Caching Multidimensional Queries Using Chunks *

Prasad M. Deshpande
University of Wisconsin, Madison
pmd@cs.wisc.edu

Karthikeyan Ramasamy
University of Wisconsin, Madison
karthik@cs.wisc.edu

Amit Shukla
University of Wisconsin, Madison
samit@cs.wisc.edu

Jeffrey F. Naughton
University of Wisconsin, Madison
naughton@cs.wisc.edu

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 more 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 previously cached query. In this paper, we propose caching small regions of the multidimensional space called *"chunks"*. Chunk-based caching allows fine granularity caching, and 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 missing from 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 an increasingly important part of business data analysis. 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 queries. The middle tier provides a multidimensional view of the data stored in the final tier, which can be a RDBMS. Queries that occur in OLAP systems are interactive and demand quick response time in spite of being complex. 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.

*This research is supported by a gift from NCR Corp., and by ARPA through Rome Air Force Laboratory contract F30602-97-2-0247

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD '98 Seattle, WA, USA
© 1998 ACM 0-89791-995-5/98/006...$5.00

1.1 OLAP Data Model and Queries

OLAP data sets are inherently multidimensional. An OLAP schema consists of a set of *dimensions* and *measures*. Each dimension is typically arranged in a *hierarchy*. A dimension can have some attributes which describe its members. In a relational system, such a multidimensional schema can be mapped onto a set of tables. Quantitatively, if there are n dimensions, the number of tables required for the mapping is $n+1$. Each dimension maps into a separate table called a *dimension table*. This table contains the information related to hierarchies on the dimension and other descriptive attributes. In addition, there is a separate table called *fact table* that relates the n dimensions with the 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*.

Example 1.1 *Consider an OLAP schema, which has three dimensions - Product, Store and Date. The dimensions have hierarchies defined on them. For example, store rolls up into city, city into state and state into country. The schema includes a single measure - Dollar Sales, which describes the sales value of particular product sold in particular store on a particular date. In a relational system, this schema is represented as shown below:*
Product = (*pname*, *pcategory*, *pid*)
Store = (*sname*, *scity*, *sstate*, *scountry*, *sid*)
Date = (*dmonth*, *dday*, *dyear*, *did*)
Sales = (*pid*, *sid*, *did*, *dollar_sales*)

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.

Example 1.2 *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, Product                              (Q.1)
WHERE pcategory = "clothes"
  AND dmonth ≥ "Jan" AND dmonth ≤ "Jun"
  AND Sales.did = Date.did
  AND Sales.pid = Product.pid
GROUP BY pname, dmonth
```

1.2 Motivation for Chunks

The *fact* table is usually much larger than the *dimension* tables. OLAP queries are computationally expensive since a substantial part of the *fact* table may have to be aggregated to get the results. For improved performance, it is imperative to cache and reuse aggregated results.

Example 1.3 *Let us assume that the results of Q.1 are cached. Now consider two possible queries Q.2 and Q.3, which are issued after the above query. Q.2 asks for aggregate sales for the months between January and May while Q.3 asks for aggregate sales for the months between April and September. These 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, Product                              (Q.2)
WHERE pname = "blaire_cotton_shirts"
  AND dmonth ≥ "Jan" AND dmonth ≤ "May"
  AND Sales.did = Date.did
  AND Sales.pid = Product.pid
GROUP BY pname, dmonth
```

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

To answer Q.2 and Q.3, the query evaluator is presented with two alternatives. The first alternative is to evaluate the queries using the results of the cached query (Q.1), while the second alternative is to issue them to the relational backend. The backend can then evaluate the query using base tables, indexes, 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, Q.2 can be evaluated entirely using the cache since it is properly contained in Q.1. However, if we use query containment, Q.3 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 can be evaluated entirely in the cache, while the other can be evaluated at 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. Large caches are common in current systems due to the large amounts of memory available.

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" around the query result. (If the query boundaries do not match the chunk boundaries, the chunks in this bounding envelop will contain extra tuples, which need to be filtered out). Since chunks are at a lower granularity than query level caching, they can be reused to compute the partial result of incoming queries. By using a fast mapping between query constants and chunk numbers, one can determine the set of chunks needed to completely answer a query. The set of chunks is partitioned into two disjoint sets, such that the chunks in one partition can be found in the cache while the chunks in the second partition have to be computed using the backend relational engine. This partitioning leads to the intended query decomposition. Since query results can be encapsulated by an integral number of chunks, the replacement policy can take advantage of the "hotness" of a chunk, which has more value than the hotness of a query. In order to reduce the total query time, it is essential to speed up the computation of the missing chunks from the backend. We introduce a "chunked" file organization for relational tables to achieve this goal. An interesting side effect of this organization is that it speeds up general query processing as well.

The paper is structured as follows: in Section 2, we describe the caching issues for the OLAP data model and related work in this field. 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 a chunk based caching scheme. Finally, we present our performance results in Section 6 and conclude in Section 7.

2 Caching for the OLAP Data Model

2.1 Locality in OLAP Queries

OLAP queries are typically repetitive and follow a predictable pattern.

Example 2.1 *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. 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 may move on to another city say Milwaukee.*

Thus, an OLAP session can be characterized using 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.

