Paper Number 213

Caching Multidimensional Queries Using Chunks

Prasad M. Deshpande **Karthikeyan Ramasamy** **Amit Shukla**
Jeffrey F. Naughton

{*pmd, karthik, naughton, samit*}*@cs.wisc.edu*

Computer Sciences Department

University of Wisconsin, Madison, WI 53706

Abstract

Caching has been proposed (and implemented) by OLAP systems in order to reduce response times for multidimensional queries. Previous work on such caching has considered table level caching and query level caching. Table level caching is suitable for static schemes. On the other hand, Query level caching can be used in dynamic schemes, but is too coarse for "large" query results. Query level caching has the further drawback for small query results in that it is only effective when a new query is subsumed by a cached previous query. In this paper, we propose caching small regions of the multidimensional space called "chunks". Chunk-based caching allows fine granularity caching, and also allows queries to partially reuse the results of previous queries with which they overlap. To facilitate the computation of chunks required by a query but not found in the cache, we propose a new organization for relational tables, which we call a "chunked file." Our experiments show that for workloads that exhibit query locality, chunked caching combined with the chunked file organization performs better than query level caching. An unexpected benefit of the chunked file organization is that, due to its multidimensional clustering properties, it can significantly improve the performance of queries that "miss" the cache entirely as compared to traditional file organizations.

1 Introduction

OLAP systems are becoming increasingly significant in modern day business in order to increase competitiveness. A typical characteristic of data sets in these systems is their multidimensional nature. However, traditional relational systems are not designed to provide the necessary performance for these types of data. Hence such systems are built by using a three tier architecture. The first tier provides an easy to use graphical tool that allows

the user to build requests. The middle tier provides a multidimensional view of the data stored in the final tier, which may be an RDBMS.

Queries that occur in OLAP systems are interactive and demand quick response time in spite of being complex. The set of queries that commonly occur include placing restrictions on dimension tables that translate into restrictions on the fact table followed by an aggregation. Various techniques can be used at different stages of the life-time of the query to speed up its execution. Precomputation and the use of specialized indexing structures have been predominantly used at the RDBMS to speed up such queries. In this paper, we propose caching query results in the middle tier as a feasible approach that complements the other strategies.

As a motivating example, consider a database containing sales data describing the dollar sales of products sold in a given region on a given date. Thus the schema could be described as:

$Product = (\ pname,\ pcategory,\ pprice,\ pid\ )$
$Store = (\ sname,\ scity,\ sstate,\ sid\ )$
$Date = (\ dmonth,\ dday,\ dyear,\ did\ )$
$Sales = (\ pid,\ sid,\ did,\ dollar_sales\ )$

Consider the following query that asks for the monthly sales of a given product category for the first two quarters.

```
SELECT pname, dmonth, sum(dollar_sales)
FROM Sales, Date                                                                (Q.1)
WHERE pcategory = "clothes" AND dmonth ≥ "Jan" AND dmonth ≤ "June" AND Sales.did = Date.did
GROUP BY pname, dmonth
```

Let us assume that the results of Q1 are cached. Now consider two possible queries Q2 and Q3 that are asked after the above query. Q2 asks for aggregate sales for the months between January and May while Q3 asks for aggregate sales for the months between April and September. The queries may be issued from multiple query streams originating from multiple users. The SQL statements for these queries are

```
SELECT pname, dmonth, sum(dollar_sales)
FROM Sales, Date                                                                (Q.2)
WHERE pname = "blaire_cotton_shirts" AND dmonth ≥ "Jan" AND dmonth ≤ "May" AND Sales.did = Date.did
GROUP BY pname, dmonth
```

```
SELECT pname, dmonth, sum(dollar_sales)
FROM Sales, Date                                                                (Q.3)
WHERE pname = "blaire_cotton_shirts" AND dmonth ≥ "April" AND dmonth ≤ "Sept" AND Sales.did = Date.did
GROUP BY pname, dmonth
```

To answer Q2 and Q3, the query evaluator is presented with two alternatives. The first alternative is to evaluate the queries using the results of the cached query while the second alternative is to issue them to the relational backend. The backend evaluates the query using base tables, indexes or materialized views or a combination of them.

The traditional approach to caching has been query level caching which caches entire query results. It uses query containment to determine whether a given query can be answered using the contents of the cache. For example, Q2 can be evaluated entirely using the cache since it is properly contained in Q1. However, if we use query containment, Q3 cannot take advantage of the cached results despite the fact that some of the partial results being in the cache, i.e, the sales for the months of April, May and June. Query caching has the further drawback of requiring redundant storage for queries with overlapping results, thereby reducing the effective size of the cache.

To circumvent the drawbacks of query level caching, we need a mechanism to determine whether partial results for the query can be answered using the cache. Additionally, we need to decompose the query so that one portion of it is evaluated entirely in the cache while the other is issued to the backend. A naive method of determining whether partial results of a query are in the cache requires that the incoming query be intersected with all cached queries. However, the cost of such an intersection operation is linear in the number of queries cached and becomes expensive as the cache size is increased and more queries are cached. This is typical because of the large amounts of memory available in contemporary systems.

In this paper, we propose a chunk based scheme that addresses these problems by dividing the multidimensional query space uniformly into chunks and caching these chunks. The results of a query are contained in an integral number of chunks, which form a "bounding envelop" about the query result. (If the query boundaries do not match the chunk boundaries, the chunks in this bounding envelop may contain tuples that are not in the answer to the query. We discuss this issue further later in the paper.) Since chunks are at a lower granularity than query level caching, we can reuse them to compute the partial result of an incoming query. By using a fast mapping between query constants and chunk numbers, we can determine the set of chunks needed for completely answering a query. We partition this set of chunks into two disjoint sets such that the chunks in one partition can be found in the cache while the others have to be computed using the backend relational engine. A consequence of partitioning the chunks is the intended query decomposition. Since query results are an integral number of chunks, the replacement policy can take advantage of the "hotness" of the chunk unlike query level caching.

The paper is structured as follows: in Section 2, we describe the OLAP data model and its caching aspects. We explain the concept of chunks and introduce the idea of chunk based caching in Section 3. We describe the chunked file format in Section 4. In Section 5, we give a detailed description of the issues involved in

implementing this caching scheme. Finally, we present our performance results in Section 6 and conclude in Section 7.

2 OLAP Data Model and Caching

2.1 OLAP Data Model and Queries

OLAP data sets are inherently multidimensional. Consider the database schema presented earlier that describes the *dollar sales* of particular product sold in particular store on a particular date. The attributes *product*, *store* and *date* functionally determine the *dollar sales*. These attributes are referred to as *dimensions* while the *dollar sales* is referred to as a *measure*. A dimension may have some attributes to describe its members. Each dimension is typically arranged in a *hierarchy*. For example, the *store* could rollup into *counties*, *counties* into *states* and *states* into *countries*. In addition, members of the hierarchy might have their own attributes. For example, the product dimension might have an attribute *price* that is the sale price of it and *category* that describes the category it belongs to.

In a relational system, such a multidimensional schema is mapped onto a set of tables. Quantitatively, if there are *n* dimensions, the number of tables required for mapping is *n+1*. Each dimension maps into a separate table called a *dimension table*. This table contains the information related with hierarchies on the dimension and other descriptional attributes. In addition, there is a separate table called *fact table* that relates the *n* dimensions with some measures. The relationship with the dimensions is accomplished by storing foreign keys for each of the dimension tables. This kind of organization is called a *star schema*. In the above example, the dimensions *product*, *store* and *date* are individually mapped into tables of the same name. The fact table *Sales* contains the foreign keys *pid*, *sid* and *did* followed by the measure *dollar_sales*.

