Applying Virtual Storage Management Method to View Materialization

Huei-Huang Chen Wan-Pei Liang

hhchen@ttu.edu.tw wpliang@it01.cse.ttu.edu.tw

Department of Computer Science and Engineering, Tatung University
No. 40, Chung-Shan N. Rd., Sec. 3, Taipei, 104, Taiwan, R.O.C.
Tel: 886-2-25925252 ext.3295 Fax: 886-2-25925252 ext.2288

Abstract

The queries in data warehouse and online analytical processing (OLAP) systems have the complex nature and demand a long execution time. One user may deliver a sequence of related queries, and other users may also submit several queries. These queries may have commonalities. Hence, query response times can be greatly improved by caching previous query results to answer the later queries.

The earlier researches on query caching can be divided into table level caching and query level caching. Then, other scholar proposed a chunked caching method for large query results. Chunked caching method separates the storage space into small regions, the caching unit. In this paper, we consider the textual feature of dimensions and propose a ranging algorithm to realize chunked caching for non-numerical data. We also use Least Recently Used (LRU) queue to record the time when a chunk in the cache is most recently referenced. Therefore, besides the benefits of query results themselves, the query variation during different period can also be considered in the computation of replacement priorities. According to the experimental result, we prove the efficiency of our proposal and find the optimal threshold to reduce overall response cost.

Keywords: View Materialization, On-Line Analysis Processing

1 Introduction

Data warehouses and On-Line Analytical Processing (OLAP) systems are becoming increasingly important in data analysis. The typical processing time of decision support and OLAP queries range from minutes to hours [1]. This is due to the nature of complex queries used for decision-making process. This research is to improve the query response time by caching query results for data warehouses and OLAP applications.

A typical characteristic of data sets in these systems is their multidimensional nature. However, traditional relational database systems are not designed to provide the necessary performance for these types of data. Data warehouses and OLAP systems act as mediator between the backend databases and users. Hence such systems are built by using three-tier architecture. The first tier provides a user-friendly interface that allows the user to build queries and parse queries into SQL statement. The middle tier provides multidimensional views of the data stored in the final tier, backend DBMS.

Queries that occur in OLAP systems demand fast or acceptable response time in spite of being complex. Various techniques can be used at different phases of the query process in order to speed up the execution time. However, the query workload consists of the queries with a lot of overlap. It is obvious that this involves repetitive

work across queries. The typical approach to eliminate this redundancy is to reuse these query results across queries.

As we know, a multi-level memory solution is used to enable reuse of loaded instructions or data. The paging algorithm is used to choose the page to be replaced for each page fault so as to minimize the access cost.

The query results reusing problem is similar to the caching concept of memory management. Deciding the granularity of cached unit and replacement policy are the two primary issues in such problems. Previous researches showed that LRU (Least Recently Used) [2] replacement policy is one of the best methodologies in paging problem. However, it needs further consideration when applying caching to view materialization. This paper proposes a chunked caching algorithm and its appropriate replacement schemes.

2 Related Works

A typical characteristic of data sets in data warehouse and OLAP systems is their multidimensional nature. However, traditional relational database systems were not designed to provide the necessary performance for such types of data. Hence such systems are built by using three-tier architecture. The first tier provides an easy to use graphic user interface that allows the users to build queries. The middle tier provides a multidimensional view of the data stored in the final tier, which is usually an 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 lifetime of the query to speed up its execution. Pre-computation and the use of specialized indexing structures have been predominantly used at the RDBMS to speed up such queries.

2.1 Query Behavior

OLAP queries are typically repetitive and follow a predictable pattern. Thus, an OLAP session can be characterized using different kinds of locality on query behaviors [3].

Temporal: The same data may be accessed again either by the same user or a different user. That is these kind of queries repeated frequently. Thus, caching the results of such queries can improve response time greatly.

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 that are related by the parent/child or sibling relationships will be accessed together.