2.2 Taxonomy of Caching Schemes

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

1. *Nature of caching* - Under this category, the caching schemes can be classified as static or dynamic. In static caching, the items to be cached are chosen statically, independent of the query stream. A static approach is specifically suited for the backend where one can precompute aggregated tables. A set of group-bys is chosen and the corresponding tables are materialized. This refers to the problem of selecting what aggregates to precompute [HRU96, GHRU97, SDN] using a given amount of space. On the other hand, in dynamic caching schemes, the cache contents vary dynamically, since new items may be inserted and old items may be removed from the cache. A dynamic approach will be significantly beneficial at the middle tier, since it adapts to the query profile.

2. *Unit of caching* - The granularity of the units cached significantly influences performance. Based on the unit of caching, we can classify the caching schemes into *query level caching*, which caches the entire results of queries, and *table level caching*, which caches entire tables. Smaller size of the unit of caching leads to better reuse of query results. On the other hand, the overhead of managing smaller units during replacement added with the overhead of checking whether the incoming query can be answered from them becomes expensive.

In this paper, we propose a dynamic caching scheme using *chunks* as a unit of caching and demonstrate its feasibility under realistic query workloads without much overhead.

2.3 Related Work

Extensive work has been reported in the literature on using views for answering queries and on caching of query results. Some of them have significant applicability to the domain of data warehousing and OLAP. The problem of answering queries using views in the presence of aggregations 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 can 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. Cache replacement and admission schemes specific to warehousing have been studied in [SSV96]. In their work, they use a profit metric for queries to decide whether a query should be cached or not. We use similar ideas in our replacement policies. The idea of using a profit metric is relevant for OLAP, since highly aggregated results are expensive to compute and should be given preference while caching.

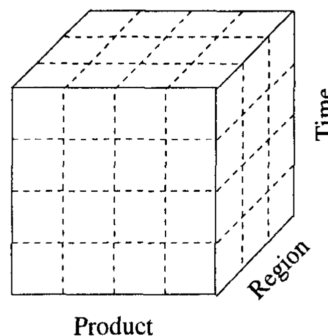


Figure 1: Chunking the multidimensional space

Caching at a granularity smaller than a query has been proposed in [DFJST]. They describe a semantic scheme of caching for client-server systems, where they cache semantic regions. The semantic approach is highly suitable to the OLAP domain where data is multi-dimensional and the notion of semantic data regions is 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. Chunk based caching can be considered as specialized semantic caching. Instead of having variable semantic regions (in terms of size and shape), chunks divide the space into uniform regions statically. There are many benefits due to the static definition of chunk regions, which are described in Section 3.2. It makes cache reuse more efficient and makes it possible to reorganize data at the backend to correspond to these chunks. The chunked-file representation, described in Section 4, reduces the cost of a cache miss, since fetching the missing chunks is efficient. None of the previously proposed schemes consider similar reorganization of data to make cache misses less expensive. The chunked-file organization can also be used for storing precomputed aggregates at the backend.

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 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 are very suitable as a unit of caching, since they capture the notion of semantic regions by dividing the multi-dimensional space space into uniform semantic regions.

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, consisting of the missing chunks, has to be computed from the backend.

In order to reduce the cost of a chunk miss, missing chunks should be computed efficiently at the backend. To achieve this, we propose a chunk based organization of data in the backend. It can be implemented either as multi-dimensional array if the database supports it or, as a chunked file (see Section 4) if the system is purely relational. Statically precomputed aggregate tables can also be organized on a chunk basis. 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

We now discuss the advantages of 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. Chunks which are not frequently accessed will be replaced eventually. The second major advantage is that previous queries can be used much more effectively. For example, Figure 2 shows a chunk based cache, in which each query represents a portion of the multidimensional space. Q.3 is not contained in either Q.1 or Q.2 or their union. Thus, methods based on query containment will not be able to use Q.1 and/or Q.2 to answer Q.3. With chunk based caching, Q.3 can use chunks it has in common with Q.1 or Q.2. 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 Q.3.)

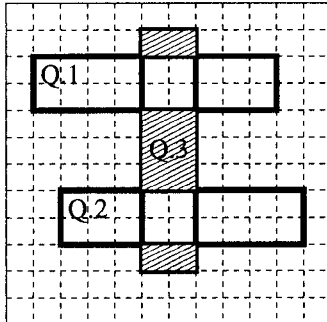


Figure 2: Reusing cached chunks

2. *Uniformity* - The notion of uniform semantic regions, which are statically defined in the form of chunks, makes query reuse less complex. By using a fast mapping, the semantic region representing the query can be mapped into a set of chunks. 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* - Since chunking can be applied at any level of aggregation, and due to the static definition of chunks, there is a very simple correspondence between chunks at different levels of aggregation. This defines a closure property on chunks which states that we can aggregate chunks at one level to obtain chunks at a different level of aggregation. This property can be used to compute the missing chunks, since only the corresponding chunks at the base level in the fact table have to be scanned, rather than scanning the entire table.

Example 3.1 *Figure 3 shows data at two levels of aggregation: - (Product, Time) and (Time), with chunking 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 can be used to reduce the cost of a chunk miss.*

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

Product — 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. Chunk based caching eliminates this replication by sharing chunks containing overlapping results, thus allowing more queries to be cached.

3.3 Chunking the Multi-dimensional Space

We will now consider the details of dividing the multi-dimensional space into chunks.