OLAP queries typically involve a selection based on some dimension values followed by a group-by on a set of dimensions. This involves a join of the *fact* table with one or more *dimension* tables followed by a group-by operation. Since *fact* table is usually much larger than the *dimension* tables, OLAP systems try to optimize scans of the *fact* table by using specialized indexing such as an bitmap index or a BTree. Aggregation is an expensive operation since a substantial part of the *fact* table may have to be accessed to get the aggregated data. In summary, caching aggregated results is important for improved perfomance in OLAP systems.

2.2 Locality in OLAP Queries

OLAP queries are typically repetitive and follow a predictable pattern. Consider an user analyzing the sales data for Wisconsin. In a typical session, the user might look at the sales data at the $(Product, City)$ level for one of the cities say *Madison*. He might find that the sales are below expectations and drill down on the *City* to get a detailed break down of individual stores in *Madison* which is at $(Product, Store)$ level. The drill down could be followed by a roll up operation back to the $(Product, City)$ level, after which the user could move on to another city say *Milwuakee*. Thus, a typical user session can be characterized using the different kinds of locality.

1. *Temporal* - The same data might be accessed repeatedly by the same user or a different user.

2. *Hierarchical* - This kind of locality is specific to the OLAP domain and is a consequence of the presence of hierarchies on the dimensions. Data members which are related by the parent/child or sibling relationships will be accessed together. For example, if an analyst is looking at data for Wisconsin, his next query is likely to be about cities in Wisconsin or about Illinois or Midwest Region.

2.3 Taxonomy of Caching Schemes

Caching schemes reported in the literature can be classified based on the granularity of unit used for caching and the nature of caching.

1. *Nature of caching* - Under this category, the caching schemes can be classified as static or dynamic. In static caching, a set of group-bys is chosen and the corresponding tables are materialized. Thus it reduces to the problem of selecting what aggregates to precompute [HRU96] [GHRU97] [SDN] using a given amount of space. The choice of these tables are made a priori and is independent of the actual query stream. On the contrary, dynamic caching schemes adapt depending on the type of query stream. A static approach is specifically suited for the backend where we precompute the aggregated tables as views. A dynamic approach will be significantly beneficial at the middle tier. In this paper, we focus on dynamic caching and issues related to it.

2. *Unit of caching* - In caching, the units that are cached significantly influence the performance. Based on the units of caching, we can classify the caching schemes: *Query level caching* caches the entire results of queries whereas *table level caching* caches entire tables. The smaller size of the unit of caching, the better the reuse. But, the overhead of managing the smaller units during replacement added with the overhead of checking whether the incoming query can be answered from them becomes expensive. We propose *chunks* as a unit of caching and demonstrate its feasibility under realistic query workloads without much overhead.

2.4 Related Work

Extensive work has been reported in the literature especially in caching and reusing of query results. Some of them have significant applicability to the domain of data warehousing and OLAP. Cache replacement and admission schemes specific to warehousing have been studied in [SSV]. In their work, a profit metric is defined for each query and on the basis of it a decision is taken whether the query has to be cached or can be replaced. The idea of using a profit metric is relevant in OLAP since, highly aggregated results are expensive to compute and should be given preference while caching as well as replacing. The problem of answering queries with aggregation using views has been studied extensively in [SDJL96]. They describe the conditions under which a new query can be answered using cached aggregate views. A query need not be answered from a single view, but it could be decomposed into multiple queries each of which is answered from a different view. However, their approach is limited by query containment, since the new query has to be computable from the set of available views. Our work is significantly different from the rest of them since we focus on how to effectively reuse the query results especially when new query results partially intersect with existing results in the cache.

We have a semantic approach to caching. Semantic query caching for client-server systems has been studied in [DFJST]. The semantic approach is very suitable to the OLAP domain where the data is multi-dimensional and the notion of semantic data regions is very natural. One of the drawbacks of semantic caching is that, the cache has to maintain information about the cached semantic regions. When a new query arrives, the algorithm has to consider its intersection with all the cached regions. The query will be split into many parts depending on how many semantic regions it intersects with. We try to avoid this overhead by using the notion of chunks, which act as uniform semantic regions. Another important contribution of our proposal is that it reduces the cost of a cache miss by using a chunked-file representation of the data at the backend. The chunk scheme maintains a very simple correspondence between semantic regions at different levels of aggregation. To compute some missing data, the backend has to process only the semantic region corresponding to the missing data rather than scanning the entire table. This combination of uniform semantic regions and the corresponding chunked representation at the backend improves the overall performance of the cache. Our solution can be easily adapted to the case where we have precomputed aggregate tables at the backend. These tables will also be stored in a chunked format. An interesting benefit of the chunked-file representation is that it improves the performance of indices like bitmap index. This speeds up general star join processing.

3 Chunk Based Caching

The idea of chunks is motivated by *MOLAP* systems which use multi-dimensional arrays to represent the data. Instead of storing a large array in simple row major or column major order, they are broken down into chunks

and stored in a chunked format [SS94] [ZDN97]. The distinct values for each dimension are divided into ranges and the chunks are created based on this division. Figure 1 shows how the multidimensional space can be broken up into chunks. Our observation is that chunks could be very suitable as a unit of caching. Chunks capture the notion of semantic regions. In fact, chunks divide the entire space into uniform semantic regions.

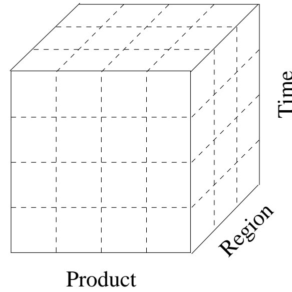

Figure 1: Chunking the Multidimensional Space.

3.1 Caching Query Results Using Chunks

In the chunk-based caching scheme, query results to be stored in the cache are broken up into chunks and the chunks are cached. When a new query is issued, chunks needed to answer that query are determined. This process is described in Section 5. Depending on the contents of the cache, the list of chunks is partitioned into two. One part is answered from the cache. The other part consists of the missing chunks which have to be computed from the backend.

It is important to reduce the cost of a chunk miss. This means that missing chunks should be computed efficiently at the backend. To achieve this, we propose a chunk based organization of data in the backend. It could be implemented either as multi-dimensional array if the database supports multidimensional array or, if the system is fully relational, a chunked file organization. The chunked file is described in Section 4. Even statically precomputed aggregate tables can be organized on a chunk basis. With this organization, computing a missing chunk is efficient. To compute any chunk, we aggregate the corresponding chunks of its parent relation according to the group-by hierarchy. Thus, only a few chunks of the parent are scanned rather than scanning the entire table. For example, in Figure 3, to compute chunk 1 of the $(Time)$ group-by, we only need scan chunks 4, 5, 6 and 7 of the $(Product, Time)$ group-by.

3.2 Benefits of Chunk Based Caching

There are many advantages to using chunks:

1. *Granularity of caching* - caching chunks rather than entire queries improves the granularity of caching. This leads to better utilization of the cache in two ways. First, frequently accessed chunks of a query get cached. The chunks which are not so useful will ultimately get replaced. The second major advantage is that previous queries can be used much more effectively. For example, Figure 2 shows a chunk based cache. Each query represents a portion of the multidimensional space. Q3 is not contained in either Q1 or Q2 or their union. Thus, methods based on query containment will not be able to use Q1 and Q2 to answer Q3. With chunk based caching, Q3 can use chunks it has in common with Q1 or Q2. Only the remaining chunks, shown by shaded portion, have to be computed. The "chunked file" organization in the relational back end, discussed in Section 4, enables these remaining chunks to be computed in time proportional to their size (rather than in time proportional to the size of Q3.)

