Exploitation of Pre-sortedness for Sorting in Query Processing: The TempTris-Algorithm for UB-Trees

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

Bayerisches Forschungszentrum [+]
für Wissensbasierte Systeme
Orleansstr.34, 81667 München, Germany

IBM Almaden Research Center[*]
K55/B1, 650 Harry Road, San Jose,
CA 95120-6099,USA

zirkel@forwiss.tu-muenchen.de, bayer@in.tum.de, marklv@us.ibm.com
http://mistral.informatik.tu-muenchen.de

Abstract

Bulk loading is used to efficiently build a table or access structure, if a large data set is available at index creation time, e.g., the spool process of a data warehouse or the creation of intermediate results during query processing. In this paper we introduce the TempTris algorithm that creates a multidimensional partitioning from a one-dimensionally sorted stream of tuples. In order to achieve that, TempTris exploits the fact that a one-dimensional order can be used as a partial multidimensional order for the creation of a multidimensional partitioning. In this way, TempTris avoids external sorting for the creation of a multidimensional index. In combination with the Tetris sort algorithm, TempTris can be used to create intermediate query processing results that can – without external sorting - be re-used to generate various sort orders. As example of this new processing technique we propose an efficient algorithm for computing an aggregation lattice. Thus, TempTris can also be used to speed up the processing of CUBE operators that frequently occur in OLAP applications.

1 Introduction

In query processing, operators often produce intermediate results in a specific sort order, e.g., a clustering index access or a sort-merge join. In practice even spool files used for bulk loading in data warehouses (DW) are often sorted with respect to one dimension, e.g., the time dimension. During further processing, the sort order of the intermediate result or spool file can be used to efficiently compute the result of further operators like projection or joining. In this paper, the sort order of the input is exploited for generating a multidimensional partitioning.

A multidimensional organization of an input stream or table has many useful applications in query processing, e.g., when query processing requires answering a set of sub-queries with multi-attribute restrictions or when further processing the stream in different sort orders than the original one.

So far there has been already some work on bulk loading for multidimensional index structures, such as R-Trees [8], Gridfiles [13] and quad trees [7]. These algorithms have an I/O complexity of $O(P \cdot \log P)$ for an input size of P pages, which is usually due to the fact that these approaches do not utilize a pre-sorted input and thus require external sorting of the input data.

For B^*-Trees and multidimensional access methods on top of these, packing algorithms can be used with minor modifications. The common method to create a space-optimal B-Tree is to sort the data with merge-sort according to the index key and write tuples in sort order into disk pages, filling up each page to the desired degree of page utilization.

Nowadays, the usual method for sorting in database systems is the sort-merge algorithm [9], i.e. the input data is written into initial sorted runs and then merged into larger and larger runs until only one run – the sorted output - is left.

The contribution of our paper is to introduce the TempTris-Algorithm, a processing technique for the creation of a multidimensional partitioning without external sorting. TempTris generalizes the bulk-loading

algorithm for UB-trees [2] by utilizing the order of the input stream. The basic idea of TempTris is to utilize the linear ordering of a tuple stream for the creation of a multidimensional space partitioning based on a new target order. A sweep line - defined by the attribute the input stream is sorted on – is used to separate the stable part of the partitioning (which can safely be written to disk) from the dynamic part (where insertions still may take place and thus would require additional I/O, if this data was already written to disk). Therefore the TempTris algorithm is adaptive in the sense that it does less work as the input stream has some degree of *pre-sortedness*.

The typical application of TempTris is creating clustering indexes and tables in a bulk loading fashion. In this way, TempTris is useful both for the creation of permanent tables and for the on-the-fly creation of intermediate results as they occur during query processing.

In contrast to all previous work, our method utilizes the fact that the input is one-dimensionally sorted for the creation of the partitioning. We give a concise evaluation by creating UB-Trees with the TempTris algorithm. We compare our technique against the traditional sort-merge techniques that are state-of-the-art for index bulk loading. We also present an efficient processing technique for computation of aggregation networks [12] that combines Tetris and TempTris and apply it to the cube operator [4] on a real data warehouse from "GfK"[1].

TempTris is the inverse operation to Tetris (which creates a linearly ordered stream of tuples from a multidimensional partitioning, see[14]). This allows for carrying over to TempTris many of our analytical and experimental results of the Tetris algorithm. In particular, with sufficient, but modest cache memory, TempTris does not require external sorting.

The rest of the paper is organized as follows: Section 2 introduces the TempTris algorithm. Section 3 gives an example of the TempTris algorithm creating the Z-region partitioning of UB-Trees. Section 4 analyzes the performance of TempTris and compares it to the performance of merge-sort, the usual technique for bulk loading. In Section 5 we discuss a new efficient processing technique for the cube operator based on a combination of TempTris and Tetris. Section 6 presents measurements results and Section 7 concludes our paper and gives an outlook on future work.

2 The TempTris Algorithm

The basic idea of TempTris is to create a new *target order* from a sorted stream of tuples (i.e, a *source order*). This target order can be used to organize a relation with a multidimensional access method on secondary storage.

During a run of TempTris a *sweep line technique* [16] is used to distinguish between *dynamic data* (that consists of regions of a partitioning that must still be kept in memory cache for processing) and *stable data* (that consists of regions of a partitioning that can already be written to disk and will not be touched again). The direction of the sweep-line is determined by the sort order of the input stream. Basically, TempTris iteratively applies the following two steps:

- Insert a tuple according to the target order into a dynamic region and split the region if necessary
- Make all dynamic regions stable, which do not intersect the sweep line

2.1 Terminology

The following notations of relational database systems and the formal concept of multidimensional regions will serve as basic terminology for describing the TempTris Algorithm:

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 $<_{dim}$ ($dim \in \{1,\ldots,d\}$) be a *total order* on Ω_{dim} and min_{dim} resp. max_{dim} the minimum resp. maximum value of Ω_{dim}.

$$\Omega = \Omega_1 \times ... \times \Omega_d = [min_1,max_1] \times ... \times [min_d, max_d]$$

is the *base space* of the relation R. Each $<_{dim}$ defines a *partial order* on Ω. R is a finite subset of Ω, i.e., $R \subseteq \Omega$. R is partitioned into a finite set of pages. Each page p stores a limited number of tuples.

