Indexing without the Index: Scalable Multidimensional Aggregation for Data Warehouses

Mirek Riedewald[1] Divyakant Agrawal[1] Amr El Abbadi[1] Flip Korn[2]

[1]University of California
Santa Barbara, USA
{mirek, agrawal, amr}@cs.ucsb.edu

[2]AT&T Labs-Research
Florham Park, USA
flip@research.att.com

Abstract

Aggregation plays an important role in data warehousing and has received much attention to date. However, existing techniques do not sufficiently address the issue of making the computation of multidimensional aggregates scale with increasing dimensionality. At the same time the benefit of taking full advantage of the hard disk geometry is often overlooked. This paper presents the Multiresolution File Scan (MFS) approach which is based on a selection of flat files which are accessed with fast sequential I/O operations. Its simple structure and low storage overhead allow MFS to scale to high dimensionality while making the best use of the increasing transfer speed of modern hard disks. We show that MFS outperforms multidimensional index structures, even if these structures are bulk-loaded and hence optimized for query processing. Our approach can incorporate a priori knowledge about the query workload and is applicable to all distributive (e.g., `COUNT`, `SUM`, `MAX`, `MIN`) and algebraic (e.g., `average`) aggregate operators.

1 Introduction

Data warehouses are designed to manage very large amounts of information, already today reaching sizes of multiple terabytes. The data is conceptually viewed as having multiple logical dimensions (e.g., `age`, `education`) with natural hierarchies defined on each dimension (cf. the well-known Star schema). Analysts examine the relationship of various measure attributes (e.g., `income`) with respect to the dimensions. Summarization and aggregation are essential in this process. A typical query is the multidimensional orthogonal range query which computes a range aggregate over some (or all) of the attributes. For example, "Find the average income of workers who have less than 13 years of education and are 20–30 years old." Popular OLAP queries like roll-up/drill-down and slice-and-dice [6] are typically sequences of related range aggregate queries.

Queries that select ranges in multiple dimensions are most efficiently supported by multidimensional access methods [9]. However, as we will discuss in Section 2.1, current aggregation techniques do not scale well with increasing dimensionality. On the other hand there is an increasing demand to support high-dimensional data sets. This data is created by business processes that span multiple organizational units of a company and involve several resources. Describing such a process in detail and discovering complex relationships of its entities requires dealing with large numbers of attributes, i.e., dimensions. For instance the `Lineitem` table of the TPC-R benchmark [36] which models a retail data warehouse with relatively low complexity already has 15 dimensions (not counting the `comment` attribute).

For large data warehouses the I/O cost, i.e., the time it takes to retrieve relevant information from hard disk, dominates the query cost. Hence our goal is to reduce this cost factor. Research regarding index structures typically focuses on *what* is accessed, i.e., on reducing the amount of accessed index pages during a search process. However, it is at least as important to optimize *how* the data is accessed since the disk geometry supports data streams (sequential I/O), rather than random accesses [30, 31]. For instance over the last decade the internal data rate of IBM's hard disks improved from about 4 MB/sec to more than 60 MB/sec. At the same time the positioning time only improved from about 18 msec to 9 msec [35]. This implies that reading and transferring large chunks of data from sequentially accessed disk sectors became about 15 times faster, while the speed for moving to a "random" position only improved by a factor of 2. The disk can reach its peak performance only if large chunks of consecutive sectors are accessed (sequential I/O). Hence it is often more efficient to read data in large chunks, even if these chunks contain irrelevant sectors.

We propose the Multiresolution File Scan (MFS) technique to address the above issues. MFS is based on a selection of flat files that represent the data set at different resolutions. Its simple structure and low storage overhead (typically below 5% of the data size) allow it to scale with increasing dimensionality. Each of the files is accessed using a pseudo-optimal schedule which takes transfer and positioning time of the disk into account. This guarantees that even in the worst case accessing relevant sectors will not be more expensive than a sequential scan. While this is important for queries that select large regions of the data set, MFS also provides efficient support for very selective queries.

We have developed a technique for selecting the files of MFS which can incorporate a priori knowledge about the query workload. MFS allows for efficient bulk-loading and provides the possibility of fast bulk-update and single-update processing. Furthermore it is flexible enough to support different sort orders of the data set. Hence it can choose good clustering orders and orders that are efficient for append-only updates and data aging.

The outline of this paper is as follows. In Section 2 we motivate the design of MFS and present its data structures and algorithms. The selection of the files for MFS is discussed in Section 3. Section 4 presents selected experimental results. Related work is discussed in Section 5. Section 6 concludes this article.

2 The MFS Technique

In this section we first discuss the issues that motivated the design of MFS. Then the actual technique is presented for the aggregate operator `SUM`. Other operators can be handled in a similar way. For the sake of simplicity we will further on assume that the data warehouse table has *d dimension* attributes and a single *measure* attribute. Our technique can be easily generalized.

2.1 Motivation

An important question which is not sufficiently addressed by researchers developing new index structures is how to take advantage of the high transfer rates of modern hard disks. Here we just provide a rather coarse overview of relevant disk features; the interested reader might consult [30, 29] for more details.

A disk read request is specified by its *start sector* and the *number of bytes* to be transferred. Unless the requested data is already in the disk cache, this request incurs two major access costs. First *positioning time*, more exactly seek time and rotational latency, is spent to move the disk head above the start sector. Once the head is correctly positioned the start sector and all following sectors are read and transferred until the specified amount of data is read. This cost is referred to as *transfer time* (for the sake of simplicity we omit a discussion of head and track switch times). Depending on the actual seek distance, modern hard disks are able to transfer 50 or more sectors during a time frame equivalent to the positioning time. Hence it is often faster to issue a single large read request which might contain irrelevant sectors, instead of using multiple small reads which each incur positioning time. Figure 1 shows a schematic illustration of this process.

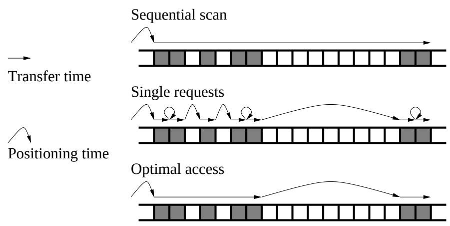

Figure 1: How to read a set of disk sectors (gray sectors are relevant, white sectors are not)

To be able to find the best access strategy, the set of target sectors must be known in advance. The depth-first traversal (DFS) of hierarchical index structures does not leave much room for optimization. Relevant index nodes are fetched one by one from hard disk and only the sectors belonging to the next single node are known in advance. This results in a positioning time penalty for each node access. The *pre-fetching* option of modern disk drives only partially alleviates the problem by automatically reading several sectors *after* the requested target sectors and keeping them in the disk cache for future requests. However, pre-fetching is not effective if requests are not accessing sectors in order or if there is a large "gap" between groups of target sectors.

There is an inherent dimensionality curse in computing range aggregates. The best known algorithms whose runtime is provably sub-linear in the size of the data set (e.g., [7, 38]), have polylogarithmic query cost and storage overhead. However, for dimensionalities $d \geq 9$ a polylogarithmic cost is practically worse than a linear cost. Let n denote the number of data points, i.e., tuples in the data warehouse's table. Then for $d = 9$ the polylogarithmic value $\log^d n$ is only less than n if $n > 10^{15}$. This means that polylogarithmic performance does not guarantee a practically better cost than a simple sequential processing of the whole data set. A storage explosion by a factor of $\log^d n$, i.e., a total storage cost of $O(n \log^d n)$ is clearly unacceptable for large data warehouses.