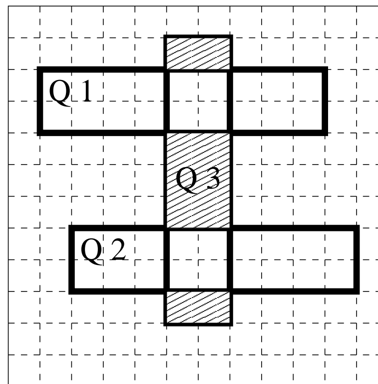

Figure 2: Reusing Cached Chunks.

2. *Uniformity* - The notion of uniform semantic regions in the form of chunks makes query reuse less complex. Unlike caching methods based on containment, we don't have to determine which of the cached queries should be combined to answer a new query. Also, its not necessary to check intersection with all cached regions, as is necessary in simple semantic based schemes.

3. *Closure property of chunks* - Chunking can be applied at any level of aggregation. The uniformity gives a very simple correspondence between chunks at different levels of aggregation. This defines a closure property on chunks which states that we can combine chunks at one level of aggregation to obtain chunks at a different level of aggregation. For example, consider Figure 3. It shows data at two levels of aggregation: - $(Product, Time)$ and $(Time)$. Chunking is applied at both levels. Thus chunk 0 of $(Time)$ corresponds to chunks $(0, 1, 2, 3)$ of $(Product, Time)$. This means that chunk 0 of $(Time)$ can be obtained by aggregating chunks $(0, 1, 2, 3)$ of $(Product, Time)$. This is a very useful property which we use for query caching.

   The closure property reduces the *cost of a chunk miss* in the cache. Using the simple correspondence between chunks at different levels, we know exactly which chunks should be aggregated to obtain the missing chunks. A chunk based organization at the backend further reduces the cost of a cache miss.

12	13	14	15
8	9	10	11
4	5	6	7
0	1	2	3

Product

Time

3
2
1
0

Time

Figure 3: Chunks at different levels.

4. *No redundant storage* - If query level caching is used, each query is cached in its entirety even though some queries will have overlapped results with other queries. Replication of such partial results reduce the effective memory available for caching. Hence, the hit ratio of the cache is adversely affected reducing the benefit of having a large cache. Using chunk based caching eliminates this replication by sharing chunks containing overlapping results. Such sharing permits more queries to be cached leading to better performance.

3.3 Ordering the Dimensions

Having considered the benefits of chunking, let us consider the implementation in more detail. To divide the multi-dimensional space into chunks, distinct values along each dimension have to be divided into ranges. However, some of the dimensions may be nonnumeric and may not have any implicit order. It is necessary to impose an order on the distinct values of a dimension. The ordering is very important since data gets clustered according to the order chosen. Hierarchical locality in OLAP queries suggests that we should use the hierarchy on the dimension to order distinct values. Thus, distinct values get clustered together according to the hierarchy. For example, for the *store* dimension, the ordering should be such that all cities in a county are nearer, the counties in a state are nearer and the states in a country are nearer. Figure 4 shows a simple hierarchy and the ordering at each level of the hierarchy. A mapping structure called *Domain Index* is used to to maintain this correspondence between a dimension value and its ordinal number.

If there are multiple hierarchies on some dimension, it may not be possible for a ordering to satisfy all the hierarchies. In general, a single order will work if the multiple hierarchies are ordered identically at all the common levels. For example, $Time$ has two hierarchies : $Day - Week - Year$ and $Day - Quarter - Year$, but at the common base level of Day, they still have a common notion of ordering, according to the date.

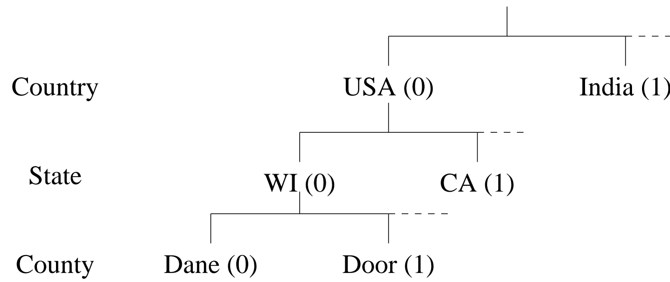

Figure 4: Ordering distinct values on a dimension

3.4 Chunk Ranges

Once the distinct values in each level of a dimension are ordered, we have to divide them into chunk ranges. The chunk ranges identify boundaries of chunk regions. Having the correct chunk boundaries is very important in order to have a mapping between chunks at different levels of the hierarchy. If the dimensions do not have any hierarchies defined on them, then we can divide the entire dimension range into uniform chunk ranges. However, this does not work when the dimensions have hierarchies defined on them. The actual structure of the hierarchy should be considered while defining the chunk ranges. For example, Figure 5 illustrates a problem that arises if the actual structure of the hierarchy is ignored. In Figure 5 dimension A has a three level hierarchy defined on it. In this case, values at level 3 are divided into uniform ranges of size 3, whereas levels 1 and 2 have been divided into ranges of size 2. Consider range $R_{3,1}$. This range rolls up into 2 ranges at level 2, i.e. $R_{2,0}$ and $R_{2,1}$. This means that a chunk which has a range $R_{2,0}$ cannot be computed from the chunks with ranges $R_{3,0}$ and $R_{3,1}$ directly since $R_{3,1}$ has some dimension values that don't map to range $R_{2,0}$.

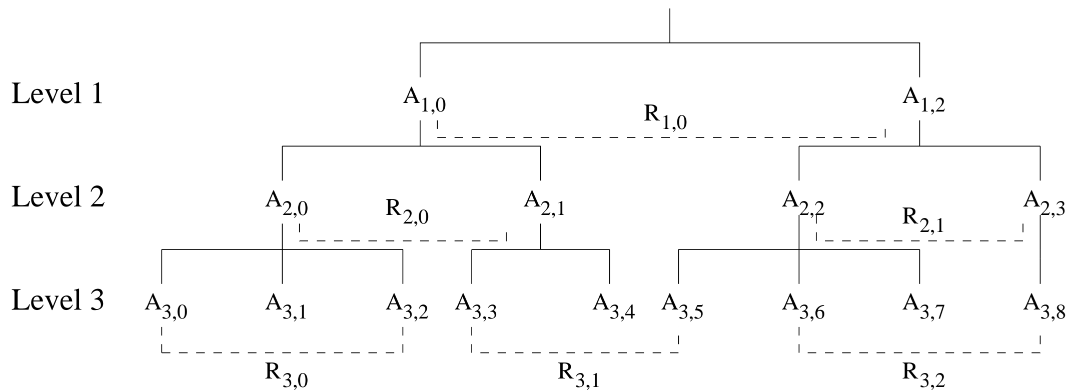

Figure 5: Uniform Ranges

For our chunk based scheme, chunk ranges at one level should map to disjoint sets of ranges at a lower level. The following algorithm creates the chunk ranges for a dimension :

Algorithm : CreateChunkRanges

hiersize = size of the hierarchy on the dimension

Divide level 1 into uniform ranges

For (l = 1 to hiersize - 1)

  For each chunk range at level l

    Let R = range of values it maps to at level l + 1

    Divide range R into uniform ranges

Figure 6 shows the chunk ranges obtained by using the above algorithm. Level 3 has a desired chunk range of 3 whereas levels 1 and 2 have a desired chunk range of 2. The desired chunk range may not match the actual chunk range due to the hierarchy.