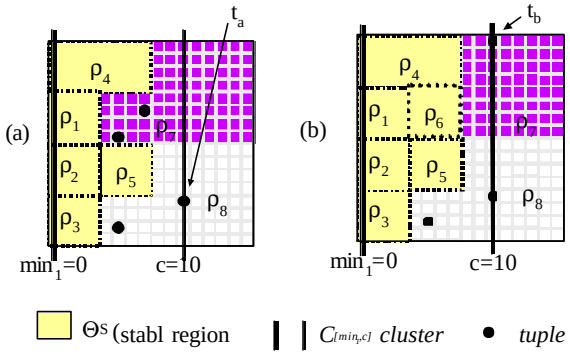

Figure 2-1: Terminology

A *region* r_i is a subspace of Ω, i.e. $r_i \subseteq \Omega$. A page p corresponds to a region r ($p \leftrightarrow r$), if all tuples stored on p are located in the region r, i.e.,

$$p \leftrightarrow r \;\hat{U}\; (x \in p \Leftrightarrow x \in r \cap p).$$

A (*disjoint*) *region set* Θ (cf. Figure 2-1a) of Ω is a set of regions $\{r_1, ..., r_k\}$ with

$$\forall_{i=1,...,k}\, r_i \in \Omega \text{ and } \forall_{j,i=1,...,k \text{ and } j \neq i}\, r_i \cap r_j = \varnothing$$

[1] GfK stands for "Gesellschaft für Konsumforschung", the largest German Market Research Company.

For a 16x16 universe Ω with A_1 as horizontal dimension, A_2 as vertical dimension and the origin (0,0) in the upper left corner, Figure 2-1a shows the geometry of 7 regions $\{r_1, r_2, r_3\ r_4, r_5, r_7, r_8\}$, Figure 2-1b shows a further region r_6 that has been created from the region set of Figure 2-1a by splitting r_7 into r_7 and r_6. For the regions r_7 and r_8, Figure 2-1 also shows the four (Figure 2-1a) respectively three (Figure 2-1b) tuples, that are stored on the pages corresponding to these regions.

A *cluster* (cf. Figure 2-1b) is a sub-space of Ω, which restricts one attribute A_{dim} to a linear interval $[a_{\dim}, b_{\dim}]$:

$$C_{[a_{\dim}, b_{\dim}]} = [min_1, max_1] \times ... \times [a, b] \times ... \times [min_d, max_d]$$

Figure 2-1 also shows the cluster $C_{[\min_1, t_{a1}]}$ for the tuple $t_{a1} = (t_{a1}, t_{a1}) = (10,12)$ with respect to attribute A_1, i.e., the cluster $C_{[0,10]}$.

A *sorted stream* $S_{R,<}$ is an ordered set that contains exactly the tuples of the relation *R*. The tuples of $S_{R,<}$ are ordered ascending by <. The function *removemin*($S_{R,<}$) removes the smallest tuple from the stream and returns it a result.

During a run of the TempTris algorithm, we call a page to be *stable* if the set of tuples stored on that page is fixed (i.e., no further tuple will be removed or inserted during further iterations); otherwise we call a page *dynamic*. We also use the term *stable region* (*dynamic region*) for the region corresponding to a stable (dynamic) page.

2.2 The TempTris Algorithm

TempTris divides the *base space* Ω into two disjoint region sets, the *dynamic region set* Θ^D and the *stable region set* Θ^S. For the following explanation we consider $A_{\dim}$ to be the attribute according to which the input stream is sorted. $t_{\dim}$ is the position of the sweep line, i.e., the attribute value of $A_{\dim}$ of the current tuple of the input stream. The stable region set is the set of regions fully contained in the cluster $C_{[\min_{\dim}, t_{\dim}-1]}$. The regions $\{r_1, r_2, r_3\ r_4, r_5\}$ in Figure 2-1a show the stable region set and $\{r_7, r_8\}$ is the dynamic region set for the sweep line with position $c = 10$. The same holds for Figure 2-1b, which contains one further stable region r_6 which has been created by splitting r_7 into r_7 and r_6.

During a run of TempTris, the stable page set corresponding to the stable region set Θ^S will be built on disk. In order to achieve that, TempTris caches the dynamic page set corresponding to the dynamic region set Θ^D in main memory.

Suppose we have a stream $S_{R,<_{\dim}}$ that is sorted with respect to attribute $A_{\dim}$. In the beginning, the *dynamic region set* Θ^D consists of one region that covers the whole *base space* Ω. The *stable region set* Θ^S is empty. Each tuple $t = removemin(S_{R,<_{\dim}})$ of the stream $S_{R,<_{\dim}}$ is iteratively inserted into a region ρ of the *dynamic region set* according to the target order. The value t_{dim} of the sort attribute of the current tuple *t* defines a *sweepline* that is used to distinguish Θ^D and Θ^S.

```
TempTris(S_{R,<dim}, dim, Θ^S)
{
    tuple t;
    region ρ, ρ_1,ρ_2;
    region-set Θ^D = {Ω};
    region-set Δ = ∅;
    Θ^S = ∅;
    while (S_{R,<dim} ? ∅)
    {
        t = removemin(S_{R,<dim});
        /* find the (unique) dynamic region containing t */
        ρ = {ρ | ρ ∈ Θ^D and t ∈ ρ }
        /* insert t into corresponding page */
        page(ρ_1) = page(ρ_1) ∪{t};
        if(|page(ρ_1)| > max_number_of_elements)
        { /* region overflow caused by page overflow */
            /* split r into r_1 and r_2 as well as the */
            /* corresponding pages */
            split(ρ ,ρ_1, ρ_2);
            /* update dynamic region set */
            Θ^D = Θ^D \ {ρ } ∪ {ρ_1 } ∪ {ρ_2 }
        }
        Δ = { ρ | ρ ∈ Θ^D and ρ ∩ C_{[min_dim ,t_dim −1]} = ρ }
        if(Δ ? ∅)
        {
            /* cache flushing: flush all new stable regions */
            /* to disk */
            /* write pages corresponding to D to disk */
            Θ^S = Θ^S ∪Δ
            /* remove pages corresponding to D from */
            /*main memory */
            Θ^D = Θ^D \ Δ
        }
    }
    /* write pages corresponding to the remaining dynamic */
    /* regions to disk */
    Θ^S = Θ^S ∪ Θ^D
}
```

Figure 2-2: TempTris Algorithm (pseudo code)

When consecutively inserting tuples, the sweepline moves forward. Θ^D grows when a region ρ is split into two regions r_1 and r_2 due to an overflow of the corresponding page. When a dynamic region is no longer intersected by the sweepline, it becomes stable and is removed from Θ^D (*cache flushing*). When the last tuple *t* of $S_{R,<_{\dim}}$ has been processed, the remaining *dynamic region set* Θ^D becomes stable and is also written to disk. At this point the multidimensional region partitioning has

been created. The TempTris algorithm is sketched in Figure 2-2.

2.3 Correctness of TempTris

The correctness of the TempTris algorithm can be proven easily. In the following we merely sketch the idea of the proof.

The stable region set grows monotonically until all tuples of the input stream have been processed. Since the input stream is sorted, no tuples will be inserted into stable regions that are "left of the sweepline", i.e., intersecting the cluster $C_{[\min_1, t_{a_1}-1]}$. Thus each tuple is stored in a page corresponding to a unique region and all regions created by TempTris are disjoint. Summing up, the regions generated by TempTris form a disjoint multidimensional region partitioning. This proves the correctness of TempTris.

