Accessing Scientific Data: Simpler is Better*

Mirek Riedewald[1], Divyakant Agrawal[2], Amr El Abbadi[2], and Flip Korn[3]

[1] Cornell University
Ithaca, NY.
mirek@cs.cornell.edu
[2] University of California
Santa Barbara, CA,
{agrawal,amr}@cs.ucsb.edu
[3] AT&T Labs-Research
Florham Park, NJ,
flip@research.att.com

Abstract. A variety of index structures has been proposed for supporting fast access and summarization of large multidimensional data sets. Some of these indices are fairly involved, hence few are used in practice. In this paper we examine how to reduce the I/O cost by taking full advantage of recent trends in hard disk development which favor reading large chunks of consecutive disk blocks over seeking and searching. We present the Multiresolution File Scan (MFS) approach which is based on a surprisingly simple and flexible data structure which outperforms sophisticated multidimensional indices, even if they are bulk-loaded and hence optimized for query processing. Our approach also has the advantage that it can incorporate a priori knowledge about the query workload. It readily supports summarization using distributive (e.g., `count`, `sum`, `max`, `min`) and algebraic (e.g., `avg`) aggregate operators.

1 Introduction

Modern scientific databases and data warehouses reach sizes in the order of terabytes [7] and soon even petabytes [4]. Hence fast aggregation techniques and efficient access to selected information play a vital role in the process of summarizing and analyzing their contents. The complexity of the problem grows with increasing dimensionality, i.e., number of attributes describing the data space. For instance the cloud data in [12] is defined by 20 attributes.

Despite the availability of high-capacity memory chips, rapidly growing amounts of information still require maintaining and accessing large data collections on hard disk. Hence the I/O cost, i.e., the time it takes to retrieve relevant information from hard disk, dominates the query cost. Our goal is to reduce this cost factor. There is a variety of indices which try to address this problem (see for instance [8] for an overview). Unfortunately, except for the R-tree [11] and its

* This work was supported by NSF grants IIS98-17432, EIA99-86057, EIA00-80134, and IIS02-09112.

T. Hadzilacos et al. (Eds.): SSTD 2003, LNCS 2750, pp. 214–232, 2003.

relatives (e.g., R*-tree [1] and X-tree [3]) few have ever gained wide acceptance in practice. Hence, instead of developing another sophisticated index, our goal is to examine how efficiently queries can be supported using the *simplest possible* data structure—a collection of flat files.

Notice that indices typically concentrate on *what* is accessed, that is, the *number* of accessed index pages during search. In this paper, we demonstrate that 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 [24, 25]. Over the last decade, improvements in disk transfer rates have rapidly outpaced seek times. For example, 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 [30]. 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. Hence, it is often more efficient to read data in large chunks, even if the chunks contain irrelevant sectors.

A naive way to take advantage of the disk geometry in answering queries is by reading the whole data set using fast sequential I/O. While this approach, known as *sequential scan*, may make sense when much of the data must be accessed, it is not efficient for very selective range queries. We propose the Multiresolution File Scan (MFS) technique to address this issue. MFS is based on a selection of flat files ("views") that represent the data set at multiple 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. Our experiments show that for processing range queries MFS achieves up to two orders of magnitude speedup over the sequential scan and is on average a factor of 2-12 times faster than the X-tree.

The simple structure of MFS makes it easy to model its performance. It also allows the incorporation of a priori knowledge about the query workload. Another benefit is the flexibility in choosing the sort order of the data set, which is particularly useful for temporal data. Often there is a strong correlation between the value of a temporal attribute and the time when the corresponding data point is inserted into the database [21]. For instance environmental measurements and sales transactions are recorded in a timely manner. Later analysis then is concerned with discovering trends, hence also involves the temporal attribute(s). MFS enables grouping of points with similar time values in neighboring disk blocks. Hence queries and updates observe a high locality of access. Similarly, removing chunks of old (and hence less relevant) data from disk to slower mass storage only affects a small portion of the data structure.

