Tertiary Storage Organization for Large Multidimensional Datasets

Sachin More[*], Alok Choudhary
Department of Electrical and Computer Engineering
Northwestern University
Evanston, IL 60028
{ssmore,choudhar}@ece.nwu.edu
tel +1-847-467-4129
fax +1-847-491-4455

1 Introduction

Large multidimensional datasets are found in diverse application areas, such as data warehousing [6], satellite data processing, and high-energy physics [9]. According to current estimates, these datasets are expected to hold terabytes of data. Since these datasets hold mainly historical and aggregate data, their sizes are increasing. Daily accumulation of raw data and jobs generating aggregate data from the raw data are responsible for this increase. Hence, estimates for the dataset sizes run into several petabytes. Though cost per byte as well as area per byte for secondary storage has been dropping, it is still not cost effective to store petabyte-sized datasets in the secondary storage [4].

Efficient storage organization for multidimensional data has been investigated extensively [8, 1, 5]. Chen et al [1] discuss organization of multidimensional data on a hierarchical storage system. The authors prove that the problem of efficient organization of multidimensional data on a one-dimensional storage system, such as tertiary storage, is NP-complete when arbitrary range queries are allowed. They present a five step strategy based on heuristics for the problem. Jagadish et al ([5]) investigated the problem of efficient organization of a data warehouse on secondary storage. The workload consists of a restricted set of range queries using hierarchies defined on the dimensions. They cast the problem as finding an optimal path through a lattice. They propose a dynamic programming based algorithm that determines how various dimensions are laid out.

We are not aware of any work that takes into consideration practical constraints like the order in which the data already exists or will be generated. Given an order in which data currently exists (or will be generated), and a limited amount of temporary storage space, we investigate issues in efficiently organizing multidimensional datasets on tertiary storage. We cast the problem as permutation of the input data stream using limited storage space. The rest of this document is organized as follows: The problem is formulated in Section 2. Section 3 describes our approach. In Section 4, we present performance results. Section 5 presents conclusions.

[*]Supported by DOE ASCI Alliance program under a contract from Lawrence Livermore National Labs B347875.

2 Background

Queries In a multidimensional dataset, each data item occupies a unique position in a n-dimensional hyperspace. A query selects a subset of the data items by selecting a subset of the domain in each dimension. A query is an instance of a *query type* [1]. A query type is a n-tuple whose values are drawn from $\{ALL, VALUE, ANY, RANGE\}$ for each dimension. ALL selects the entire domain of the dimension. $VALUE$ selects exactly one value from the domain. ANY is similar to $VALUE$, but choose all domain values with equal probability. $RANGE$ choose a set of values from the domain of the dimension. We assume that approximate execution probabilities of query types are known.

Native order and storage order A data source generates data items in a known order (e.g., in temporal order, when *time* is one of the dimensions). We call this ordering of the data items *native order*. Depending on the expected query types, this native order may not be the most efficient way to store the dataset. We call the order in which the data items are stored as the *storage order*. Data items need to be *permuted*, to transform the native order into storage order.

Storage model The storage model consists of secondary storage of size D pages and tertiary storage of size at least T pages, where T is the size of the dataset and $T > D$. The tertiary storage consists of a tape drive controlled by a robotic arm and a set of magnetic tapes. The data items generated by the data source temporarily reside on the secondary storage before they are stored on tertiary storage. The secondary storage pages can be viewed as temporary storage that is used to carry out data permutation.

3 Data clustering algorithm

Given native ordering of data items and the expected query workload, we derive a storage order that optimizes the I/O time of the workload subject to some constraints and assumptions [7]. The input data is read exactly once in the native order. We estimate the available temporary storage size to be of the order of the size of expected answer sets for queries in the workload. We use the knowledge about the expected query types and their execution probabilities to compute the storage order.