A similar dimensionality curse has also been observed for popular practical indexing techniques which do not guarantee sublinear query performance, but nevertheless often successfully reduce search costs. For instance, on high-dimensional data sets queries using R-trees wind up searching multiple search paths due to the high overlap of page extents which dramatically increases search time [3]. Other empirical studies for nearest neighbor queries have also shown that balanced tree structures can be outperformed by a sequential scan of the complete data set [4, 37].

Another possibility to speed up queries is to materialize pre-computed aggregates and to use them to reduce on-the-fly computation and access costs. However, we have shown that for a d-dimensional data set any pre-computed value is contained in at most about $\frac{1}{2^d}$ of all possible range queries (cf. Appendix A). Assuming that all range queries are equally likely, the probability that materializing a value will benefit a query in a 15-dimensional data set therefore is at most about 0.00003. Hence effective pre-aggregation requires knowledge about the query workload.

2.2 Overview of MFS

The main goals of MFS are to take maximum advantage of the high transfer rates of modern hard disks and to keep the storage overhead low. At the same time very selective queries should not have to access large amounts of irrelevant data. To achieve this MFS uses a collection of flat files. Each file represents the data set at a different resolution which is selected based on attribute hierarchies. The file with the highest resolution contains the exact information of the warehouse data set. The next coarser file partitions the data space into regions and stores for each region the sum over the measure values of the data points in the region. The next coarser file in turn stores sums over regions of regions, etc. Figure 2 shows a simple example for a data set with the two dimensions `time` and `location`. Let the measure attribute be the amount of sales for the corresponding time and location. The base data set is at (month,state) granularity. For simplicity the measure value of each base tuple is set to 1.

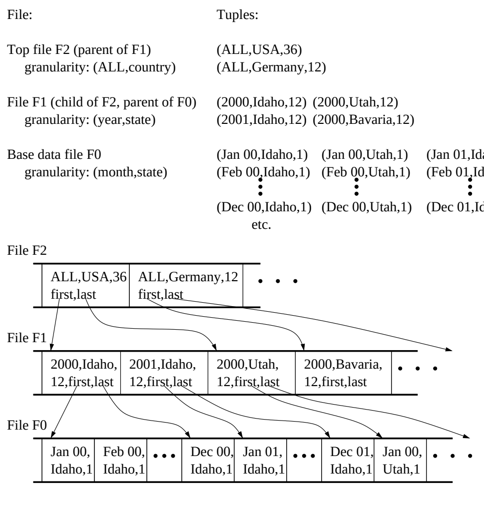

Figure 2: Basic structure of MFS

Note that the different granularities establish a conceptual hierarchy among the files. Figure 2 illustrates this hierarchy by placing the file with the lower resolution above the file with the next higher one. We will therefore refer to the file with the lowest resolution as the *top* file. Also, a file and the next coarser file are referred to as child and parent. Let F_i be the file at level i, level 0 containing the base data.

To allow efficient pruning during range query processing, the database schema is augmented by two attributes: `first` and `last`. They point to the first and last sector on disk that contains information that refines the aggregate information of a high-level aggregate tuple.

2.3 Processing Queries

A query is processed as follows. First the top file is sequentially scanned. This scan returns all tuples that intersect the query together with the pointers to the chunks of consecutive sectors that contain the corresponding higher resolution tuples in the child file. Based on this information a pseudo-optimal set of disk read requests is generated (to be discussed shortly) to retrieve the target sectors with minimal I/O costs. After processing these sectors, all intersected tuples of the next child file are known together with the location of their child tuples, etc. This way the query proceeds by pseudo-optimally reading all relevant tuples in the higher-resolution files. Figure 3 shows the algorithm in pseudo-code. Note that for each intersected tuple in a low-resolution file *all* its child tuples in the next file have to be accessed. Hence the best clustering strategy is to place all tuples with the same parent tuple into consecutive locations in the file.

Algorithm `Query`
Input: MFS-file; query
Output: progressively refined approximate results with absolute error bounds

1. Read the file with the lowest resolution sequentially to get the measure values and child pointers of all tuples that are intersected by the query;
2. Compute the approximate query result with error bounds and output it;
3. While (the file has a child file of higher resolution)
   Compute the pseudo-optimal schedule to access the relevant sectors in the child file;
   Read the sectors and output a new approximate result whenever all child tuples of a parent tuple are read;

Figure 3: Algorithm for querying MFS

In the *worst case* the query cost is identical to the cost of sequentially reading each file of MFS. This guarantee is actually much stronger than it appears. Index structures that access tree nodes one-by-one might perform much worse even if they access less disk sectors in total. If the ratio between positioning time and transfer time per index page is 10, an index that uses single read requests for each page only beats the corresponding sequential scan of the complete data set if it accesses less than 10% of the pages (cf. Section 2.1). Our experimental results indicate that in practice MFS typically performs much better than the worst case suggests.

As a by-product of maintaining files at different granularities, MFS can return approximate results with *absolute* error bounds "for free". The algorithm in Figure 3 reflects this procedure. The tuples in the low-resolution files represent aggregate values for *regions* of the data space. They can be used in the same way as the nodes of hierarchical indexes (cf. [26, 20]). The underlying idea is that low-resolution tuples which are completely contained by a range query contribute their aggregate value to the query and hence their child tuples in the lower files do not have to be examined. By accessing files with increasing resolution, the approximate result and the error bounds are progressively refined. Based on the output, the user can interrupt the query processing whenever the result is accurate enough or let the query run till completion and return the exact value. For a more detailed discussion please refer to [26, 20]

The pseudo-optimal disk access schedule is computed using a very fast approximation algorithm which assumes a simple disk model. Nevertheless the experimental results in Section 4 show that it is very effective. Let $R = \frac{t_{\text{positioning}}}{t_{\text{transfer}}}$ denote the ratio between the "typical" positioning time and the transfer time per sector. Given a set of target sectors, the pseudo-optimal schedule is generated as follows. The first read request starts at the target sector with the lowest number and contains all target sectors with higher numbers such that the number of non-target sectors between any neighboring target sectors in the request is less than R. The second read request starts at the target sector with the lowest number which is not in the first request and contains all following sectors such that there are no gaps of R or more sectors between neighboring target sectors, and so on. The motivation for allowing gaps of less than R consecutive non-target sectors in a read request is that it is faster to read and transfer the non-target sectors instead of issuing smaller reads which each incur positioning time. Figure 4 shows an example for $R = 2$, i.e., when gaps of at most one non-target sector are allowed within the read request. Realistic values are $R = 20$ or higher for modern disks.

2.4 Bulk-Loading MFS

For now we assume that the granularities for the files of MFS are known. Recall that the goal is to cluster together tuples which share the same parent tuple in the next coarser file. Hence the selection of file granularities defines a partial order among the base tuples. For construction purposes one can choose any *total* order for the base tuples which is consistent with the partial ordering. This total order is similar to the well-known Z-order (cf. [9]).