The outline of this paper is as follows. In Section 2 we motivate the design of MFS and then present its data structures and algorithms in Section 3. The selection of the files for MFS is discussed in Section 4, including a discussion

on file selection for temporal data. Section 5 shows how MFS design principles can improve the query cost of standard indices. Detailed experimental results are presented in Section 6. Related work is discussed in Section 7. Section 8 concludes this article.

2 Problem Definition and Motivation

Scientific and warehouse data is often conceptualized as having multiple logical *dimensions* (e.g., `latitude`, `longitude`) and *measure attributes* (e.g., `temperature`, `cloud cover`). Complex phenomena like global warming can be analyzed by summarizing measure attribute values of points which are selected based on predicates on the dimensions. A typical and important query in scientific and business applications is the *multidimensional range aggregate*. It computes an aggregate over range selections on some (or all) of the dimension attributes. Suppose we have geographical data with schema `(latitude, longitude, temperature)`. An example of a range aggregate on this data is, "Find the minimum and maximum temperatures in the region of Santa Barbara County."

In the following we will concentrate on the `count` operator. Other aggregate operators can be handled in a similar way. Note that applying the `null` "aggregate" operator is equivalent to *reporting* all selected data points without performing any aggregation. While MFS can efficiently answer such reporting queries, the focus of this paper is on its performance for range aggregates. For the sake of simplicity we will further on assume that the data set has d *dimension* attributes and a single *measure* attribute (for the `null` operator no measure attribute is necessary). Our technique can be easily generalized.

Hard Disk Geometry. Here we only provide a high-level view of relevant disk features; the interested reader might consult [24, 23] 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.

To be able to find the best access strategy, the set of target sectors must be known in advance. The *pre-fetching* option of modern disk drives only partially addresses this problem by automatically reading several sectors *after* the requested target sectors and keeping them in the disk cache for future requests.

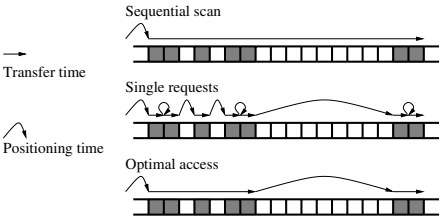

Fig. 1. Reading disk sectors (relevant sectors are shaded

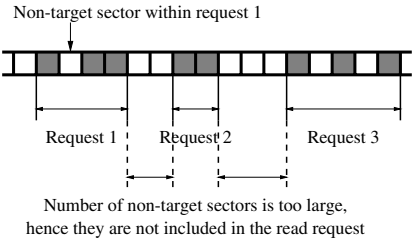

Fig. 2. Pseudo-optimal access schedule for $R = 2$

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.

Dimensionality Curse for Indexing and Aggregation. There is an inherent dimensionality curse in indexing and computing range aggregates. The best known algorithms whose runtime is provably sub-linear in the size of the data set, e.g., [6], have polylogarithmic query cost and storage overhead. However, for dimensionality $d \geq 9$ a polylogarithmic cost is practically worse than a linear cost. Let n denote the number of data points. 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 scan 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 sets.

A similar dimensionality curse has also been observed for popular practical indexing techniques [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 [5, 32].

Another possibility to speed up aggregate queries is to materialize pre-computed values and to use them to reduce on-the-fly computation and access costs, e.g., as has been proposed for the data cube operator [10]. Unfortunately for a d-dimensional data set any pre-computed value is contained in at most about one out of 2^d possible range queries (proof omitted due to space constraints). Hence the effectiveness of materializing pre-computed aggregates is very sensitive to changes in the query workload.

3 The MFS Technique

The discussion in the previous section shows that even the most sophisticated index cannot *guarantee* good query performance for multidimensional data sets with 8 or more dimensions. Together with the limited practical impact complex indices have had in the past, there is a strong motivation to try the opposite direction—supporting queries using the *simplest* possible data structure.

