENUE DESC, O_ORDERDATE
```

Figure 5-1: Query Q3 of the TPC-D benchmark

The operator tree for Q3 generated by a standard RDBMS is illustrated in Figure 5-2. We use σ for the selection operator, $\bowtie$ for the join operator, γ for grouping, ω for ordering and M for a merge operator on a sorted stream of tuples. The query is processed by first applying the restrictions on each table and then performing a hash join or a sort-merge join on the intermediate result. The join order of Figure 5-2 is due to the fact that the LINEITEM relation is four times larger than the order relation and 40 times larger than the CUSTOMER relation. The intermediate result of the second join is used for grouping with aggregation and final ordering.

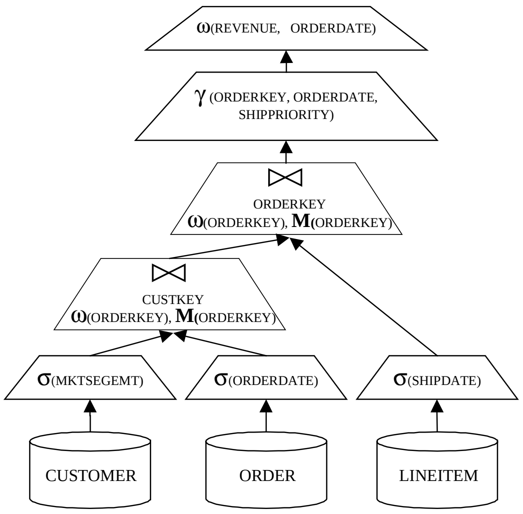

ω(REVENUE, ORDERDATE)
γ (ORDERKEY, ORDERDATE, SHIPPRIORITY)
⋈ ORDERKEY ω(ORDERKEY), M(ORDERKEY)
⋈ CUSTKEY ω(ORDERKEY), M(ORDERKEY)
σ(MKTSEGEMT)
σ(ORDERDATE)
σ(SHIPDATE)
CUSTOMER
ORDER
LINEITEM

Figure 5-2: Standard Operator Tree for Q3

With UB-Trees on CUSTOMER, ORDER and LINEITEM Figure 5-3 and Figure 5-4 illustrate the Tetris operator $\tau_{\sigma,\omega}$, which combines selection and sorting. Reading the restricted part of each relation in sort order of the join attribute causes a sorted stream of tuples. This stream is transferred to the merge operator M and processed further.

We measured the sorted table accesses of query Q3 for TPC-D scaling factors (SF) from 0.1 to 1 (SF = 1 means a size of 1GB for LINEITEM). We do not want to enter the debate whether sort-merge joins or hash joins do perform better [Mer81, DKO+84]. We chose a large main memory for our test environment, since according to [CHH+91] sort-merge join and hash join have a similar performance for computer systems with large main memories. Consequently we assume a sort merge-join.

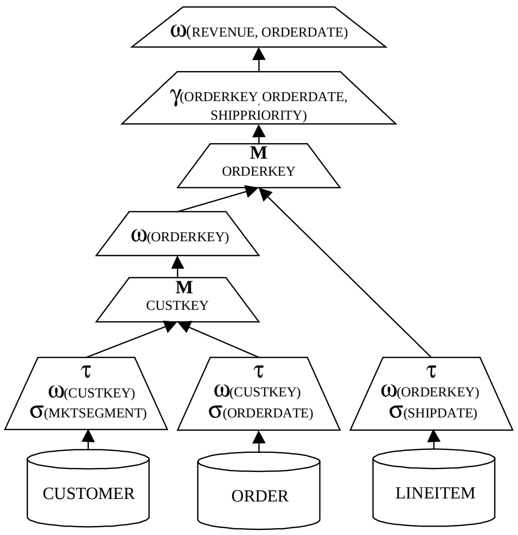

ω(REVENUE, ORDERDATE)
γ(ORDERKEY, ORDERDATE, SHIPPRIORITY)
M ORDERKEY
ω(ORDERKEY)
M CUSTKEY
τ ω(CUSTKEY) σ(MKTSEGMENT)
τ ω(CUSTKEY) σ(ORDERDATE)
τ ω(ORDERKEY) σ(SHIPDATE)
CUSTOMER
ORDER
LINEITEM

Figure 5-3: Tetris Operator Tree for Q3

Since the LINEITEM table is the major bottleneck for Q3, we focus on this relation for our performance comparison. We created four instances of LINEITEM: an IOT on SHIPDATE, an IOT on ORDERKEY and secondary indexes on each restricted or sorted attribute.