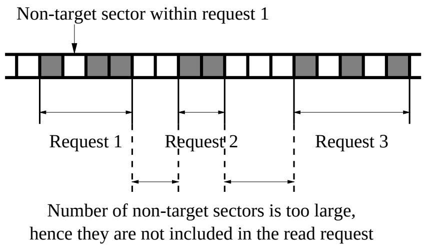

Figure 4: Pseudo-optimal access schedule for $R = 2$ (target sectors are gray)

The bulk-loading algorithm first sorts the input file according to the total order. Then all files of MFS are constructed simultaneously by scanning the sorted file only *once*. This is possible because the total order ensures that all base tuples which share a common ancestor in a low-resolution file are stored before all base tuples for the next ancestor. The algorithm has extremely low requirements for temporary storage. All it needs to keep in memory is the currently processed tuple for each file, together with its current aggregate value according to the corresponding base tuples read so far. The values of the pointers `first` and `last` are generated during the scan as well. In Figure 5 the algorithm is shown in pseudo code. The total bulk-loading cost is equivalent to the cost of one sort and one scan of the data set. Both operations take advantage of the disk geometry by transferring large chunks of sequentially stored data. Note that the bottom file of MFS is already complete after the sorting step. Hence the scan (step 3) only writes the tuples of the small low-resolution files. The write cost therefore is negligible compared to the cost of scanning the sorted base data file.

2.5 Updating MFS

In the following discussion the term *update* denotes any change to the base data set, including insertion and deletion of tuples.

In applications like data warehouses which are mainly designed for the *analysis* of large data sets, updates are performed in large batches at times when the system utilization is low. MFS is well-suited to deal with such batch updates. Note that the update operations are equivalent to tuples of the base table: they specify the dimension values of a data point and a change regarding its measure attribute. Hence we can easily construct a temporary MFS-structure Δ by applying the MFS bulk-loading algorithm to the update batch. Then the old version of MFS is merged with Δ by scanning both data sets. If the output of the merging process is written to a new location on disk, Δ and the old version of MFS can be used until the new MFS is ready. Queries are answered by

```
Algorithm Bulk-load
Input: File containing the input data;
    total order ≤ of the input tuples
Output: Files of MFS
1. Sort the input file baseFile according to the total order;
2. Read first tuple from baseFile and generate list
    Ancestors of its ancestor tuples in the
    low-resolution files;
3. While (there are more tuples in baseFile)
    Read tuple from baseFile;
    currentFile = baseFile;
    Do
        If (tuple is a child of the corresponding parent
            tuple parent in Ancestors)
            Update aggregate value of parent;
            continue = false;
        Else
            Create new parent tuple newParent of tuple;
            Set first pointer of newParent;
            Replace parent by newParent in Ancestors;
            Set last pointer of parent and append it to
                the corresponding file;
            tuple = newParent;
            currentFile = parent of currentFile;
            continue = true;
    while (continue);
4. Append tuples from Ancestors to the respective files;
```

Figure 5: Algorithm for bulk-loading MFS (writes are buffered)

combining the partial results obtained from accessing the old version and Δ using the MFS query algorithm. The total bulk-update cost is equivalent to the cost for sorting the update batch plus a sequential scan of the file of the old version of MFS and two scans of the files of Δ. Sorting and scanning both can take maximum advantage of fast sequential accesses.

It is not difficult to process even single updates on MFS. The MFS structure does not require a total order of the tuples to be maintained in the files. This order was only used by the bulk-loading algorithm. Hence overflowing chunks of tuples which share the same parent tuple can be relocated. Details are omitted due to space constraints. In general for query-centered applications like data warehouses it seems more appropriate to collect single updates in a small dynamic structure and then to apply them in batches.

We can take specific advantage of MFS's flexible tuple order to support append-only data [27] and data aging. In many practical applications data is appended rather than inserted at general locations. For instance, warehouses record sales transactions in timely manner, i.e., append new tuples with a later time stamp. Similarly data about current weather conditions is reported by sensors in order of increasing time stamps. Queries often are also time-related, e.g., comparing sales amounts of different months or years. Also, in data warehouses *data aging* processes manage rapidly growing amounts of data by retiring old information from hard disk to slower mass storage [15]. MFS can take this time-oriented behavior into account by choosing higher resolutions for the time-related dimensions in the upper level files. This ensures that tuples that fall into a small time range are also within a small range in the files and hence result in high locality of append-only updates and data aging.

3 Selecting the Granularities for the MFS Files

Our goal is to design a technique that automatically selects the number of files and their granularities for MFS such that the average query cost is minimized.

3.1 Notation

Let the data set $\mathcal{D}$ have d dimensions and n tuples. Each dimension i has $l(i)$ hierarchy levels. The levels are numbered 0, 1,..., $l(i)-1$; 0 being the level with the highest resolution and $l(i)-1$ corresponding to the artificial ALL value that stands for the whole domain of attribute i (cf. [11]). For instance `time` could have the 3 levels month (=0), year (=1), and ALL (=2). If no natural attribute hierarchy exists, any hierarchical partitioning, e.g., according to the decimal system, can be used.

We use N_i to denote the size of the domain of dimension i. Hence the total data space has a size of $\prod_{i=1}^{d} N_i$. We assume that MFS has l files $F_0, F_1, \ldots, F_l$ in decreasing order of their resolution. Let n_k denote the number of tuples in F_k.

The smallest unit of measurement in the data set is a tuple of the original data set. We can conceptually view such a base tuple as a cell of side length 1 that contains a value. Tuples of lower resolution files correspond to hyper-rectangular regions of such cells. For instance (2000,Idaho) has a side length of 12 in the `time` and 1 in the `location` dimension. We will use $x_i^{(k)}$ to denote the side length in dimension i for the region of a tuple in file F_k. Consequently for the base file tuples it holds that $x_i^{(0)} = 1$ for all dimensions i.

3.2 Estimating Query Performance for a Given MFS

Given a certain MFS configuration we need a fast algorithm that evaluates the expected query performance. We propose a lightweight analytical solution.

As described in Section 2.3 a query first completely scans the top file F_l and then reads all child tuples of those tuples in F_l which intersect the query, and so forth. Hence the selectivity of the query in file F_{k-1}, $1 \le k \le l$, is affected by the granularity of its *parent* file F_k. Figure 6 illustrates this for a query (shaded) that intersects the four inner tuples of file F_k which are double hatched in the figure. After reading F_k the algorithm only knows the location of the child tuples in file F_{k-1}, but not which of them actually are contained in the query. Hence it examines all 16 child tuples in F_{k-1} even though the query only contains 9 of them.

In the figure the accessed tuples in F_{k-1} are marked by a bold frame.

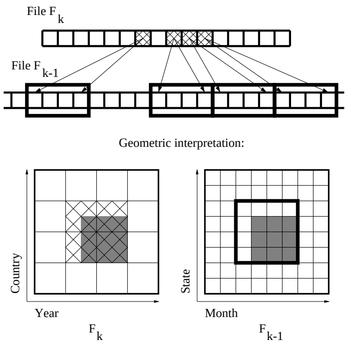

Figure 6: Increased query area in F_{k-1} caused by the low resolution of F_k

Let query q select a multidimensional range that has size q_i in dimension i. The selectivity of q in file F_{k-1} $(0 < k \leq l)$ can then be computed as $\prod_{i=1}^{d} \frac{q_i + x_i^{(k)} - 1}{N_i}$. Based on the selectivity, the average number of accessed bits can be estimated as $n_{k-1} s_{k-1} \prod_{i=1}^{d} \frac{q_i + x_i^{(k)} - 1}{N_i}$. Here s_{k-1} denotes the number of bits required to encode a tuple of file F_{k-1}, including the measure value and the two pointers to the child tuples. For more details please refer to Appendix B.