3.3.1 Ordering the Dimensions

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 non-numeric 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. A mapping structure called *Domain Index* is used to to maintain this correspondence between a dimension value and its ordinal number.

Example 3.2 *For the* store *dimension, the ordering should be such that all stores in a city are clustered, the cities in a state are clustered and the states in a country are clustered together. Figure 4 shows a simple hierarchy and the ordering at each level of the hierarchy.*

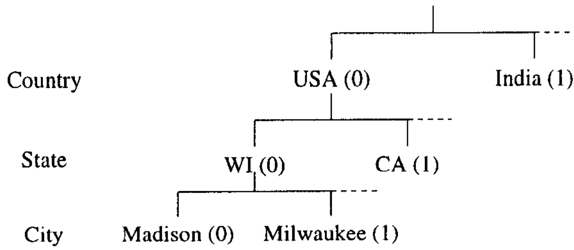


Figure 4: Ordering distinct values on a dimension

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.

3.4 Chunk Ranges

Once the distinct values in each level of a dimension are ordered, we have to divide them into chunk ranges, which identify the 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 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 chunk ranges.

Example 3.3 *Figure 5 illustrates a problem that arises if the actual structure of the hierarchy is ignored. In this case, dimension A has a three level hierarchy defined on it. 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* $R_{2,0}$*.*

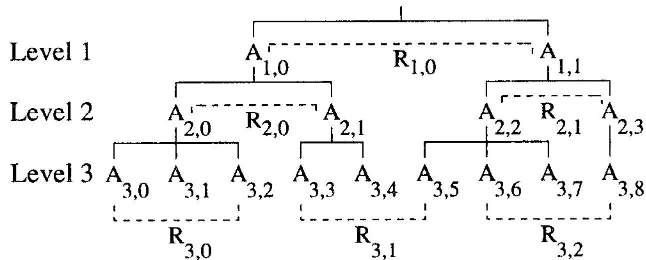


Figure 5: Uniform ranges

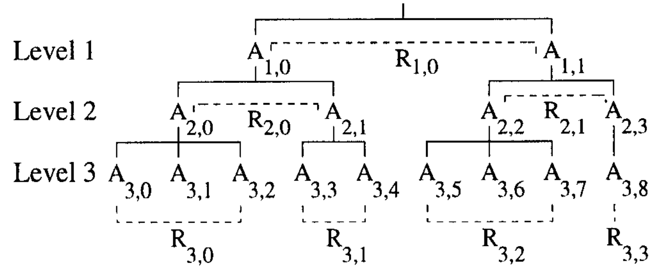


Figure 6: Ranges according to hierarchy

In 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
```

Example 3.4 *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.*

Since the chunk ranges are no longer uniform, it is necessary to record the range number (also called chunk index number) for each distinct value. This can be stored along with the ordinal number in the domain index.

4 Chunked File Organization

As discussed previously, the cost of a chunk miss can be reduced by organizing data on a chunk-basis at the backend. One way to achieve this is to use chunked multi-dimensional arrays. However, significant effort is required to incorporate the support for multi-dimensional arrays in a relational system. In addition, this may result in the loss of relational access to the data. We show that the concept of chunking need not be restricted to arrays, and can be applied to relational tables using a *chunked-file* organization.

In a chunked-file organization, the data is stored as tuples rearranged on a chunk basis. Thus all tuples in a chunk are 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 *chunk index* can be implemented using B+ Trees. The chunked file provides two interfaces: a relational interface, so that it can be considered as a regular table and can be used in SQL statements, and a chunk based interface, that allows direct access to individual chunks. 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 little effort. The various options are explored 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 size of the entire table.
2. *Multi-dimensional clustering* - chunked organization achieves excellent multi-dimensional clustering of 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, as shown in the next section.

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 extract 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 bitmap performance.

Example 4.1 *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. The improvement will be greater 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.*

Using a simple back of the envelope mathematical calculation, one can quantify this improvement. We ran some experiments (described in Section 6) to verify this improvement in a more general situation as well.

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 of the multi-dimensional space, distinct values along each dimension are broken up into

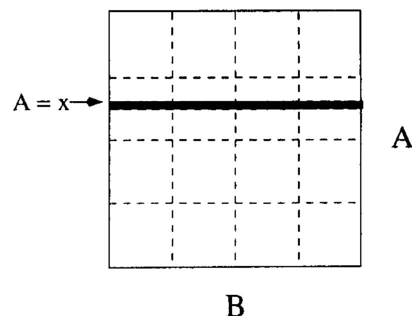


Figure 7: Clustering improves bitmap performance

ranges. The size of the ranges determines the number of chunks and their.

Example 5.1 *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, 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 by [SS94] in the context of multi-dimensional arrays. The missing chunks are computed by scanning the corresponding chunks at the backend. Since disk I/O is in units of pages, the average chunk size should be a multiple of the page size to reduce the overhead of extra I/O. Section 6 describes some experiments to determine the optimal chunk range.

5.2 Query Processing

5.2.1 Query Analysis

We assume a *"star join template"* for queries, i.e. each query is a join of a *Fact* table with some dimension tables, filtered using some selections and followed by an aggregation:

```
SELECT ⟨ dimension-list ⟩ ⟨ aggregate-list ⟩
FROM ⟨ FactName ⟩, ⟨ DimensionTable-list ⟩
WHERE ⟨ condition-clause ⟩
GROUP BY ⟨ dimension-list ⟩
```

When a query is issued, it is necessary to check if it can be answered from the cache. The selection predicates are analyzed to produce a list of chunk numbers required to answer the query. Then, the cache can be interrogated to

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

Product

Figure 8: Chunk numbering

see if any of the required chunks are present in it. For cached chunks to be used to answer a query, the following conditions have to be satisfied:

1. *Level of Aggregation* - The level of aggregation of a query is determined by the *dimension-list* in the group-by clause. Cached chunks can be used for a query when they are at the same or lower level of aggregation than the query. If the cached chunks are at a lower level, they need to be aggregated to get the query results. The problem of figuring out the set of chunks (different chunks may be at different levels), which can be aggregated to get a result chunk is quite complex. In our implementation we assume that all aggregations are done in the backend. Thus, we reuse the cached results only if they are at the same level of aggregation as the query.

2. *Condition Clause* - 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 predicate on non group-by attributes of the query and that corresponding to the cached chunk match exactly. This is because, these selections have been factored in before performing the aggregations. On the other hand, selections on group-by attributes are mapped to a set of chunk numbers, which are used to lookup in the cache. This mapping is possible for any condition clause, which uses only the operators $<$, $>$ and $=$ combined in a boolean formula. This is because, the query represents a union of regions in the multi-dimensional space. The set of chunks forms a bounding envelop around the query region. Since this envelop may not match entirely with the query results, post-processing is necessary to filter the extra tuples.

5.2.2 Computing Chunk Numbers

Selection predicates on group-by attributes are converted into a list of chunk numbers, which can be used to determine if the required chunks are present in the cache. We assume that selection predicates are either range or point predicates, i.e. they either select ranges of values along the dimensions or a single point. To calculate 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 and is identical to the common function used to store arrays. Let us call this function as *getChNum()*.

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, using the domain index. Let RC_i denote the set of chunk indices for dimension i. If there are k group-by dimensions, we get k such sets $RC_1, RC_2, \ldots RC_k$. 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 is as follows:

Algorithm : ComputeChunkNums
$CNums = \emptyset$
For s in $(RC_0 \times RC_1 \ldots \times RC_k)$
  $num = getChNum(s)$
  $CNums = CNums \cup \{num\}$

$CNums$ is the list of chunks required to answer the query.

5.2.3 Query Splitting

After determining the list of chunk numbers, we can lookup the chunk cache to see if they have been cached. Depending on the cache contents, the list is split into two: (1) *CNumsPresent*, of chunks present in the cache and (2) *CNumsMissing* of the missing chunks. Chunks in *CNumsPresent* can be obtained from the cache, while chunks in *CNumsMissing* have to be computed from the backend. Then, a modified form of SQL, qualified with chunk numbers, is used to compute these chunks from the backend database. For each missing chunk, it is necessary to determine which chunks of the base table (or some precomputed table) have to be aggregated to compute it. 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 of the query. 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 Algorithm *ComputeChunkNums*. Once the missing chunks are computed and fetched, we have all the chunks necessary to answer the query. It is necessary to do post-processing on the chunks which are not contained in the query region (the boundary chunks), since they will have more tuples than required by the query. The new chunks can be inserted into the cache using cache policies, which we discuss later in the paper.

5.3 Chunked File

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

1. *Comprehensive Implementation* A new chunked file type is added to the backend database, providing a chunk-based interface in addition to the normal relational interface. This can be done by making the backend "chunk aware", so that it understands chunk based queries and sends back results on a chunk by chunk basis.

2. *Simple Implementation* It is possible to get the chunked file functionality without implementing a new file type. This is achieved with minimal effort using the following three steps:
   (a) Add a new attribute to the relation to denote the chunk number
   (b) Sort the file on the chunk number attribute, so that it

gets clustered on a chunk basis.

(c) Create an index such as the B-Tree 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, and the middle tier has to separate the result tuples into different chunks in order to cache them.

We implemented the chunked file in the Paradise [DKLP+94] Database System using option 1 above. The chunked file was implemented by using a B-Tree as a chunk index on a *fact file*. A *Fact file* [RJZN97] is a relational file optimized for storing and accessing the records in a *fact* table, which 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 achieve a chunk based organization, tuples in the *fact file* are clustered based on their chunk numbers. This can be done while bulk-loading the *fact file*. The B-Tree holds one entry for each chunk and points to the start of the chunk in the *fact file*. Updates can be supported by reserving extra space in each chunk. When all the extra space is used up, we have to reorganize the *fact file*.

5.4 Replacement Schemes

Replacement schemes are required because old chunks have to be replaced to make room for new chunks in the cache. Simple LRU is one of the options, but, it is not very suitable for OLAP queries because chunks at different levels of aggregation have different costs of computation. For example, it is 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 any cache replacement policy.

Schemes which make use of a profit metric consisting of the size and execution cost of a query are considered in [SSV96]. 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 perform aggregations in the middle tier, a group-by benefits only itself. 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 small, they have a higher benefit. The benefit is thus proportional to the cost of computing a chunk. We combined the CLOCK scheme with the notion of benefit. Let Benefit(C) denote the benefit of a chunk. We associate one more quantity, called Weight(C) with each chunk C in the cache. The replacement algorithm is as follows:

Algorithm : ClockBenefit