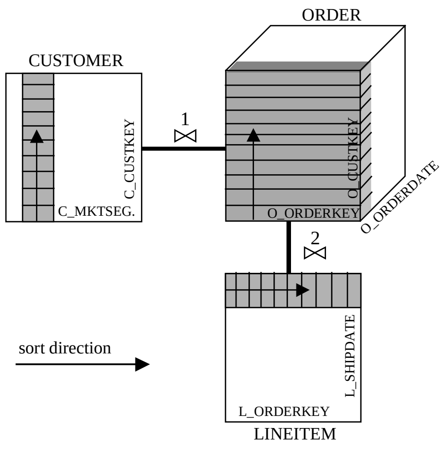


Figure 5-4: Processing Q3

The optimizer favored a FTS over secondary indexes, which our theoretical considerations and measurements proved to be the right decision (forcing Oracle to process Q3 with a secondary index on SHIPDATE or ORDERKEY took more than 6 hours for SF = 1). Thus we exclude secondary indexes from further considerations.

Figure 5-5 and Table 5-1 show that the Tetris algorithm for UB-Trees is most preferable to answer this query. The 50% restriction on SHIPDATE is not selective enough for an IOT on SHIPDATE to be competitive. The presorted IOT on ORDERKEY does not require a merge sort and therefore shows response times similar to a FTS with merge sort. Using Tetris for sorting LINEITEM is more than three times faster than FTS or any IOT. The first response of Tetris is already produced after few seconds, two to three orders of magnitude faster than with FTS or any IOT. While the intermediate storage requirements of Tetris are not exactly zero as for an IOT on ORDERKEY, they are extremely low: Compared to an FTS or an IOT on SHIPDATE they are several orders of magnitude lower. Since for FTS and IOT on SHIPDATE storage requirements grow linearly with *tablesize*, the main memory is exceeded soon: For our measurements we several times had to enlarge the temporary Oracle tablespaces.

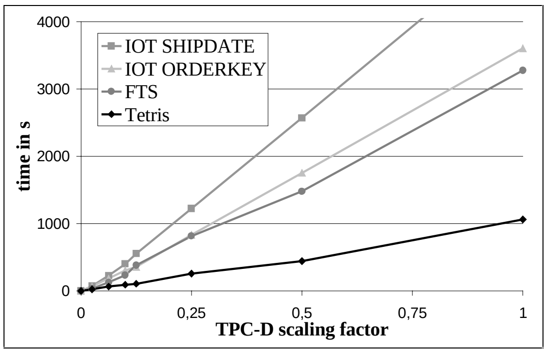


Figure 5-5: Q3 Response Times

As predicted by our cost functions the Tetris cache of Figure 5-6 is almost 300 times lower than the intermediate storage of the IOT on SHIPDATE or FTS; even for a LINEITEM table sizes of several Terabytes the Tetris cache easily fits into the RAM of current computers.

