Storage Optimization for Large Multidimensional Datasets

Sachin More and Alok Choudhary

Technical Report No. CPDC-TR-9911-018

November 4, 1999

Center for Parallel and Distributed Computing
Department of Electrical & Computer Engineering
Northwestern University
Technological Institute
2145 Sheridan Road, Evanston IL-60208

Abstract

Large multidimensional datasets are found in diverse application areas like data warehousing, satellite data processing and high energy physics. Due to the enormous size of the data to be stored, tertiary devices are the only cost-effective storage option. Due to the sequential access interface provided by these devices, storage pattern optimization becomes important to avoid high latency and attain streaming data bandwidth during query processing. This paper examines issues involved in determining efficient storage patterns for large multidimensional datasets in a database environment. We propose and evaluate a heuristic-based online algorithm for tertiary storage management under limited secondary storage assumption.

1 Introduction

Large multidimensional datasets are found in diverse application areas like data warehousing [10], satellite data processing and high energy physics [17]. 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 due to daily accumulation of raw data (commonly referred to as *data loading* operation in data warehouses) and aggregation data generated by jobs that process the raw data. Hence, estimates for the dataset sizes in near future run into several petabytes.

Though cost per byte as well as area per byte for secondary storage has been dropping for secondary storage, it is still not cost effective to store petabyte-sized datasets in the secondary storage. The two main reasons reasons are: tertiary storage is still very cheap compared to secondary storage and this assumption is expected to hold for next few years. Also, a single query usually accesses a very small subset of the data [8]. Hence a very high percentage of the data is *cold* data.

This paper investigates storage optimization strategies for large multidimensional datasets using a two-level storage strategy comprising of secondary and tertiary storage. We cast the problem as permutation of input data stream using limited storage space. The main contribution of this work is a heuristic-based online algorithm for efficient tertiary storage management in a database environment.

2 Related work

Various issues in tertiary storage management have been addressed by the database community. [1] evaluates issues in extending database technology for storing/accessing data on tertiary storage. [18] proposes a database architecture that uses hierarchical storage. [13, 14, 12, 16] examines issues in query processing when data resides on tertiary storage. Data striping on tertiary storage has been evaluated in [3, 4]. Tertiary storage space organization issues are addressed in [2, 5].

[2] discusses organization of multi-dimensional 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 data organization problem. In the first step, the dataset is split into disjoint subsets such that for any query type, the answer comes from a single subset. Next the subsets are broken into basic blocks, a query always access an entire basic block or none of it. Next, the ordering among the basic blocks is determined. The fourth step combines neighboring basic blocks to create *clusters*, where a cluster becomes a file in the hierarchical storage system (e.g., UniTree). Finally, the clusters are allocated to individual media based on the capacity of individual media.

[9] investigates 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 which determines how various dimensions are laid out. Further optimization is achieved by using a technique called *snaking*.

While finding an efficient storage pattern for multidimensional data has been investigated in literature ([15, 2, 9]), we are not aware of any work that evaluates whether such storage pattern is achievable in practice. Given an order in which data currently exists (or will be generated) and a limited amount of temporary storage space, we investigate issues in attaining an efficient storage pattern for multidimensional datasets.

3 Background

3.1 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 taking a subset domain of each dimension. A query is an instance of a *query type* [2]. A query type is a n-tuple whose values are drawn from the domain $\{ALL, ANY, VALUE, RANGE\}$. We make following assumptions about user queries:

- *Each query accesses a very small part of the dataset.*[1]
- *All queries are drawn from a set of known query types.*
- *Approximate execution probabilities of query types are known.*

[1] Aggregation queries used to produce summary information are notable exceptions. This work is aimed towards storage optimization for *ad-hoc* queries posed by end-users.

3.2 Data sources

The data items are generated by a data source. A data source could be physical devices like sensors, satellites etc. (in case of high energy physics experiments and satellite data), simulation programs or data loading programs (in case of data warehouses). 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*. In order to transform the native order into storage order, some amount *data permutation* needs to be done.

The transformation from native order to storage order can be trivially achieved if:

- *Unlimited number of passes over input data are allowed.*[2] Figure 1 shows an example of using an unlimited number of passes to compute a desired permutation of the input.. Note that no temporary storage space is needed and *any* native order can be transformed into *any* storage order if unlimited number of passes over input data are allowed.
- *Enough temporary storage space, up to the size of the dataset, is available.* See figure 2. The exact amount of temporary storage space needed for the transformation depend on how much the storage order differs from native order. In figure 2, temporary space that can hold at least nine data items to carry out the transformation. Given enough memory, *any* native order can be transformed into *any* storage order using exactly one pass over the input data.
- *A combination of both above* The number of passes needed to be made over the input data can be reduced if some amount of temporary storage space is available.

