Query Result Caching in Data Warehouses and Data Marts*

Prasan Roy[1] **Jinesh Vora**[1] **Krithi Ramamritham**[1]
S. Seshadri[†2] **S. Sudarshan**[1]

[1] IIT Bombay
{prasan,juvr,krithi,sudarsha}@cse.iitb.ernet.in
[2] Bell Laboratories, Murray Hill
seshadri@research.bell-labs.com

April 16, 1999

Abstract

In data warehouse and data mart systems, queries often take a long time to execute due to their complex nature. A single user may submit a sequence of related queries, and queries submitted by different users may also have commonalities. Query response times can be greatly improved by caching final/intermediate results of previous queries, and using them to answer later queries. Further, in a distributed environment, caching query results locally can also reduce communication time.

In this paper we describe a caching middleware system, which provides both the above functionalities. It can be used in centralized warehouse settings, as well as in distributed data mart settings. The system incorporates several novel features, revolving around a tight coupling of the query optimizer and the cache manager. In particular, the tight coupling allows us to implement better cache management techniques. The system also supports deferred updates, and the query optimizer chooses between incremental update of stored query results and recomputation in a cost-based manner.

1 Introduction

Data warehouses and On-Line Analytical Processing (OLAP) systems are becoming increasingly important parts of data analysis. The typical processing time of decision support and OLAP queries range from minutes to hours. This is due to the nature of complex queries used for decision making process. Data warehouses act as mediator between data sources and data consumers. Various OLAP, data mining, querying and reporting tools access data at the central warehouse. This keeps the warehouse heavily loaded. An alternative mart architecture distributes the load of the data warehouse across multiple data marts. Figure 1 shows three different warehouse architectures. The data marts architecture is getting popular since low end machines can be used to perform data warehouse activities. The aim of our project is to improve the query response time by caching query results for data warehouses and data marts. This is an *ongoing project* and we give an overview of our project in this paper.

*Work partly funded by a Govt. of India DST grant and a UPP grant from IBM. Prasan Roy's work was funded by an IBM PhD fellowship.

†Work primarily done while at IIT Bombay

In a traditional database engine, every query is processed independently. In case the query workload consists of sequence of queries with a lot of overlap, it is clear that this involves repetitive work across queries. A natural way to eliminate this redundancy is to reuse these (partial) query results across queries. These partial results are stored in an *associative cache* (logical cache). Caching can be used to reduce query computation costs(CPU/IO) as well as communication costs in a distributed architecture. Logical caching differs from standard page caching in two major ways.

- In page caching techniques physical units of data (a page) are used to cache data. In contrast, query results are cached in our scheme. The query optimizer must be extended to consider the possibility of using cached results of previous queries. We extend the multi query optimizer [RSSB98], which is based on a DAG representation as in Volcano optimizer [GM91].

- Cache replacement based on just recency of usage is not sufficient. We need to consider other factors such as size and cost of computation of query results. To make sure that only those query results are cached, which will give appropriate return on investment of caching data, we use cache admissibility criterion. Both cache replacement and cache admissibility are based on estimated benefits.

We propose a novel associative caching technique, where the query optimizer and the cache manager are tightly coupled together to perform better. We present a heuristic to select the most beneficial views(query results) to cache, using a benefit cost based approach. To consider which query results(or partial query results) to keep in the cache, we use representative set of weighted queries based on the queries in recent past. Based on the representative set, we compute the benefits of query/intermediate results, to decide on the best set of query/intermediate results to cache.

Typically, updates to the data warehouses are batched and applied on a periodic basis. Our architecture allows on-line update propagation to data warehouses and data marts. We attempt to perform on-line updates with least overheads and also making sure that the required updates are always propagated using the best alternative available.

We perform lazy/deferred updates when a stored query result that has been updated, is required to be used. Our optimizer then chooses the best alternative amongst throwing away the old result and recomputation of the cached result, throwing away the old result and assume it was not cached, and performing incremental update. We also make sure that we never use outdated stored results in cache.

