Dynamic Caching of Query Results for Decision Support Systems

Junho Shim*
Dept of Electrical and Computer Engineering
Northwestern University
Evanston, IL 60208, USA
shimjh@ece.nwu.edu

Peter Scheuermann
Dept of Electrical and Computer Engineering
Northwestern University
Evanston, IL 60208, USA
peters@ece.nwu.edu

Radek Vingralek
STAR Lab
InterTrust Technologies Corp
460 Oakmead Parkway
Sunnyvale, CA 94086, USA
rvingral@intertrust.com

Abstract

The response time of DSS (Decision Support System) queries is typically several orders of magnitude higher than the response time of OLTP (OnLine Transaction Processing) queries. Since DSS queries are often submitted interactively, techniques for reducing their response time are becoming increasingly important. We argue that caching of query results is one such technique particularly well suited to the DSS environment. We have designed a query cache manager for such an environment. The cache manager can lookup query results from the cache either based on an exact query match or using a *query split* algorithm to efficiently find query results which subsume the submitted query. The cache manager dynamically maintains the cache content by deciding whether a newly generated query result should be admitted to the cache and if so, which query results should be evicted from the cache to free space for the new query result. The decisions are aimed at minimizing the query response time. The decisions are explicitly based on a cost function that considers the execution cost of each query, the size of each query result, the reference frequency to each result, the cost of maintenance of each result due to updates of the base tables, and the frequency of such updates. Experimental evaluation shows that our cache manager can improve performance on TPC-D like workloads.

1 Introduction

Data warehouses are usually dedicated to the processing of data analysis and decision support system (DSS) queries [8, 7]. Unlike online transaction processing (OLTP) queries which access only a few rows in each table, DSS queries make a heavy use of aggregation and thus are much more complex and need to access a substantial part of the data stored in the warehouse. Consequently, the response time of DSS queries is several orders of magnitude higher than the response time of OLTP queries.

Since in many cases users submit the DSS queries interactively, it is important to reduce the DSS query response time. Compared to OLTP systems, data warehouses are relatively static with only infrequent updates [8]. In addition, DSS queries typically retrieve relatively small results containing aggregate data such as averages, sums, counts, etc. Therefore, query result caching is particularly well suited to the DSS query processing environment.

Testing whether the cache content can be used to process a query is algorithmically difficult, since it requires a test for query subsumption, which is NP-hard [13]. However, in many special cases, the subsumption test can be performed efficiently. For example, a containment test based on a syntactic exact match has been described in [9]. Harinarayan, Rajaraman and Ullman [7] developed a fast containment test for a special class of queries, *data cubes* that are aggregate queries with empty `WHERE` and `HAVING` clauses.

We believe that a practical implementation of a query result cache manager can take advantage of the fact that most queries submitted to DSS systems are not in completely arbitrary form. For example, many DSS tools [4, 8] do not generate complex, nested queries, but rather a series of simple aggregate queries which reuse each other's results via creation of temporary tables. Furthermore, physical design of many data warehouses often follows a star schema [8], which further increases regularity of the queries submitted against data warehouses. We define a *canonical form* of a query which we argue is sufficiently generic to fulfill the needs of most applications, but at the same time allows a fast containment testing.

In this paper we describe a design of an intelligent

*Current affiliation: Computer Associates Intl, One Computer Plaza, Islandia, NY 11788, USA query result cache manager for DSS environment. The cache manager looks up query results from the cache based either on an exact query match or using a *query split* algorithm if the query is in the canonical form. The cache manager dynamically maintains the cache content by deciding whether a newly generated query result should be admitted to the cache and if so, which query results should be evicted from the cache to free space for the new query result. The decisions are aimed at minimizing the query response time. The decisions are explicitly based on the execution cost of each query, the size of each query result, the reference frequency to each result, the cost of maintenance of each result due to updates of the base tables, and finally, the frequency of such updates. Unlike similar systems described in the literature [7, 9, 14, 1, 5], the cache manager maintains the subsumption relationship between both materialized and non-materialized query results to improve the efficiency of cache replacement and admission algorithms. Experimental evaluation of our cache manager shows that on a TPC-D like workload the cache manager can improve performance of similar systems on average by at least 37% and by as much as 70% on a skewed workload.

System Overview

We present a high level overview of the cache manager design and functionality. We will refer to various system components whose relationship can be found in Figure 1.

When the cache manager receives a query, the query transformer (*split process*) examines the query if the query is in the canonical form. If it is not, the cache manager tries to look up a query result in the cache based on an exact query match. If, on the other hand, the query is in the canonical form, it is then transformed according to the split algorithm. The cache content is examined for query results that may subsume the transformed query. If several candidates are found, a selection is made so that the average query response time is minimized.