2.4 Basic Performance Observations

TempTris writes each region only once. Thus for generating a partitioning of P pages, TempTris needs to perform P page write operations, resulting in an I/O-complexity linear in the size of the input stream if sufficient but modest memory is available.

The dynamic region set maintained by TempTris in worst-case contains P regions, thus requiring to store the entire stream in main memory to create the partitioning. This extreme case happens, if all tuples have the same value in the sort attribute. In this case, the sweep line does not move and no regions can be made stable until the very end of the algorithm. In this case the sort order is not useful for creating the partitioning. If P does not fit in main memory, TempTris should call merge-sort to perform the sorting in this worst-case scenario. Note that no unnecessary I/Os are caused by TempTris in this case: The main memory cache of TempTris then can be used to create the initial runs for merge-sort. If a part of the region is written to disk the incremental loading algorithm presented in [2] can be applied.

For query optimization that means, one can call TempTris each time one would call merge-sort for creating a partitioning of a sorted stream. If TempTris cannot exploit the sort order efficiently, it – without having caused unnecessary I/Os - can dynamically switch to merge-sort instead.

However, if the sort order is useful, one can expect the region and page cache to contain about $P/|A_{\text{dim}}|$ entries with $|A_{\text{dim}}|$ being the distinct number of values in the sorting dimension contained in the input stream. A detailed analysis of the cache for uniformly distributed data is given in Section 4.4.

2.5 Possible Optimizations of TempTris

Page utilization of the stable region set created by TempTris is a critical performance issue: First, with a better page utilization fewer disk pages will be needed to store the multidimensional partitioning. Second, with a smaller amount of overall disk pages, queries on the partitioning will be faster, since then fewer disk pages are accessed.

TempTris as described in the previous section immediately splits disk pages when they overflow and thus may only guarantee a worst-case page utilization of 50%. However, the average page utilization with this strategy will also not be much better. Applying improved splitting algorithms as described in [10] for B-Trees, the average storage utilization can be increased to up to 81%.

A further improvement is to use a different page concept for the dynamic regions stored in main memory cache. In this case, it is not necessary to have a fixed page capacity, but instead fill up the pages with tuples until the cache overflows (i.e., dynamic page size). The split into pages of fixed size then takes place when the cache is flushed. This allows for creating large sets of pages with a utilization of 100%. Then pages can be bulk written to disk in sequential order, which may also be exploited by sequential reads in further processing.

An algorithmic improvement of TempTris is to avoid calculating the stable candidates of the dynamic region set, i.e., the set Δ in Figure 2-2, only after a region split or movement of the sweep line, thus avoiding unnecessary calculations and checks, reducing the overall CPU time required for TempTris.

3 The TempTris-Algorithm for UB-Trees

In the following we give an example of TempTris creating the Z-region partitioning of UB-Trees (i.e., a d-dimensional UB-Tree). To be able to exploit the order on attribute A_{dim} for the Z-region creation, A_{dim} must be one of the index attributes of the UB-Tree.

3.1 UB-Tree, Z-ordering

The UB-Tree [1],[17] uses a space-filling curve to create a partitioning of a multidimensional universe while preserving multidimensional clustering. Using the Lebesgue-curve (*Z-Curve*,Figure 3-1.a) it is a variant of the zkd-B-Tree [15] partitioning the universe into Z-regions.

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[2] and we denote the

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

binary representation of attribute value x_j by $x_{j,s-1}x_{j,s-2}...x_{j,0}$. A *Z-address* $a = Z(x)$ 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 3-1.b shows the corresponding Z-addresses.

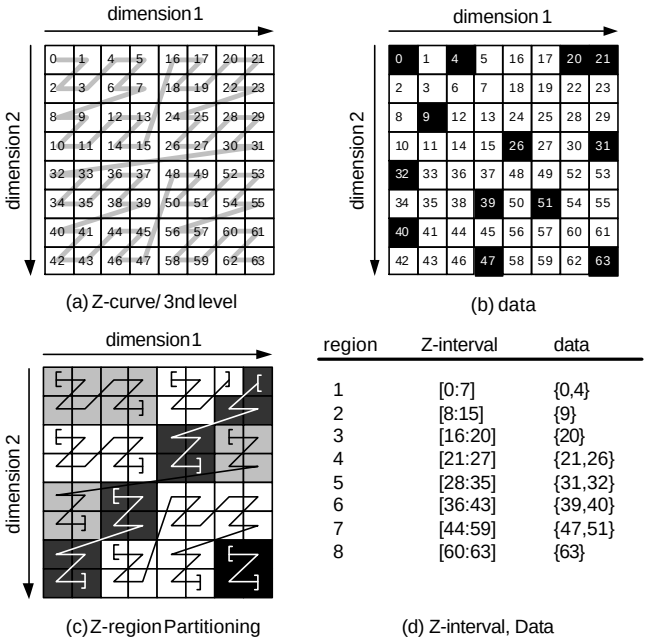

Figure 3-1: Z-Curve, Z-addresses, Z-regions

A *Z-region* $[a : b]$ is the space covered by an interval on the *Z-Curve* and is defined by two Z-addresses a and b. Figure 3-1c shows a partitioning with eight Z-regions.

3.2 Example: Creating a Z-Region Partitioning with TempTris

The following example illustrates TempTris by creating a two-dimensional *Z-region set* Θ^S. Figure 3-2 shows some steps of TempTris for a two-dimensional space with A_1 as horizontal dimension, A_2 as vertical dimension and the origin (0,0) in the upper left corner. For our example we assume a page capacity of 2 tuples.

In the beginning, the *dynamic region set* consists of only one region $\Theta^D = \{\Omega\} = \{r_1\}$, Θ^S is the empty set. We denote the position of the sweep line in the sort dimension by *c*. In Figure 3-2a TempTris has not started reading any tuples and therefore only region $\Omega \in \Theta^D$ is intersecting the sweep line at $c = 0$.

Figure 3-2b shows three dynamic regions created by TempTris having read 5 tuples from the input stream, i.e, $\Theta^D = \{ r_1, r_2, r_3\}$. $t_5 = (1,3)$ was the last tuple inserted. Thus the current position of the sweepline is c= 1.