The overall query cost for MFS is the sum of the cost for scanning F_l and the query cost for reading relevant tuples in each file below. With the above formula this cost is estimated as

$$n_l s_l \quad + \quad \sum_{k=l}^{1} n_{k-1} s_{k-1} \prod_{i=1}^{d} \frac{q_i + x_i^{(k)} - 1}{N_i}.$$

This formula provides a good estimate for the average time it takes to read the data from disk. Note that we are actually not interested in the absolute value. All we need to find is the MFS configuration with the lowest cost relative to all other examined configurations.

The only problem with the above formula is to determine the n_k, i.e., the number of tuples in all files F_k. One can either materialize the files to obtain the exact values, or use much less costly approximation techniques like the one proposed by Shukla et al. [32]. For *fractal* data sets (cf. [23]) there is an even faster purely analytical way of determining the n_k. Let D_0 be the Hausdorff fractal dimension. Then n_k can be accurately estimated as $n_k = n \left(\frac{\sqrt{d}}{d_k}\right)^{D_0}$ where d_k denotes the diameter of the region that corresponds to a tuple of file F_k.

3.3 Addressing the Complexity

So far we have a formula that computes the cost for a certain MFS configuration. However, an exhaustive testing of all possible MFS configurations is costly. Each dimension has at least two hierarchy levels (highest resolution and ALL). Hence there are $2^d - 1$ candidates for the parent file of base file F_0. We propose to use a greedy heuristic.

First the best MFS configuration is selected bottom-up as follows. We start with the base data set in F_0 and assume that F_1 has the same granularity as F_0. Then the algorithm checks for each dimension in F_1 if decreasing its resolution by one hierarchy level reduces the cost. We choose the configuration of F_1 with the lowest cost as the new granularity candidate. The same process is repeated with this candidate as the starting point until we find a (local) minimum. F_1 then is set to the found granularities and the process continues in the same manner with F_2 as the new top file, and so on until no improvement is found. A simple example is provided in Appendix C.

The greedy process tends to over-estimate intermediate costs. When a new candidate for the top file is examined, the cost of a scan is assumed. Later, when another file is selected to be placed above it, the actual cost would be smaller. To correct this problem we use a second bottom-up process. It starts with the configuration proposed by the first process. First it greedily tries to *increase the resolution* of file F_1, dimension by dimension as discussed before (just in the opposite direction, i.e., refining instead of making coarser). Once a local minimum is found it continues with $F_2, \ldots, F_l$. As long as at least one file's granularity changes, the process iterates over all files again.

The greedy process only examines a number of MFS configurations which is linear in $\sum_{i=1}^{d} l(i)$, the total number of hierarchy levels of the dimensions. In contrast to the exponential exhaustive search it easily scales to high dimensionality.

3.4 Taking the Query Workload into Account

The formula derived in Section 3.2 computes the average cost given a certain query specification. A workload containing several of these query specifications with their respective probabilities can be easily taken into account by computing the cost for each query type and weighting it with its probability.

Often there exists less specific information about the most likely queries, e.g., "the majority of the queries selects on product categories rather than single products". This type of knowledge can also be incorporated. The average query enlargement due to a low file resolution was computed for the general case as $\prod_{i=1}^{d}(q_i + x_i^{(k)} - 1)$ (cf. Section B). If the query actually aligns in dimension i with the grid structure defined by the $x_i^{(k)}$ in file F_k, the corresponding term changes from $(q_i + x_i^{(k)} - 1)$ to q_i, i.e., there is no enlargement in that dimension.

Knowledge about the *location* of a query can also be taken into account, e.g., for density-biased queries. Instead of only using the n_k the algorithm can use the number of tuples in the most accessed sub-regions of the data space. Similar to multidimensional histograms [24] there is a tradeoff between accuracy of estimation and storage and maintenance costs. Examining this tradeoff is beyond the scope of this paper.

If there exists no knowledge about the workload, MFS is constructed by assuming queries that select small ranges in each dimension, e.g., 10% of the domain. This ensures that these queries are well supported. Queries that select large regions, and hence retrieve large amounts of tuples, benefit from the fast transfer property of MFS anyway.

4 Experimental Results

We evaluated the performance of MFS for several real and synthetic data sets. To allow a fair and detailed comparison, we report the I/O costs measured with the DiskSim 2.0 simulation environment [10]. This tool is publicly available at `http://www.ece.cmu.edu/ ganger/disksim` and was shown to model the real access times very accurately. In our experiments we used the settings for the most detailed and exact simulation.

We compare MFS to the X-tree which was shown to outperform other multidimensional indexes [3]. In the following we will refer to this structure as the *dynamic* X-tree. To examine the best query performance that can be achieved by classical index structures, we also include a comparison to a *bulk-loaded* X-tree [2]. For this structure only the accesses to its *leaf nodes* are measured, while for the other data structures, including MFS, *all* accesses are taken into account. Note also that the bulk-loaded tree has a high page utilization (in the experiments it was 90% and higher) and there is generally no overlap of page extents (except, if page boundaries touch). A sophisticated unbalanced split strategy is used to create the page extents. In [2] it was shown that this query-optimized structure by far outperforms other multidimensional indexes like the Hilbert R-tree [17]. However, processing updates on this tree would destroy the optimized structure. For both indexes the page size was set to 8 KB. MFS works at the level of single disk sectors since its approach is not based on index pages.

Note that the indexes used in the experiments actually represent *idealized* structures that have improved query performance. Each data dimension was mapped to a single dimension in the corresponding index, i.e., a d-dimensional data set is stored in a d-dimensional X-tree. In practice this requires a fixed total order of the values of each dimension, hence this type of X-tree could not support efficient dimension updates (see Section 3 of [8] for details). Our X-trees actually are structurally equivalent to the DC-tree [8] without paying its price of using more space consuming minimum describing sequences instead of bounding boxes.

We initially also compared MFS to *augmented* index structures that maintain pre-computed aggregates in their internal nodes [26, 20]. Augmented indexes speed up aggregate queries if a query completely contains an inner tree node's bounding box. On the other hand storing additional values requires more space and thus reduces the node's fanout. In our experiments the augmented version of the X-tree performed virtually identical to the non-augmented one, sometimes even worse. The reason is that the fairly high dimensionality makes it very unlikely that a query completely contains the bounding box of a tree node. This is consistent with our analytical results about the benefit of pre-aggregation on high-dimensional data sets (cf. Section 2.1).

4.1 Experimental Setup

The results presented here were obtained for the Seagate Cheetah 9LP hard disk. The most important parameters are summarized in Table 1. All parameters, including a detailed seek time lookup table for different seek distances, are part of the simulation tool and set to match the exact performance of the real disk.

Parameter	Value
Capacity	9.1 GB
Rotations per min	10045
Seek time	0.83 to 10.63 msec
Controller maximum transfer rate	4.65 MB per sec

Table 1: Important parameters of the disk used for the experiments

We compared the different approaches for a variety of data sets. Here we discuss results for a representative group of real and synthetic data sets. For all data sets we used natural attribute hierarchies for the granularities of MFS. If there was no natural hierarchy, an artificial hierarchy that recursively divides the domain by 10 (large domains), by 4 (medium domains) or by 2 (small domains) was used.