Making multiple passes over input data may be impossible (when the data source is an application) or very expensive (when the data source is another tertiary device). Hence we assume that only one pass is made over the input data. Further, we assume that the amount of temporary storage space available is order of the size of the answer sets generated by set of expected (ad-hoc) queries.

3.3 Storage model

The storage model (figure 3) 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 jukebox* containing a set of tape drives and a set of magnetic tapes. The secondary storage space is assumed to be random-access

[2]Multiple passes over output tapes is infeasible given the append-only nature of tape devices.

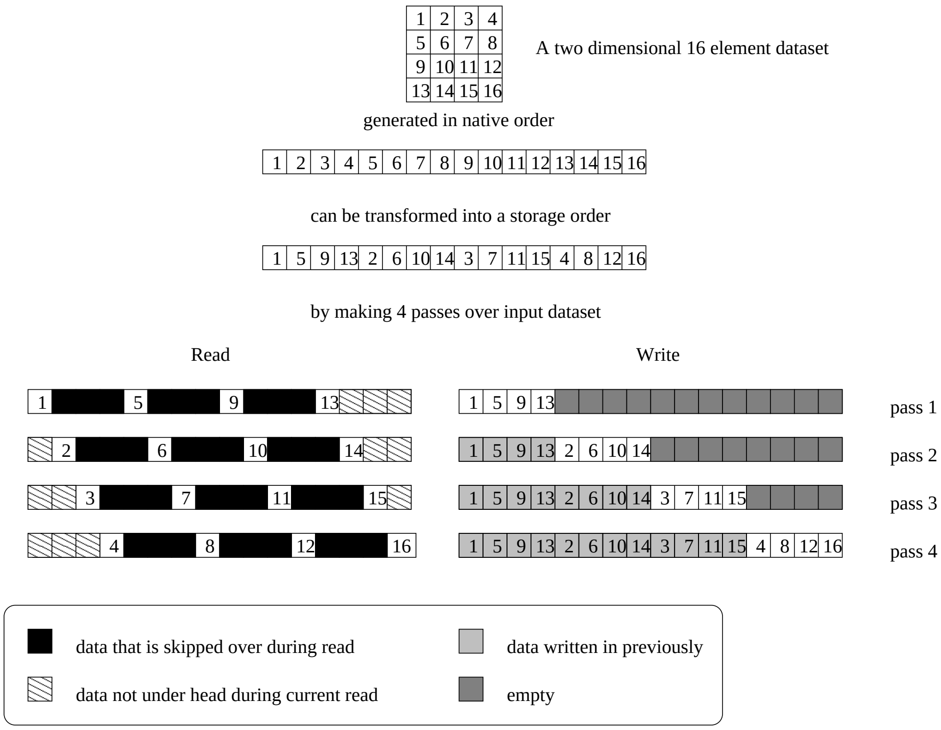

Figure 1: Illustration of data permutation using multiple passes over input data.