The relationship between the transformed query and the selected result set is maintained using a *query attachment graph* in metadata storage. The graph also maintains the relations between the query result and other non materialized result sets which may benefit from the given query result should it be materialized in the cache. Such relationship is necessary for the cache replacement and admission algorithm to determine which query results should be materialized in the cache. The cache replacement and admission algorithms dynamically select query result sets for caching so that the average query response time is minimized. Both algorithms make use of a "profit metric" that considers with every query result its average rate of reference, its average update rate, its size and the benefit of cost saving for the processing of associated queries which are stored in metadata storage using the query attachment graph. The cache replacement also uses a complementary cache admission algorithm to determine whether a retrieved set currently not materialized in the cache should be admitted into the cache. The cache admission algorithm materializes a new query result in the cache only if it increases the overall profit. Whenever cache replacement occurs, the query attachment graph is also updated to reflect the new relationship among queries. Garbage collection process is employed at some intervals to prevent the graph from having many useless nodes and incident links.

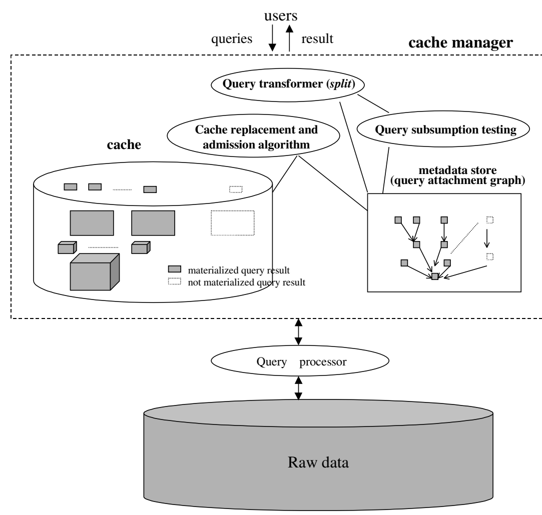

Figure 1: **Overview of our cache manager**

The remainder of this paper is organized as follows. In Section 2 we review the related work. In Section 3 we define the canonical form of a query and describe the query split algorithm. In Section 4 we describe the dynamic cache replacement and admission algorithms. In Section 5 we present the results of an experimental performance evaluation of the cache manager. Finally, Section 6 concludes our paper.

2 Previous Work

Answering queries from the materialized query result sets was studied in [2, 6, 3, 11]. Query subsumption for restricted forms of queries without grouping and aggregation were presented in [2]. Answering queries with aggregation using views has been described in [6, 11, 5].

[6] shows a syntactic transformation of the query to test whether the view definition is identical to the definition of the query. [11] illustrates how fractions of materialized views can be used to answer other queries. However, their algorithms for the query subsumption appear to be too expensive to be used in our cache manager in that the studied class of queries is more general than ours. In [5] queries are answered using fixed sized chunks. The chunks can be either cached or computed from raw tables. As the authors point out, determining the right size of a chunk is critical for success of this approach.

The performance of any cache manager is critically dependent on its cache replacement and admission algorithm. So our initial work concentrated on the design of an efficient cache replacement and admission algorithm for query results [9]. It is shown in [9] that a profit metric which considers for each retrieved query set its average rate of reference, its size, and cost of execution is much more effective for caches in a warehousing environment. However, [9] does not consider any query transformation for subsumption testing. It employs an exact query match for arbitrary forms of queries. Also the profit metric is limited in that it neither considers the subsumption dependency among queries nor the maintenance cost of the cached result sets.

Selective materialization of data cubes was studied in [7, 14, 1]. All these algorithms use greedy heuristics based on various benefit functions to minimize the query execution time, and select a static set of data cubes to materialize. [7] chooses a static set of data cubes using a linear cost model without considering the size of the data cubes. Furthermore, all queries are assumed to be referenced with an equal probability and update costs are not considered. The algorithms described in [14, 1] are closer to ours in that they consider reference frequency and maintenance cost of data cubes. Yet, the cost models do not consider size of the materialized views and they do not provide a mechanism for relating views to each other. The performance of [7, 14]'s algorithms is compared to ours and the results are shown in Section 5.

3 Query Split Process and Query Attachment Graph

3.1 Canonical Form of the Query and Query Split Process

The DSS queries we are interested in have a common structure. They *join* tables and *group* attributes in their SQL representations as follows:

select *select_list*, *agg_list*
from t_1, t_2, ... , t_n
where *join-condition* AND *select-condition*
group by *group-by_list*
having *group-condition*; (This clause is optional.)

where *select_list*, *agg_list* are column lists in `SELECT` clause, t_1,...,t_n is a list of tables, and the *join-condition* is a list of equality predicates among columns in tables t_1,...,t_n connected by `AND`. The *select-condition* clause consists of a boolean combination of predicates in which each predicate involves only a single attribute and contains only one of the range comparison operators $\{<, >, \geq, \leq, =, \neq,$ `between` $\}$. The *group-by_list* is the list of attributes over which aggregations are performed and the *group-condition* is the optional condition clause for `group by` clause.