```
Input : chunk N to be inserted in the cache
While ( space not available for N )
    Let C be the chunk corresponding to
        current CLOCK position
    if (Weight(C) ≤ 0)
        Evict C from the cache
    else
        Weight(C) = Weight(C) − Benefit(N)
    Advance Clock position
Insert N into the cache
Weight(N) = Benefit(N)
```

The weight is reset to its initial benefit value whenever the chunk is reaccessed. Our performance results confirm that Algorithm *ClockBenefit* performs much better than simple LRU.

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

Table 1: Distinct values at different levels. Dimensions D0 and D2 have a hierarchy size of 3, whereas D1 and D3 have a hierarchy size of 2

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 a chunk based caching scheme performs when compared with a query based caching scheme. We also conducted experiments to study improvements in bitmap performance from a chunked file organization.

6.1 Caching Experiments

6.1.1 Experimental Setup

The schema we used in our experiments had four dimensions, one metric and each dimension has a hierarchy defined on it. The hierarchy sizes are 3, 2, 3 and 2 respectively. The number of distinct values at different levels of the hierarchy are shown in Table 1. A smaller level number means a higher level of aggregation. For example, D0, level 1 represents a higher level of aggregation than D0, level 3. Using this distribution of distinct values, we assumed that data is uniformly distributed and generated a base table with 500,000 tuples, each of size 20 bytes. The cube size for this base table is 300MB. The buffer pool size at the backend was 8MB. We used a cache size of 30MB in middle tier cache manager. Our experiments were run on a dual processor Pentium 133Mhz having 128MB of memory. 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 the effects of file system caching are eliminated. Each experiment consisted of running a query stream with 1500 queries.

6.1.2 Query Profile

We implemented a simple query generator which attempts to model the query stream of an OLAP systems, and studied two types of locality in query streams:

1. *Designated Hot Region* - A certain percentage of the database is designated as the hot region. Queries are generated such that a large percentage of them access data in this hot region. The rest of the queries are uniformly distributed over the database. This is designed to model locality of access, such as an analyst located in Wisconsin being primarily interested in data for Wisconsin. We used the following distributions in our experiments:

- 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 attempt to model hierarchical locality. Queries adjacent in the query stream access data at the same level of aggregation but the selection predicates are different; they either overlap or access adjacent members of the dimension. For example, one query may ask for sum(sales) for Madison, and the next query might ask for sum(sales) for Milwaukee. The query stream consists of a mix of such queries with randomly generated queries, which are uniformly distributed over all the group-bys. The degree of locality can be tuned by varying the mix of random queries and proximity queries. We used the following distributions in our experiments:

   - QRandom : 100% queries are randomly generated
   - QEqual : 50% queries are random, and 50% are proximity queries
   - QProximity : 20% queries are random, and 80% are proximity 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 [SSV96], and is a measure of the percentage of the total cost of the queries saved due to hits in the cache. It uses an estimate of the cost of executing a query at the backend to compute the savings in cost due to a cache hit. Consider a query stream consisting of a mix of n queries $q_1, q_2, \ldots q_n$.

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

c_i : cost of execution of query q_i at the backend
h_i : number of times references to q_i were satisfied in the cache
r_i : total number of references to query q_i
In case of chunk based caching, where only a part of the query may be satisfied from the cache, we compute the CSR using the same formula in terms of chunks instead of queries. However, this is not the exact CSR for chunks, since chunks on the boundary of the query region may fetch data not required by the query, and counting the entire savings of these chunks is not appropriate. The CSR calculated by the above formula will be close to the exact CSR, assuming that percentage of boundary chunks w.r.t. the total number of chunks is less. CSR 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 Experimental Results

We now describe the experimental evaluation of the chunk based caching scheme.

Comparison with Query Caching

We implemented a query caching scheme based on containment. The replacement policy is benefit based, and is similar to the benefit based replacement policy 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 compares the performance of the two methods for query streams with proximity queries. Figure 10 shows the performance of the caching schemes for query streams with a designated hot region. In both these figures we plot the execution times as well as the CSR.

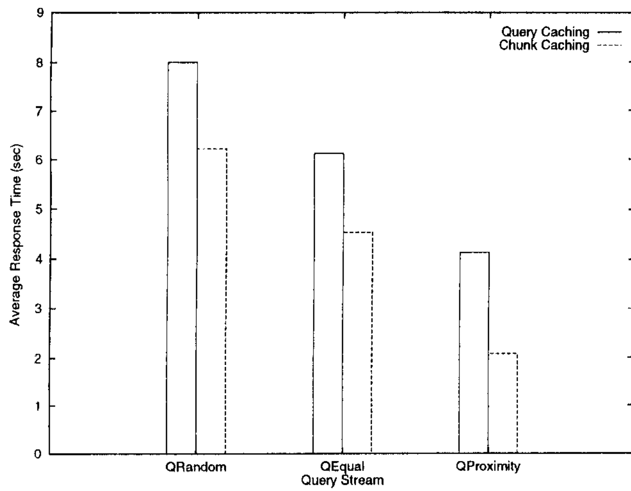
In all these cases, chunk based caching does well because of two reasons. It avoids redundant storage when compared to query level caching, where overlapping results are stored multiple times. In addition, chunk based caching exploits overlap between queries which query level caching cannot use. This is reflected in a higher CSR and lower average execution time for chunk based caching. The ratio of their performance increases as the locality of the query stream increases (either due to increase in proximity queries or due to increase in queries accessing the hot region), leading to an improvement factor of about 2. This shows that the chunk based scheme can exploit locality better than query level caching. We ran a simulation in order to verify that query level caching suffers because of redundant storage. For the Q100 query stream, we used a cache size of 20% of the cube size (60 MB). 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 was 0.98.

Varying the cache size