Table Size Scaling Factor (SF)	326 MB (0.25)	651MB (0.5)	1302MB (1)
Tetris 1st response	1,3s	1,3s	3,3s
Tetris Slices	256	256	512
Time IOT ORDERKEY	834.3s	1753.6s	3604.1s
Time IOT SHIPDATE	1223.7s	2569.8s	5286.4s
Time FTS-Sort	816.5s	1479.4s	3276.4s
Time Tetris	257.5s	441.2s	1062.2s
Cache Tetris	1.4MB	2.1MB	2.6MB
Temp Storage IOT/FTS	183MB	326MB	751MB

Table 5-1: Interactive response times and cache sizes for sorting 50 % of LINEITEM

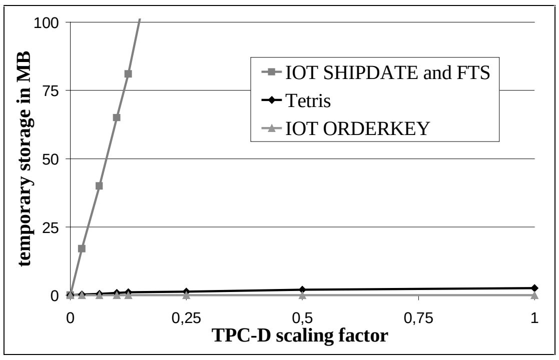


Figure 5-6: Q3 Cache Size

5.2 Joins, Grouping and Restrictions

Restrictions, joins and grouping are efficiently handled by the Tetris algorithm. Then external sorting is avoided usually, while multi-attribute restrictions are utilized to reduce I/O. We illustrate processing of these operations with the Tetris algorithm by Q4 of the TPC-D benchmark.

```
SELECT O_ORDERPRIORITY, COUNT(*)
FROM ORDER
WHERE
  O_ORDERDATE >= DATE '[date]' AND
  O_ORDERDATE < DATE '[date]' + INTERVAL '3'
MONTH AND
  EXISTS (
    SELECT *
    FROM LINEITEM
    WHERE
      L_ORDERKEY = O_ORDERKEY AND
      L_COMMITDATE < L_RECEIPTDATE )
GROUP BY O_ORDERPRIORITY
ORDER BY O_ORDERPRIORITY
```

Figure 5-7: Q4 of the TPC-D benchmark

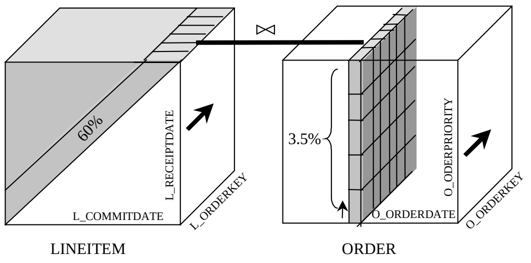


Figure 5-8: Processing Q4

Assuming a three dimensional organization of ORDER and LINEITEM, query processing with the Tetris algorithm is shown in Figure 5-8. Q4 groups the restricted ORDER table depending on tuple existence in the LINEITEM table. Efficiently processing this query means processing ORDER in ORDERKEY order while using the 3.5%-restriction on ORDERDATE. To evaluate the existential restriction, LINEITEM is processed in ORDERKEY order and semi-joined to ORDER. The Tetris-algorithm can be used to process the triangular search space defined by COMMITDATE < RECEIPTDATE in ORDERKEY order. CPU cost for comparisons may be saved by processing each ORDERDATE-slice in ORDERPRIORITY order.

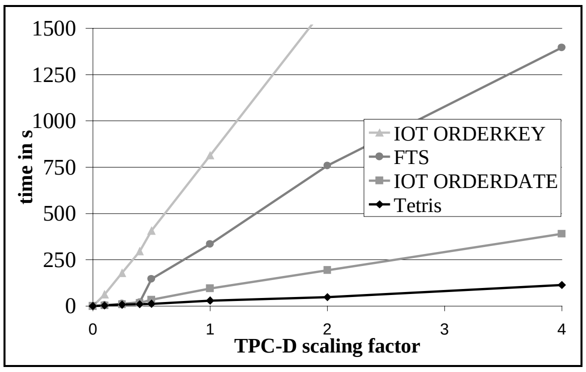


Figure 5-9: Q4 Response Times

We just report the response times and cache sizes of sorting ORDER in Table 5-2, since the enhancement of the

Tetris algorithm for non-rectangular query spaces has not been implemented yet.