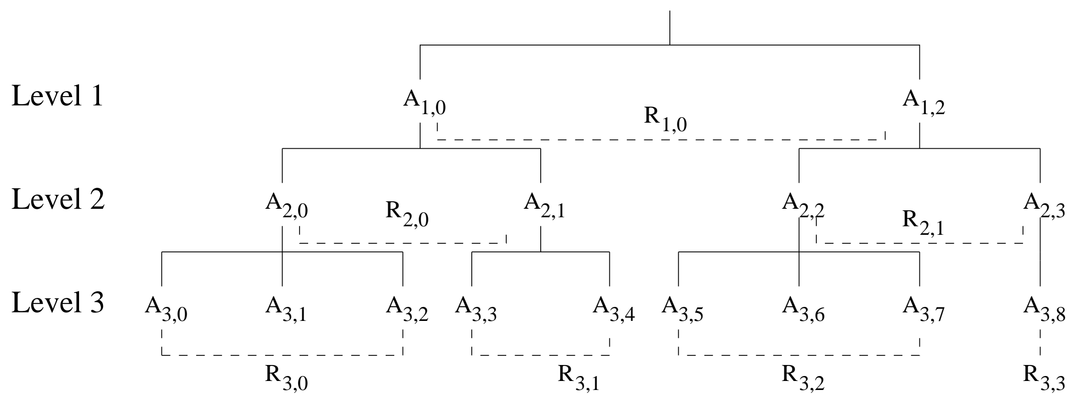

Figure 6: Ranges according to hierarchy

4 Chunked File Organization

One way to organize data on chunk-basis at the backend is to use chunked multi-dimensional arrays. However, it requires significant effort to incorporate a multi-dimensional array in a relational system. Also, the relational interface to the data may be lost. But this concept of chunking need not be restricted to arrays alone. It can be applied to relational tables as well.

In a chunked-file organization, the data is still stored as tuples. But they are rearranged on a chunk basis. Thus all tuples in a chunk get clustered together in the file. A *chunk index* is created so that given a chunk number it is possible to access all tuples corresponding to that chunk. The chunked file provides two interfaces. It has a relational interface, so that it can be considered as a regular table and can be used in SQL statements. It also

has a chunk based interface that allows access to an individual chunk directly. This satisfies the requirements of caching. To compute a missing chunk, we use the *chunk index* to access the corresponding chunks and aggregate them. The chunked file can be implemented in an existing relational system with very less effort. The options are described in Section 5.

4.1 Benefits of Chunked File Organization

The chunked file organization has the following benefits:

1. *Reduce cost of a chunk miss* - cost of accessing a chunk is proportional to the size of the chunk rather than the entire table.

2. *Multi-dimensional clustering* - chunked organization achieves very good multi-dimensional clustering for relational tables. This is very useful for OLAP queries, which access data with hierarchical locality. For example, selection queries which use bitmaps can be speeded up (Section 4.2).

4.2 Improving Bitmap Performance

Although we developed the chunked file organization primarily so that missing chunks could be computed efficiently, a nice side effect of this file organization is that due to its clustering properties it improves the performance of star join queries as compared to unordered files. This is independent of the value of chunked files for supporting chunked caching. In this section we explain this benefit and give a simplified analysis that provides intuition as to why this works.

A star join query involves selection based on some dimension values followed by aggregation. Bitmap indices are often used to speed up the selection. The bitmaps corresponding to the different dimension values are ANDed or ORed depending on the selection condition. The resultant bitmap is used to pull out tuples from the *fact* table.

When the query selectivity is high, only a few bits in the result bitmap are set. If there is no particular order among the *fact table* tuples, we can expect each bit to access a tuple in different page. Thus there will be as many I/Os as there are bits set. The clustering achieved by the chunked-file organization reduces the number of pages accessed, thus improving the bitmap performance. For example, consider a *fact* table with two dimensions A and B, as shown in Figure 7. If there is a selection condition $A = x$, we can see that tuples satisfying this condition have been clustered together thus causing fewer I/Os.

A simple analysis of a greatly simplified scenario gives intuition as to why this works:

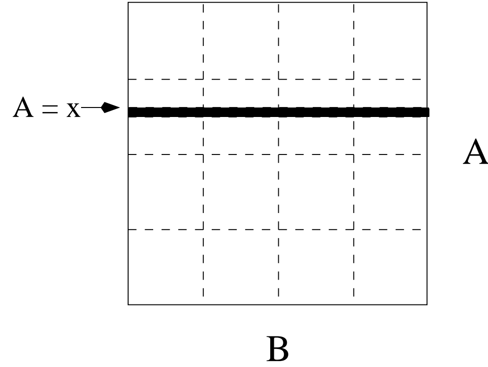

Figure 7: Clustering helps bitmaps

A, B - dimensions

D - number of distinct values of A and B

N - number of tuples

T - number of tuples per page

P - number of pages

C - number of chunks

d - density of data

Note that we are assuming that both dimensions have the same number of distinct values. Suppose query is a selection on A, i.e. $A = x$. The expected number of tuples satisfying this condition is $n = \frac{N}{D}$. Thus number of bits set in the bitmap is n. We use the following result from probability theory [Fell57] :

If r elements are chosen uniformly randomly from a set of k elements, the expected number of distinct elements obtained in the sample is $f(r,k) = k - k(1-1/k)^r$.

$f(r,k)$ has the following properties:

$$
\begin{aligned}
f(r,k) &\leq r, k \\
f(r,k) &\approx r, if r \ll k \\
f(r,k) &\approx k, if r \gg k
\end{aligned}
$$

For a randomly ordered file, tuples corresponding to the set bits may be distributed over all the P pages. Thus expected number of pages accessed is $p = f(n, P)$. Now, consider the file organized on chunk basis. Assuming, a average chunk size of one page, number of chunks $C = P$. Thus, number of chunk ranges on both dimensions A and B is $\sqrt{P}$. From the figure we can see that the tuples corresponding to the set bits can only be from these $\sqrt{P}$ chunks. Thus, expected number of pages accessed is $p_c = f(n, \sqrt{P})$. From the properties of f, it follows that $p_c \leq p$. Thus less pages are accessed for a chunked file. We also have $d \times D \times D = N$, since d is the density of data.

$$
\begin{aligned}
D &= \sqrt{\frac{N}{d}} \\
n &= \sqrt{N \times d} \\
N &= P \times T \\
n &= \sqrt{P \times T \times d}
\end{aligned}
$$

Thus we have,

$$
\begin{aligned}
p &= f(\sqrt{P \times T \times d}, P) \\
p_c &= f(\sqrt{P \times T \times d}, \sqrt{P})
\end{aligned}
$$

Thus the actual improvement depends on the density of data. If $1 << T \times d << \sqrt{P}$, then $p \approx \sqrt{P \times T \times d}$ and $p_c \approx \sqrt{P}$, thus showing an improvement. The improvement will be more if there is a range of values selected on A, i.e. $x \leq A \leq (x+k)$, since tuples will map to the same chunks. Thus they are likely to cause the same number of I/Os for chunked file, but more for a randomly ordered file.

We ran some experiments to verify this effect in a more general situation; the results were surprisingly good and are described in Section 6. In future work we will also explore other methods to improve query performance using chunked files.

5 Implementation

There are some important issues that come up in the implementation of the chunked-file and the chunked based caching scheme. We will discuss them in this section.

5.1 Chunk Size

To determine the chunking, distinct values along each dimension are broken up into ranges. The size of this range affects how many chunks there are and also the size of chunks. For example, consider two dimensions A and B with 100 distinct values each. Let the database size be N tuples. If we use a chunk range of 5 along each dimension, the number of chunks becomes 400 and the average chunk size is $\frac{N}{400}$ tuples. But if the chunk range is of size 10, we get 100 chunks with each chunk of size $\frac{N}{100}$.