Though the storage order defines an ordering on data items, the process used to arrive at the order need not be a sorting process. Given a set of data items, it is important to identify the subsets of the data items that are accessed together. This is an important observation, since the success of transforming the native order into a storage order depends on amount of space available to carry out the transformation. We show that, given a dataset and a workload, there exist multiple storage orders that are equally good. For a given native order and amount of temporary storage space, only some of them are achievable [7]. In general, data items need to be clustered based on their coordinate values in a subset of dimensions. The clustering process needs to *match* coordinate values of the data items rather than *sort* the data items based on their coordinate values. This observation forms the basis of the data clustering process in this paper.

3.1 Computing affinity between data items

The actual process of matching data items uses a measure of *affinity* between data items. Affinity between two data items, I_i and I_j, is the probability that if I_i is accessed by a query then I_j is also accessed, and vice versa. Two data items that are always accessed together have an affinity equal to 1. Two data items that are never accessed together have an affinity equal to 0. We use the following formulae to compute affinity between two data items, I_1 and I_2, is $\sum_{Q_i=Q_1}^{Q_k}(p_i \times \prod_{D_j=D_1}^{D_n} \lambda_i^j)$. Where Q_i is the i^{th} query type and p_i is its execution probability. D_j refers to the j^{th} dimension. $Q_i(j)$ is the selection criteria of Q_i in dimension D_j. If $Q_i(j) = VALUE$ then v_i^j is the value parameter for Q_i from the domain of D_j. If $Q_i(j) = RANGE$ then r_i^j is the range parameter for Q_i from the domain of D_j. $I_1(j)$ and $I_2(j)$ are the coordinates of I_1 and I_2 in D_j. λ_i^j is calculated using the following table:

	λ_i^j
$Q_i(j) = ALL$	1
$Q_i(j) = ANY$ and $I_1(j) = I_2(j)$	1
$Q_i(j) = VALUE$ and $I_1(j) = I_2(j) = v_i^j$	1
$Q_i(j) = RANGE$ and $\lvert I_1(j) - I_2(j)\rvert \leq r_i^j$	1
otherwise	0

3.2 The heuristic approach to data clustering

[1] proves that a simpler version of the problem, one without temporary space constraint, is NP-complete. Hence, we use heuristics to design our algorithm and evaluate them experimentally. The generic algorithm inputs some data items in native order in each iteration. It also produces, in each iteration, some output data items in storage order. This process is repeated until all data items are output. A heuristic approach needs to answer the following questions:

1. If there are $d \leq D$ pages free in the temporary storage, how many data items to input in each iteration?

2. If there are k data items in the temporary storage how many and which data items to output in each iteration?

Greedy heuristic The greedy heuristic answers these questions as follows: *Input as many data items as possible to fill up the temporary storage during each iteration. Output data items only if the temporary storage is full.* While outputting, it chooses the data item that has maximum affinity to the last data item that was output. The rationale behind the greedy heuristic is as follows: It is important to keep the temporary storage filled to capacity, since that gives a wider choice to the output phase in choosing data items to be output, which will result in better decisions. Hence, the input phase should read in as many data items as possible, and output phase should output as few data items as possible. The greedy heuristic results in a simple and low complexity algorithm.

Look-back-m heuristic This is a generalization of the greedy heuristic. The greedy heuristic looks at only one data item, the one that it just output, to decide which data item to output next. The look-back-m heuristic remembers the last m data items that were selected for output. It uses a larger history to improve the quality of the solution. For each data item in the temporary storage, the algorithm computes its affinity to these m data items, called *backward affinity*. It outputs the data item that has the maximum backward affinity. Remembering the last m data item has the side effect of reducing the space that holds incoming data items. If M pages (output bin) are used to hold the previous m data items, then only $D - M$ pages (input bin) are available to hold the incoming data items. This reduces the choice available to select the next data item to be output.

Forward tuning During the execution of the Look-back-m heuristic, multiple candidate data items in the input bin can have same backward affinity. For example, when there are no data items in the output bin, in the initial phase of the algorithm, all data items in the input bin have zero backward affinity. The algorithm randomly chooses from candidate data items. The forward tuning technique uses data items in the input bin to improve the performance. For each data item in the input bin, it computes its affinity to the remaining data items in the input bin, called *forward affinity*. It outputs the data item having maximum forward affinity.