Immediate/eager updates of cached queries may result in wasted effort, if the cached result is evicted before it is used, but it can reduce query latency. However, we note that updates can be applied ahead of time(eagerly), if idle cpu/disk time is available. Thus our solution subsumes immediate update also.

We are currently, implementing these techniques on top of the MQO we developed earlier. We use an existing commercial database to store data and perform query evaluation.

2 System Model and Assumptions

Figure 1 outlines three possible data warehouse architectures. In the figure, the data sources are typically transaction processing systems, such as inventory systems and sales processing systems. The eventual consumers of data are people, but other programs can be intermediate consumers of data. In a centralized architecture – the first case in the figure – data from all the sources is *materialized* in the warehouse. Typically, updates from the sources are batched and applied to such a centralized warehouse on a periodic basis. Further, consumers of data execute their queries at the centralized data warehouse.

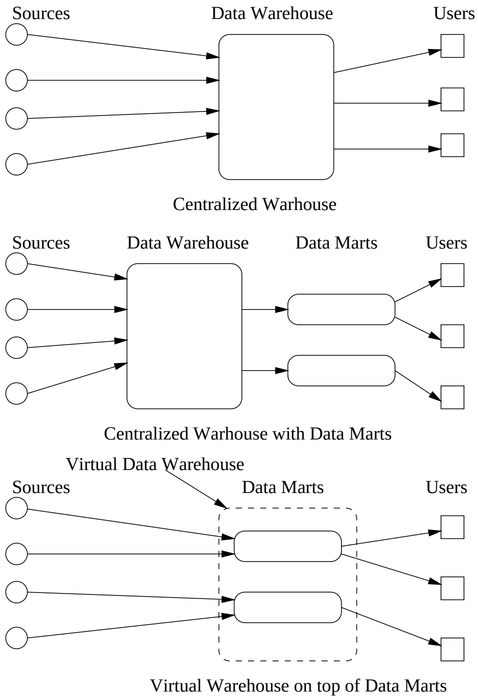

Figure 1: Data Warehouse Architectures

It is also possible to have (additional) data marts in a data warehouse architecture. The data marts store part of the data constituting the entire data warehouse. They may store data fetched from a centralized data warehouse – as in the second architecture in the figure – or they may store data from local sources – as in the third architecture. In the third architecture, the data warehouse is *virtual*, that is, there is no one site or machine that constitutes the warehouse. Rather, data marts communicate with each other, and each data mart presents a view to the user as though all data from all other data marts are available locally. In general, data marts are a practical way of providing data warehousing support to local users in a large distributed organization.

We present our work in context of a generic architecture with one or more servers(data marts) and a *cache middleware*, which performs query processing. There can be multiple instances of the caching middleware, but we assume they are entirely independent of each other. Queries are optimized and evaluated locally at the cache middleware, which has enough computational power and memory/disk for query optimization, evaluation and caching to occur locally. The cached data at each cache middleware can reside in the primary or secondary memory; this can be appropriately modeled by access cost functions. For simplicity, we assume that only one of memory/disk is used for the cache, i.e., the cache is not hierarchical. The cache is assumed to be of a known fixed size.

All update transactions are assumed to act at the servers. These transactions render the cached data at the cache middleware inconsistent with respect to the database at the server. We only consider the deferred (lazy) refresh policy, wherein the inconsistent views at the cache middleware are refreshed only when they are accessed in course of the evaluation of an input query. We allow for the option of incremental refresh of the cache in face of the above updates.

For simplicity of presentation, we assume that only a single query is processed at a time and updates and query processing do not happen concurrently. This is to sidestep issues related to concurrency-control which is orthogonal to the context of our work.