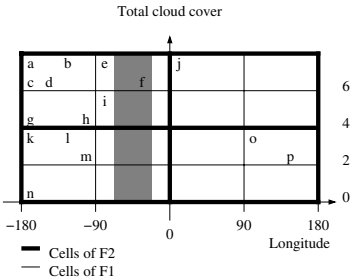

Fig. 3. Data set and query (shaded)

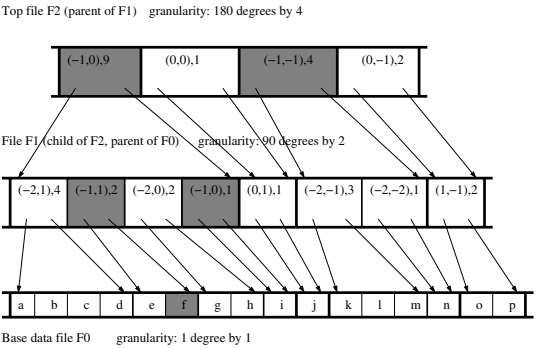

Fig. 4. Basic structure of MFS (tuples intersected by query are shaded)

3.1 Overview of MFS

MFS consists of a collection of flat files and pointers. Each file represents the data set at a different level of resolution. The file with the finest resolution contains the exact information. The next coarser file partitions the data space into regions and stores for each region the number of data points in the region.[1] The next coarser file in turn stores the count value over regions of regions, etc.

Figure 3 shows an example for a data set with the two dimensions `longitude` and `total cloud cover` [12]. The letters indicate weather stations that measure the cloud cover for certain locations (projected to their longitudes for presentation purposes). The user is interested in finding out how many stations measured a given cloud cover range for a given longitude range. Note that in practice there are far more dimensions.

The original data set is at the granularity of single degrees by 1 cloud intensity level. Figure 4 shows a possible MFS where the coarser files are at 90 degrees by 2 levels and 180 degrees by 4 levels resolution, respectively. The tuples in the files are shown as (`longitude`,`cloud cover`),measure and correspond to the cells indicated in Figure 3. For instance the first tuple in top file F_2 corresponds to the upper left quarter of the whole data space, i.e., to range $[-180, -90)$ by $[4, 8]$ in `longitude` and `total cloud cover` dimension, respectively. Note that coarser granularity enables a more efficient encoding of the dimension values since less bits are necessary to distinguish between the values.

The granularities establish a conceptual hierarchy among the files. Figure 4 illustrates this hierarchy by placing the file with the coarser resolution above the file with the next finer one. We will therefore refer to the file with the coarsest 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

[1] Recall that we describe MFS for the `count` operator, other operators are handled similarly.

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

1. Scan top file to find tuples intersected by query;
2. Compute approximate query result with error bounds and output it;
3. While (the file has a child file of finer resolution)
   Compute pseudo-optimal schedule to access all relevant sectors in child file;
   Access child file according to schedule to find tuples intersected by query;
   Output new approximate result when all child tuples of a parent are read;

Fig. 5. Querying MFS ("breadth-first")

data. Knowledge about the workload can be taken into account by selecting the appropriate file granularities.

An example for a query that selects the longitude range from −22 to −66 degrees and the whole cloud cover range is indicated in Figure 3. In Figure 4 the tuples that intersect the query at the different granularities are shaded. To allow efficient pruning during range query processing, each non-base tuple has two pointers `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.

3.2 Processing Queries

The query procedure is similar to a breadth-first tree traversal. The main contribution of MFS are the emphasis on file organization and access order rather than on the number of accessed "index" pages alone. Another main aspect is the use of a simple and fast, yet very efficient disk access scheduler.