4 Performance results

We base our experiments on the Sequoia 2000 Storage Benchmark ([10]) for the results presented in this section. We use the national dataset accumulated over 100 half-months. This is a four-dimensional dataset. The dimensions are *time*, *band*, *X*, and *Y*, where X and Y are the two dimensions from the raster image. We make the following assumptions about the native order of the dataset: all the raster images are chronologically sorted, since they were captured in that order. Raster images for a half-month are not sorted in any particular order. The raster images are created in row-major order (that is Y changes faster than X in the image). Two query types with distinct access patterns are used. Instances of *Query Type 1* select all images belonging to a band. Each instance accesses $\frac{1}{5}^{th}$ of the dataset. Instances of *Query Type 2* select all images belonging to a half-month. Each instance accesses $\frac{1}{100}^{th}$ of the dataset. Based on these query types, we experimented with two kinds of workloads. *Workload 1* consists of majority (90%) of type 1 queries. *Workload 2* consists of majority (90%) of type 2 queries.

The results presented here compare the performance of our algorithms (*Look-back-m* and *Forward Tuning*) with three other algorithms. The naive algorithm, which we refer to as *Unoptimized*, preserves the native order while storing data. The *band-date* algorithm sorts data items on the *date* dimension, and within the date dimension they are sorted on the band dimension. The *date-band* algorithms reverses the inner and outer sort dimensions of the band-date algorithm. We also present the *best case* times, calculated by storing data for each query at the start of the media using as much data replication as necessary. The best case times represent a lower bound on the workload execution time.

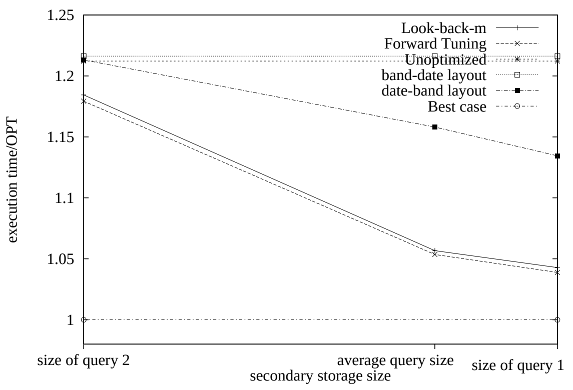
We use a simulator that uses the analytical model of Exabyte EXB-8505XL tape drive and EXB-210 tape library described in [2] and uses the *SORT* algorithm ([3]) for I/O scheduling during query execution. The time taken to execute the workload by an algorithm is plotted as a ratio of the actual execution time divided by *best case* time. We evaluate the performance of the algorithms by using temporary storage equal to the size of the instances of query type 1, query type 2, and the average query size of the workload.

Workload 1 Figure 1(a) shows that the performance of various algorithms is between 4 to 22 percent of *best case*. The wide range in performance is expected since the workload is dominated by larger queries that require major changes in the native order in order to be executed efficiently. The band-date scheme performs badly since it more or less preserves the native order; in fact its performance is worse than the *unoptimized* case. The date-band scheme improves its performance when given more temporary storage space. The date-band scheme favors query type 1, which dominates this workload. Hence, a closer transformation to the intended order (from native order) with increased amount of memory improves the performance. Our algorithms (*Look-back-*m and *Forward Tuning*) outperform other algorithms; moreover, they are within 6 percent of *best case* if the amount of memory available is more than average query size. The *Forward Tuning* algorithm does not provide any appreciable performance benefits over *Look-back-*m.