In this experiment, we varied the cache size for chunk based caching for the query stream QEqual. Figure 6.1.4 shows that as the cache size is increased, the CSR increases, since more chunks can be found in the cache. The execution times reduce as expected.

Varying chunk dimension range

Distinct values on each dimension are divided into ranges in order to chunk the multi-dimensional space. As we have previously seen, size of the chunk dimension range is critical to cache performance. 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 QEqual, we studied performance by varying the ratio of the chunk dimension range to the total dimension range from 5% to 20%. Figure 12 plots the ratio of the response times and the CSR w.r.t. the base case of 10% chunk range. It can be seen that chunk range of 5% has a CSR slightly greater than that of the 10% chunk range, whereas 20% chunk range has a lower CSR. This is due to the fact that smaller chunk ranges lead to smaller chunks and better cache use. However, the response time graph does not match the CSR graph. In spite of having a higher CSR, the average response time in the case of 5% chunk range is higher than in the case of 10% chunk range. This is due to the overhead caused by the larger number of chunks when the chunk range is 5%. This shows that, as chunks get smaller, eventually the overhead due to the large number of chunks overcomes the benefit achieved due to better cache utilization.



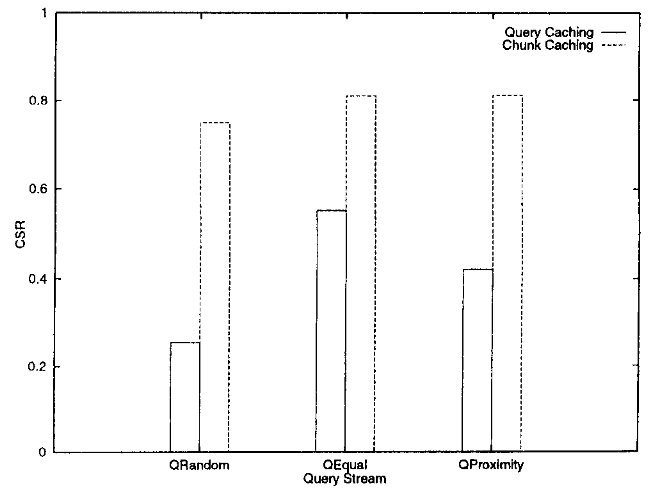
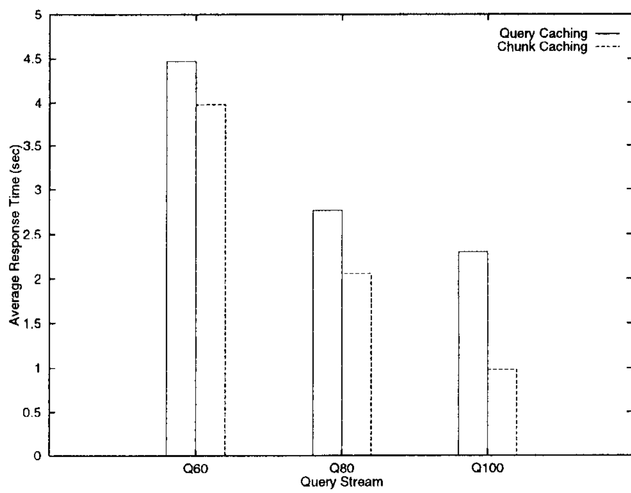


Figure 9: Query streams having varying percentage of proximity queries



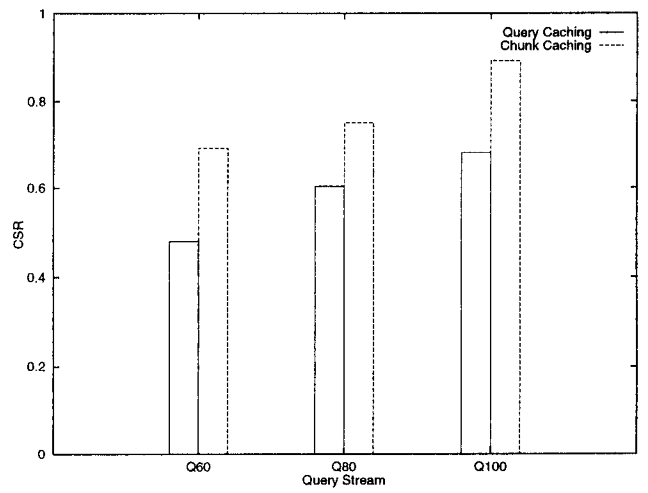
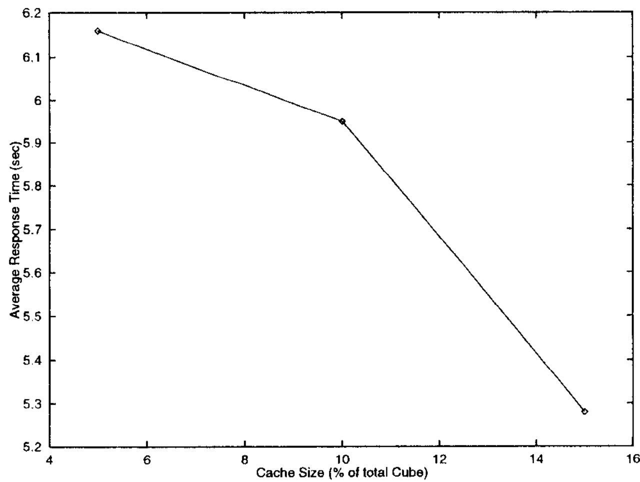


Figure 10: Query streams having varying percentage of queries accessing the designated hot region



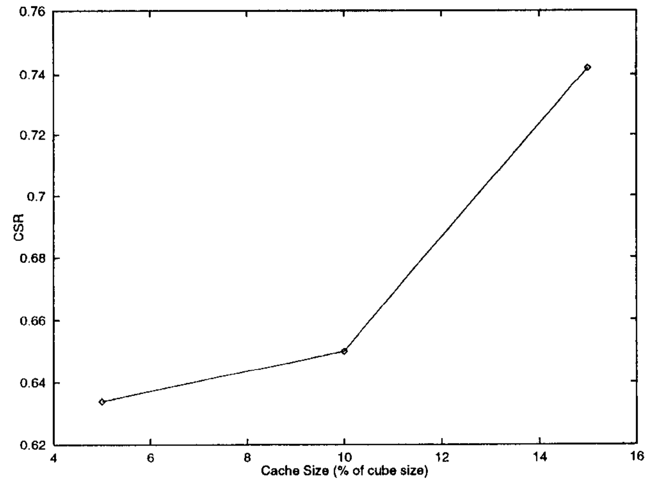


Figure 11: Effect of varying the cache size on execution time and CSR for chunk-based caching

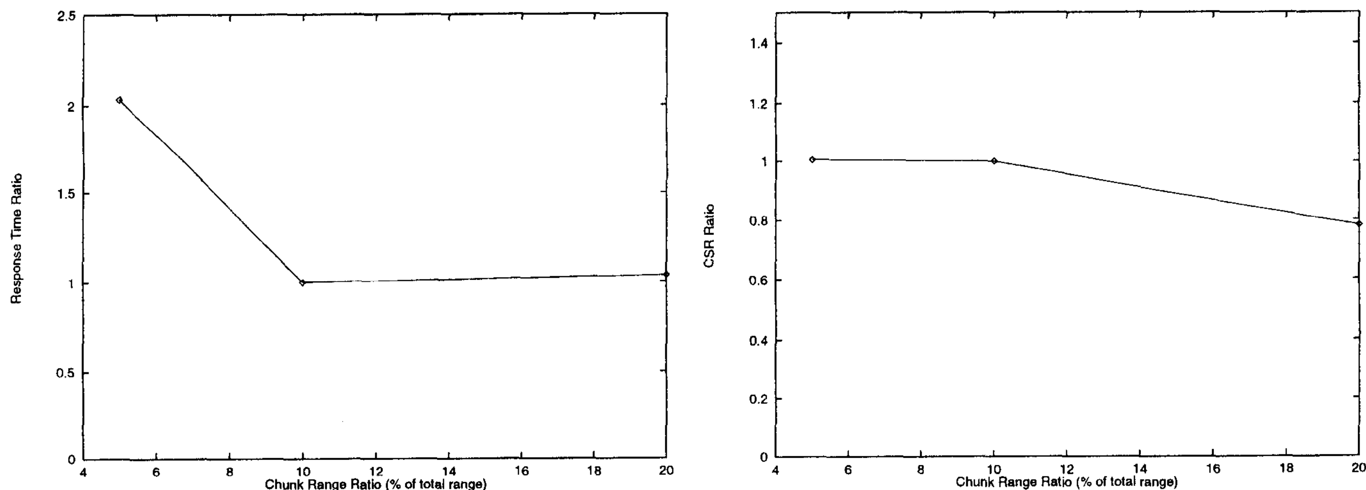


Figure 12: Effect of varying the chunk range on execution times and CSR

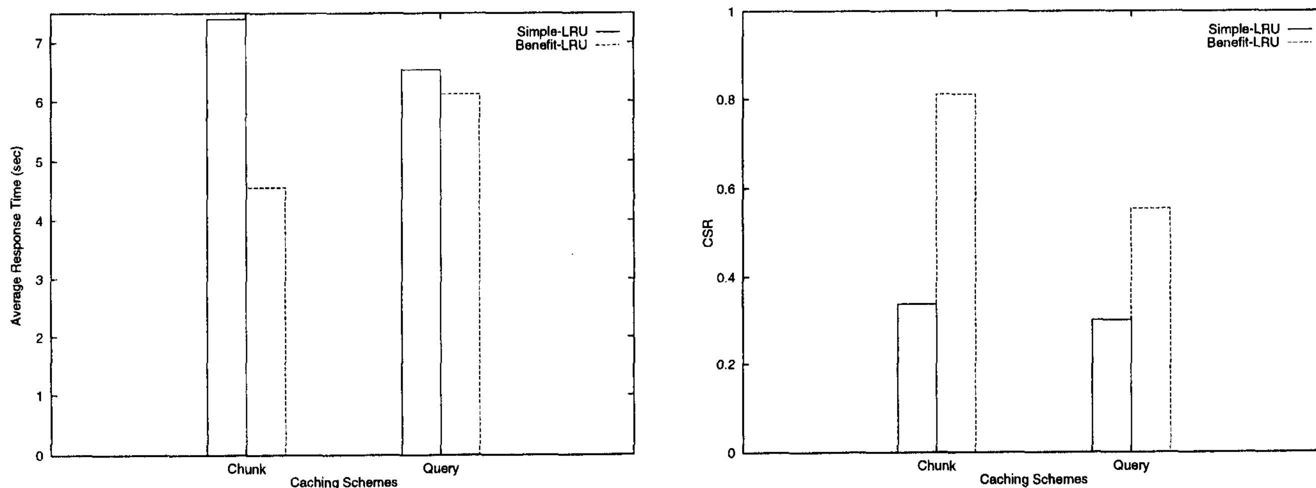


Figure 13: Effect of replacement policies on cache performance

Replacement Policies

In this experiment, we compared simple LRU and LRU combined with benefits. The LRU was approximated by CLOCK in both cases. Query stream QEqual was used for this experiment. Figure 13 shows that a replacement policy which consider benefits performs much better than simple LRU. This is expected since group-bys at a higher level of aggregation are expensive to compute and should be given higher preference for caching. The difference is more pronounced in the case of chunk based caching, since chunks may compute more data than asked by the query. When such a chunk is evicted, the excess effort in computing the extra data is wasted. Since, highly aggregated chunks are expensive to compute, this waste can be minimized by giving them higher priority in the cache, which is achieved by using a benefit based policy.

6.2 Bitmap Experiments