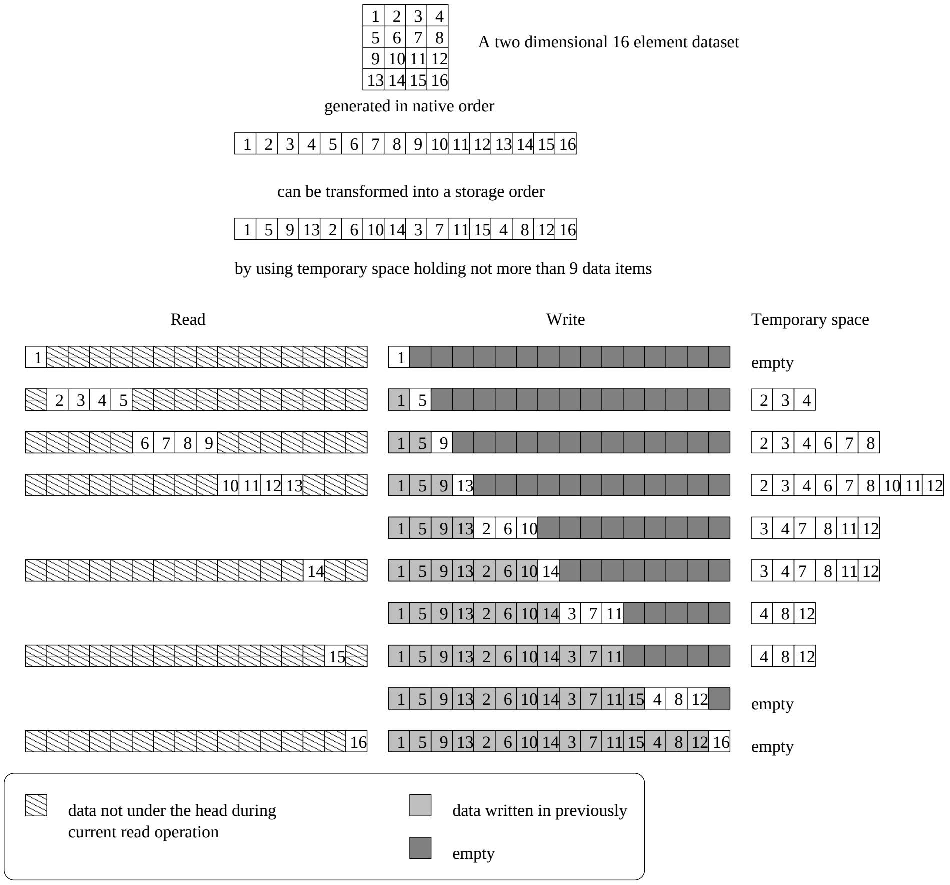

Figure 2: Illustration of data permutation using temporary space.

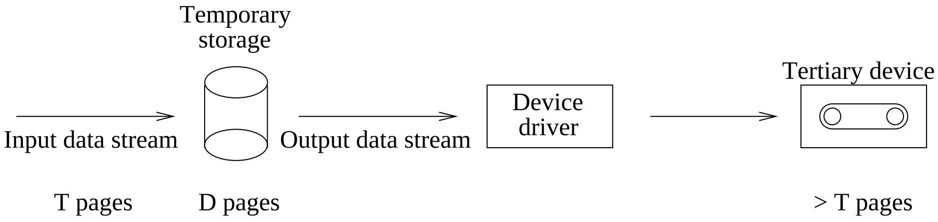

Figure 3: System model

storage, which is a reasonable assumption for the current technology. 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 are temporary storage that is used to carry out data permutation.

4 The data permutation problem

We formulate our problem as follows:

> Given N data items in their native order, $I_1, I_2, \ldots I_N$, each of size S pages and their affinities, find the most efficient storage order when only D pages of temporary space is available.

[2] proves that a simpler version of this problem, one without temporary space constraint, is NP-complete. Hence we use heuristics to design our algorithm and evaluate the heuristics experimentally.

Given native ordering of data items and expected query workload, we derive a storage pattern that optimizes the I/O time of the workload under the following constraints/assumptions:

- The input data is read exactly once in the native order.
- Amount of temporary storage space available is order of size of expected answer sets for the workload.
- Given two data items it is possible to compute the probability of one of them being accessed when the other one is accessed, that is, the expected workload is known.
- The output data items are stored in temporary storage before they are sent to tertiary storage device driver which groups consecutive data items into *physical blocks*[3] of data that are written out to the device.

Even though storage order defines an ordering on the 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

[3] The size of the physical blocks depends on the characteristics of the tertiary storage device.

the data items that are accessed together. Consider the two dimensional dataset shown in figure 4. If the expected workload accesses the dataset one row at a time with equal probability, all the four orderings shown are deemed equally good. Note that, permuting elements within a row (ordering 1), permuting the order of the rows (ordering 2) or both do not affect the effectiveness of the solution. Also, in ordering 3, the data items are not sorted on the domain values in any of the dimension. In fact, there are $24^5 = 7962624$ distinct storage orders for this dataset that are equally good for the above workload.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

A two dimensional 16 element dataset

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	Row-major ordering
2	1	3	4	5	7	6	8	10	11	12	9	16	15	14	13	Ordering 1
9	10	11	12	5	6	7	8	13	14	15	16	1	2	3	4	Ordering 2
10	11	12	9	5	7	6	8	16	15	14	13	2	1	3	4	Ordering 3

Figure 4: The two dimensional dataset shown at the top is always accessed single row at a time with equal probability. A row-major ordering (a complete ordering) on the dataset is not found qualitatively better than partial orderings 1, 2, 3.