The **TPC-R** data set is taken from the `Lineitem` table of the TPC-R benchmark [36] and has 1,199,969 tuples. It was generated by using dbgen with scale factor 0.2. The dimensions are all 15 attributes of `Lineitem`, except for `comment`. Only the attributes in Table 2 have hierarchies defined by the Star schema of the TPC-R database.

Weather is a real data set which contains 1,036,012 cloud reports for September 1986 [13]. We selected the 9 dimensions `Day`, `Hour`, `Brightness`, `Latitude`, `Longitude`, `Station-ID`, `Present-weather`, `Change-code`, and `Solar-Altitude` for the base table. For `Day` the natural hierarchy (`ALL`, `year`, `month`, `day`) was selected. Data set **WeatherL** has the same attributes, but contains 4,134,221 cloud

Dimension	Levels of Hierarchy
Orderkey	ALL, Region, Nation, Customer
Partkey	ALL, Brand, Type, Part
Suppkey	ALL, Region, Nation, Supplier
Shipdate	ALL, Year, Month, Day
Commitdate	ALL, Year, Month, Day
Receiptdate	ALL, Year, Month, Day

Table 2: Dimensions of the TPC data set which have a natural hierarchy

reports for four months of 1986 (September till December).

Gauss is a synthetic 20-dimensional data set with 1 million tuples. It has 1,000 dense clusters of varying center and variance. The clusters follow a discrete normal distribution.

We measured the performance for several workloads of multidimensional aggregate range queries. The results discussed here were obtained for a workload that was constructed as follows. For a query for each dimension one of the four range predicates $\min \leq x \leq A$ (prefix range), $A \leq x \leq B$ (general range), $x = A$ (point range), and $\min \leq x \leq \max$ (complete range) is chosen with probability 0.2, 0.4, 0.2, and 0.2, respectively. Values A and B are uniformly selected. This way of generating queries models realistic selections and ensures a wide variety of selectivities. Table 3 shows the 20 percent quantiles for the selectivity distribution of the queries that were used to produce the figures. For instance, the value for the 60% quantile and Figure 9 indicates that 60% of the queries in this experiment had a selectivity of 3.8e-5 or below, and that the other 40% had a higher selectivity.

Figure	20%	40%	60%	80%	100%
7, 8, 13	1.5e-9	3.5e-7	3.3e-5	4.1e-3	0.91
9	3.5e-9	6.1e-7	3.8e-5	3.7e-3	0.79
12	1.13-11	6.5e-9	7.4e-7	8.1e-5	0.24
10, 11	4.4e-15	7.9e-13	5.9e-11	1.0e-8	0.06

Table 3: Quantiles of the query selectivity for the experiments

Note that the queries as generated above are the most expensive for MFS because they do not take hierarchies into account. In practice users often query according to hierarchies. For example it is more likely that a query selects the 31-day period from May 1 till May 31 (i.e., the month May), than from say May 5 till June 5. MFS benefits from such hierarchy-aligned queries because its "grid" structure is based on hierarchies, therefore the query intersects less tuples (cf. cost formula in Section 3.2).

4.2 MFS Properties

MFS did **not** take the real query workload into account during construction. The file selection algorithm simply assumed that each query selects a range of 10% of the domain in each dimension. The motivation for this setting it to make sure that MFS supports very selective queries well. Queries that select large regions of the data space will access many tuples anyway and are supported well by the fast sequential accesses. The resulting MFS configurations had between 4 and 5 files. The overall storage overhead was below 2.4% compared to the base data set, the only exception being **Gauss** with 19% overhead.

4.3 A Detailed Analysis of the I/O Costs

We start with a set of experiments that establishes the importance of sequential accesses. We processed 10,000 range queries for the **Weather** data set. For the dynamic X-tree the standard depth-first search algorithm (DFS) was used. This algorithm accesses relevant index pages one by one, i.e., is not optimized for sequential accesses. To reduce the positioning cost, pre-fetching of sectors was enabled. The bulk-loaded X-tree also accesses pages one-by-one and uses pre-fetching. The bulk-loading was set to guarantee that leaf pages that share the same parent node are clustered together and that sectors are accessed in ascending order. This makes the disk's standard pre-fetching strategy very effective. Note that without pre-fetching both indexes perform much worse. For the dynamic X-tree the average query cost increases from 2.8 sec (cf. Table 5) to 5.1 sec; for the bulk-loaded tree the costs increase from 0.79 sec to 2.0 sec. MFS on the other hand does not need pre-fetching because the query algorithm already computes the pseudo-optimal access schedule for the disk.

The results are presented in Figures 7 and 8 for a slow and fast disk controller, respectively. The figures show the query cost distribution over *all* queries. The numbers on the x-axis are the cost quantiles, the y-axis shows the corresponding I/O cost in seconds. For instance, if the cost for query quantile x=80% for a technique is 0.5 sec, this implies that 80% of the 10,000 queries processed with this technique had a cost of 0.5 seconds or less, and 20% had a cost of more than 0.5 sec. We report all quantiles, hence the graphs show the *exact* cost distribution.

The graphs of Figure 7 were obtained for the original configuration of the hard disk (cf. Table 1). MFS clearly outperforms the competitors, virtually never exceeding the cost of a sequential scan. Table 4 shows that MFS in the average not only noticeable reduces the number of head movements, but also the overall amount of transferred data. This causes a much lower average query I/O cost (cf. Table 5). The reason for the lower amount of transferred data is that MFS packs the data densely into the sectors and that it works at sector rather than page granularity. For the indexes the smallest unit of transfer are 8 KB, i.e., 16 sectors. However, using smaller pages reduces the index node fanout and leads to worse overall index

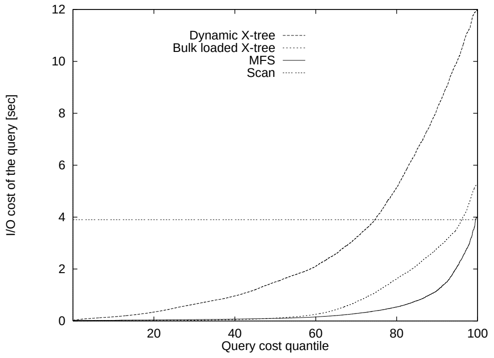

Figure 7: Query cost distribution for **Weather** (slow controller)

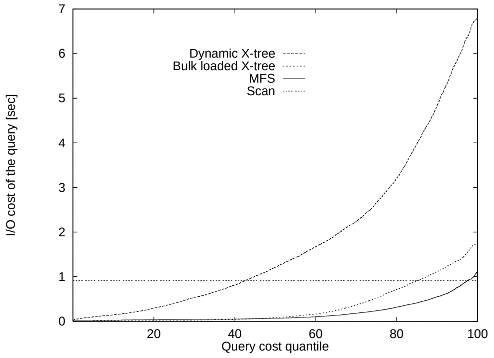

Figure 8: Query cost distribution for **Weather** (fast controller)

performance for high-dimensional data [4].