Having smaller chunk ranges is good for better granularity of caching. With the chunk based caching scheme, a query is computed in terms of its chunks. The chunks at the boundary of the query region have some extra

tuples and cause extra computation. If these extra tuples are not reused later then the effort of computing them is wasted. This is particularly true for highly aggregated queries, since each tuple of the result is an aggregation of a large number of base tuples and is expensive to compute. Thus, having small chunk ranges is important to reduce the extra computation. However, if the chunk ranges are too small, then the total number of chunks increases. This, again increases the overhead since the query gets split into many chunks and also the size of the chunk index at the backend increases. This suggests that the chunk range at any level in the hierarchy should be a proportional to the number of distinct values of the dimension at that level. This agrees with a similar observation made in [SS94], in the context of multi-dimensional arrays. Section 6 describes some experiments to determine the optimal chunk range.

5.2 Query Processing

5.2.1 Query Analysis

When a new query is issued, it is necessary to check if it can be answered from the cache. Also, the selection predicates have to be analyzed to compute a list of chunk numbers which can answer this query. We will assume a star join template for queries, i.e. each query is a join of the *Fact* table with the dimension tables filtered by some selections and followed by aggregation. The template looks like:

`SELECT` ⟨ proj-list ⟩ ⟨ aggregate-list ⟩
`FROM` ⟨ FactName ⟩ ⟨ dimension-list ⟩
`WHERE` ⟨ select-list ⟩
`GROUP BY` ⟨ dimension-list ⟩

For the cached result chunks to be reused, following conditions have to be satisfied:

1. *Level of Aggregation* - Cached results can be either at the same or lower level of aggregation than the query. However, in our implementation we restrict all aggregations to the backend. We reuse the cached results only if they are at the same level of aggregation as the query.

2. *Project List* - The project list of the query should be a subset of the project list of the cached result.

3. *Select List* - Selection predicates can be classified into two : selection on group-by attributes and selection on non group-by attributes. It is necessary that the selection on non group-by attributes of the query and the cached result match exactly. This is because, these selections have been factored in before doing the aggregations. On the other hand, selections on group-by attributes need not match exactly. Based on the intersection of group-by selection for the query and the cached results, a part of the cached results can be used. This works because selection on group-by attribute is a post-aggregation filter.

5.2.2 Computing Chunk Numbers

The selection predicates on the group-by attributes have to be converted into a list of chunk numbers, so that we can check if these are present in the cache. We will assume that the selection predicates are range or point predicates, i.e. they select ranges of values along the dimensions. To get the chunk numbers, we need a function which will compute the chunk number given a chunk index along each dimension. For example, in Figure 8, the chunk number corresponding to index values $(0, 0)$ is 0, chunk number for $(1, 2)$ is 6. This is computed using a row major ordering scheme. Lets call this function as *getChNum()*.

12	13	14	15
8	9	10	11
4	5	6	7
0	1	2	3

Time

Product

Figure 8: Chunk Numbering

Let R_i denote the selection on group-by dimension i, i.e. R_i is a set of values selected for dimension i. These values are converted into chunk indices. Let RC_i denote the set of chunk indices for dimension i. If c_i be the range size used to divide dimension i into chunks. Then,

$$RC_i = \{(\frac{x}{c_i}) \mid x \in R_i\}$$

If there are k group-by dimensions, we get k such sets. The next step is to take a cross product of these sets, and for each item in the cross product, use the function *getChNum()* to get the corresponding chunk number. The algorithm looks like:

Algorithm : ComputeChunkNums

$CNums = \emptyset$

For s in $(RC_0 \times RC_1 \ldots \times RC_k)$

$\quad num = getChNum(s)$

$\quad CNums = CNums \cup \{num\}$

$CNums$ is the final list of chunks which has to be looked up from the chunk cache.

5.2.3 Query Splitting

Once we have the list of chunk numbers, we lookup the cache to see if they have been cached. Depending on the cache contents, the list is split into two: $CNumsPresent$ of chunks present in the cache and $CNumsMissing$ of the missing chunks. Chunks in $CNumsPresent$ can be directly used from the cache. To compute the missing chunks, a query is issued to the backend. We use a modified form of SQL to specify this to the backend. For each missing chunk, it is necessary to determine which chunks of base table (or some pre-computed table) to aggregate to get that chunk. This process is similar to the one described in Section 5.2.2. The chunk number is converted to a set of chunk indices (inverse of the getChNum() function). These chunk indices correspond to the aggregated level. They are converted to a range of chunk indices at the base level, using the domain index. The base chunk numbers are computed from this range of chunk indices using **ComputeChunkNums** from Section 5.2.2.

Once the missing chunks are computed and fetched, we have all the chunks necessary to answer the query. It might be necessary to do some post-processing on these chunks, since chunks will have extra tuples than what are needed. The new chunks can be inserted into the cache according to the cache policies.

5.3 Chunked File

There are two alternatives in implementing a chunked file in the backend.

- In a complete implementation, a new chunked file type is added to the backend. This provides a chunk-based and normal relational interface. The backend is made aware of chunks in query processing. So if a chunk based query is asked from the backend, the results can be fetched on a chunk by chunk basis.
- It is possible to get the chunked file functionality without implementing a new file type. This is achieved in three steps:
  1. Add a new attribute to the relation to denote the chunk number
  2. Sort the file on the chunk number attribute, so that it gets clustered on a chunk basis.
  3. Create an index such as the BTree on the chunk number attribute. This gives a chunk based access to the file

  In this approach, the backend is not really chunk aware, i.e. it cannot return the results in terms of chunks. The middle tier has to separate the result tuples into different chunks in order to cache them.

We did a full implementation of the chunked file in the PARADISE [DKLP+94] database system. Chunked file is implemented by using a BTree as a chunk index on a *fact file*. *Fact file* is a relational file which is optimized

for storing and accessing the records in a *fact* table [RJZN97]. It exploits the fixed length property of *fact* table records. It eliminates the slot overhead in the pages and provides a fast path for skipped sequential access. To get a chunk based organization, the tuples in the *fact file* are clustered on a chunk basis. This is achieved at the time of bulkloading the file. The BTree holds one entry for each chunk and points to the start of the chunk in the *fact file*. To allow for updates, some extra space can be kept in each chunk.

5.4 Replacement Schemes

Old chunks need to be replaced when new chunks are to be added to the cache. Simple LRU is one of the options. But, LRU is not very suitable for OLAP queries. This is because chunks at different levels of aggregation have different cost of computation. It is much more expensive to compute a chunk at a high level of aggregation since it requires scanning and aggregating a lot of base level chunks. This cost has to be incorporated into the cache policies. Schemes which make use of a profit metric consisting of the size and execution cost of a query are considered in [SSV].

We use a similar replacement scheme which considers the benefit of a chunk. The notion of benefit of a group-by was introduced in [HRU96]. Since we do not do any aggregation in the middle tier, a group-by benefits itself but not any other group-by. The benefit of a chunk is measured by the fraction of the base table that it represents. For example, if there are n chunks for group-by (A, B), then the benefit of each chunk is $\frac{D}{n}$ where D is the size of the base table. Since the number of chunks at higher levels of aggregation is less, their benefit is more. The benefit is thus proportional to the cost of computing the chunk. We have combined the CLOCK scheme with the notion of benefit. When a new chunk is put into the cache, its weight is set to its benefit. Whenever the clock arm moves through it, its weight gets reduced by an amount equal to the benefit of the new chunk being considered for caching. If the weight is already zero, the chunk can be replaced. The weight is reset to its initial benefit value whenever the chunk is reaccessed. This scheme performs much better than simple LRU, which we confirm in our performance results.