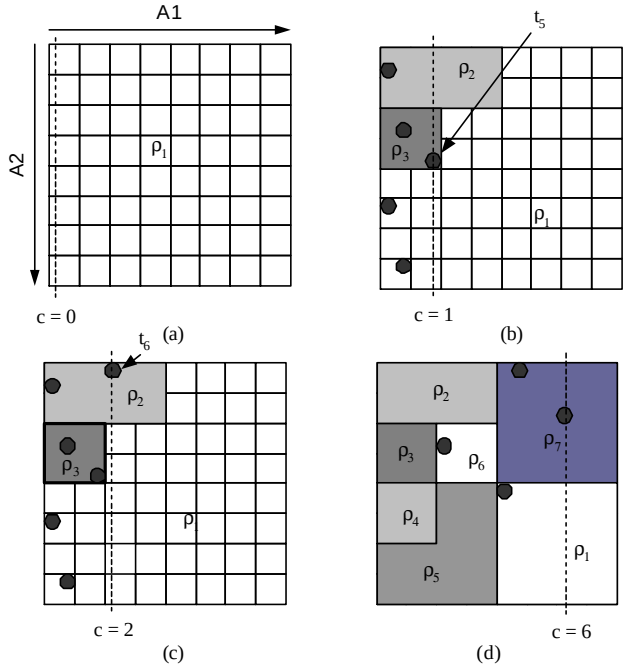

Figure 3-2: TempTris Algorithm for UB-Trees

In Figure 3-2c, t_6 is inserted and moves the sweepline to $c = 2$. Now the dynamic region denoted by r_3 does not intersect the cluster $C_{[2,7]}$ anymore. As a consequence, no more tuples may be inserted into r_3. Thus this region is made stable, i.e., the page corresponding to that region is removed from main memory and written to disk. The stable region set now consists of $\Theta^S = \{r_3\}$, the dynamic region set is $\Theta^D = \{ r_1, r_2\}$.

In Figure 3-2d the *stable region set* consists already of 5 regions, $\Theta^S = \{ r_2, r_3, r_4, r_5, r_6\}$. If the last tuple t is processed, the remaining dynamic region set $\Theta^D = \{r_1, r_7\}$ is made stable and the UB-tree is completed.

3.3 Maintenance of the Cache

The dynamic regions are organized according to Z-order and Tetris-order. For each ordering we provide an index structure for efficient access. With the Z-index a new tuple can be inserted in the corresponding page like a point query. The task of the tetris-order is to handle the sweep line that can be mapped to a range-query.

In [14] the Tetris-order is introduced that creates a total order with respect to A_i from Z-addresses. The Tetris-address extracts an attribute from a Z-address and concatenates it with the reduced Z-address. Formally the Tetris-address is defined as follows:

$$T_j(x) = x_j \circ Z(x_1,...,x_{j-1},x_{j+1}, \ldots, x_d)$$

Figure 3-3 presents the Tetris-order for the two dimensional case.

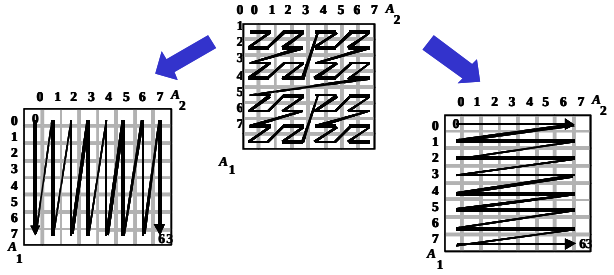

Figure 3-3: Two Tetris Orderings for the two dimensional case

If we use the Tetris-order defined by the sorting dimension all tuple x located in stable regions Θ^S have a smaller Tetris-value $T_j(x)$ than the smallest Tetris value of the sweep-line c.

$$\{T_j(x) \mid x \in r \text{ and } r \in \Theta^S\} < \min\{T_j(x) | x_j = c\}$$

Therefore, for each dynamic region r is indexed according to the maximum Z-value and Tetris-value. Figure 3-4 depicts the Z-value and Tetris-value of the region [12,35]. The maximum Z-value is 35 and the maximum Tetris-value has the ordinal 31.

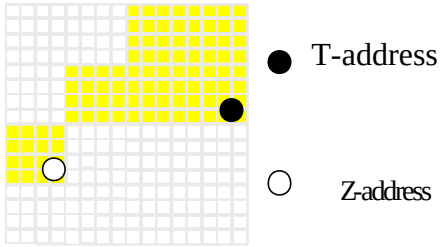

Figure 3-4: Z-address and Tetris-address

The computation of the maximum Tetris value can be done in linear time by bit-operations and is therefore of O(n) whereas n is the number of bits that represent Z(t).

```
SELECT region FROM T-Index
WHERE T(region) < T(sweep-line)
```

Figure 3-5: Determination of the D set

With this property the Δ set, i.e. the region which becomes stable, can be very efficiently determined by a range query (cf. .Figure 3-5).

4 Performance Analysis

For the creation of a Z-region partitioning from a sorted stream $S_{<,\dim}$ we define cost functions for processing times and intermediate temporary storage. Our analysis considers the TempTris algorithm and external sorting according to Z-values [2].

4.1 The Cost Model

In accordance with [6] 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 or write caching strategy of the file system reads or writes a physical cluster of L consecutive pages from disk with one random access. This takes time $t_\pi + t_\tau \cdot L$. Reading or writing k pages in consecutive order therefore takes

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

4.2 Cost Functions

Using the cost model of Section 4.1 we calculate the cost of sorting a relation of P pages using a main memory of M pages and a merge degree of m for the merge-sort algorithm. The bulk loading mechanism using merge-sort [2] divides the load process into 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).

$$c_{read}(P) = c_{write}(P) = \left(t_p \frac{1}{L} + t_t\right) \cdot P$$

We sometimes do not distinguish between read and write operations and then use $c_{r/w}(P)$ for the cost of reading or writing P pages.

$$c_{TempTris}(P) = c_{read}(P) + c_{write}\left(\frac{P}{page\_utilization}\right)$$

$$c_{ms} = \begin{cases} \underbrace{2 \cdot c_{r/w}(P)}_{\text{initialphase}} + \underbrace{2 \cdot c_{r/w}(P)}_{\text{Sortphase}} & \text{,if } \frac{P}{M} < m \\ \underbrace{2 \cdot c_{r/w}(P)}_{\text{initialphase}} + \underbrace{2 \cdot c_{r/w}(P) \cdot \log_m \frac{P}{M}}_{\text{Sortphase}} & \text{,otherwise} \end{cases}$$

Figure 4-1: Cost functions

Using a full table scan for the retrieval phase to create the initial runs (c_{read} + c_{write}) in conjunction with merge-sort algorithm results in the formula for c_{ms}. Using a full table scan for retrieval the phase in conjunction with TempTris results in the formula for $c_{TempTris}$. If M >P sorting takes place in main memory. Then the merge sort

factor of c_{ms} is reduced to zero. If $\frac{P}{M} < m$ then the sorting can be done with one run and the merge-sort algorithm results in the formula:

$$c_{ms} = 4 \cdot c_{read/write}(P)$$

To reduce the I/O cost of the merge sort algorithm the merge degree should be set to maximum size with respect to the main memory and therefore the merge degree is set to

$$m = \frac{P}{M}$$

For a main memory of size M the maximum table size that can be computed in linear time is

$$\max\_tablesize = \frac{M^2}{L}$$

4.3 Processing Time

Current operating systems usually fetch L = 8 pages physical disk pages with one random access. Four our cost analysis to create a multidimensional partitioning as target sorting from a one-dimensional source sorting we assume t_p = 10 ms and t_τ = 1 ms, a main memory cache of 32 MB and a merge degree of m = 2047 (best case). Therefore the sort merge algorithm can sort 64 Gbyte in linear time. We set the constant *page_utilization* of $c_{TempTris}$ to 81% (see Section 2.5).