Table Size Scaling Factor (SF)	322 MB (1)	750MB (2)	1.5GB (4)
Tetris 1st response	0.1s	0.2s	0.3s
Tetris Slices	256	256	512
Time IOT ORDERKEY	813.8s	1627.5s	3254.9s
Time IOT ORDERDATE	95.4s	194.2s	390.4s
Time FTS-Sort	335.2s	758.6s	1396.7s
Time Tetris	29.7s	47.8s	113.9s
Cache Tetris	0.2MB	0.2MB	0.3MB
Temp Storage IOT/FTS	12.9MB	30.1MB	60.1MB

Table 5-2: Interactive response times and cache sizes for sorting 3.5% of ORDER

The restrictions on ORDER are selective enough for an IOT on ORDERDATE to be superior to FTS and IOT on ORDERKEY. The Tetris algorithm is superior to FTS and any IOT, since it utilizes restrictions and sorts the data at the same time. Even for this quite selective ORDERDATE restriction the Tetris algorithm is more than three times faster than the IOT on ORDERDATE. The Tetris algorithm for UB-Trees also is 11 times faster than an FTS and about 30 times faster than an IOT on ORDERKEY.

5.3 Multi-attribute Restrictions

Like standard range query algorithms, the Tetris algorithm may be used to efficiently process multi-attribute restrictions. Query Q6 of the TPC-D benchmark is a typical example for this kind of query.

```
SELECT SUM(L_EXTENDEDPRICE*L_DISCOUNT)
FROM LINEITEM
WHERE
  L_SHIPDATE >= [date] AND
  L_SHIPDATE <= [date] + INVERVAL 1 YEAR AND
  L_DISCOUNT BETWEEN [d] –0.01 AND [d] + 0.01
  AND L_QUANTITY < [quantity]
```

Figure 5-10: Query Q6 of the TPC-D benchmark

Q6 is processed by either using an IOT on SHIPDATE to materialize the result and then check the conditions on DISCOUNT and QUANTITY, or perform a FTS, if no such index exists. Figure 5-11 shows the portion of the LINEITEM relation that the Tetris Algorithm and an IOT on SHIPDATE process in order to answer Q6. Although a FTS retrieves the entire relation, prefetching strategies substantially reduce the number of random accesses and make the FTS superior to any IOT.

Performing an index intersection on three secondary B*-Trees is not very efficient, since the selectivity of an individual attribute is relatively low (20%, 33% resp. 50%). An intersection of bitmap indexes is not a good choice either, since the number of distinct values for SHIPDATE, DISCOUNT and QUANTITY is quite high. Since 1/30th of all tuples of LINEITEM satisfy the restrictions of Q6, 200k tuples have to be retrieved to process the query for a TPC-D scaling factor of 1. Since bitmap indexes and secondary B-Trees do not cluster the data, an FTS is preferable to both access methods. Multidimensional indexes cluster data symmetrically with respect to all index attributes. With 8kB pages 80 tuples of the LINEITEM relation are stored together on one page. Accessing 200k tuples then means 2.5k random disk accesses. Thus a multidimensional index is useful for this type of query.

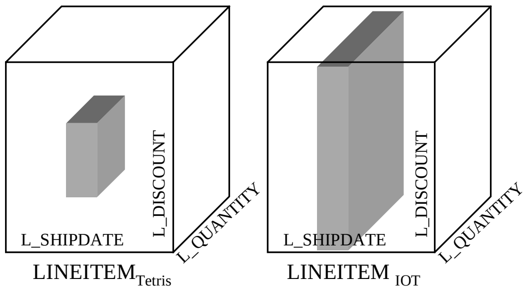


Figure 5-11: Processing Q6 with Tetris and IOT

For Q6 we created five instances of LINEITEM, namely a UB-Tree, an IOT on each restricted attribute and a table with three secondary B-Trees, one on each restricted attribute. As expected it was not possible to make the optimizer perform an index intersection. The optimizer always preferred a FTS instead. Forcing the optimizer to use a single secondary index on SHIPDATE (the most selective attribute) was much less efficient than a FTS. Since this verifies our theoretical expectations, we exclude secondary indexes also from our performance comparison for Q6.