Cost	MFS	Bulk X-tree	Dyn. X-tree	Scan
Positioning	40.8	311.0	819.3	1
Transfer (KB)	1199	2486	6554	18,500

Table 4: Average number of positioning operations and transferred bytes per query for **Weather**

Figure 7 indicates that the 50% fastest queries on the bulk-loaded index are faster than the fastest 50% of the queries on MFS. However, the absolute I/O-cost difference there is between 10 and 30 msec and mostly caused by not counting the inner node accesses for the index. MFS on the other hand shows clear performance advantages for the expensive queries that take 500 msec and longer and cause noticeable delays for the user. For instance, 78.4% of the queries on MFS take less than 500 msec, compared to only 65.8% on the bulk-loaded index. The dynamic X-tree performs much worse than both MFS and the bulk-loaded tree. About 25% of the queries take longer on the X-tree than by simply scanning the whole data set even though the selectivity of the queries is very low (cf. Table 3).

It is interesting to note that the disk controller is actually quite slow (it was the one used in 1998) and hence constitutes the bottleneck during data transfer. Modern SCSI controllers are able to transfer more than 160 MB per second. Hence we performed the same experiment as above, but now with a controller that is able to transfer 50 MB per second. Now the disk can make true use of its internal speed. Figure 8 shows that all techniques benefit from the higher transfer rate of the controller. However, the high percentage of positioning time limits the benefit for the dynamic X-tree compared to the competitors (see also Table 5). The bulk-loaded tree benefits similarly to MFS. The reason is that most of its 311.0 average positionings (cf. Table 4) are actually removed thanks to the pre-fetching. Note that the pre-fetching is only effective because we made sure that the leaf pages are laid out and accessed on disk in the right order, i.e., following the principles established for MFS.

Data set	MFS	Bulk X-tree	Dyn. X-tree	Scan
Weather (fast controller)	0.18	0.33	1.83	0.91
Weather	0.40	0.79	2.80	3.91
Gauss (fast Controller)	0.50	0.56	2.26	1.16
Gauss	1.35	1.10	2.64	5.02
WeatherL	1.36	2.82	9.07	15.52
TPC-R	0.56	3.60	6.81	5.74

Table 5: Average query costs (in seconds) for the different techniques and data sets

Table 6 summarizes the results for **Weather** from a different point of view. Each of the selectivity ranges in the table contains exactly 20% of the queries. We report the average I/O cost for each of these query groups. As expected the bulk-loaded index has a slight advantage for queries with a very low selectivity (in part caused by not counting inner node accesses). Apart from that MFS clearly outperforms the competitors, especially for the fast controller.

4.4 Results for Various Data Sets

We performed the same experiments as above for multiple data sets. The results show the same trends as for **Weather**.

As Figure 9 illustrates, the cost distribution for **WeatherL** is almost identical to **Weather**. Note that the worst case cost for the dynamic X-tree increased from 12 to 63 sec, i.e., by a factor of 5. For the other techniques the worst case cost increased almost exactly by a factor of 4, i.e., the size ratio of the two weather sets.

We performed an "extreme" test for MFS with **Gauss**. The data set contains dense clusters which are advantageous for the X-trees which construct their pages tuple-oriented. MFS on the other hand uses space-oriented partitioning based on average query costs. In addition to that the queries select extremely

Technique	$\sigma \leq$ 1.2e-9	1.2e-9 $< \sigma \leq$ 3.4e-7	3.4e-7 $< \sigma \leq$ 2.9e-5	2.9e-5 $< \sigma \leq$ 3.7e-3	3.7e-3 $< \sigma$
MFS	0.046	0.099	0.25	0.56	1.05
Bulk X-tree	0.027	0.10	0.36	1.26	2.19
Dynamic X-tree	1.27	1.77	2.92	2.47	5.57
MFS (fC)	0.037	0.067	0.14	0.27	0.42
Bulk X-tree (fC)	0.026	0.072	0.20	0.54	0.83
Dynamic X-tree (fC)	0.96	1.24	1.94	1.62	3.37

Table 6: Average query cost (in seconds) as a function of the selectivity σ; for **Weather**

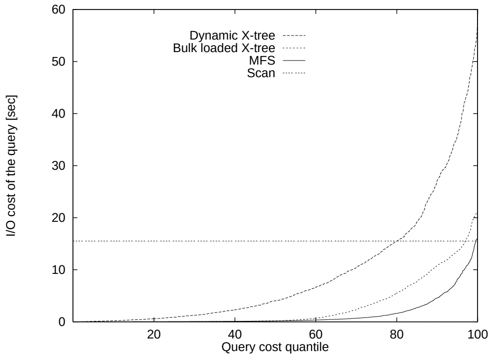

Figure 9: Query cost distribution for **WeatherL**

small regions (80% have a selectivity below 10^{-8}, cf. Table 3) which benefits the X-trees as well. Consequently, the bulk-loaded X-tree shows better cost distribution in Figure 10. However, MFS's strength of taking advantage of the transfer speed of modern hard disks shows in Figure 11. With a faster controller it now outperforms all competitors.

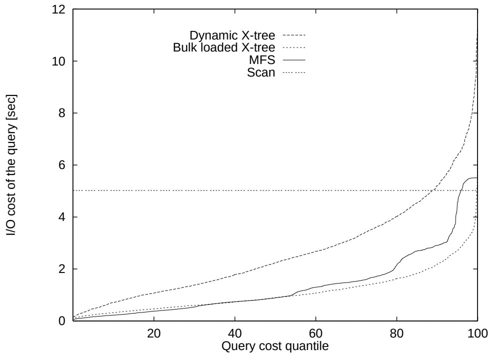

Figure 10: Query cost distribution for **Gauss** (slow controller)

On **TPC-R** MFS shows the greatest performance advantage (cf. Figure 12). The reasons are the high dimensionality and the less skewed data distribution which both give MFS a comparative advantage. The dynamic X-tree performs worse than the sequential scan for almost 50% of the queries.

4.5 Support for Single Updates

We examined the effect of leaving empty space in the MFS files to enable incorporation of single updates. We constructed MFS by leaving k empty sectors in

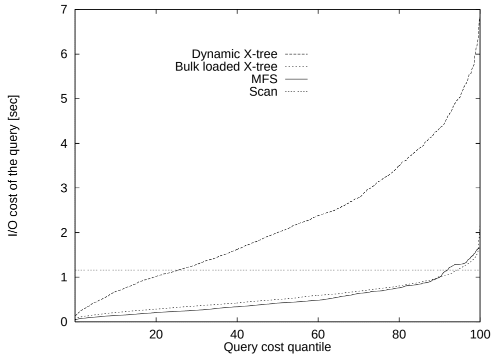

Figure 11: Query cost distribution for **Gauss** (fast controller)

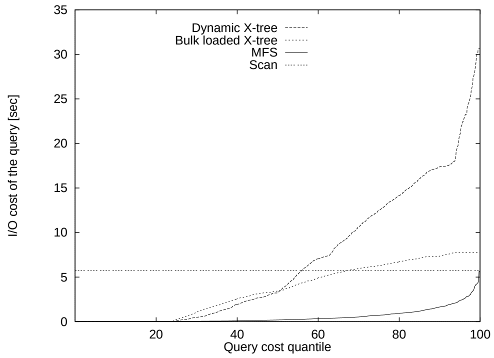

Figure 12: Query cost distribution for **TPC-R** (slow controller)

between chunks of j non-empty sectors. An update would sequentially read the appropriate chunk of j non-empty sectors sequentially and write the updated chunk back. The variables j and k determine the query-update cost tradeoff. Larger j and smaller k increase the update cost, but avoid query overhead caused by reading empty sectors (or seeking over a chunk of empty sectors). Figure 13 shows the query cost distribution for the same set of queries we used in Section 4.3. Variables j and k were such that the storage utilization of MFS was only 75%. We examined the combinations (j=2000,k=400), (j=400,k=80), and (j=80,k=20). As expected, the larger the absolute values, the lower the effect of leaving empty space. Note that for j=2000 and j=400 the cost distribution is almost identical to MFS without empty sectors. This shows that moderate amounts of single updates (up to 25% of the data size) can be supported with virtually no effect on the queries.