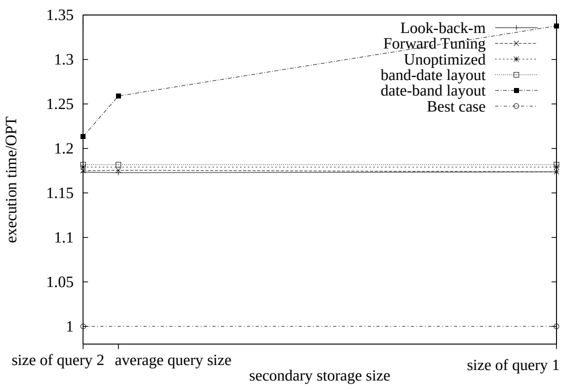
Workload 2 Workload 2 is dominated by query type 2, and the native order is favorable to it. The date-band scheme is not favorable to the majority of the queries; hence, we see a reduction in performance as the amount of available memory increases (increasing the closeness to a perfect date-band order) (figure 1(b)). Other algorithms perform similar to each other, the range of performance being around 18 percent of *best case*. The reason is that these algorithms tend to retain the native order; hence, the increase in the temporary storage size has no effect on the performance. None of the algorithms get close to *best case* because they do not take into account the size of the instance of the query types in the workload. A more sophisticated algorithm would have realized that 10% of the queries in the workload access approximately 66% of the total data accessed by all the queries in the workload and tried to optimize the storage layout for the *minority* query type (query type 1). Knowledge about the amount of data accessed by an instance of a query type requires advance information about the domain of each dimension. The algorithms presented in this paper were designed to work without such knowledge.

5 Conclusions

This paper investigates issues in efficient tertiary storage organization for multidimensional datasets. We show that the order in which the dataset is generated (or currently stored) affects the storage layout decisions due to limited amount of temporary storage available for data reorganization. We further show that efficient storage layout can be designed in this situation by considering data items that are accessed together rather than sorting the data items based on their coordinates. The experimental results have shown our techniques, based on heuristics, to be effective. They also reveal that when taking decisions about data



(a) Workload 1



(b) Workload 2

Figure 1: Performance Results

layout, one must consider the amount of data accessed by different queries in the workload besides the query characteristics and their execution frequencies.

References

[1] L. T. Chen, R. Drach, M. Keating, S. Louis, D. Rotem, and A. Shoshani. Efficient organization and access of multi-dimensional datasets on tertiary storage systems. *Information Systems*, 20(2):155–183, 1995.

[2] B. K. Hillyer, R. Rastogi, and A. Silberschatz. Scheduling and data replication to improve tape jukebox performance. In *Proceedings of the 15th International Conference on Data Engineering, Sydney, Austrialia*, pages 532–541, 1999.

[3] B. K. Hillyer and A. Silberschatz. Scheduling non-contiguous tape retrievals. In *Proceedings of Sixth NASA Goddard Conference on Mass Storage Systems and Technologies and Fifteenth IEEE Mass Storage Systems Symposium, University of Maryland, College Park, Maryland*, March, 1998.

[4] B. Inmon. The Role of Nearline Storage in the Data Warehouse: Extending your Growing Warehouse to Infinity. Technical white paper. StorageTek.

[5] H. V. Jagadish, L. V. S. Lakshmanan, and D. Srivastava. Snakes and sandwiches: Optimal clustering strategies for a data warehouse. In *Proceedings ACM SIGMOD International Conference on Management of Data, Philadephia, Pennsylvania, USA.*, pages 37–48. ACM Press, 1999.

[6] R. Kimball. *The Data Warehouse Toolkit*. John Wiley and Sons, Inc., 1996.

[7] S. More and A. Choudhary. Tertiary Storage Organization for Large Multidimensional Datasets. Technical Report CPDC-TR-9911-018, Center for Parallel and Distributed Computing, Northwestern University, November 1999.

[8] S. Sarawagi and M. Stonebraker. Efficient organization of large multidimensional arrays. In *Proceedings of the Tenth International Conference on Data Engineering, Houston, Texas, USA.*, pages 328–336. IEEE Computer Society, 1994.

[9] A. Shoshani, L. M. Bernardo, H. Nordberg, D. Rotem, , and A. Sim. Storage management for high energy physics applications. In *Computing in High Energy Physics*, 1998.

[10] M. Stonebraker and J. Dozier. SEQUOIA 2000: Large Capacity Object Servers to Support Global Change Research. Technical Report SEQUOIA 2000 Technical Report No. 1, Electronics Research Lab, March 1992.