This is an important observation, since success of transforming the native order into a storage order depends on amount of space available to carry out the transformation. And the above example shows that a lot of flexibility is available while determining a storage order. For example, consider the case where the dataset in the figure 4 is generated by a parallel application. Lets assume that the application is executing on four processors, where

processor	*generates*	*elements*		
1	1	2	5	6
2	3	4	7	8
3	9	10	13	14
4	11	12	15	16

with two rows worth of temporary space available for data permutation available. A row major ordering cannot be achieved if processors 1 and 2 cannot finish outputting before processors 3 and 4 start outputting. But equally good storage order can be achieved with a less stricter condition that there is no overlap between output from processor set $\{0, 1\}$ and $\{2, 3\}$.

This flexibility available in storage order is important. Consider the two dimensional dataset shown in figure 5. This dataset is always accessed single column at a time with all columns accessed with equal probability. The middle column shows two different native orderings in which the dataset is produced. Using a two element buffer, each can be transformed into an optimal storage, which is different for the two native orders. Hence, under a limited memory constraint, it is not possible to construct an optimal storage order without taking into consideration the native order.

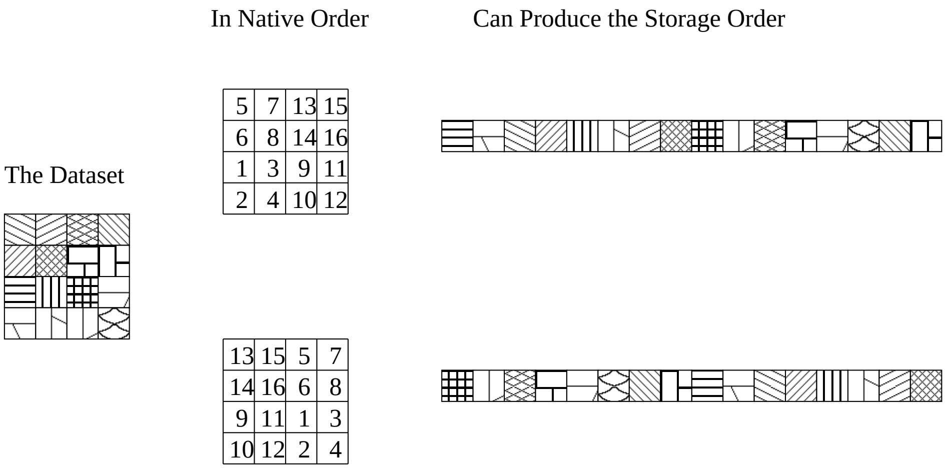

Figure 5: The two dimensional dataset shown at the the extreme left is always accessed single column at a time with equal probability. The middle column lists two different native orders. The right column shows storage order achieved with a two element temporary buffer for each native order. Note that the two storage orders produced (one for each native order) are not the same, can not be produced with a two element buffer using the other native order and still optimal for the workload under consideration.

From the above discussion it is evident that, the most crucial information about the dataset and workload under consideration is that any storage order that stores together data items whose column dimension match, is optimal. In general, data items need to be clustered based on their coordinate values in a subset of dimensions. This observation forms the basis of the clustering process in this paper.

4.1 Computing affinity between data items

In order to compute an efficient storage strategy, one must consider the relationship between data items with respect to the expected set of queries on the dataset. Given two data items in the dataset, I_i and I_j, and a set of expected query types and their execution probabilities, we define $p(I_i, I_j)$ as the *affinity* between I_i and I_j. Affinity between two data items is the probability that if I_i is accessed by a query then I_j is also accessed and can be compute as:

$$\sum_{Q_i=Q_1}^{Q_k}(p_i \times \prod_{D_j=D_1}^{D_n} \lambda_i^j)$$

where,

Q_i	i^{th} query type
p_i	execution probability of Q_i
D_j	j^{th} dimension
λ_i^j	$= 1$ when $Q_i(j) = ALL$
	$= 1$ when $Q_i(j) = ANY$ and $I_1(j) = I_2(j)$
	$= 1$ when $Q_i(j) = VALUE$ and $I_1(j) = I_2(j) = v_i^j$
	$= 1$ when $Q_i(j) = RANGE$ and `I_1(j)-I_2(j)` $\leq r_i^j$
	$= 0$ otherwise
$Q_i(j)$	selection criteria of Q_i in D_j
$I_1(j)$	coordinate of I_1 in D_j
$I_2(j)$	coordinate of I_2 in D_j
v_i^j	value parameter for Q_i from the domain of D_j
r_i^j	range parameter for Q_i from the domain of D_j

For each query type, the coordinates of the data items in each dimension are compared to compute the affinity. The final value of the affinity is computed by taking a weighted sum of the affinities for each query type. To reduce the computational complexity of the process, we compute affinities between *basic blocks*[4], proposed in [2], rather than individual data items when all data items in basic blocks are found contiguous in the native order (see section 5.3).