2.2 Caching Methodologies

Caching had been implemented by OLAP systems in order to reduce response times for multidimensional queries. The early works on such caching have considered

table level caching and query level caching [3] [4] [5].

- Table level caching is more suitable for static schemes.
- Query level caching can be used in dynamic schemes.

However, query level caching is still too coarse for "large" query results, and it 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 1993, Deshpande et al. proposed a chunk-based caching idea to deal with huge data. They divide results with a regular rule. Several researchers examined the benefits of chunks [2] [3] [6] as following:

1. **Granularity** - caching chunks rather than entire query result reduces the granularity of caching. This leads to better utilization of the storage space for caching in two ways. First, frequently accessed chunks of a query get cached. Chunks that are not frequently accessed will be replaced eventually. The second major advantage is that previous queries can be used much more effectively.

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 do not have to determine which of the cached queries should be combined to answer a new query.

3. **Closure property** - 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. The definition 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 rather than scanning the entire table.

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 reduces the effectiveness of memory available for caching. Chunk based caching eliminates this replication by sharing chunks containing overlapping results, thus allowing more queries to be cached.

2.3 Replacement Policy

All database systems retain disk pages in memory buffers for a period of time after they have been read in from disk and accessed by a particular application. The purpose is to keep popular pages memory resident and reduce disk I/O. The critical buffering decision arises when a new buffer slot is needed for a page about to be read in from disk, and all current buffers are in use: "Which page should be dropped from buffer?" This is known as the page replacement policy. The algorithm utilized by almost all commercial systems is known as LRU, for Least Recently Used, and this

name involved the type of replacement policy it impose. When a new buffer page is needed, the LRU policy drops the page from buffer that has not been accessed for the longest time.

3 Application Methodologies

3.1 Query Processing

First of all, we introduce the general query processing and the activities needed in each stage. The query processing we proposed is shown in **Figure 3.1**. We take advantage of the four engines, Query Analysis Engine, Query Splitting Engine, Computing Engine, and Replacement Engine. The Chunking Methodology and the Replacement Schemes will be discussed further later.

3.2 Query

3.2.1 Overview

In **Figure 3.1**, user submits a query through Graphic User Interface (GUI) and the query is transformed to SQL statement format before sending to the Query Analysis Engine. Then, the Query Analysis Engine parses the selection predicates to generate a list of chunk numbers required to answer the query.