6 Performance

We implemented a middle tier cache manager that uses the chunk based scheme. The backend is the PARADISE system that has been enhanced with the chunked file. We performed several experiments to study how the chunk based caching scheme performs compared to the query based caching. We also conducted some experiments to study the improvement in bitmap performance due to the chunked file organization.

Level	D0	D1	D2	D3
1	25	25	5	10
2	50	50	25	50
3	100	-	50	-

Table 1: Distinct Values of Dimensions

6.1 Caching Experiments

6.1.1 Experimental Setup

The data for our experiments was generated randomly with some parameters. The data has 4 dimensions and a metric. The dimensions have hierarchies defined on them. The hierarchy sizes for the dimensions were 3, 2, 3 and 2 respectively. The number of distinct values at different levels of the hierarchy are listed in Table 1. Level numbers decrease as we go to higher levels of aggregation in the hierarchy. The base table has 500,000 tuples, each of size 20 Bytes. The buffer pool size at the backend was 8MB. Due to resource and time constraints, we used a small data size. The cube size for this base table is 300MB. We used a cache size of 30MB in our experiments. All our experiments were run on a dual processor pentium 90Mhz having a memory of 128MB. The cache manager and the database server were run on the same machine. We used a raw device to store the fact table data so that file buffer cache effects are eliminated. All the experiments were performed with a buffer pool size of 8MB in the database. For each experiment, a query stream of 1500 queries was run.

6.1.2 Query Profile

We implemented a simple query generator which tries to model the query stream of OLAP systems. The query generator can generate a random stream of queries. It introduces locality in the query stream in two ways:

1. Designated Hot Region - A certain percentage of the database is designated as the hot region and the queries are generated such that a large percentage of them access the data in the hot region. We used the following distributions:
   - Q60 : 60% of the queries access 20% of the cube
   - Q80 : 80% of the queries access 20% of the cube
   - Q100 : 100% of the queries access 20% of the cube
2. Proximity - Proximity queries access data at the same level of aggregation but the selection predicate access adjacent members. These queries try to model hierarchical locality. The query stream consists of a mix of these three queries with the randomly generated queries.

	Proximity	Random
Random	0	1
EQPR	0.5	0.5
Proximity	0.8	0.2

Table 2: Locality Parameters

The degree of locality of the query stream can be tuned by changing the probability corresponding to each of the query categories. We peformed experiments with different degrees of locality. Table 2 shows the parameters that we used in our experiments. The query streams consisted of 1500 queries.

6.1.3 Performance Metrics

We used two metrics in order to evaluate the effectiveness of the caching schemes :

1. The average execution time of the last 100 queries in the query stream.
2. Cost Saving Ratio - This metric was defined in [SSV]. It is a measure of the percentage of the total cost of the queries saved due to hits in the cache.

$$CSR = \frac{\sum_i c_i h_i}{\sum_i c_i r_i}$$

   c_i : cost of execution of query q_i
   h_i : number of times references to q_i were satisfied in the cache
   r_i : total number of references to query q_i
   This is a more appropriate metric for OLAP queries, than the more common "hit ratio", since query costs can vary widely depending on the level of aggregation.

6.1.4 Comparison with Query Caching

We implemented a query caching scheme based on containment. The replacement policy is benefit based, as described for chunks. A bitmap index is built on the *fact table* at the backend, so that queries which miss the cache can be answered efficiently. Figure 9 shows the performance of the two methods for different types of locality. Figure 10 shows the performance of the caching schemes when 60 to 100% of the queries access 20% of the cube.

In all these cases, chunk based caching does well because of two reasons. It avoids redundant storage when compared to query level caching, where overlapping results are stored multiple times. Also, it can exploit