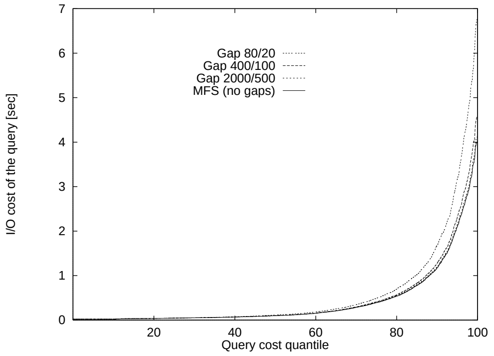

Figure 13: Query cost distribution for configurations of MFS with empty space for single updates

5 Related Work

There exists a large amount of work on indexing data warehouses. However, scalability with increasing dimensionality is not examined. Popular data structures like bitmap indexes [5, 22, 39] and B+-trees [1] are one-dimensional secondary indexes. We can combine them easily with MFS to specifically support queries that select ranges in only a few dimensions. This is possible because MFS uses simple flat files. One can construct one-dimensional indexes for the dimensions in the MFS files in the same way this would be done for any database table.

Jagadish et al. [16] and Markl et al. [21] propose efficient clustering strategies for mapping data warehouse tables to files. MFS's flexibility allows it to support both clustering orders. Hence MFS provides an efficient technique to speed up aggregate range queries over the clustered tables. Note that [21] proposes the UB-tree to index the clustered data. While it is efficient for low-dimensional data, its search performance degrades with increasing dimensionality due to the one-by-one page accesses. Furthermore the UB-tree requires a fixed total order of the base tuples which restricts the possibility of dimension updates.

Roussopoulos et al. [18, 28] use packed R-trees to store tuples of multiple views of the base data in a single data structure. The techniques were shown to be effective for low to medium dimensionality. With increasing dimensionality the R-tree performance is expected to degrade as shown in this paper for the X-tree. Note that the packing requires the tuples to be sorted in row-major order (alphabetically) which was shown to be inferior to lattice path clustering in [16].

Seeger et al. [31, 30] propose algorithms for determining the fastest way of retrieving a set of disk pages when this set is known a priori. Our technique to select the pseudo-optimal schedule is similar to the approach of [31] for limited gaps and unlimited buffer.

Previous work regarding the prediction of range query performance focussed on B-trees and the R-tree family [33, 34, 25]. Our cost model arrives at similar results regarding the selectivity estimation of a query. However, it is specific to our technique and can take advantage of the regular and simple structure of MFS. For data structures like B-trees and R-trees the shape of the bounding box of a tree node can only be approximated which leads to high errors with increasing dimensionality [19].

The well-known *data cube* [11] is a d-dimensional generalization of the `GROUP BY` operator. It computes all 2^d possible results of grouping the d-dimensional base data set by any subset of its dimensions. MFS provides a new way of supporting range aggregate queries for one or more of the groups. It can be similarly combined with approaches that select views to be materialized [14, 12].

6 Conclusions and Future Work

Recently frameworks have been proposed for using index structures to support aggregate queries [20, 26]. These frameworks have been shown to be successful for low to medium dimensionality. Our technique can be seen as a step into the same direction, extending the approach to higher dimensions. Instead of augmenting existing index structures with pre-computed aggregates, MFS is a new solution that explicitly takes the hard disk geometry into account. It provides evidence that a technique based on the sequential scan has substantial benefits for aggregation on high-dimensional data sets.

We have shown that MFS is a lightweight data structure which outperforms dynamic and often even bulk-loaded (and hence query-optimized) balanced index structures. With increasing disk transfer rates MFS's advantage will even grow. Also, in contrast to most bulk-loaded structures, MFS can be updated without losing its query-optimized structure. The analytical query cost model we provided for MFS allows it to be integrated in the database query optimizer.

A great advantage of MFS is its ability to support different clustering orders of the tuples in its files. Hence it can support efficient warehouse clustering, data aging, and append-only databases. Its flat files can also be combined with one-dimensional secondary indexes like bitmaps to support joins. Examining the exact benefits of this combination is part of our future work. Furthermore we plan to combine MFS with features of more tuple-oriented data structures to better support very skewed data sets.

References

[1] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

[2] S. Berchtold, C. Böhm, and H.-P. Kriegel. Improving the query performance of high-dimensional index structures by bulk-load operations. In *Proc. Int. Conf. on Extending Database Technology (EDBT)*, pages 216–230, 1998.

[3] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, pages 28–39, 1996.

[4] C. Böhm and H.-P. Kriegel. Dynamically optimizing high-dimensional index structures. In *Proc. Int. Conf. on Extending Database Technology (EDBT)*, pages 36–50, 2000.

[5] C. Y. Chan and Y. E. Ioannidis. An efficient bitmap encoding scheme for selection queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 215–226, 1999.

[6] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *SIGMOD Record*, 26(1):65–74, 1997.

[7] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.

[8] M. Ester, J. Kohlhammer, and H.-P. Kriegel. The dc-tree: A fully dynamic index structure for data warehouses. In *Proc. Int. Conf. on Data Engineering (ICDE)*, pages 379–388, 2000.

[9] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[10] G. R. Ganger, B. L. Worthington, and Y. N. Patt. *The DiskSim Simulation Environment Version 2.0 Reference Manual*, 1999.

[11] 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 subtotals. *Data Mining and Knowledge Discovery*, pages 29–53, 1997.

[12] H. Gupta and I. S. Mumick. Selection of views to materialize under a maintenance cost constraint. In *Proc. Int. Conf. on Database Theory (ICDT)*, pages 453–470, 1999.

[13] C. J. Hahn, S. G. Warren, and J. London. Edited synoptic cloud reports from ships and land stations over the globe, 1982-1991, 1996. Data available at http://cdiac.esd.ornl.gov/ftp/ndp026b.

[14] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 205–216, 1996.

[15] W. H. Inmon. What is a data warehouse? White Paper, 2000. Available at http://www.billinmon.com/cif/edw/edw_content.html.

[16] H. V. Jagadish, L. V. S. Lakshmanan, and D. Srivastava. Snakes and sandwiches: Optimal clustering strategies for a data warehouse. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 37–48, 1999.

[17] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, pages 500–509, 1994.

[18] Y. Kotidis and N. Roussopoulos. An alternative storage organization for ROLAP aggregate views based on cubetrees. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 249–258, 1998.

[19] C. A. Lang and A. K. Singh. Modeling high-dimensional index structures using sampling. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 389–400, 2001.

[20] I. Lazaridis and S. Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 401–412, 2001.