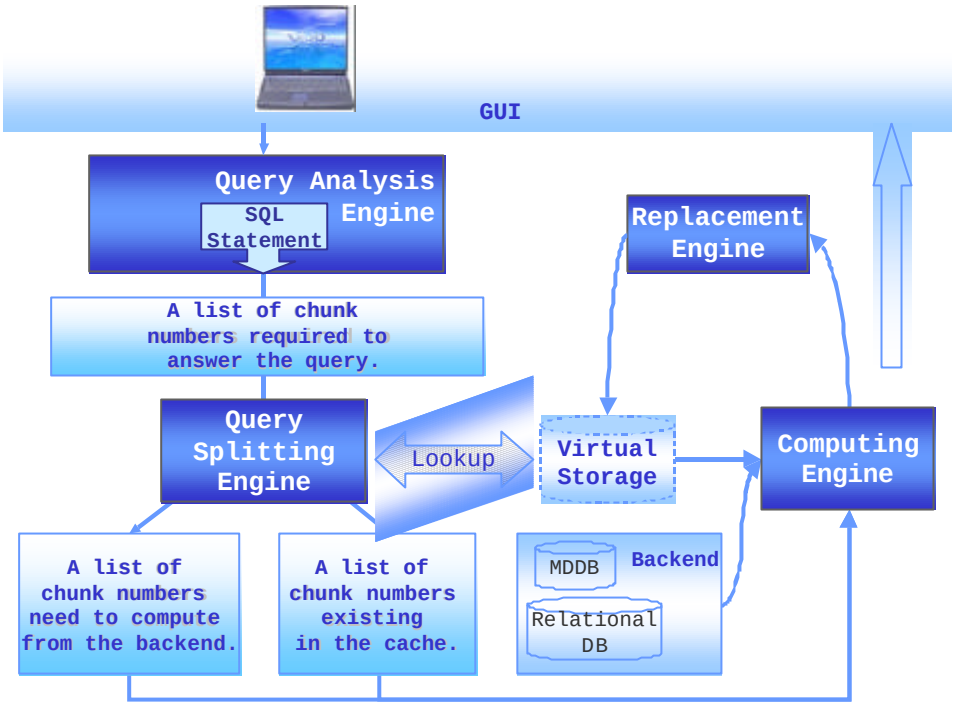

Figure 3.1: Query processing diagram.

The star join queries can be formalized to the "star template" (**Figure 3.2**), which is each query is a join of a Fact table with some dimension tables, filtered with some selections and followed by aggregates.

```
SELECT      〈dimension-list〉 , 〈aggregate-list〉
FROM        〈Fact Table〉 , 〈Dimension Table-list〉
WHERE       〈condition-clause〉
GROUP BY    〈dimension-list〉
```

Figure 3.2: A star template.

3.2.2 Query Analysis

In Figure 3.2, the *dimension-list* in the group-by clause determines the aggregation level of the incoming query. For the cached chunks at a lower level, it is necessary to aggregate them all and then we can obtain the results of the query. However, it is considerably complicated for computing the chunk set by aggregating

the chunks related to the current query may at different levels in order to get a result chunk. For simplifying our scale, only the cached results at the same level of aggregation as the query will be reused.

3.2.3 Query Region and Bounding Envelop

The attributes in dimensions mostly have the textual feature. Thus, the condition clause is with operators "=" combined in a Boolean formula. Deshpande et al [3] provides a good idea of chunking on dimension attributes, which divides the dimension values into groups and composes the multi-dimensional chunks with those sections. However, the proposed grouping mechanism leads to extra computations during the point-query processing.

To explain the extra computation, an example with drill-down may be helpful. In Figure 3.3, the basic storage unit in the cache is "chunk". A drill-down query is, from the viewpoint of a tree structure, to find children of a unique attribute value. Due to the grouping, or named ranging, the probability of the situation shown in Figure 3.3 is higher. Since this envelop is possible to not match entirely with the query results, post-processing is necessary to filter the extra tuples. Next, our recommended chunk-ranging manner will be presented, which can mitigate the extra computation problem.

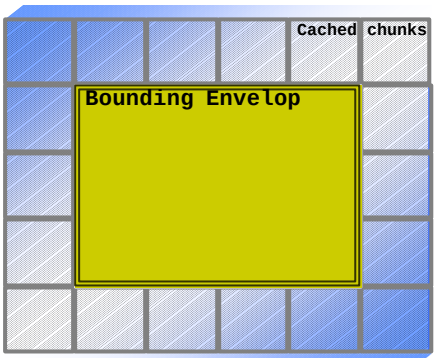

Figure 3.3: A bounding envelop.

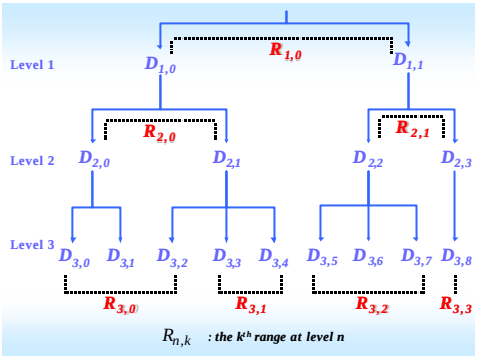

Figure 3.4: Ranges with *CreateChunkRanges*.

3.3 Ranging Chunks

3.3.1 Previous Work

Caching chunks method improves the granularity of caching. According to that chunk based caching scheme, it is still involving some extra tuples between the query region and the bounding envelop in answering a query with related chunks. This kind of extra tuples cause additional computations. And if these extra answers are not used later, the additional computations will become wastes. Accordingly, the smaller chunk ranges certainty can reduce the effort of extra computations.