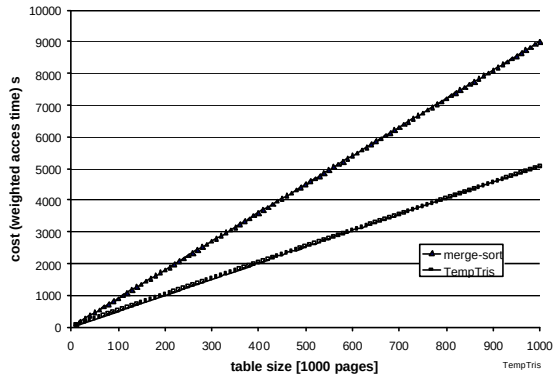

Figure 4-2: Sorted writing with Merge-Sort and TempTris

Figure 4-2 shows the cost of creating a multidimensional partitioning of a sorted input stream with a page size varied up to 1 million disk pages. As the figure shows, according to our cost model TempTris is roughly two times faster than a competing merge-sort algorithm.

4.4 Cache of the TempTris algorithm

TempTris requires less temporary memory than *sort-merge*. As *sort-merge* accesses and writes the entire input stream at least once for each run, the intermediate storage for *sort-merge* is P pages. If the data is uniformly distributed, TempTris requires a main memory of

$$cache_{TempTris}(P,d) = \frac{P}{2^{\frac{\log_2 P}{d}}} = \sqrt[d]{P^{d-1}}$$

pages to create a d-dimensional *UB-Tree* . That can be computed by the size of the table P divided by the number of splits in attribute A_{dim} (i.e. $P/2^{recusive\_splits}$). Figure 4-3 shows the size of the temporary storage to create the UB-Tree required by merge-sort and TempTris. The table size varies from 10K to 1G pages. For a page size of 2kB, creating a four dimensional space partitioning with TempTris for a 2GB input stream requires 64 MB temporary space. For a 2 dimensional partitioning of that size, only 2MB of cache memory are necessary.

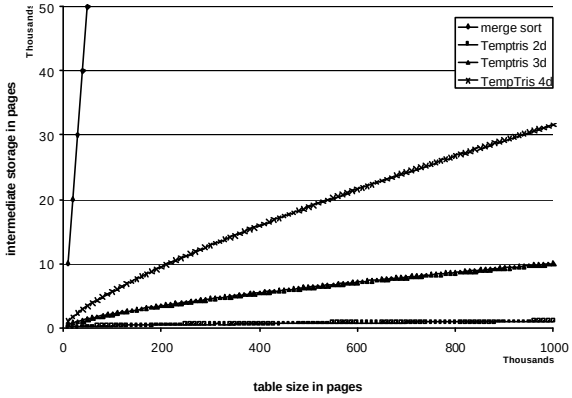

Figure 4-3: Intermediate storage sizes of Merge-Sort and TempTris

4.5 Result Table Sizes

As already mentioned before TempTris with optimizations can be expected to achieved a storage utilization somewhere between 80% and 100%.

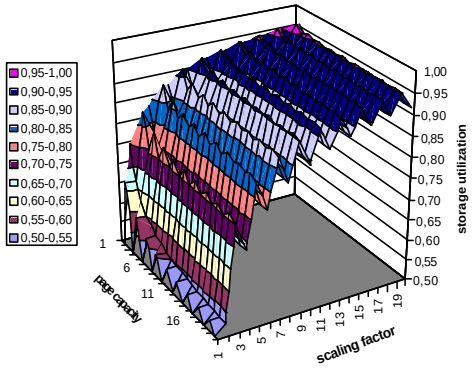

Figure 4-4:Storage utilization

This optimization technique based on sharing. As the TempTris algorithm maintains the dynamic region in the memory we generalize this technique and define the memory page v as follows:

$$v = u \cdot p$$

u denotes the scale factor of the memory page. Figure 4-4 presents the worst case storage utilization for different scale factors u and page capacities k. Using the optimization technique any storage utilization can be achieved. With a scale factor of 5 we get a storage utilization at least of 85%.

This of course is worse than bulk-loading with merge-sort, which always guarantees a storage utilization of 100%. Thus TempTris is useful, if

- a page utilization of 100% is not desired. This is true in many OLTP databases, where in anticipation of further insertions a certain percentage of each page in the database is intentionally left free, e.g. by setting the parameter PCTFREE in Oracle.
- The multidimensional partitioning is used for a small set of queries and then is dropped again. Then the benefits of TempTris for creating the partitioning several times faster outweigh the query response time, which can be expected to be up to 20% worse than the response time of a 100% storage utilization as created by sort-merge.

5 Processing an Aggregation Lattice

We now show a new technique for computing aggregation lattices by combining TempTris and Tetris. Aggregation lattices are often used to compute a set of aggregations efficiently [5]. [12] provides a good overview of aggregation techniques.

The main optimization for computing an aggregation lattice comes from the fact that some aggregates do not have to be computed from the base table, but can be derived from other already computed aggregates (e.g., the total sales for a year can derived from the sales of the 12 month periods).

```
SELECT Month2_Period, Outlet, Item,
SUM(Sales) AS Sales
FROM fact
WHERE Years > 1996 AND
       Years < 1999 AND
      Country = 'Germany'
 GROUP BY Month2_Period, Outlet, Item
WITH CUBE
```

Figure 5-1: A CUBE statement

A common application for this processing technique is the CUBE-Operator [4] frequently used in data warehousing applications. The CUBE-Operator computes 2^d aggregates of a set of d attributes. Figure 5-1 shows an example of a CUBE statement performed on a real DW for a schema provided by the market research company GfK.

For aggregation, conventional algorithms first sort or hash the relation according to the grouping attribute(s). After sorting, the aggregation can be computed very easily by simply reading the tuple stream in sort order. However, sorting is a bottleneck if the result set does not fit in the main memory and external sorting is necessary.

In order to compute the above three dimensional cube, 8 groupings have to be computed. The result of CUBE is then a union of these groupings. The corresponding aggregation lattice together with the size of each sub-aggregate in pages is shown in Figure 5-2. Each node of the network represents one grouping of the cube.

Usually, this aggregation network is processed as follows: For each node the conventional aggregation algorithm is performed. With level 0 being the base fact table, the result set of a node of level i is the input for the node at level i+1.

5.1 Cube Calculation by Sorting

In the following we calculate the I/O-cost for processing the aggregation network, assuming that no I/Os are necessary if an intermediate result fits into main memory. For our further considerations we assume a main memory cache of 16000 pages.