Our work is based on the data marts architecture. However, our technique can be used also in

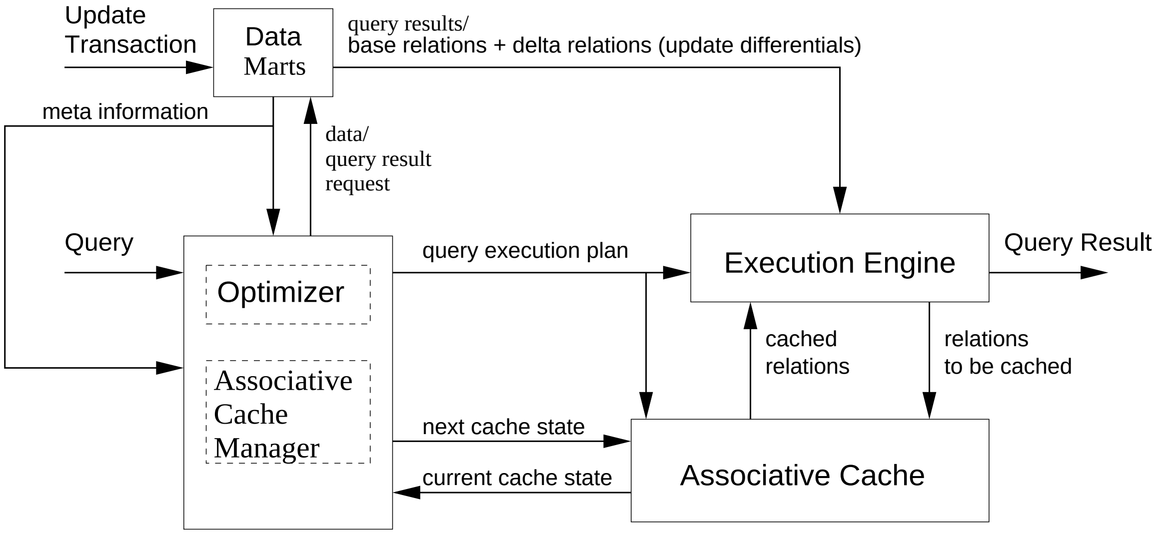

Figure 2: Architecture of Caching Middleware

centralized data warehouses. In fact, centralized data warehouse is a special case of this system.

3 Associative Caching: Overview and Related Work

The functional requirements of the associative cache are as follows. The associative cache, in general, should:

- interface with the query optimizer to come up with an efficient evaluation plan for the given query that makes effective use of the cached results;
- interface with the evaluation engine to
  - supply the cached result to the execution engine if specified by the evaluation plan;
  - if needed, cache partial results generated by the execution engine along with the semantic information(description of partial result);
- monitor the workload and adapt the cache contents appropriately;
- support invalidation or refresh of cached results in the face of updates to the underlying database.

For reference, we show the above interactions in Figure 2, which outlines the architecture of cache middleware. Static and dynamic maintenance of materialized views has been the focus of quite a lot of recent research [GM95, RSS96, KR99, CR94], much of which is relevant in the context of associative caching and will be covered later in this section.

3.1 Query Optimizer-Associative Cache Interface

In order to make effective use of cache contents, the optimizer should be cache-aware. This involves considering rewritings of the query in terms of the partial results present in the cache. We consider the viable alternatives. The optimizer produces a plan that uses cache contents, computing the query result from local and remote queries. *Local queries* are those computed from the cached data.

The subqueries that are computed at the server are called *remote queries.* The optimizer uses cost functions to decide which is the best alternative taking into account the query evaluation costs at the local and remote site, and communication costs.

There can be multiple rewritings of the same query in terms of the cache contents – different rewritings can use different sets of cached results. The optimizer should consider all possible rewritings (or a heuristic subset) and choose the cheapest alternative in a cost based manner. Some work in this direction is that of Chaudhury et al. [CKPS95], which extend the System-R dynamic programming algorithm to consider the search space of left-deep plans which can have the materialized views as the right input to the joins instead of base relations. An alternative is to extend the Volcano optimizer paradigm for considering cached materialized views. The multi-query optimization algorithm of [RSSB98] provides efficient ways of doing this. We use this algorithm in our system.