4.2 Heuristic approach

We start with a sketch of a generic algorithm for our problem:

```
while (there are more data items)
  if (there is space left in the temporary storage)
    input "some" data items
  endif
  if (there are data items in the temporary storage)
    output "some" data items
  endif
endwhile
```

The algorithm consists of a loop that executes till there more data items to be stored. At any point during the execution of the algorithm, the temporary storage space hold some data items that

[4]A basic block consists of data items where any query in the workload either accesses all the data items or none. Basic blocks do not replicate data items. Given two data items that belong to the basic block, affinity between them is always 1. Given two data items that belong to distinct basic blocks B_1 and B_2, affinity between them is always equal to affinity between any two data items chosen (one each) from B_1 and B_2.

have been input previously. Each iteration of the loop consists of an *input phase* and an *output phase*. During the input phase, the algorithm inputs data items (in native order) without overflowing the temporary storage. In the output phase, the algorithm outputs some data items (in the storage order) making room for more data items to be input in the next iteration.

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 of size S pages to input in each iteration?
2. If there are k data items ($k \times S \leq D$) in the temporary storage, how many and which of the k data items to output in each iteration?

4.2.1 Greedy heuristic

The greedy heuristic answers the question 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 greedy heuristic is as follows: It is important to keep the temporary storage filled to the capacity, since that gives a wider choice to the output phase in choosing data items to be output, which will result in better decisions. This means the input phase should read in as many data items as possible and output phase should output as few data items as possible. It also tries to maximize the affinity between consecutive data items while choosing the next data item to be output. The greedy heuristic results in a simple and low complexity algorithm.

4.2.2 Look-back-m heuristic

This is generalization of the greedy heuristic. The greedy heuristic looks at only one data item, the one that it just output, to make the decision which data item to output next. Look-back-m heuristic remembers the last m data items that were selected for output. This reduces the greediness of the algorithm and tries to improve the quality of the solution by looking at more amount of history. Contrast this with the original greedy heuristic that tries to improve the quality of the solution by looking at as many candidate data items as possible. 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 maximum backward affinity. We name this algorithms *Greedy 0*. Remembering last m data item has the side effect of reducing the space that holds incoming data items. If M pages are used to hold the previous m data items, then only $D - M$ pages are available to hold the incoming data items. This reduces the choice available to select the next data item to be output.

4.2.3 Forward tuning

During execution of the Look-back-m heuristic, multiple candidate data items are available to be output next. For example, when no data items are yet output, in the initial phase of the algorithm, all data items in the secondary storage have zero backward affinity. The algorithm randomly chooses one of such candidate data items. Forward tuning technique aims to improve performance by using data items that have *not* been output yet. For each data item in the temporary storage, it computes its affinity to m other data items in the temporary storage, called *forward affinity*. It outputs the element having maximum forward affinity.

5 Simulation results

5.1 Description of the Sequoia 2000 benchmark

We base our experiments on the Sequoia 2000 Storage Benchmark ([19]) for the results presented in this section. The benchmark provides three separate datasets, *regional*, *national*, and *world*. We use the national dataset over a period slightly less than four years (200 weeks) which is about 64GB in size. The spatial resolution of the dataset is $0.5km \times 0.5km$. The schema for the tables used in these queries is:

```
create RASTER( location = box,
time   = int4, band     = int4,
data   = int2[][])
```

time is a four byte integer and denotes the half-month over which the raster data was accumulated. The data set contains data for 200 weeks. The *location* attribute specifies the bounding box for the raster data and is represented by the coordinates of its top-left and right-bottom corners represented by four 4-byte integers. *band* specifies the wavelength band at which the data was captured; the values range over $1, 2, 3, 4, 5$. *data* is a two dimensional array of size 10240×6400. Each point is a two-byte integer.

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 such. 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).

5.2 Workload description

5.2.1 Queries

We use the following queries classes:

- **Query Class 1** *Select all images belonging to a band.* This is a *bad* query class for the native order. The data of interest is spread over the entire input data stream.
- **Query Class 2** *Select all images belonging to a half-month.* This is a *good* query class for the native order. The data of interest is localized in the input data stream.