[21] V. Markl, F. Ramsak, and R. Bayer. Improving OLAP performance by multidimensional hierarchical clustering. In *Proc. Int. Database Engineering and Applications Symp. (IDEAS)*, pages 165–177, 1999.

[22] P. E. O'Neil and D. Quass. Improved query performance with variant indexes. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 38–49, 1997.

[23] B.-U. Pagel, F. Korn, and C. Faloutsos. Deflating the dimensionality curse using multiple fractal dimensions. In *Proc. Int. Conf. on Data Engineering (ICDE)*, pages 589–598, 2000.

[24] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, pages 486–495, 1997.

[25] G. Proietti and C. Faloutsos. I/o complexity for range queries on region data stored using an R-tree. In *Proc. Int. Conf. on Data Engineering (ICDE)*, pages 628–635, 1999.

[26] M. Riedewald, D. Agrawal, and A. El Abbadi. pCube: Update-efficient online aggregation with progressive feedback and error bounds. In *Proc. Int. Conf. on Scientific and Statistical Database Management (SSDBM)*, pages 95–108, 2000.

[27] M. Riedewald, D. Agrawal, and A. El Abbadi. Efficient integration and aggregation of historical information. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2002. To appear.

[28] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cubetree: Organization of and bulk updates on the data cube. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 89–99, 1997.

[29] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, 1994.

[30] B. Seeger. An analysis of schedules for performing multi-page requests. *Information Systems*, 21(5):387–407, 1996.

[31] B. Seeger, P.-A. Larson, and R. McFayden. Reading a set of disk pages. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, pages 592–603, 1993.

[32] A. Shukla, P. Deshpande, J. F. Naughton, and K. Ramasamy. Storage estimation for multidimensional aggregates in the presence of hierarchies. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, pages 522–531, 1996.

[33] Y. Tao, D. Papadias, and J. Zhang. Cost models for overlapping and multi-version structures. In *Proc. Int. Conf. on Data Engineering (ICDE)*, 2002. To appear.

[34] Y. Theodoridis and T. K. Sellis. A model for the prediction of R-tree performance. In *Proc. Symp. on Principles of Database Systems (PODS)*, pages 161–171, 1996.

[35] D. A. Thompson and J. S. Best. The future of magnetic data storage technology. *IBM Journal of Research and Development*, 44(3):311–322, 2000.

[36] Transaction Processing Performance Council. TPC benchmarks. Available at http://www.tpc.org.

[37] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, pages 194–205, 1998.

[38] D. E. Willard and G. S. Lueker. Adding range restriction capability to dynamic data structures. *Journal of the ACM*, 32(3):597–617, 1985.

[39] M.-C. Wu and A. P. Buchmann. Encoded bitmap indexing for data warehouses. In *Proc. Int. Conf. on Data Engineering (ICDE)*, pages 220–230, 1998.

A Proof for the Number of Queries Containing an Aggregate

We show that for a d-dimensional data set any pre-computed value is contained in at most about $\frac{1}{2^d}$ of all possible range queries. First we start with the one-dimensional case.

Let A denote a one-dimensional data set with a domain of size N. Without loss of generality, let domain $\mathcal{D} = \{0, 1, \ldots, N-1\}$. Let $A[x]$ be the measure value of a point $x \in \mathcal{D}$ in A. A pre-computed aggregate value v is the result of applying an aggregate operator to the measure values of selected points in A. A range query *contains* a pre-computed aggregate v iff all the cell locations used to compute it are selected by the query. Here we assume, that a pre-computed aggregate only reduces the query cost for queries that contain it. This holds for all semigroup operators. For groups the existence of an inverse operation theoretically enables the query to take advantage of aggregates which are not contained. However, in practice most techniques simply test for containment to avoid an increase in the complexity.

Clearly, there exist $N(N+1)/2$ possible range queries in a one-dimensional data set. A value $A[x]$ is contained in exactly $(x+1)(N-x)$ of these range queries. Aggregates involving multiple points of A are contained in even less queries. The expression $(x+1)(N-x)$ is maximized for $x = (N-1)/2$, resulting in $(N+1)^2/4$. Consequently, at most $(N+1)^2/4$ out of $N(N+1)/2$ range queries, i.e., approximately 50%, contain $A[x]$. The generalization to d dimensions yields that any pre-computed value is contained in at most about $\frac{1}{2^d}$ of all possible range queries.

B Proof for the Selectivity Formula

The goal is to estimate the selectivity in file F_{k-1} for a query which selects ranges of size q_i in dimension i. The selectivity depends on the q_i and the low-resolution "grid" of cells defined in the parent file F_k (cf. Figure 6).

For simplicity we will first discuss the selectivity increase caused by the low-resolution grid for a data set which only has the single dimension `time`. Assume that file F_{k-1} is at month granularity and that the tuples in file F_k are at year granularity. Let a query select a range of 4 months. If the range starts in January or any other month before October, the query intersects one year. Stated differently, the selectivity of the query in F_{k-1} is determined by the 12 months, not the actual query size of 4 months. If the query range starts in October, November, or December, it will even intersect two years. Hence there are 9 cases that require examining 12 child tuples (the 12 months of the single intersected year), and there are 3 cases when 24 child tuples are to be read. On the average, ignoring boundary effects, the query would retrieve $(9 \cdot 12 + 3 \cdot 24)/12 = 15$ month-tuples in the child file F_{k-1}. Its selectivity increases from $4/N$ to $15/N$, where N denotes the domain size of the dimension.

This analysis easily generalizes to d dimensions. Let $x_i^{(k)}$ denote the side length in dimension i of the region corresponding to a tuple in file F_k. In the example, if the base data set in dimension i is at month granularity and F_1 is at year granularity, then $x_i^{(1)} = 12$. The general formula for the size of the enlarged query region in file F_k therefore is $\prod_{i=1}^{d}(q_i + x_i^{(k)} - 1)$.

We can estimate the average number of accessed tuples in F_{k-1} as the product of the query selectivity and the number of tuples in the file. With the above formula we obtain $n_{k-1} \prod_{i=1}^{d} \frac{q_i + x_i^{(k)} - 1}{N_i}$. To get a better estimate of the actual query *cost* we also take the size of the tuples into account. Let s_k be the number of bits required to encode a tuple of file F_k, including the measure value and the two pointers to the child tuples. Then the average number of *bits* accessed by the query in file F_{k-1} is computed as $n_{k-1} s_{k-1} \prod_{i=1}^{d} \frac{q_i + x_i^{(k)} - 1}{N_i}$.

C A Simple Example for the Initial Granularity Selection

Let the data set have 2 dimensions, both being `time`. The data set, and hence also F_0, is at (month, month) granularity. For F_1 we first compute the cost by setting its granularity to (month,month) as well. Then we evaluate the costs for choosing (year,month) or (month,year) instead. Assume that (month,year) has the lowest cost, hence it becomes the new granularity candidate for F_1. Now we examine if choosing (year,year) or (month,ALL) instead would reduce the costs even further. Let both costs be higher than for (month,year), i.e., we reached a local minimum and therefore fix the granularity of F_1 as (month,year). Now we continue searching for the appropriate granularity for F_2. We start with (month,year), then examine (year,year) and (month,ALL). Assume that none of the costs improves over the costs without F_2. Hence the process terminates and proposes only a single additional file F_1 at (month,year) granularity.