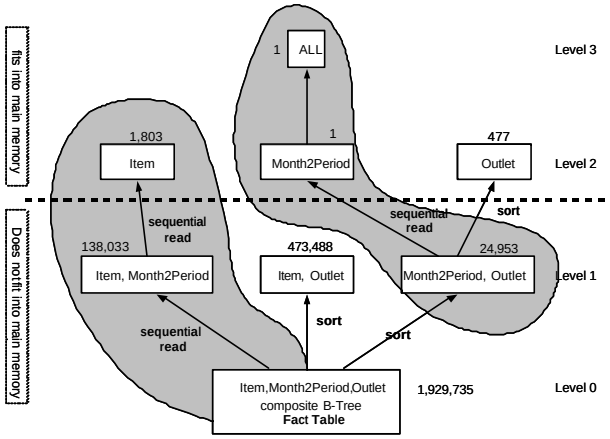

Figure 5-2: Aggregation Lattice; External sorting

We selected the calculation paths through the aggregation lattice by the heuristic "Re-use existing sort orders and use the smallest sub-aggregate for calculating an upper node of the aggregation lattice", which has proven to be a reasonable heuristics in many practical applications.

The nodes of the aggregation lattice have the following size:

Aggregate	Size in pages	label
Item, Month2Period,	1929735	P1

Outlet		
Item, Month2Period	138033	P2
Item, Outlet	473488	P3
Month2Period, Outlet	24953	P4
Item	1803	P5
Month2Period	1	P6
Outlet	477	P7

We assume two passes over the data for sorting and the cost of reading and writing a page to be identical, i.e., sorting P pages has a cost of

$$c_{sort}(P) = 4 \cdot c_{read/write}(P)$$

We also assume to have created a composite clustering B-Tree on Item, Month2Period, Outlet in this lexicographic order on the base fact table. Therefore, computing the aggregates <Item, Month2Period> requires a sequential read of the B-Tree consisting of P_1 pages and a sequential write of the result, i.e., P_2 pages:

$$c_{\langle Item, Month2Period \rangle} = c_{read}(P_1) + c_{write}(P_2)$$

The calculation of <Item> re-uses the aggregation result <Item, Month2Period> and accordingly requires sequential reading and writing:

$$c_{\langle Item \rangle} = c_{read}(P_2) + c_{write}(P_5)$$

To compute the node ⟨Item, Outlet⟩ the fact table must be sorted requiring to read P_1 pages for the creation of the initial runs. Then those runs are written to disk again resulting in writing P_1 pages. Merging and aggregating the runs again requires P_1 read operations. Finally writing the aggregation result is another P_3 page write operations:

$$c_{\langle Item, Outlet \rangle} = 2 \cdot c_{read}(P_1) + c_{write}(P_1) + c_{write}(P_3)$$

To compute ⟨Month2Period, Outlet⟩ sorting is necessary with the cost

$$c_{\langle Month2Period, Outlet \rangle} = 2 \cdot c_{read}(P_1) + c_{write}(P_1) + c_{write}(P_4)$$

Deriving <Month2Period> from this sub-aggregate requires only sequential read and write operations:

$$c_{\langle Month2Period \rangle} = c_{read}(P_4) + c_{write}(P_6)$$

Finally, the node ⟨Month2Period, Outlet⟩ can also be used to calculate outlet, this time requiring a sort operation:

$$c_{\langle Outlet \rangle} = 2 \cdot c_{read}(P_4) + c_{write}(P_4) + c_{write}(P_7)$$

Thus the total cost for aggregation using sorting is:

$$c_{cube}^{aggregate/sort} = 7 \cdot c_{read/write}(P_1) + 2 \cdot c_{read/write}(P_2) + c_{read/write}(P_3) + 5 \cdot c_{read/write}(P_4) + + c_{read/write}(P_5) + c_{read/write}(P_6) + c_{read/write}(P_7)$$

5.2 Cube Calculation by Combining TempTris and Tetris

Now we present our new approach that combines TempTris with Tetris. We organize the fact table of the "GfK"-schema as a UB-Tree. The Tetris algorithm [14] is used for sorted reading from the fact table, which is organized as a UB-Tree, as well as from the temporary UB-Tree created by TempTris storing the aggregation result of <Month2Period, Outlet>. Tetris does not require external sorting, if the main memory cache is sufficient:

According to [14], Tetris requires a main memory cache of at most $\sqrt[3]{P_1^2} = 15501 < 16000$ pages for processing the 3 dimensional cube statement without external sorting. The same maximum number of cache pages is required by TempTris as shown in Section 4.4.

In contrast to the aggregation lattice in Figure 5-2, the organization of the fact table as UB-Tree allows for using Tetris to create the aggregation results of <Item, Month2Period> and <Item, Outlet> without external sorting.

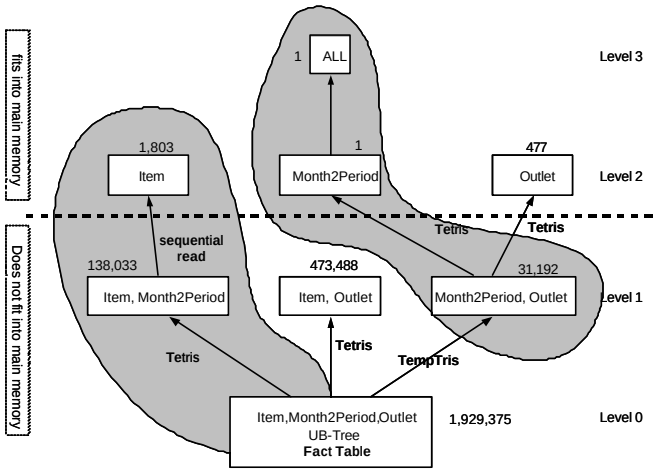

Figure 5-3: Aggregation Lattices; Temptris

In addition, Tetris produces a sorted stream on Month2Period or Outlet which is used by TempTris to create a temporary UB-Tree on <Month2Period, Outlet>. As TempTris usually does not achieve a 100% page utilization, the size of this UB-Tree is larger then the result obtained by merge-sort. Assuming a page utilization of 81% as described in Section 2.5, the UB-

Tree created by TempTris contains 31,192 pages in contrast to the composite clustering B-Tree created by merge-sort, which contains 24,953 pages. The temporary UB-Tree is necessary to apply Tetris again for the creation of the <Month2Period> and <Outlet> aggregations. This strategy avoids external sorting for calculating the cube.