Nevertheless, if the chunk ranges are too small, the total amount of chunks will be increasing and the load of overheads will be heavier. Moreover, the storage of chunk indexes requires more and results in the negative influences. We come now to the point at which it is necessary to deal further with disk I/O. For the I/O operations on the basis of page unit, it is reasonable to set chunk ranges with multiple of a page size. Thus, we can diminish the overheads for additional I/Os. What is immediately

apparent is that the chunk ranges at each hierarchy are according to heuristics.

Assume a dimension D has only one hierarchy with 3 levels. When we only want to get the aggregate of $\boldsymbol{D_{2,1}}$, the computation involved two ranges (i.e. $\boldsymbol{R_{3,0}}$ and $\boldsymbol{R_{3,1}}$), through *CreateChunkRanges* algorithm proposed in [3]. Actually, the number of attributes corresponding with $\boldsymbol{D_{2,1}}$ at level 3 is equal to the uniform ranges. It is rational to binding them with a single range, to improve the computation.

3.3.2 Correction

Figure 3.5 shows our proposed algorithm to generate chunk ranges for a dimension by the idea mentioned above.

```
Algorithm : RangingChunks
h = numbers of the hierarchical levels on the dimension
Divide level 1 into uniform ranges
For (l = 1 to h-1)
        For each value V_i at level l
        {
                Let R = range of values mapped to V_i at level l+1
                Divide range R into uniform ranges
        }
```

Figure 3.5: The *RangingChunks* algorithm

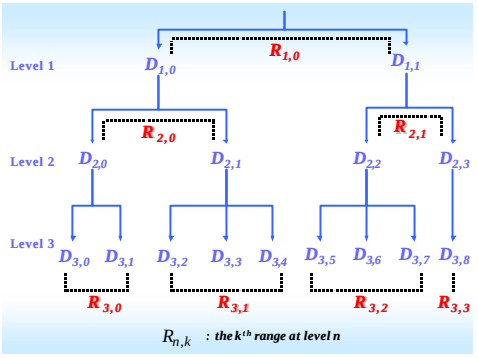

Figure 3.6: Ranges with *RangingChunks*.

In our chunk-based scheme, distinct value at one level should map to disjoint sets of ranges at a lower level. The following figure shows the chunk ranges assigned by using the *RangingChunks* Algorithm. Since the chunk ranges are not fixed, it is certainly to record the range number for each distinct value. This can be stored along with the ordinal number in the Domain Index.

3.4 Replacement Schemes

3.4.1 Overview

The purpose of cache replacement algorithms is to maximize the usability of the cache, be attempting to cache the most frequently referenced chunks. Simple LRU (Least Recently Used) is one of the options. However, it cannot reflect the usage rate of the chunks and this is very important. Thus, this kind of issues what we called frequency needs to be considered within our cache replacement policy. In the next subsection, the "benefit" considered in this paper will be introduced.

3.4.2 Replacement Benefit

An experimental validation of the cost model is shown in [9], and it expresses the linear relationship by the formula: $T=m*S+c$. There T is the running time of the query on a view of size S, c given the fixed cost, and m is the ratio of the query time to the size of the view. Since the aggregations are computed from the backend, the benefit of a chunk is measured by the fraction of the base table that it represents.

We take advantage of the basic benefit concept, which view benefits as cost savings, and define our benefit formula. For different group-bys, the numbers of chunks will also be different. The space cost to hold the chunks in the cache cannot be ignored. We use the numbers of chunks for the group-by representing the space factor.