We consider two different workloads.

- **Workload 1** consists of majority (90%) of "bad" queries for the native order.
- **Workload 2** consists of majority (90%) of "good" queries for the native order.

5.2.2 Workload generation

A workload is generated using the following parameters:

- *Query Instances* This denotes the total number of queries that this workload will consist of.
- *Query Classes* Describes the different query classes that will be used to create the queries that make up the workload.
- *Query Class Probability Estimates*

The query classes and their probability estimates have already been discussed earlier in this section. The number of query instances determines the accuracy of the workload generation process. If too few query instances are generated, it can result in mis-representation of query classes in the workload. We use the rule of thumb: For all query classes, if the number of distinct query instances that belong to a query class is n and the probability estimate of the query class is p, the workload should contain at least $\frac{n}{p}$ queries. This assures that the expected number of occurrences for any query instance is at least 1.

5.3 Identifying basic blocks

The basic blocks for the dataset and the query set, consist of individual images since query 1 as well as query 2, access the entire image. Since data for each image is found contiguous in the native order, we compute and use affinities between images (basic blocks, in other words).

5.4 Indexing scheme

We use a hashing based indexing scheme. For each data item, we use key attribute(s) to index into the hash table. Each entry in the hash table consists of a list of tuples of the form $\{media\_id, block\_number\}$. The list stores blocks on different media that contain data items whose key value hashes into the table entry. For query 1, we use *band* attribute for the key and for query 2, we use the *time* attribute.

Forward seek past k blocks	$4.834 + 0.378k$ seconds	$k \leq 28$
	$14.342 + 0.028k$ seconds	$k > 28$
Reverse seek past k blocks	$4.99 + 0.328k$ seconds	$k \leq 28$
	$13.74 + 0.0286k$ seconds	$k > 28$
Reverse seek to the start of the tape	21 seconds	extra
Reading k blocks after a forward seek	$0.38 + 1.77k$ seconds	
Reading k blocks after a reverse seek	$1.77k$ seconds	
Ejecting a tape	19 seconds	
Fetching a new tape from library	20 seconds	
Loading a tape	42 seconds	

Table 1: Analytical model for EXB-8505XL tape drive and EXB-210 tape library from [5].

5.5 Analytical tape model

Most of the literature ([14, 16, 12, 3]) uses a linear approximation of the locate time for tape drives. [6] found that such linear approximation is inaccurate. We use the analytical models of Exabyte's EXB-8505XL tape drive and EXB-210 tape library described in [5]. The model uses a logical blocks size of 1MB and is described in table 1. The simulator used for the experiments contains a single tape drive. We use the *SORT* algorithm ([7]) for I/O scheduling during query execution.

5.6 How is lower bound on the query execution time calculated?

We use a loose lower bound on the query execution time to compare various strategies. This bound is obtained by calculating the time the query is will take under the *best* possible storage layout. The *best* layout for a query is obtained by contiguously storing data needed by the query at the start of the media. If size of data is more than size of media, we assume that data is stored contiguously on a set of media with all but the last media full. For a workload, each query has its data stored on distinct set of tapes. A data item is replicated as many times as the number of query instance (and not query classes) that access it[5]. Query execution time under this scheme consists entirely of time to load/unload necessary media and reading the data at the streaming rate provided by the device.

5.7 Strict order strategies

We compare our heuristics with two strict order layouts. The layouts are chosen in such a way that they are beneficial for either query 1 or query 2. The *band-date* layout orders the data such that the data

[5]Note that our algorithms do not replicate data.

points are sorted on the date dimension and within the date dimension they are sorted on the band dimension. The *date-band* layout reverses the inner and outer sort dimensions of the band-date layout.

The band-date layout is beneficial to query 2 while date-band layout is beneficial to query 1. Since query 1 dominates workload 1, date-band layout is expected to perform better than band-date layout for workload 1. On the other hand band-date layout is expected to perform better than date-band layout for workload 2 since query 2 dominates that workload.

In the native order, the data is already sorted on the date dimension, hence band-date layout is very close to the native order (in fact, perfect band-date layout is achievable when the amount of temporary storage available is order of the size of the smallest query in the workload). Whereas, date-band layout is more difficult to achieve, given the limited amount of temporary space (calculations show that a perfect transformation needs temporary space of the order of the dataset size).