In order to limit the complexity of the problem, some earlier work has looked at simpler variants of the more general problem. We consider two such cases here. The first case involves caching only the results of selects on the base relations [DFJ+96]. Restricting to such simple expressions eases the task of expression lookup and query rewrite. The other alternative is to consider only a limited set of queries in the workload, for example cube slices (cube expressions with select predicates) [DRS98, KR99]. However, the general case is important, since many DSS queries are quite complex and cannot be handled by the above schemes. We consider this general case and perform multi query optimization with cached contents efficiently. We describe this in detail later. WATCHMAN [SSV96] checks for reuse in a trivial manner, with exact query match only. ADMS [CR94] performs query caching of SPJ(select-project-join) queries only. We handle more general class of queries.

3.2 Dynamic Adaptation to Query Workloads

Static version of caching would involve characterizing the entire workload apriori and using it to obtain a set of partial results that are permanently resident in the cache. [HRU96, RSS96, LQA97, Gup97] give a theoretical analysis of this problem and present heuristics for the same. These partial results are computed beforehand (alternatively, cached the first time they are evaluated by the execution engine). The major disadvantage of static cache contents is that it cannot cater to changing workloads. The data access patterns of the queries cannot be expected to be static. To answer all types of queries efficiently, we need to dynamically change the cache contents according to the current locality of reuse. Hence, a more intelligent cache would monitor the workload and adapt its contents appropriately. This involves tuning decisions regarding what new result to admit into the cache (called *admissibility policy*) and what to replace from the cache (called *replacement policy*) in order to accommodate a new result.

Taking the cue from the popular caching policies for paging systems, query caching systems proposed earlier [DFJ+96, KB96, SSV96, CR94, KR99], maintain statistics for each cached result, which is used to compute a replacement metric for the same; the replacement metric is variously taken as the cached result's last use, its frequency of use in a given window, its rate of use, etc. However, naively using the caching policies of paging systems for associative caches is not a good idea.

- The foremost problem is that in the paging systems, the objects cached
  - are of an uniform size and
  - have an uniform access cost.

Neither of these two properties hold for query results. There has been some amount of work in extending page caching policies to work in systems that cache objects of non-uniform size that have arbitrary sizes and access cost. The motivation has been in the context of associative caching of full query results [SSV96], caching of selections of base relations [DFJ$^+$96], cube slices [DRS98, KR99], etc. The common approach in most of these is to use access cost per unit size instead of access cost in the replacement policies.

- Further, in paging systems, the requests are in terms of pages which uniquely corresponds to a cached object. Thus, it makes sense to say whether a page has been referenced or otherwise, and so it is meaningful to maintain reference statistics for the pages in the cache and use the same for cache maintenance decisions. However, this is not the case in associative caches which cache intermediate results. A query can have a number of different rewritings in terms of the cached results. Which one is chosen depends on the optimizer. Evicting one cached result may still permit another to be used for the same query. Thus, we think that using query references to maintain statistics for the cached results is not good enough.

In our solution, the workload is monitored by maintaining statistics for the queries encountered in the recent past; this is the *representative set* of queries. Caching decisions are then influenced by this representative set.

The frequency of usage of a cached result depends on other cached results and the optimizer's choice, and is not fundamental as mentioned above. Instead, we make the choice of what to store as follows. For each of the cached query result we have a notion of cost benefit for the representative set of queries, and at a given point of time we only keep the most beneficial set of query results that satisfy the cache size constraints.

There has been some previous work to improve cache replacement and admissibility decisions by [SSV96], where they consider exact match for reuse with only complete query results and not the intermediate results. [SV98] has considered caching of intermediate results in context of distributed systems, to check reuse by query rewrite, after the plan is generated. They do not retain intermediate results in cache as we do, once the query has been answered. [KR99] performs caching of query results and try to dynamically adjust to changing workload, but they only consider cube queries.

4 Update Propagation

Updates at the servers are propagated to the cache middleware in a deferred manner. The inconsistent views in the cache are refreshed incrementally. This is done as described below.