A query is processed by first sequentially scanning the (small) top file. Then the relevant tuples in its child file are read using a pseudo-optimal access schedule that takes the disk parameters into account. Then the next child file is pseudo-optimally accessed and so forth. This way the query proceeds breadth-first, file by file in increasing order of resolution. The `first` and `last` pointers provide the information for computing the pseudo-optimal schedule. The overall algorithm is shown in Figure 5. Approximate output is generated similar to [20].

The **pseudo-optimal disk access schedule** is computed using a very fast approximation algorithm which assumes a simple disk model. This algorithm is a simplified version of a scheduler proposed in [25], nevertheless the experimental results in Section 6 show that it is very effective. Let $R = \frac{t_{\text{positioning}}}{t_{\text{transfer}}}$ denote the ratio between the "average" 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 2 shows an example for $R = 2$.

3.3 File Organization

In general the tuples in a file could be sorted in *any* order. However note that during query processing for an intersected tuple in file F_i all child tuples in the next file F_{i-1} are accessed to refine the result. In order to increase locality of access these "sibling" tuples should be stored in neighboring disk sectors. Hence each file (except for the top file) consists of chunks of sibling nodes which could be stored in any order on disk, e.g., Z- or Hilbert order.

The requirement of clustering tuples that share the same parent tuple establishes a *partial order* which is determined by the selected file granularities. Tuples in a file therefore can be stored in any order which is consistent with this partial order.

3.4 Bulk-Loading MFS

For now assume the granularities for the files of MFS are known. As discussed in Section 3.3 the tuples in a file can be stored in any order that is consistent with the partial order established by the file granularities. For bulk-loading a consistent *total order* "$\leq$" is selected which has the following property. For any two base tuples t_i and t_j it holds that if $t_i \leq t_j$ then for all ancestors a_i and a_j of t_i and t_j it also holds that $a_i \leq a_j$. Such an order trivially exists (just group descendent tuples together recursively).

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 of the ancestor order ensured by "$\leq$". The algorithm only needs little memory to hold the currently processed tuple with all its ancestors. The values of the pointers `first` and `last` are generated during the scan as well. In Figure 6 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, therefore during step 3 only tuples of the small coarse-resolution files are written to disk. The write cost thus is negligible compared to the cost of scanning the sorted base data file.

3.5 Updating MFS

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

```
Algorithm Bulk-load
Input: File containing input data; total order ≤ of input tuples
Output: Files of MFS
1. Sort input file baseFile according to total order ≤;
2. Read first tuple from baseFile and generate list Ancestors of
     its ancestor tuples in the coarse-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;
```

Fig. 6. Bulk-loading MFS (writes are buffered)

is simple and inexpensive. As discussed in Section 3.3 the files only have to maintain a partially ordered set. Hence whenever an update leads to overflowing sibling chunks or gaps within the file, selected sibling chunks can be moved to a different location in the file. A standard garbage collection algorithm can maintain the gaps within a file at a low cost [5]. Details are omitted due to space constraints. Updates to the dimensions, e.g., inserting a new product, can be processed in the same way.

MFS's regular structure also simplifies batch updates. First a small delta-MFS is built for the updates (using MFS' granularities). If MFS is maintained according to the total order "≤", both MFS and delta-MFS can be merged in a single scan. Otherwise the sub-"trees" of common top-level tuples are merged using pseudo-optimal accesses.

Note that one can leave "gaps" of empty sectors within the files of MFS to enable fast single updates with very low re-organization cost. Thanks to the use of the pseudo-optimal scheduler these gaps have virtually no effect on the query cost (cf. Section 6).

4 Selecting the Granularities for the MFS Files

The main design challenge for MFS is to select the right number of files and their granularities. We show how MFS can take workload information into account and discuss how to construct it if no such knowledge exists.

4.1 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 finest resolution and $l(i) - 1$ corresponding to the artificial ALL value that stands for the whole domain of attribute i (cf. [10]). 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. We assume that MFS has l files F_0, F_1,...,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. Notice that our model can also be applied to continuous domains by assuming that a point has a size of 1 as well. Tuples of coarser resolution files correspond to hyper-rectangular regions of base level cells. We will use $x_i^{(k)}$ to denote the side length in dimension i for the region of a tuple in file F_k.