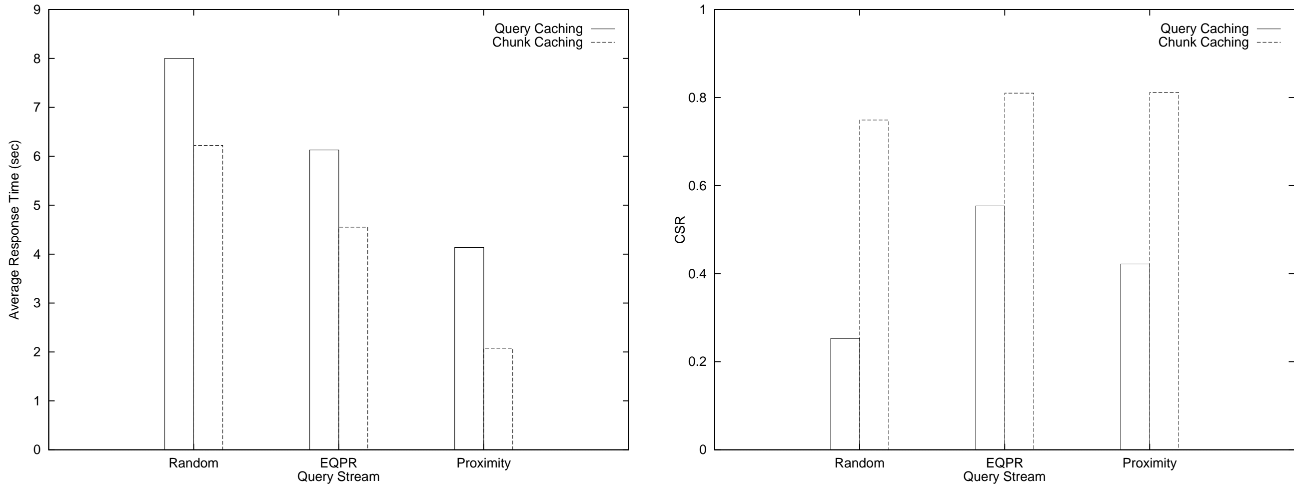

Figure 9: Different Types of Locality

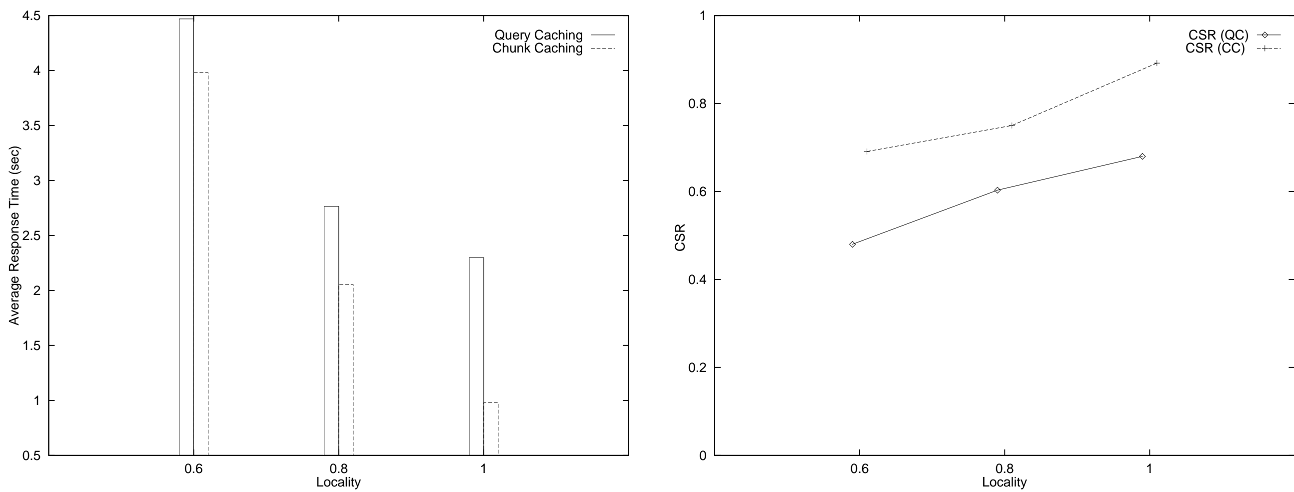

Figure 10: Percentage of Locality

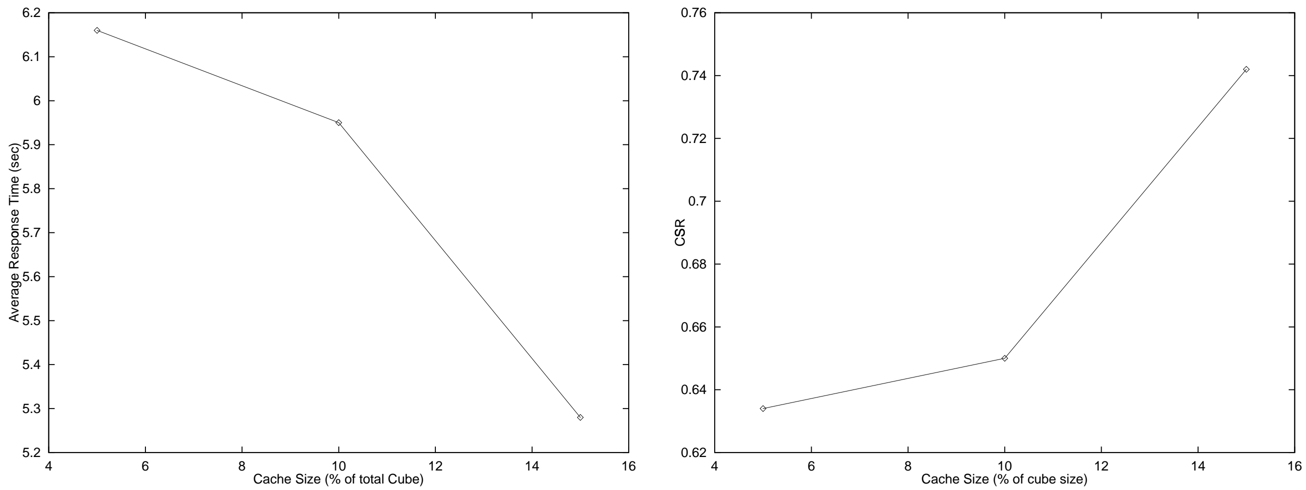

Figure 11: Effect of Cache Size

overlap between queries which the query level caching cannot. The ratio of their performance increases as the locality of the query stream increases. This shows that the chunk based scheme can exploit locality in a better way. On an average, we got an improvement factor of 2. We ran a simulation in order to verify that query level caching suffers because of redundant storage. For the 100Q query stream, we used a cache size of 20% of the cube size. Since 100% of the queries in this stream access 20% of the cube size, we should expect a CSR of 1 after a sufficiently long time. We simulated the behavior for 5000 queries. The CSR for query based scheme was 0.42 showing that there is some redundant storage in the cache. For the chunk based scheme, the CSR is 0.98.

6.1.5 Varying the cache size

In this experiment, we varied the cache size for chunk based caching for the query stream EQPR. As expected, the CSR and the execution times improve as the cache size is increased.

6.1.6 Varying chunk dimension range

As explained previously, chunk dimension range is critical for the performance of the cache. In our experiments, the chunk dimension range at any level is kept proportional to the number of distinct values at that level. For query stream EQPR, we studied performance by varying the ratio of the chunk dimension range to the total dimension range. Figure 12 shows that the performance improves initially as the ratio increases, but then it worsens again when the overhead increases due to large number of chunks.

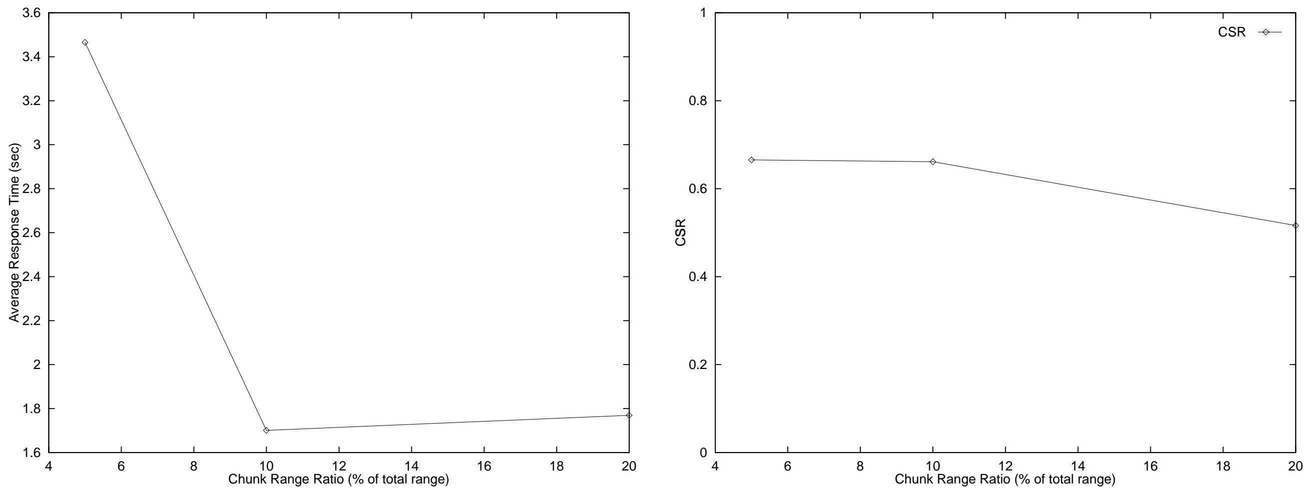

Figure 12: Effect of Chunk Range

6.1.7 Replacement Policies

In this experiment, we compared simple LRU and LRU combined with benefits. The LRU was approximated by CLOCK in both cases. This is necessary since the number of chunks cached is significant. Query stream EQPR was used for this experiment. Figure 6.1.7 shows that a replacement policy which consider benefits performs much better than simple LRU. This is expected since group-bys at higher level of aggregation are expensive to compute and should be given preference for caching.

6.1.8 Bitmap Performance

As shown analytically, a chunked file organization will improve the performance of bitmaps. The actual improvement depends on various factors such as the actual selection predicate, selectivity of the query, number of dimensions on which the selection is made etc. We tried queries with different selectivities. Figure 14 shows the query times for a randomly ordered file and a file ordered on chunk basis.