4.1 Invalidation/Refresh of Cached Results

Recall from Section 2 that we assume that updates do not occur concurrently with query execution. As such, the only effect of the updates would be that some of the results currently in the cache would get inconsistent with the underlying database. The inconsistency can be detected either by having the cache poll data sources or having the data sources send invalidation messages. Which one is used does not affect the rest of our discussion. When a query evaluation plan decides to use a cached result that is out of date, two things are possible:

1. recomputation of the cached result, and
2. performing incremental updates of the cached result.

Yet another possibility is to throw out the result affected by updates, and use a different plan. The optimizer estimates these costs and chooses the cheaper alternative as we will see. There has been plenty of work on incremental materialized view maintenance, part of which can be applied in this restricted context (see Gupta and Mumick [GM95] for a comprehensive survey). For this purpose, the server supplies the update differential ("delta") relations corresponding to the updates which can be used by the execution engine to refresh the cache contents incrementally.

Our solution provides for deferred (lazy) incremental refresh of the cache. In fact, we account for view updation/refresh cost in the query plan which uses an outdated view. So, if a view is expensive to update/recompute (whichever is cheaper) and does not have comparable benefit of being retained in the cache, it gets evicted. Furthermore, our solution enables us to share updation work for different views triggered by different queries (a consequence of lazy refresh) over a period of time.

We give details of our solution in the next section.

4.2 Deferred vs Immediate Updates

In the immediate update policy, all the cached results that depend on the updated relation at the server need to be updated immediately, typically as a part of the updating transaction. Naturally, immediate maintenance imposes significant overheads on the update transactions. Further, notice that this policy requires an active server with a notification facility; no such assumption is made for the deferred materialization policy.

To support the deferred update policy, updates to a base relation are logged in an associated differential ("delta") relation at the server. (The differential relation can possibly be a virtual or a materialized view on the system log.) This differential relation is used to refresh the cached intermediate relation. The details of what auxiliary information to store in the base relation tuples and the associated differential relations, and how, using these, the above two steps are carried out are specified by the incremental update algorithm being used (cf. Gupta and Mumick [GM95])

With each tuple in the differential base relation, we store a timestamp of the instant when the corresponding update occurred. Also, each cached result stores a timestamp of the instant it was materialized or last refreshed. The timestamp is used to determine which tuples in the differential relations are required for the refresh of the cached result.

The system typically cannot allow the differential relations (or logs) to grow unboundedly; they need to be truncated if they grow to an unacceptably large size. This truncation might rule out the possibility of incremental refresh of some inconsistent cached results; the timestamps are needed to figure this out. In such a case, recomputation is the only option for refreshing the view.

The immediate policy can be emulated by eagerly refreshing cached results, using the same mechanisms as for the deferred policy. As mentioned earlier, we may use idle time of CPU/disk for performing eager updates. However, the deferred recomputation may actually be cheaper.

5 Associative Cache Management

Section 5.1 discusses how the cache manager characterizes the workload at any given point in time. The manner in which the above characterization is used to arrive at cache maintenance decisions is discussed in Section 5.2. We next present our approach to deferred refresh of inconsistent cached results in Section 5.3. Our approach is well integrated with the rest of the cache manager architecture and results in certain novel features being available for free. Since the proposed architecture takes a cost based approach to cache management, we find that the cache manager can be integrated with the optimizer.

5.1 Monitoring Workload

In this section, we describe the mechanism using which the cache manager adapts the cache maintenance decisions to the current workload.

The simplest way of representing the workload is to keep the last N queries. Large values of N may increase the cost of cache maintenance decisions. Some possibilities for reducing the number of queries in the representative without affecting the quality of caching decisions greatly are as follows:

- we can discard queries having little overlap with others
- we can replace similar queries by parameterized queries, eg., $\sigma_{A=5}(E), \sigma_{A=6}(E), \sigma_{A=9}(E)$, can be replaced by $\sigma_{A=*}(E)$
- we can keep only frequently occurring queries
- we can also give different weightage to different queries in the representative set, eg.,more recent or more frequent queries can be given higher weightage.