Table Size Scaling Factor (SF)	326 MB (0.25)	651MB (0.5)	1302MB (1)
Time IOT QUANTITY	460,7s	921,4s	1842,8s
Time IOT DISCOUNT	339,2s	678,4s	1356,8s
Time IOT SHIPDATE	208,1s	416,3s	832,5s
Time FTS	47,7s	93,9s	187,6s
Time Tetris	12,0s	21,3s	30,5s

Table 5-3: Interactive response times for Q6

Table 5-3 and Figure 5-12 again show the superiority of a multidimensional organization over classical access methods by a sixfold speedup of the Tetris algorithm for UB-Trees over an FTS and by a speedup of two to three orders of magnitude over any IOT.

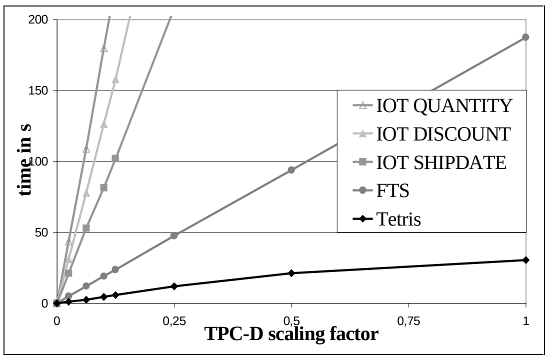


Figure 5-12: Performance of Q6

6 Conclusions and Future Work

When sorting a relation or joining relations, restrictions in multiple attributes should be efficiently utilized in order to reduce I/O-cost and CPU-cost. The Tetris algorithm uses a multidimensional access method and a sweep plane technique to combine the sort process and the evaluation of multi-attribute restrictions in one single processing step. We have shown that for dimensionalities typical for relational databases only I/O-time linear in the size of the result set and sublinear temporary storage are necessary to perform the Tetris algorithm. In contrast to a merge-sort algorithm results are produced in a continuous flow of operation. Therefore sorting is no longer a blocking operation. Compared to existing techniques, the first results are available much earlier and thus allow better interactive response times and better internal pipelining of the data. The benchmark results for three queries of the TPC-D benchmark show speedups of up to two orders of magnitude in response time. Depending on the query, temporary storage requirements are reduced by several orders of magnitude. Further analysis indicates the usability of our approach for an even broader range of queries.

In our future work we are particularly interested in a detailed study of relational query processing with multidimensional indexes. We are in the process of investigating a methodology for query optimization with multidimensional indexes, both for heuristics-based and cost-based query optimizers. We will apply these techniques to practical application scenarios of our commercial project partners. Applications with real-world customer data then will prove the practical usability of our methods.

Acknowledgments

We thank our project partners SAP, Teijin Systems Technology, NEC and Hitachi for funding this research work. We also thank our master student Roland Pieringer for his effort in doing the performance measurements reported in this paper.