4.2 Estimating Query Performance for a Given MFS

Given a certain MFS configuration we evaluate its expected query performance by using a fast and lightweight analytical solution. As described in Section 3.2 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}, $1 \leq k \leq l$, is affected by the granularity of its *parent* file F_k. Let query q select a multidimensional range that has size q_i in dimension i. The expected selectivity of q in file F_{k-1} $(0 < k \leq l)$ can then be computed as $\prod_{i=1}^{d} \frac{q_i + x_i^{(k)} - 1}{N_i}$ (proof omitted due to space constraints). 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.

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 an 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.

To compute the result we have to determine all n_k, the number of tuples in file F_k. One can either materialize the files to obtain the exact values, or use

efficient approximation techniques like the one proposed by Shukla et al. [26]. For *fractal* data sets (cf. [17]) 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.

4.3 Addressing the Complexity

So far we have a formula that computes the cost for a certain MFS configuration. However, the number of possible MFS configurations is exponential in the number of dimensions. We therefore use a greedy heuristic to explore this space.

The algorithm first selects the granularity for F_1 assuming MFS only consists of F_0 and F_1. It greedily reduces the resolution of F_1 in that dimension which leads to the greatest reduction in query cost. Once a local minimum is reached, the algorithm adds a new top level file F_2 with F_1's resolution and greedily decreases F_2's resolution, and so on until adding a new top level file does not result in any improvement.

This greedy process over-estimates the cost for non-top level files during the search. When a new candidate for the top file is examined, the cost of a scan is assumed for it. Later, when a file with coarser resolution is added above it, the actual cost contribution is much lower. To correct for this problem we use a second bottom-up process which greedily *increases* the resolution of the files. This process iterates over all files until no improvements are possible.

The overall process in the worst case 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.

4.4 Taking the Query Workload into Account

The formula derived in Section 4.2 computes the average cost given a certain query specification. A workload containing several of these query specifications with their respective probabilities can be 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". Such knowledge about queries aligning in some dimensions can be incorporated by correcting the query volume enlargement caused by the coarser parent file.

Knowledge about the *location* of a query can also be taken into account, e.g., for density-biased queries. One simply needs to partition the data space into smaller regions and then perform the cost analysis using the appropriate query workload and weight for each region. Similar to multidimensional histograms [18] there is a tradeoff between accuracy of estimation and storage and maintenance costs. Examining this tradeoff is beyond the scope of this paper.

4.5 Default Configuration

So far we have discussed how to select the file granularities when a query workload is given. If there is no such knowledge, one could choose any typical mix of range queries. During the experimental evaluation we noted that choosing very selective queries to configure MFS resulted in best overall performance. Choosing such queries generated a larger number of coarse-resolution files, allowing efficient pruning during search. Queries that select large ranges typically access a large number of disk blocks in any MFS-configuration and are well supported by the pseudo-optimal disk scheduling anyway. In the experiments (cf. Section 6) MFS was simply configured assuming each query selects 10% of the domain in each dimension. The resulting data structure performed very well for queries with varying selectivity, indicating the robustness of MFS.

4.6 MFS for Temporal Data

MFS's flexible tuple order enables it to efficiently support management of real-time observations. Such observations could be measurements of environmental factors (e.g., temperature, amount of clouds), experimental data, or sales transactions in retail stores. All these applications have in common that the observations are time stamped in order to be able to discover *trends* like global warming or unusual sales patterns. As discussed in [21], there is a strong correlation between the time of the observation and the time when a tuple is inserted into the database. In other words, the data set consists of a *current* part which is heavily updated and a much larger *historical* part which is mainly queried and rarely updated.