We are currently studying these alternatives.

5.2 Cache Maintenance

The decisions on what results need to be kept in the cache at an instant is made using a cost-benefit analysis. The representative query set (cf. Section 5.1) at an instant is used to decide on the utility of the results that can potentially be cached at that instant, For the purpose of this section, we consider the cache as consistent throughout. The extensions due to integrating cache coherence into our framework will be outlined in Section 5.3.

Our objective is to determine what results to maintain in the cache at each instant so that the overall cost of evaluating the input stream of queries gets minimized. A crucial issue here is to prevent unnecessary overheads. We think it is a bad idea to eagerly compute a result for the sake of caching; it is clearly so in case the result incurs equal cost when evaluated eagerly or lazily since

- eager evaluation does not lead to any savings in the overall cost, and
- space that could be utilized to cache some other result in the interim gets used up.

Given these considerations, we restrict our choice of results to be considered for caching at any instant to a *candidate set* that consists of:

- the set of results computed during the evaluation of the query being processed at that instant; and
- the set of results already in the cache.

At an instant, given the candidate set of queries and the representative set of queries, we need to decide on what subset of the candidate set of results to cache (equivalently, which equivalence nodes to mark) given the cache space constraints, so that the savings are maximized for the representative set of queries at the given instant.

We define a cost metric $W(S,t)$ on a given instant t as

$$W(S,t) = \sum_{Q \in R(t)} wt(Q,t)C(Q,S)$$

where $wt(Q, t)$ is the weight of Q at the instant t, $R(t)$ is representative set at t, and $C(Q, S)$ is the best cost (see discussion below) of evaluating the query Q given containing all results in S. The problem of choosing the cache configuration at an instant t can therefore be stated as that of selecting the subset S of the candidate set that minimizes $W(S, t)$ while satisfying the space constraints. A greedy heuristic for the above problem is stated next.

A Greedy Heuristic: With each unselected intermediate query result in the candidate set, we associate a *benefit*, defined as the reduction in $W(t)$ if this query result is additionally cached. The greedy heuristic starts with all query results unselected and iteratively selects the result in the candidate set with the maximum benefit per unit space (that is, the benefit of the node divided by the estimate of the size of the corresponding result) till either all the results in the candidate set are selected or the next result chosen for selecting will cause the total space occupied by the selected query results to exceed the cache size.

A naive approach for implementing the above would be to invoke the Volcano optimizer every time the benefit of a query result needs to be determined. This can clearly be very expensive. In the paper on multi-query optimization [RSSB98], we have proposed a bottom-up propagation strategy that can be used to find the benefits of the query results in an incremental manner. Substantial improvements on efficiency can further be obtained if we assume that the benefit per unit space of a query result does not increase as other query results are selected. This is based on an incremental greedy heuristic proposed and evaluated in [RSSB98].

5.3 Refreshing Cached Results

In this section, we consider the important issue of keeping the cache consistent in presence of updates to the underlying database. As stated in Section 2, we consider only deferred maintenance of the cached results with respect to the updates at the server. Further recall that there is a timestamp associated with each cached result giving the time at it was computed or last refreshed; and that the state of the underlying database remains stable during the period a query is being processed.

An outdated cached result can be updated in one of the following ways:

- recompute it from scratch,
- compute the differential for the result from the updated underlying relations and their differentials, or
- a combination of the above, obtained by following the one of the previous two alternatives for some parts of the plan and the other for the rest.

Depending upon the extent of update and on what else is stored in the cache, any of the above could be cheaper. Further, in case multiple results need to be refreshed, the refresh plans for these results should be able to exploit the overlap and avoid repeating work by carrying over partial results of the differential computations. Thus, it is required that the space of refresh plans be searched in a cost-based manner to determine the best alternative.

Our solution rests upon modeling cache refreshes as subquery evaluations so that optimizer considers the space of refresh plans for the referred outdated cached results in an integrated manner with the space of evaluation plans of the input query.