With TempTris and Tetris the I/O-cost for calculating the cube is as follows:

$$c_{\langle Item, Month2Period \rangle} = c_{read}(P_1) + c_{write}(P_2)$$

$$c_{\langle Item \rangle} = c_{read}(P_2) + c_{write}(P_5)$$

$$c_{\langle Item, Outlet \rangle} = c_{read}(P_1) + c_{write}(P_3)$$

$$c_{\langle Month2Period, Outlet \rangle} = c_{read}(P_1) + c_{write}(P_4)$$

$$c_{\langle Month2Period \rangle} = c_{read}(P_4) + c_{write}(P_6)$$

$$c_{\langle Outlet \rangle} = c_{read}(P_4) + c_{write}(P_7)$$

Therefore total cost for our new approach is:

$$c_{cube}^{temptris/tetris} = 3 \cdot c_{read/write}(P_1) + 2 \cdot c_{read/write}(P_2)$$
$$+ c_{read/write}(P_3) + 3 \cdot c_{read/write}(P_4) +$$
$$+ c_{read/write}(P_5) + c_{read/write}(P_6) + c_{read/write}(P_7)$$

For our example this means that the approach for aggregation by sorting accesses 14,384,745 disk pages, roughly twice the number of pages that Tetris and TempTris access (6,615,899 disk pages). This improvement is due to two factors: First, Tetris saves 4 times reading the base table. Second, creating the intermediate result for <Month2Period, Outlet> by TempTris allows for also applying Tetris here, saving two times reading this intermediate result.

If the cube consists of more than 3 dimensions, the combination of Tetris and TempTris will benefit even more. In this case, there are more opportunities for applying TempTris and Tetris resulting in further savings of disk accesses.

6 Performance Evaluation

The measurements presented here have been made with the prototype implementation of the UB-Tree. It is realized as middleware between a database management system and a database application. We have implemented the TempTris Algorithm for Transbase UB-Tree middleware. An Intel Pentium III 500 CPU with 512 MB RAM has been used for these measurements. The databases were created on 9 GB hard disk with an average position time of 7,9 ms and a transfer rate of 0,6 ms per page. The machine runs under SuSE Linux 6.2, kernel version 2.2.10 SMP. To get no unpredictable cache effects we disable all systems caches .

Set	01	02	03	04	05	06
size	32 MB	64MB	128MB	256MB	512MB	1024MB
Tuple	335544	655360	1310720	2621440	5242880	10485760

Table 6-1:Size of the data sets

To show the performance gain of the TempTris algorithm we create 6 three dimensional *data cubes* with uniformly distributed data. Each cube is created from a different data set. The size of a tuple was 100 bytes and the page size was set to 2 kB. The size of each set is depicted in Table 6-1.

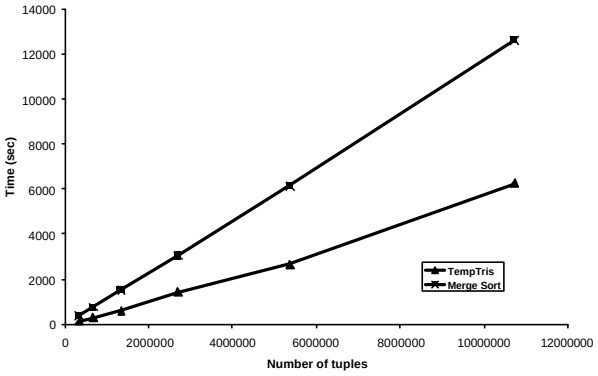

Figure 6-1:Loading performance

Each cube is created by the TempTris algorithm and external sorting. The implementation of the external sorting does not use replacement selection [3], that based on a heap to create runs that are larger than memory. With replacement selection the expected number of runs is about half as many runs as created by quicksort [9]. However, the advantage of having fewer runs must be balanced with the different I/O pattern and the disadvantage of more complex memory management [3].

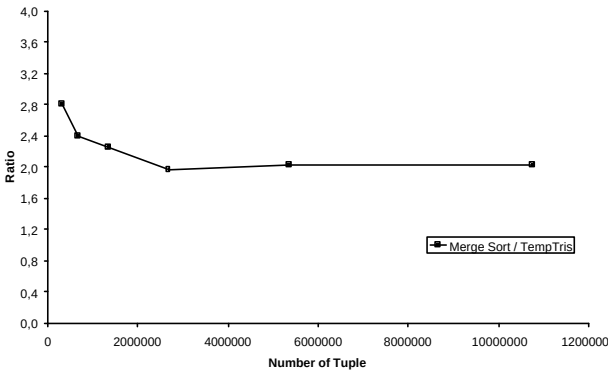

Figure 6-2:Ratio merge-sort and TempTris

In [11] it is shown that it is possible to create 1.8 times larger runs than the workspace. But this has no influence

on the measurement. As we set the available cache M used by the merge sort and TempTris algorithm at most half of the size of the measured sets and smaller than $M \cdot m$, external sorting is required and is done in linear time. To cerate a storage utilization at least of 95 % we use a scaling factor for TempTris algorithm with u = 10.

Figure 6-1 shows the measured times for the creation of 6 UB-trees from the different sets. One time with the TempTris algorithm and one time with the external sort algorithm. The TempTris algorithm is superior to merge sort algorithm. Both algorithms create the multidimensional index in linear time as predicted with our theoretical model (cf. Figure 4-1).

Figure 6-2 shows the performance ratio between the external sorting and TempTris. With the TempTris algorithm we gain an average speedup factor of 2. As we use a page scaling factor u = 10 the page utilization is about 95% (cf.Figure 6-3).

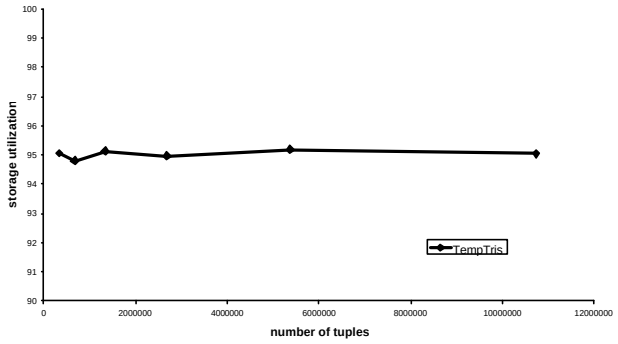

Figure 6-3: Storage utalization

As we said in paragraph 4.4 the creation of a multidimensional partitioning of a UB-tree with the TempTris algorithm requires for uniformly distributed data only $\sqrt[d]{P^{d-1}}$ temporary main memory pages. d denotes the number of dimensions.

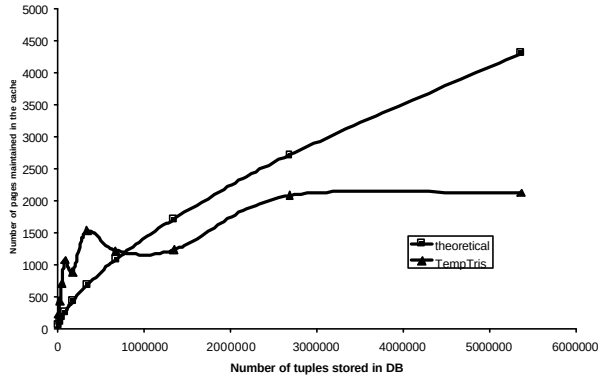

Figure 6-4: Cache size

Figure 6-4 shows the measured and the theoretical cache size. For sets smaller than 1 M the cache size is above the predict cache size. The reason for this is the freedom of the split address. For larger sets the cache size is above of the predicted one and even grows essential slower.