MFS can explicitly support such applications by choosing a tuple order which groups tuples with similar time stamps into neighboring disk blocks. This is done by selecting a sufficiently fine resolution for the temporal attribute at the top file. For instance, if the temporal attribute is at granularity `month` in the top file, all tuples belonging to the same month are automatically clustered together. Newly arriving observations therefore only affect the small part of MFS which corresponds to the current month, guaranteeing a high locality. Parts that contain the previous months are rarely updated and hence can be packed without leaving empty space into disk blocks. Queries that search for trends and hence compare aggregate values for different time periods observe similar locality since the values are essentially clustered by time periods.

The same benefits apply to *data aging* processes in data warehouses which manage rapidly growing amounts of data by retiring old information from hard disk to slower mass storage.

5 Improving Indices

New indexing techniques proposed in the literature only rarely address the exact layout of the structure on disk and the scheduling of node accesses taking this

layout into account. Our experiments indicate that these aspects should become integral parts of index structure proposals since they considerably affect the performance.

Figure 9 shows how the query cost of the X-tree index can be reduced by applying MFS' design principles. The first graph shows the query cost distribution for the standard depth-first (DFS) traversal of a dynamically generated X-tree. The second graph indicates the improvement obtained for exactly the same index structure, now using breadth-first (BFS) traversal and our pseudo-optimal scheduler for accessing the next tree levels. Finally the curve for the bulk-loaded X-tree was obtained by clustering the tree level-wise and making sure that the children of an inner node are stored in neighboring disk blocks. The experimental setup is explained in detail in Section 6.

6 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 [9]. 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 *dynamic* X-tree [3] and a *bulk-loaded* X-tree [2]. The latter 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). Furthermore the bulk-loaded tree also is given the advantage of having all its non-leaf nodes in fast cache. The leaf nodes also are laid out such that they are traversed in order, i.e., the disk arm does not have to move forth and back while processing a query. For both indices 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.

We initially also compared MFS to *augmented* index structures that maintain pre-computed aggregates in their inner nodes [20, 16]. However, due to the fairly high dimensionality of the data, the results were virtually identical to the non-augmented indices, sometimes even worse, and hence are not included. Similarly the performance comparison for indexing, i.e., for *reporting* all selected points rather than computing an aggregate, led to almost identical results and therefore is omitted as well.

6.1 Experimental Setup

The results presented here were obtained for the Seagate Cheetah 9LP hard disk (9.1 GB, 10045 rpm, 0.83–10.63 msec seek time, 4.65MB/sec max transfer rate). 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. The results for other disks are similar.

Table 1. Quantiles of the query selectivity for the experiments

Figure	20%	40%	60%	80%	100%
7, 8 9, 10	1.5e-9	3.5e-7	3.3e-5	4.1e-3	0.91

We compared the different approaches for a variety of data sets. Due to lack of space only the results obtained for two real data sets are discussed here. Experiments with other data, e.g., the TPC benchmark [31] and synthetic skewed data consisting of Gaussian clusters, were similar.

Weather is a real data set which contains 1,036,012 cloud reports for September 1986 [12]. We selected the 9 dimensions `Day`, `Hour`, `Brightness`, `Latitude`, `Longitude`, `Station-ID`, `Present-weather`, `Change-code`, and `Solar-Altitude`. For `Day` the natural hierarchy (`ALL`, `year`, `month`, `day`) was selected. For the other dimensions we recursively partition the domain by 10 (large domains), by 4 (medium domains) or by 2 (small domains). Data set *WeatherL* has the same attributes, but contains 4,134,221 cloud reports for four months of 1986 (September till December).

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 1 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 7 indicates that 60% of the queries in this experiment had a selectivity of 3.3e-5 or below, and that the other 40% had a higher selectivity.