We model the operation of incremental refresh by the *refresh operator* which is defined only for cached relations. This operator takes as input the differential relation and results in the refreshed relation (the outdated result is an implicit input). The refresh operator does a selection of data from "delta" relations based on timestamps.

Observe that as the differentials of underlying base relations for a cached result increase in size due to ongoing updates, the incremental refresh cost for these results increases. This enables the cache manager to factor in the impact of update rates of base relations in the cache maintenance decisions. Notably, a cached result may run out of favor if it has high maintenance overheads, and may eventually get replaced without getting refreshed. This is a consequence of the deferred maintenance policy.

Maintenance cost is estimated as the cost of computation of the differential from the delta relation. This cost is the cost estimated by the optimizer, to implement the refresh operator. The benefit of caching the query result can then be computed as:

$$B(Q,t) = wt(Q,t)C(Q,S) - M(Q,t)$$

where $M(Q,t)$ is the maintenance cost of Q at time t.

6 Extensions

In this section. we consider certain extensions to the proposed architecture. We state the main ideas and present partial solutions and a discussion of the relevant issues for each extension considered below.

6.1 Discarding Parts of Query Results

In order to have a more effective utilization of the cache pool, a possibility is to consider how to discard parts of views while retaining others. For example with selection, we can retain a selection on a view and discard the rest. So for a cached view we can apply a selection condition over the view and partition the relation horizontally. Then consider this new view description instead of the original one. For example, if we have a view

```
select prod_type, prod_id, sales_amt
from sales, product
group by product_type, prod_id
```

we may replace it by

```
select prod_type, prod_id, sales_amt
from sales, product
group by product_type, prod_id
having product_type < 10
```

There are issues in introducing these partitioned nodes:

- what nodes do we introduce, what attributes do we select on etc.;
- how to maintain statistics and for what partitions, etc.

6.1.1 Replacing data by Aggregates

Another possibility to save cache space is to keep aggregates of cached result as compared to the whole result. This can be done provided the benefit of aggregation is good and the cost of computing it is less. For example, replace

```
select prod_type, prod_id, sales_amt
from sales, product
```

by

```
select prod_type, prod_id, sales_amt
from sales, product
group by product_type, prod_id
```

The issues discussed in the context of selection also apply here. These are exceptions to our general rule of no eager computations.

6.2 Sharing Query Processing Between the Data Marts/Warehouses and the Caching Middleware

In the current system architecture, the server provides the base relations to the client and all the query processing is performed at the client. At times it may be more efficient to ship some data from the client to the server, perform evaluation of the partial query result and ship the partial result back to the client. This will then be used to generate the final result. Consider for example, the query is $A \bowtie B \bowtie C$, and relations A and C are cached. Also say that the size of relation A is much lesser than the size of relations B or C. In such a case, it would be more efficient to ship relation A to the server, compute $A \bowtie B$ at the server, and ship that result back to the client to compute the final result, $A \bowtie B \bowtie C$. The obvious benefit here is that amount of data transfer is reduced, and hence the total cost of computing the query result. Thus distributing the query processing between client and server can improve performance. This can be done by appropriately modifying the cost functions of the query optimizer. This also increases the search space of the optimizer. So it is necessary to evaluate whether such an overhead during optimization is fruitful.

6.3 Concurrency Control

When we consider a complete virtual data warehouse, data is distributed at various data marts, and query results have to be evaluated in the view of updates at these data marts. So we need to propagate updates from the various data marts concurrently. Then it would be more practical to relax the restriction that updates and query evaluation cannot occur concurrently. To handle such a scenario, we need to invalidate all the transactions that were active and accessed the updated data. For this to be possible we need to indicate the updates immediately, even though the updates would be propagated only if required in a deferred manner. An update notification that does not affect any of the active transaction can be used to mark the data in cache to be invalid, and then perform update propagation when required. This is one solution. We need to consider more alternatives to this, evaluate the different possibilities and come up with a strategy that allows update propagation with minimal overheads.