Converting native order to a strict order (e.g., date-band, band-date) can be thought of as sorting a stream of data items. Given two data items, it is possible to decide which data item is *earlier* than the other in the storage order. We use the algorithm described in [11], which creates initial (sorted) runs for merge sort under finite memory assumption. This creates multiple runs of data items. Data items in a run are in the perfect sorted order (date-band or band-date as the case may be). The overall quality of the solution depends on the amount of temporary storage available and how much the native order differs from the final storage order that is desired. [11] argues that the size of each run is on an average twice the size of the temporary storage under random input data. We found that if the storage order matches closely with the native order (e.g., band-date layout), we always get a perfect transformation under our assumptions about amount of temporary storage available. On the other hand, if the native is reverse-sorted with respect to the desired storage order (e.g., date-band layout), the quality of solution is very poor under our assumptions about amount of temporary storage available.

5.8 Performance evaluation

The results presented here compare the performance of six different algorithms:

1. *Unoptimized* The storage order is same as the native order. The amount of temporary storage available does not affect the performance.
2. *Best Case* This is a lower bound on execution time for the workload and forms the baseline against which all other algorithms are compared. See section 5.6 for description of how best case time is calculated.
3. *look-back-m* The value of m, amount of history, is set to 1. Our experiments revealed that bigger value of m does not better the

performance, and sometimes degrades the performance.

4. *Forward tuning*
5. *band-date*
6. *date-band* These are the strict order strategies described in section 5.7.

Time taken to execute the workload by an algorithm is plotted as a ratio of actual execution time divided by the best case time. We evaluate the performance of the algorithms under three different temporary storage scenarios:

- The size of the temporary storage is equal to the size of the instances of query class 1.
- The size of the temporary storage is equal to the size of the instances of query class 2.
- The size of the temporary storage is equal to the average query size of the workload.

5.8.1 Workload 1

The results of our experiments with workload 1 are presented in figure 6. The performance of various algorithms is between 4 to 22 percent of the best case. The wide range in performance is not surprising since the workload is dominated by queries that require major changes in the native order in order to be executed efficiently, which depends on how well an algorithm is able to carry out the transformation. The band-date scheme performs badly as expected 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 class 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 under all cases, moreover, they are within 6 percent of the optimal if amount of memory available is more than average query size. The *forward tuning* algorithm does not provide any appreciable performance benefits. Further, we found that forward tuning can actually degrade performance (see 7).

5.8.2 Workload 2

See figure 7. Workload 2 is dominated by query class 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 reduction in performance as amount of available memory increases (increasing the closeness to a perfect date-band order). The rest of the algorithms perform very similar to each other, the range of performance being around 18 percent of the best case. This is due to the fact that the native order

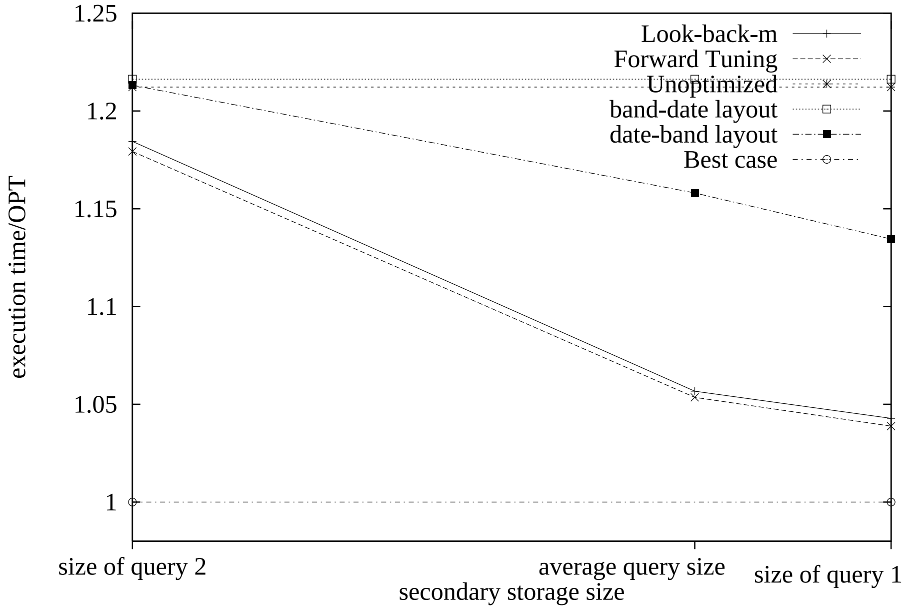

Figure 6: Performance Results : Workload 1

is already very close to the storage order needed for majority of the queries to perform well.