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

Table 2. Average query costs (in seconds)

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
WeatherL	1.36	2.82	9.07	15.52

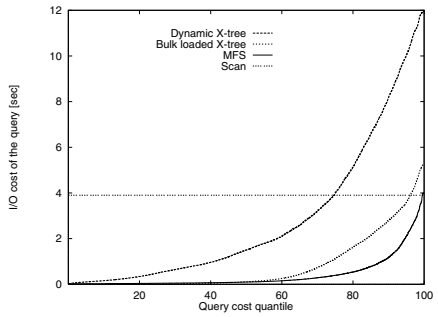

Fig. 7. Query cost distribution for *Weather* (slow controller)

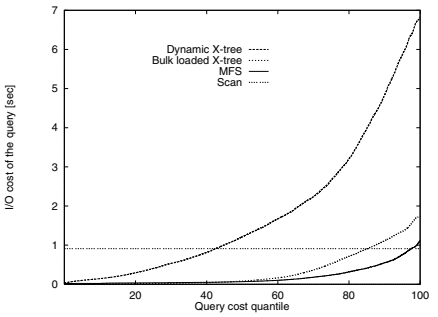

Fig. 8. Query cost distribution for *Weather* (fast controller)

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 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 (cf. Section 4.5). The resulting MFS configurations had between 4 and 5 files. The overall storage overhead was below 2.4% compared to the base data set.

6.2 Performance of MFS

We report results obtained for 10,000 range queries. Notice that for the indices the disk's pre-fetching feature was enabled. Without it the average query cost increased from 2.8 to 5.1 sec (dynamic X-tree) and from 0.79 to 2.0 sec (bulk-loaded X-tree).

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. 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. MFS clearly outperforms the competitors, virtually never exceeding the cost of a sequential scan. Note 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 1).

MFS and Future Hard Disks. To examine the fitness of the data structures with respect to current hard disk trends, we increased the disk's transfer rate by a factor of 4.3 without changing the parameters that affect positioning time. Figure 8 shows that all techniques benefit from the higher transfer rate, but the high percentage of random accesses limits the benefit for the dynamic X-tree compared to the competitors (see also Table 2). The bulk-loaded tree benefits similarly to MFS. The reason is that most of its disk positionings are actually masked thanks to the pre-fetching. Recall 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.

Scalability with Data Size. The cost distribution for the larger *WeatherL* is almost identical to *Weather*, hence the graphs are omitted. The average query costs are reported in Table 2. It turned out 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, the size ratio of the two weather sets.

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 between chunks of j non-empty sectors. An update would sequentially read the appropriate chunk of j non-empty sectors 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 10 shows the query cost distribution for the same set of queries we used in Section 6.2. Variables j and k were such that the storage utilization of MFS was only 75%. We examined the combinations (j=2000,k=500), (j=400,k=100), 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. The experiment shows that moderate amounts of single updates (up to 25% of the data size) can be supported with virtually no effect on the queries.

7 Related Work

The DABS-tree [5] and the VA-file [32] (including its variants) explicitly address high-dimensional data spaces, but focus on nearest neighbor queries. The underlying idea is to have simple two-level schemes where the first level is sequentially scanned and then the relevant data in the second level is accessed. MFS extends

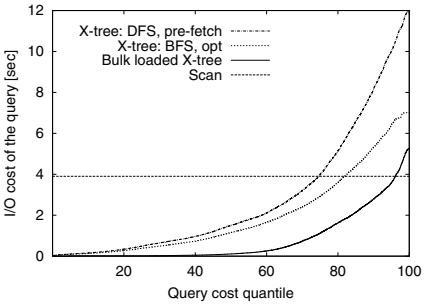

Fig. 9. Index performance for *Weather* (slow controller)

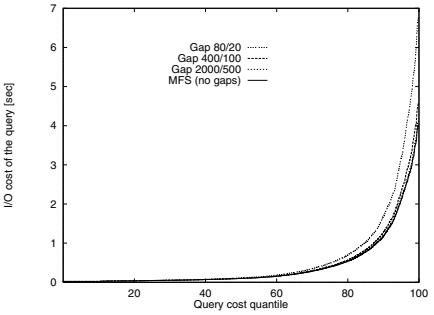

Fig. 10. Query cost distribution for MFS with empty space for single updates

these approaches and addresses range queries. It is more general by allowing multiple "levels" depending on the query workload.