However, when designing replacement schemes, the mainly concern is maximizing the overall benefit. As discussed above, our benefit rating can be formalized as:

$$Benefit\ Rating = \Sigma(Frequency * Saved\ Execution\ Cost) / Total\ Chunks$$

* *Saved Execution Cost* is the number of related tuples in the fact table.
* *Total Chunks* are the amount of cells in the cache. (3.1)

Frequency

Now, we probe into the computation issues on frequency. As is known to all, the definition of frequency is the occurred number of times during the unit period. We divide those computation methodologies into four classifications:

1. Referenced Number of Times／Fixed Period: The divisor is a fixed time scale *T*, while the dividend is the referenced number of times for a chunk during the past *T* period.

2. Total Referenced Number of Times／Total Period: Total Period means the time distance from the fixed initial time, configured by the query system, to current time. And the Total Referenced Number of Times is the retrieved times of a chunk since the beginning. This category is stretched from the previous one.

3. Fixed Number of Times／Passing Period: The dividend is the fixed referenced times, K; the divisor is the time period during the past K_{th} reference of a chunk.

4. Total Number of Times／Passing Period: The Passing Period mentioned here is from the first retrieved time of a chunk to, so far, the last one. And this method is the

extension of the previous categories.

The significant drawback of the first two frequency computations involving current time-point is that the updates lead to highly temporal cost. When it comes to the third type of computation, all the information of referenced time-points needs to be recorded. The narrow usage of that information is a defect. Consequently, these methods are extremely inefficient with large number of chunks.

In Total Number of Times／Passing Period, only the first referenced time and the total retrieved number of times require to be recorded. When there is an incoming query, what we have to do is update the information of related chunks. This approach is more efficient than the others. Equation **3.1** represents the frequency formula we used. The formula is used to compute the replacement information of a chunk only when it is referenced by the current query. l_i denotes frequency, K_i is the total referenced times of a chunk *i*, t is the current time and t_i is the time when chunk *i* is first referenced. Based on our observation above, we will rewrite the Benefit Rating formula as Eq. (3.2), where $|D_i|$ is the tuples of the base table and $|C_i|$ is the numbers of chunks for a group-by query *i*. This is our replacement criterion.

$$l_i = \frac{K_i}{t - t_i} \tag{3.2}$$

$$Benefit\ Rating = \frac{\sum_i l_i * |D_i|}{\sum_i |C_i|} \tag{3.3}$$

LRU Queue

However there are two important issues arise. When a chunk is referenced only once, its benefit will be infinity for the divisor equals to zero and it will occupy the cache space all the time. The second problem is the time-consuming of the policy. The entire replacement priority queue needs to be reordered as each query is processing. The reordering operation takes alarming time with a large-scale priority queue. Therefore, we proposed the LRU queue to cooperate with the replacement priority queue. In the LRU queue, the lesser usage a chunk has, the higher priority it is given. All the chunks are ordered in this queue, whereas, the ordering operation is so different from the replacement priority. There is no need to take as much time as replacement queue for reordering.

Furthermore, we advise a threshold T, which is used to pick the first T-th chunks in LRU queue. Then, only these picked chunks are considered in the replacement priority queue. In other words, we take account of the T-th least-recently-used chunks to be the list of candidates, and then use our benefit rating to decide which to be evicted. This fashion will reduce time requirement by a wide margin. In the section 4, we will aim at the analysis of the trade-off between saving time and hit ratio.

4 Simulation Experiment

4.1.1 Experimental Environment

The data source of the experiment is generated by APB-1 OLAP Benchmark Release II program, which is provided by OLAP Council [7]. The operating system is Microsoft Windows 2000 Server, while the database is Microsoft SQL Server 2000. In hardware, the CPU is 1G Hz Pentium 4 and the RAM is 1G Bytes DDR.

Out of the source data, only one star schema (Fig. 4.1) is used. The data statistics used below is collected in advance. Next, let us consider the hierarchies of dimensions.