6 Conclusions

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

6.1 Acknowledgments

The authors thank Liddy Shriver, Rajeev Thakur and Sanjay Goil for their help in reviewing this manuscript.

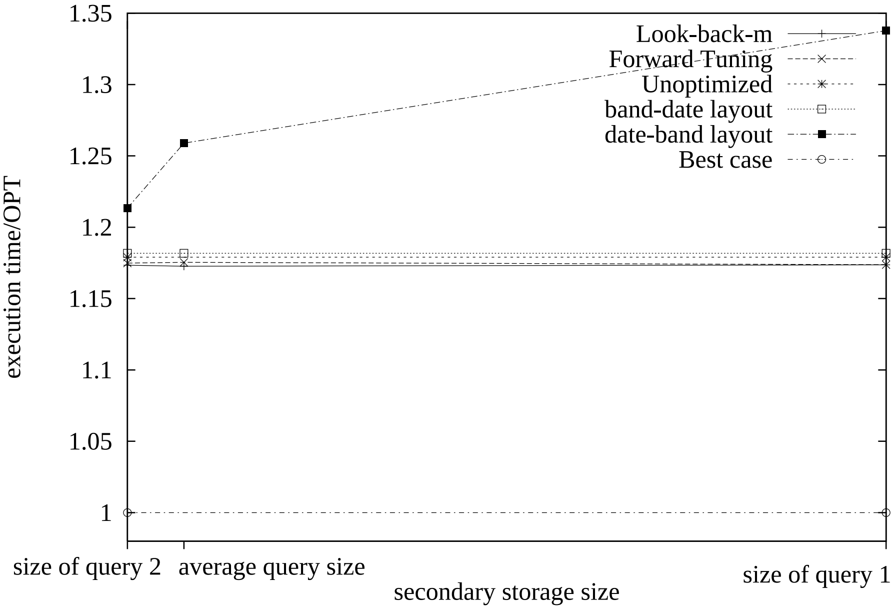

Figure 7: Performance Results : Workload 2

References

[1] M. J. Carey, L. M. Haas, and M. Livny. Tapes hold data, too: Challenges of tuples on tertiary storage. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C.*, pages 413–417. ACM Press, 1993.

[2] 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.

[3] A. L. Drapeau and R. H. Katz. Striping in large tape libraries. In *Proceedings of the 1995 ACM/IEEE Supercomputing Conference, San Diego, California, USA*. IEEE Computer Society Press, 1993.

[4] L. Golubchik, R. R. Muntz, and R. W. Watson. Analysis of striping techniques in robotic storage libraries. In *Proceedings of the Fourteenth IEEE Symposium on Mass Storage Systems, Monterey, California, USA*, pages 225–238. IEEE Computer Society Press, 1995.

[5] 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.

[6] B. K. Hillyer and A. Silberschatz. On the modeling and performance characteristics of a serpentine tape drive. In *Proceed-*

ings of 1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Philadelphia, Pennsylvania, U.S.A., pages 170–179, 1996.

[7] 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.

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

[9] 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.

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

[11] D. Knuth. *The Art of Computer Programming*, volume 3, chapter 5, pages 254–262. Addison-Wesley, Reading, Massachusetts, 1973.

[12] J. Myllymaki and M. Livny. Disk-tape joins: synchronizing disk and tape accesses. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Ottawa, Canada.*, pages 279–290, 1995.

[13] S. Sarawagi. Database systems for efficient access to tertiary memory. In *Proceedings of the Fourteenth IEEE Symposium on Mass Storage Systems, Monterey, California, USA.*, pages 120–126. IEEE Computer Society Press, 1995.

[14] S. Sarawagi. Query processing in tertiary memory databases. In *Proceedings of 21th International Conference on Very Large Data Bases, Zurich, Switzerland*, pages 585–596. Morgan Kaufmann, 1995.

[15] 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.

[16] S. Sarawagi and M. Stonebraker. Reordering query execution in tertiary memory databases. In *Proceedings of 22th International Conference on Very Large Data Bases, Mumbai (Bombay), India*, pages 156–167. Morgan Kaufmann, 1996.

[17] 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.

[18] M. Stonebraker. Managing persistent objects in a multi-level storage. In *Proceedings of the 1991 ACM SIGMOD International*

Conference on Management of Data, Denver, Colorado., pages 2–11. ACM Press, 1991.

[19] 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.