7 Project Status

We are currently working on algorithms for cache management and extending the multi-query optimizer to handle updates. Even though the basic algorithm is clear we need to apply appropriate heuristics to reduce the search space of the optimizer. We are also implementing the system.

The basic architecture for implementation is as follows. Figure 2 shows the components of the system. The optimizer/cache manager is implemented by extending the multi-query optimizer.

We are using a commercial database to maintain cache data. We tune the database to switch off logging so that there are no unnecessary logging overheads. We can optionally tune the database to keep the data in memory, such that it can then behave as a main memory cache with query evaluation, and data caching features. We use standard ODBC interface to communicate with servers.

8 Conclusion

In this paper we have considered the general problem of answering queries efficiently in data warehouses and data marts, by means of caching. A novel cost based technique that solves the problem of dynamic query caching for data marts has been presented. The advantages of our technique is that there are low overheads, since the cache manager and query optimizer are tightly coupled. Efficient computation of benefits is done using incremental computation techniques proposed as part of the multi-query optimization technique of [RSSB98]. The reason that we can adapt to dynamic workloads for the general caching problem is because, unlike other query caching systems we also cache query plans, and integrate the incremental refresh of cached views with the cached representative plan. We have seen why deferred, incremental updates, based on the decisions made by the optimizer provide a good way handling on-line updates.

We are working to prune the optimizer search space further, by selecting a subset of the past queries for deciding the representative set of queries. Extending our system for concurrency control is also important. We are also considering how to distribute the task of query evaluation across data marts to reduce query response time and communication costs. We believe we can easily extend our optimizer to consider distributed query evaluation plans.

References

[CKPS95] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. In *Intl. Conf. on Data Engineering*, Taipei, Taiwan, 1995.

[CR94] C. M. Chen and N. Roussopolous. The implementation and performance evaluation of the ADMS query optimizer: Integrating query result caching and matching. In *Extending Database Technology (EDBT)*, Cambridge, UK, March 1994.

[DFJ+96] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Intl. Conf. Very Large Databases*, pages 330–341, Mumbai,India, 1996.

[DRS98] Prasad M. Deshpande, Karthikeyan Ramaswamy, and Amit Shukla. Caching multidimensional queries using chunks. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 259–270, 1998.

[GM91] Goetz Graefe and William J. McKenna. Extensibility and Search Efficiency in the Volcano Optimizer Generator. Technical Report CU-CS-91-563, University of Colorado at Boulder, December 1991.

[GM95] Ashish Gupta and Inderpal Singh Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. In *Data Engineering Bulletin*, pages 3–18, 1995.

[Gup97] H. Gupta. Selection of views to materialize in a data warehouse. In *Intl. Conf. on Database Theory*, pages 98–112, 1997.

[HRU96] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. In *SIGMOD Intl. Conf. on Management of Data*, pages 205–216, Montreal, Canada, June 1996.

[KB96] Arthur M. Keller and Julie Basu. A predicate-based caching scheme for client-server database architectures. In *The International Journal on Very Large Databases*, volume 5, pages 35–47, January 1996.

[KR99] Yannis Kotidis and Nick Roussopoulos. DynaMat: A Dynamic View Management System for Data Warehouses. In *SIGMOD Intl. Conf. on Management of Data*, 1999.

[LQA97] W.L. Labio, D. Quass, and B. Adelberg. Physical database design for data warehouses. In *Intl. Conf. on Data Engineering*, 1997.

[RSS96] Kenneth Ross, Divesh Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *SIGMOD Intl. Conf. on Management of Data*, May 1996.

[RSSB98] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. Technical report, Indian Institute of Technology, Bombay, October 1998.

[SSV96] P. Scheuermann, J. Shim, and R. Vingralek. WATCHMAN: A Data Warehouse Intelligent Cache Manager. In *Intl. Conf. Very Large Databases*, pages 51–62, Mumbai,India, 1996.

[SV98] Subbu N. Subramanian and Shivakumar Venkataraman. Cost based optimization of decision support queries using transient views. In *SIGMOD Intl. Conf. on Management of Data*, Seattle, WA, 1998.