In order to evaluate our algorithms with real world data we have loaded a data warehouse from a leading German consumer-market analysts institute. They store data pre-aggregated to two month periods. The data was stored in a cube with the three dimensions *Product*, *Segment* and *Period*.

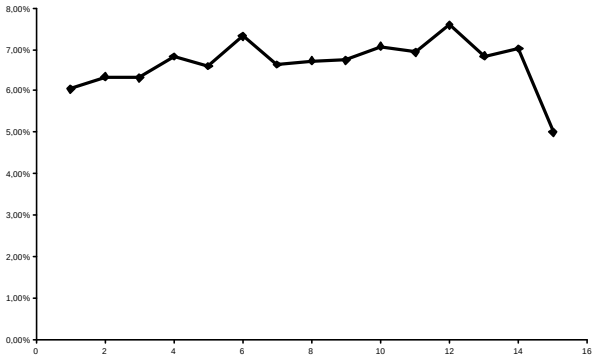

Figure 6-5:Data Distribution of DW

The data warehouse used for our measurements consists of 43 million fact tuples belonging to fifteen two-month periods. Figure 6-5 depicts the data distribution in percent according to the periods. The size of a tuple was 56 bytes. In our measurements we have only considered the fact table, since this is the biggest table and new data contributes mainly to this table.

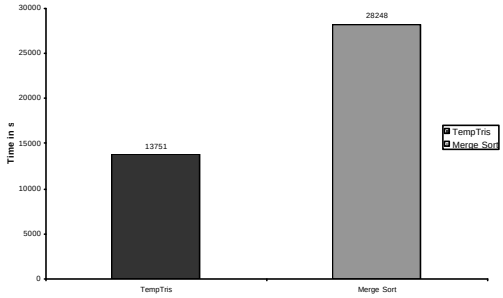

Figure 6-6: Loading Performance of DW

The page size was 2kB but to the overhead of the page management only 31 tuples fit on one page. As we use a scaling factor u = 4 the page utilization is about 87%.

Figure 6-6 shows the measured times for loading the fact table of the DW with the TempTris algorithm and external sorting. As predicted in out theoretical model the TempTris algorithm is superior to merge sort Algorithm. Both algorithms create the multidimensional index in

linear time but the TempTris Algorithm shows a speedup of about a factor 2 compared to external sorting.

7 Conclusion and Future Work

We have presented the TempTris algorithm as an efficient way for creating a multidimensional partitioning from a sorted stream of tuples. This allows for bulk loading a multidimensional index faster than the usual merge-sort algorithm. The main advantage of TempTris is that a multidimensional partitioning can be created without external sorting, if sufficient, but modest cache memory is presented.

We described the general algorithm and illustrated TempTris by an example creating the multidimensional Z-region partitioning of UB-Trees. We also gave a performance analysis, which showed a performance improvement of TempTris over a competing sort merge algorithm by a factor of two. In addition, TempTris requires only a root function of the input data set size as cache, whereas merge-sort requires temporary storage in the size of the input set.

We have presented performance measurements that proved our predicted speed up factor of TempTris algorithm of 2.

In combination with Tetris, the TempTris algorithm can be used to efficiently compute the CUBE operator, as we have also illustrated in this paper by giving the cost for computing a data cube from a three-dimensional real-world schema of GfK. In this example, TempTris calculated an aggregation lattice with two times less page accesses than the competing method using sort/hash-grouping.

TempTris is a general approach: it can be used to create the regular tiling of Grid-Files, the rectangular partitioning of R-Trees or more complex partitioning patterns like the Z-regions of the UB-Tree. The main application of TempTris are bulk loading of a multidimensional index, especially the computation of the cube operator with aggregation lattices.

TempTris is the inverse function of Tetris. Thus, many of the result on Tetris presented in [14] can be taken over to TempTris, especially the cost-functions of the cache size and the guarantees for processing an input stream with one sweep.

Acknowledgements

We thank our project partners Microsoft Research, Teijin Systems Technology, and the European Union for funding this research work. We also thank Sebastian Hick for doing the prototype implementation.

References

[1] R. Bayer, *"The universal B-Tree for multidimensional Indexing,"* Institut für Informatik, TU München, München TUM-I9637, 1996.

[2] R. Fenk, A. Kawakami, V. Markl, R. Bayer, and S. Osaki, "Bulk loading a Data Warehouse built upon a UB-Tree," presented at IDEAS, Yokohama, Japan, 2000.

[3] G. Graefe, "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys*, vol. 25, pp. 73-170, 1993.

[4] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh, "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total," *Data Mining and Knowledge Discovery*, vol. 1, pp. 29-53, 1997.

[5] V. Harinarayan, A. Rajaraman, and J. D. Ullman, "Implementing Data Cubes Efficiently," presented at SIGMOD, Montreal, Canada, 1996.

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

[7] G. R. Hjaltason, H. Samet, and Y. Sussmann., "Speeding up Bulk-Loading of Quadtrees," presented at ACM International Workshop on Advances in Geographic Information Systems, 1997.

[8] I. Kamel and C. Faloutsos, "On Packing R-trees," presented at CIKM, 1993.

[9] D. E. Knuth, *Sorting and Searching*. Reading Massachusetts: Addison-Wesley, 1998.

[10] K. Küspert., "Storage utilization in B*-trees with a generalized over ow technique.," *Acta Informatica*„ vol. 19, pp. 35-55, 1983.

[11] P. Larson and G. Graefe, "Memory Management During Run Generation in External Sorting," presented at SIGMOD,International Conference on Management of Data, Seattle, Washington, USA, 1998.

[12] W. Lehner, "Aggregatverarbeitung in Multidimensionalen Datenbanksystemen.," . Erlangen: Friedrich-Alexander Universität, 1998.

[13] S. T. Leutenegger and D. M. Nicol, "Efficient Bulk-Loading of Gridfiles.," *IEEE Transactions on Knowledge and Data Engineering*„ vol. 9, pp. 410-420, 1997.

[14] V. Markl, M. Zirkel, and R. Bayer, "Processing Operations with Restrictions in Relational Database Management Systems without external Sorting," presented at ICDE, Sydney, Australia, 1999.

[15] J. A. Orenstein and T. H. Merret, "A Class of Data Structures for Associate Searching," presented at

Proc. of ACM SIGMOD-PODS Conf, Portland, Oregon, 1984.

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

[17] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer, "Integration the UB-Tree into a Database System Kernel," presented at the 26 International Conference on Very Large Databases, Cairo, Egypt, 2000.