4.1.2 Data Model

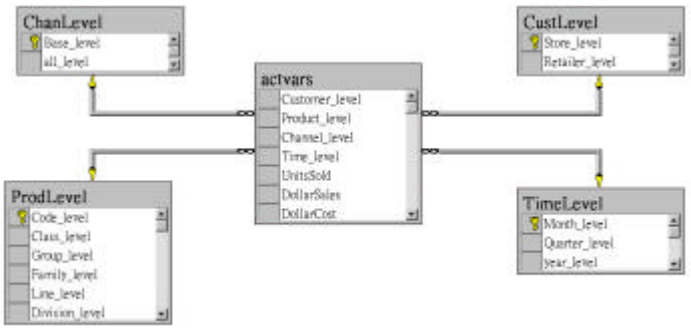

Figure 4.1: Star schema.

The logical database structure is made up of four dimensions: time, scenario, measure, and three aggregation dimensions that define the database size (product, customer, and channel). The tuples of these dimensions are shown in Table 4.1, and the total number of records in the fact table, sales, is 914,643.

	Product	Customer	Channel	Time
Tuples	9000	900	9	24

Table 4.1: Distribution of attributes.

4.1.3 Query Simulation

OLAP queries are ad hoc and very dynamic. The types of information requested span the full scope of the available data. Queries must be able to take advantage of the business relationships represented in the database. The time periods, products, customers, and channels must be dynamically generated from their respective hierarchies. Ten query classifications are formulated along with the distributions. Table 4.2 shows the distributions and the details can be found on http://www.olapcouncil.org/.

	1	2	3	4	5	6	7	8	9	10
Distribution	10 %	10 %	15 %	3 %	5 %	5 %	15 %	20 %	15 %	2 %

Table 4.2: Distribution of query classifications.

We produce 20,000 queries randomly but still match the distribution of query class, while the cache is assigned to hold at most 1000 records. The following figure shows the results of our experiment.

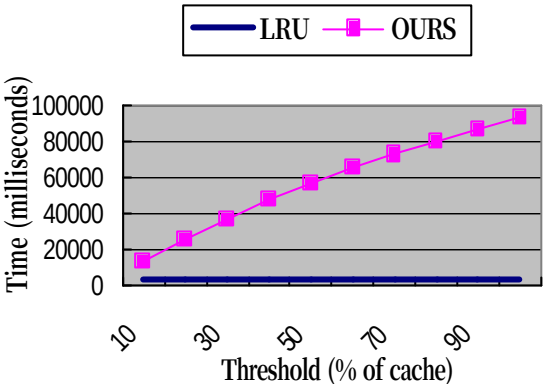

Figure 4.2: Execution time of policies.

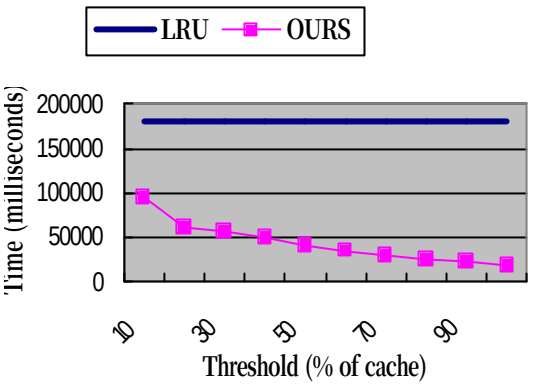

Figure 4.3: Computation time from backend.

The results show that our method spends more time on replacement determination while saves much more time on the computation for aggregations. And the Fig. 4.4 shows total time on replacement and computation. Obviously, the method we proposed cost lesser time than 60% of the time with LRUQ Replacement Policy. As shown in Fig. 4.5, our method has higher hit ratio if the threshold is greater than 0. Thus, we can say that our proposed methodology is a better one.

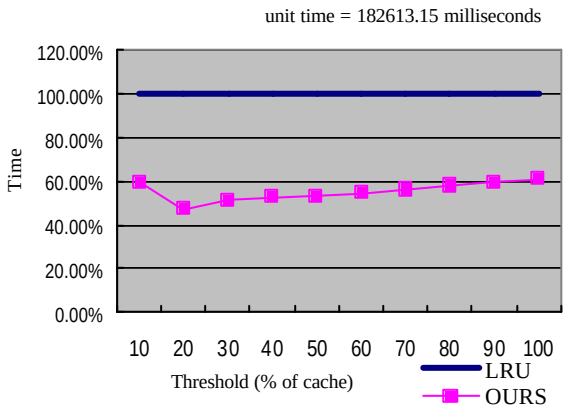

Figure 4.4: Total time cost of the two policies.