As explained previously, 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. In our experiments, we used a simple selection query, and varied the selectivity of the query. Figure 14 shows the speedup obtained in bitmap performance for a file ordered on chunk basis over a randomly ordered file. The speedup reduces as the query becomes less selective, since more tuples satisfy the predicate and a larger percentage of pages in the fact table are accessed. It eventually reduces to a scan of the entire file, in which case the chunked organization does not matter.

7 Conclusions and Future Work

We have introduced a new chunk based scheme for caching queries that works very well in the OLAP domain, where data is multi-dimensional. Chunk-based caching allows fine granularity caching, and allows queries to be partially answered from the cache. We introduced a new organization for relational tables, called "chunked file", that reduces the cost of a chunk miss. Our experiments show that for workloads that exhibit query locality chunk based caching combined with a chunked file organization performs better than traditional query caching. One of the important issues in chunk based caching is to choose an appropriate chunk range size. Smaller chunk ranges lead to smaller chunks and bet-

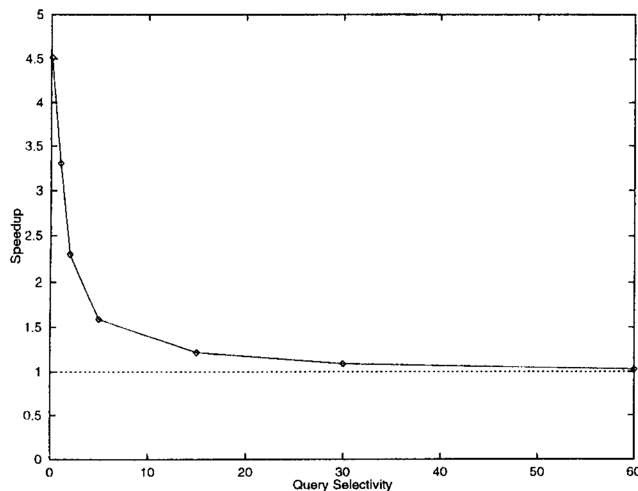


Figure 14: Comparing bitmap performance of a file ordered on chunk basis with a randomly ordered file

ter caching, but the overhead due to the larger number of chunks increases. The replacement policy is also very critical for performance. We used a benefit-based LRU policy and showed that it performs much better than simple LRU. The chunked file organization can be implemented with little effort in existing systems, while maintaining their relational nature. We showed that the chunked file organization also improves the performance of bitmap indices, since it clusters the data.

In our current implementation, all aggregations are restricted to the backend. For future work, we are planning to explore the possibility of aggregating chunks in the cache to get a missing chunk rather than going to the backend. This implies that the notion of chunk benefit has to be improved. Another possible enhancement is to use more aggressive caching schemes, which fetch data at more detail than what is required. This will be particularly useful for drill down queries. Finally we are planning to explore other applications of the chunked file organization. Since it also partitions data on all the dimensions, the pre-existing partitioning could be used to do more efficient aggregations for star join queries.

Acknowledgements

We would like to thank the Paradise team for their help with the use of the Paradise Database System.

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.

[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.

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

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

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

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

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

[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.

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

[SSV96] P. Scheuermann, J. Shim and R. Vingralek WATCHMAN : A Data Warehouse Intelligent Cache Manager *Proc. of the 22nd Int. VLDB Conf.*, 1996.

[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.