Roussopoulos et al. [14, 22] 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 [13]. MFS supports lattice path clustering.

Seeger et al. [25, 24] 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 [25] for limited gaps and unlimited buffer.

Previous work regarding the prediction of range query performance focused on B-trees and the R-tree family [28, 29, 19]. 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 [15].

Recent work by Tao and Papadias [27] improves index performance by adaptively changing the number of pages assigned to index nodes. Their motivation, like for MFS, is to reduce the number of disk positioning operations (seeks). Since the technique is based on detailed cost models, its effectiveness will rapidly decrease with less accurate cost prediction in medium to high dimensions. MFS takes a more radical approach away from improving indices and using complex (and hence fragile) cost models.

8 Conclusions and Future Work

Efficient access and summarization of massive multidimensional data is an essential requirement for modern scientific databases. The MFS technique shows that this can be achieved with a very simple lightweight data structure. The key to MFS' performance is the consequent orientation on the geometry and parameters of hard disks. With rapidly increasing transfer rates of modern hard disks, MFS will further benefit.

In the experiments MFS clearly outperformed existing popular indices, even when they are bulk-loaded and hence optimized for query performance. We also showed how principles developed for MFS can be applied to indices in order to increase their performance.

MFS' simple and flexible structure enables it to take query workloads into account. By choosing an appropriate tuple order, timestamped data (e.g., experimental measurements, sales transactions) and data aging can be managed with guaranteed locality of access. For the same reason provably efficient clustering strategies like snaked lattice paths [13] are supported as well. Thanks to its use of simple flat files, MFS can also be combined with popular secondary indices like B+-trees. Examining the actual tradeoffs involved in choosing certain tuple orders and secondary indices is part of our future work.

References

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 322–331, 1990. 215

[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. 225

[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. 215, 217, 225

[4] P. A. Bernstein et al. The Asilomar report on database research. *SIGMOD Record*, 27(4):74–80, 1998. 214

[5] 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. 217, 221, 228

[6] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988. 217

[7] Winter Corporation. Database scalability program. http://www.wintercorp.com, 2001. 214

[8] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998. 214

[9] G. R. Ganger, B. L. Worthington, and Y. N. Patt. *The DiskSim Simulation Environment Version 2.0 Reference Manual*, 1999. 225

[10] 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-totals. *Data Mining and Knowledge Discovery*, pages 29–53, 1997. 217, 222

[11] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 47–57, 1984. 214

[12] C. J. Hahn, S. G. Warren, and J. London. Edited synoptic cloud reports from ships and land stations over the globe, 1982-1991. http://cdiac.esd.ornl.gov/ftp/ndp026b, 1996. 214, 218, 226

[13] 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. 229, 230

[14] 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. 229

[15] 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. 229

[16] 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. 225

[17] 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. 223

[18] 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. 223

[19] 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. 229

[20] 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. 219, 225

[21] 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*, pages 13–24, 2002. 215, 224

[22] 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. 229

[23] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, 1994. 216

[24] B. Seeger. An analysis of schedules for performing multi-page requests. *Information Systems*, 21(5):387–407, 1996. 215, 216, 229

[25] 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. 215, 219, 229

[26] 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. 223

[27] Y. Tao and D. Papadias. Adaptive index structures. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, pages 418–429, 2002. 229

[28] Y. Tao, D. Papadias, and J. Zhang. Cost models for overlapping and multi-version structures. In *Proc. Int. Conf. on Data Engineering (ICDE)*, pages 191–200, 2002. 229

[29] 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. 229

[30] 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. 215

[31] Transaction Processing Performance Council. TPC benchmarks. http://www.tpc.org. 226

[32] 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. 217, 228