7 Conclusions and Future Work

We have introduced a new chunk based scheme for caching of queries. This scheme works very well in the OLAP domain where data is multi-dimensional. For better performance, we introduced a chunked file organization for data at the backend. Experiments show that chunk based caching combined with a chunked file organization performs better than traditional query and table caching. We used a benefit and LRU based replacement scheme

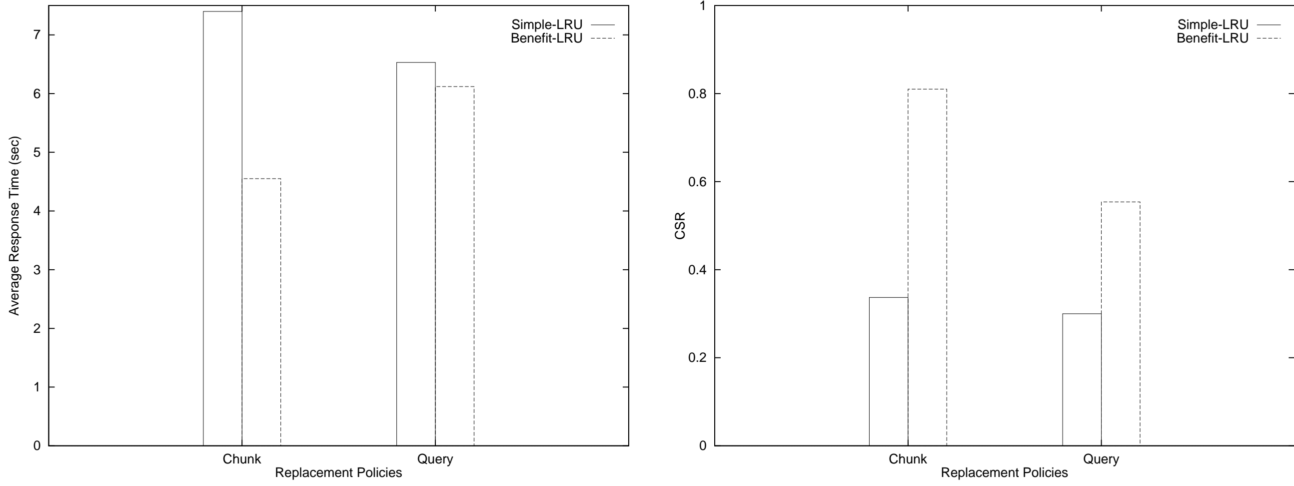

Figure 13: Replacement Policies

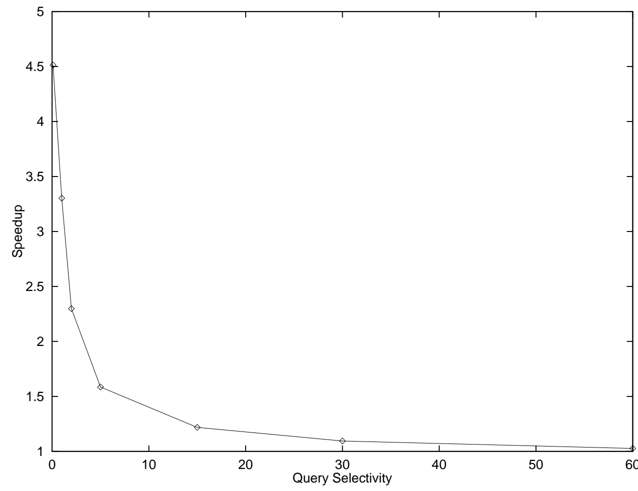

Figure 14: Bitmap Performance

which performs much better than simple LRU. At present, we do not do any aggregations of chunks in the middle tier. However, sometimes it may be possible to aggregate the chunks in the cache to get a missing chunk rather than going to the backend. For future work, we are planning to explore these possibilities. This implies the the notion of chunk benefit has to be improved. It is also possible to have more aggressive caching schemes, which fetch data at more detail than what is required. This will be particularly useful for drill down queries.

The chunked file organization can be implemented with little effort in existing systems. It maintains the relational nature of data. The chunked file organization improves the performance of bitmap indices, since it clusters the data. The chunked file also partitions the data on all the dimensions. We will be exploring ways to exploit this partitioning to do more efficient aggregations for star join queries.

References

[AAD+96] S. Agarwal, R. Agrawal, P.M. Deshpande, A. Gupta, J.F. Naughton, R. Ramakrishnan, S. Sarawagi. On the Computation of Multidimensional Aggregates, *Proc. of the 22nd Int. VLDB Conf.*, 506–521, 1996.

[ARBOR] Arbor Software Corporation. The Essbase Product Family, `http://www.arborsoft.com/essbase/datasht/esb_fmly1.html`

[BPT97] E. Baralis, S. Paraboschi, E. Teniente. Materialized View Selection in a Multidimensional Database, *Proc. of the 23rd Int. VLDB Conf.*, 1997.

[DFJST] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava, M. Tan Semantic Data Caching and Replacement *Proc. of the 22nd Int. VLDB Conf.*, 1996.

[DKLP+94] D. DeWitt, N. Kabra, J. Luo, J. M. Patel, J. Yu. Client-Server Paradise. *Proc. of the 1994 VLDB Conf.*, 1994.

[Fell57] William Feller, *An Introduction to Probability Theory and Its Applications*, Vol. I, John Wiley & Sons, pp 241; 1957.

[GBLP96] J. Gray, A. Bosworth, A. Layman, H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals, *Proc. of the 12th Int. Conf. on Data Engg.*, pp 152-159, 1996.

[GHRU97] H. Gupta, V. Harinarayan, A. Rajaraman, J.D. Ullman. Index Selection for OLAP. *Proc. of the 13th ICDE*, 208–219, 1997.

[Gupt97] H. Gupta. Selection of Views to Materialize in a Data Warehouse. *Proc. of the Sixth ICDT*, 98–112, 1997.

[HRU96] V. Harinarayanan, A. Rajaraman, J.D. Ullman. Implementing Data Cubes Efficiently, *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 205–227, 1996.

[RK96] R. Kimball. *The Data Warehouse Toolkit*, John Wiley & Sons, 1996.

[KT95] Kenan Technologies. An Introduction to Multidimensional Database Technology, *A white paper available from http://www.kenan.com/*

[MSI95] MicroStrategy Inc. The Case for Relational OLAP, *A white paper available from http://www.strategy.com/*

[OQ97] P. O'Neil, D. Quass, Improved Query Performance with Variant Indexes. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 38–49, 1997.

[OG95] P. O'Neil, G. Graefe, Multi-Table Joins Through Bitmapped Join Indices. *SIGMOD Record*, 8–11, September 1995.

[RJZN97] K. Ramasamy, Q. Jin, Y. Zhao and J. F. Naughton. Bit-Map Indices: Implementation Issues and Performance Results. Working Paper.

[SS94] S. Sarawagi and M. Stonebraker. Efficient Organization of Large Multidimensional Arrays. *Proc. of the 11th Int. Conf. on Data Engg.*, 1994.

[SDNR96] A. Shukla, P.M. Deshpande, J.F. Naughton, K. Ramasamy, Storage Estimation for Multidimensional Aggregates in the Presence of Hierarchies, *Proc. of the 22nd Int. VLDB Conf.*, 522–531, 1996.

[SDN] A. Shukla, P.M. Deshpande, J.F. Naughton, , *Submitted for SIGMOD 1998*.

[SSV] P. Scheuermann, J. Shim and R. Vingralek WATCHMAN : A Data Warehouse Intelligent Cache Manager *Proc. of the 22nd Int. VLDB Conf.*, 1996.

[SDJL96] D. Srivastava, S. Dar, H. V. Jagadish and A. Y. Levy. Answering Queries with Aggregation Using Views *Proc. of the 22nd Int. VLDB Conf.*, 1996.

[TPCD95] TPC benchmark D, proposed revision 1.0. San Jose, April 1995.

[Ull96] J.D. Ullman, Efficient Implementation of Data Cubes Via Materialized Views A survey of the field for the 1996 KDD conference.

[ZDN97] Y. Zhao, P.M. Deshpande, J.F. Naughton. An Array–Based Algorithm for Simultaneous Multidimensional Aggregates, *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 159–170, 1997.