References

[Bay96] Bayer, R.: The universal B-Tree for multidimensional Indexing. Technical Report TUM-I9637, 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, LNCS, Springer, 1997

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

[BKK96] Berchtold, S. D.; Keim, D.; Kriegel, H.-P.: The X-Tree: An Index Structure for high-dimensional Data. 22nd Int. Conf. on Very Large Data Bases, 1996

[BM98] Bayer, R.; Markl, V.: A Multidimensional Index and its Performance on Relational Database Management Systems, Report, FORWISS 1998

[CHH+91] Cheng, J.; Haderle, D.; Hedges, R.; Iyer, B.R.; Messinger, T.; Mohan, C.; Wang, Y.: An efficient hybrid hash join algorithm: a DB2 prototype, 7th Intl. Conf. on Data Engineering, Kobe 1991, pp. 171-180

[DKO+85] DeWitt, D.J.; Katz, R.H.; Olken, F.; Shapiro, L.D.; et al.: Implementation Techniques for Main Memory Database Systems, SIGMOD Intl. Conf. on Management of Data, 1984, pp. 1-8

[FNP+79] Fagin, R.; Nievergelt, J.; Pippenger, N.; Strong, H. R.: Extendible Hashing – a fast access method for dynamic files. ACM TODS 4(3), 1979, pp. 315 - 344

[Gut84] Guttman: A dynamic Index Structure for spatial Searching. SIGMOD Intl. Conf. on Management of Data, 1984, pp. 47 – 57

[Gra93] Graefe, G.: Query Evaluation Techniques for Large Databases, Computing Surveys 25, pp. 73-170

[GG97] Gaede, V.; Günther, O.: Multidimensional Access Methods. Humboldt Universität, Berlin, 1997, http://www.wiwi.hu-berlin.de/~gaede/survey.rev.ps.Z

[Gün93] Günther, O.: Efficient computations of Spatial Joins, 9th Int. Conf. on Data Engineering, Vienna, 1993

[GHR+97] Gupta, H.; Harinarayan, V.; Rajaraman, A.; Ullman, D. J.: Index Selection for OLAP. Intl. Conf. on Data Engineering, 1997

[HNK+90] Harada, L.; Nakano, M.; Kitsuregawa, M.; Takagi, M.: Query Processing Methods for Multi-Attribute Clustered Relations, 16th Int. Conf. on Very Large Databases, 1990, pp.59-70

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

[Inf97] Informix Software Inc.: A New Generation of Decision Support Indexing for Enterprisewide Data Warehouses, http://www.informix.com/informix/corpinfo/zines/whitpprs/wpxps.pdf, 1997

[Jag90] Jagadish, H.V.: Linear Clustering of Objects with multiple Attributes. ACM SIGMOD Intl. Conference on Management of Data, 1990, pp. 332 – 342

[LS90] Lomet, D.; Salzberg, B.: The hB-Tree: A Multiattribute Indexing Method with good guaranteed Performance. TODS, 15(4), 1990, pp. 625 – 658

[Mar99] Markl, V.: MISTRAL: Processing relational queries with multidimensional access techniques, Dissertation, Technische Universität München, 1999

[MB98] Markl, V.; Bayer, R.: The Tetris-Algorithm for Sorted Reading from UB-Trees, In: “Grundlagen von Datenbanken”, 10th GI Workshop, 1998

[ME92] Mishra, P.; Eich, M.H.: Join Processing in Relational Databases, Computing Surveys, Vol. 24 No.1, 1992, pp. 194-211

[Mer81] Merret, T.H.: Why sort-merge gives the best implementation of then natural join, SIGMOD Record 13, 1981, pp. 39 - 51

[NHS84] Nievergelt, J.; Hinterberger, H. ; Sevcik, K. C.: The Grid-File. TODS, 9(1), March 1984, pp. 38-71

[OQ97] O´Neill, P.; Quass, D.: Improved Query Performance with Variant Indexes. SIGMOD Intl. Conf. On Management of Data, Tucson, 1997, pp. 38-49

[OM84] Orenstein, J. A.; Merret, T. H.: A Class of Data Structures for Associate Searching. SIGMOD Intl. Conf. on Management of Data, 1984, pp. 294-305

[Ora97] Oracle Corp.: Oracle 8 Documentation, Oracle Corporation, 1997

[Rot91] Rotem, D.: Spatial Join Indices, Intl. Conf. on Data Engineering, 1991, pp. 500-509

[Sam90] Samet, H.: The Design and Analysis of Spatial Data Structures, Addison Wesley, 1990

[TPC97] TPC benchmark D. Transaction Processing Performance Council, http://www.tpc.org, 1997