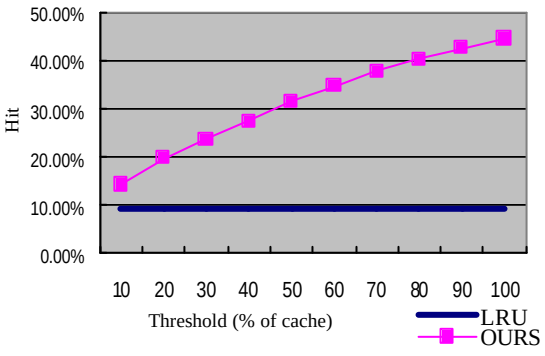

Figure 4.5: Hit ratio of the two policies.

Furthermore, the hit ratio curve of our policy is rising with thresholds, in other words, more inputs of the benefit rating formula causes more efficient caching. That figures out the benefit rating we proposed is a **good** scheme. We observe the trade-off between replacement

determining time and the missing chunks' computation time. Then find that our replacement policy will gain the most profit with the threshold 20%.

5 Conclusion and Future Work

We have introduced a new chunk raging rule for caching queries that works well in the query systems, where data is in multidimensional nature. Chunk-based caching allows line granularity caching, and allows queries to be partially answered from the cache. However, we propose a well suitable computation for frequency, which is used to be the weight of benefit rating. According to the benefit rating for each chunk, we can determine whether remain the chunk or evict it for placing a new one.

We also proposed a LRU queue to deal with the costly replacement priority computation. This method only takes the front portion, which is before what we called threshold, of least recently used chunks in the cache and calculates the benefit rating of these chunks to determine the replacement. We had proved the optimal threshold is 20% of cache, and the query response time is reduced via our replacement scheme.

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 signifies that the notion of chunk benefit has to be improved. Finally, the other possible enhancement is to consider the sparse data problem. Having an intelligent scheme to drop the spare data space, there will be more useful

aggregates resided in the cache.

References

[1] P. Roy, J. Vora, K. Ramamritham, S. Seshadri, and S. Sudarshan, "Query Result Caching in Data Warehouses and Data Marts", April 16, 1999.

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

[3] P.M. Deshpande, K. Ramasamy, A. Shukla, and J.F. Naughton, "Caching Multidimensional Queries Using Chunks", *Proc. ACM SIGMOD Int. Conf. on Management of Data, 259-270, 1998.*

[4] Y. Kotidis and N. Roussopoulos, "DynaMat: A Dynamic View Management System for Data Warehouses", *Proc. ACM SIGMOD Int. Conf. on Management of Data, 371--382, 1999.*

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

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

[7] OLAP Council APB-1 Benchmark Release II, November 1998, *http://www.olapcouncil.org/.*

[8] H. Gupta, "Selection of Views of Materialize in a Data Warehouse", *Proc. of the 6th ICDT, 98-112, 1997.*

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

[10] Y. Zhao, P.M. Deshpande, and J.F. Naughton, "An Array-Based Algorithm for Simultaneous Multidimensional Aggregates", *Proc. ACM SIGMOD Int. Conf. on Management of Data, 195-170, 1997.*

[11] A.Y. Levy and D. Suciu, "Deciding Containment for Queries with Complex Objects", *Proc. of the 16th ACM SIGMOD Symposium on Principles of Database Systems, 20-31, 1997.*

[12] G. Slivinskas, C.S. Jensen, and R.T. Snodgrass, "Adaptable Query Optimization and Evaluation in Temporal Middleware", *Proc. ACM SIGMOD Int. Conf. on Management of Data, 2001.*