Queries in the above form are called *canonical* queries. The class of canonical queries is fairly broad. For example, twelve of the total of seventeen TPC-D queries have a variant in a canonical form [12]. As an example, a query $Q1$ shown in Figure 2 is in a canonical form with the following setting.

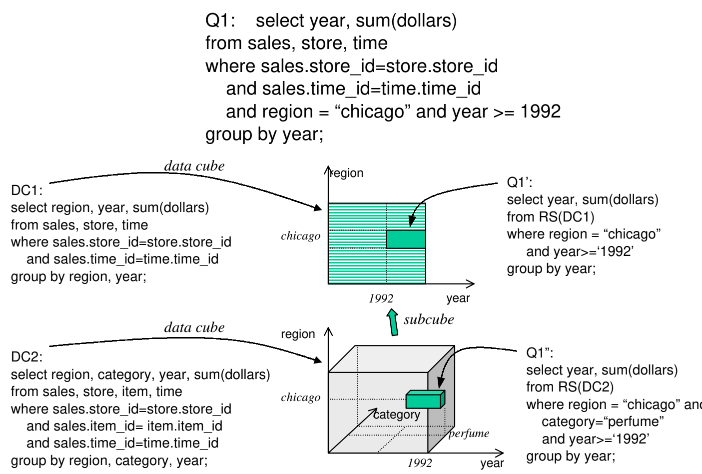

Figure 2: A canonical query and data cubes

select_list: `year`, *agg_list*: `sum(dollars)`
t_1: `sales`, t_2: `store`, t_3: `time`
join-condition: `sales.store_id = store.store_id AND`
`sales.time_id = time.time_id`
select_condition: `region = "Chicago" AND year >= 1992`
each predicate contains only one attribute, `region` and `year` respectively, and each predicate has only one range comparison operator, $=$ and $\geq$, respectively.
group-by_list: `year`, *group-condition*: `NULL`

Similarly, all data cube queries are in canonical form. Since the *select-condition* and *group-condition* of data cube queries are empty, it is sufficient to use the list of *group-by* attributes and *agg_list* to identify a specific data cube. Data cubes are related in a derivability relation that forms a lattice structure [7]. Figure 3 shows a lattice of data cubes. The *agg_list* `sum(dollars)` is omitted for readability, and `none` is used to represent the data cube that gives the total aggregation with no group-by attributes. An *immediate alive ancestor(IA)* is the ancestor that was already materialized in the cache and there is no other materialized ancestor between the given data cube and IA along the lattice hierarchy. For example, if neither `(region, year)` nor `(region, category)` are materialized but `(region, category, year)` is found in the cache, `(region, category, year)` is the immediate alive ancestor of `(region)`.

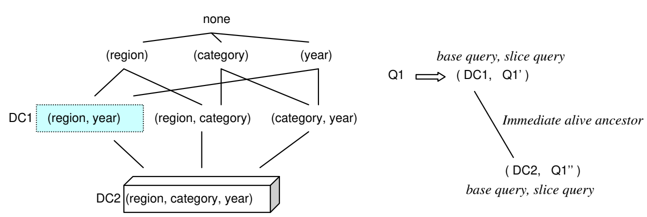

Figure 3: An example lattice and base, IA, and slice queries

A data cube whose cells contain the aggregations over all the attributes appearing in the *select-condition* of a query can be used for the evaluation of the query. Moreover, if the corresponding data cube is not yet materialized in the cache, the lattice structure can be exploited to determine its immediate alive ancestor(IA) so that the given query can be evaluated on the found IA. This entire process is called *split* (Figure 4) and its details follows.

Given a canonical form query Q, the *query split* algorithm transforms Q into two queries Q_1 and Q_2 ($Q \Longrightarrow (Q_1, Q_2)$). The result set of Q_1 is a data cube and Q_2 is a rewrite of query Q so that it can be evaluated using the result set of Q_1, RS_1.

Q_1:	select *select_list*$_1$, *agg_list*	Q_2:	select *select_list*, *agg_list*
	from t_1, t_2, $\ldots$, t_n		from RS_1
	where *join-condition*		where *select-condition*
	group by *group-by_list*$_1$		group by *group-by_list*

where *select_list*$_1$ and *group-by_list*$_1$ are column lists in which each column name is resolved to be one-to-one mapped to a column name appearing in *select-condition*. We call Q_1 a *base query* and Q_2 a *slice query*. In the previous example, $Q1$ can be transformed into ($DC1$, $Q1'$) where *select_list*$_1$ and *group-by_list*$_1$ are `(region, year)`. (See Figure 2).

$Q \Longrightarrow (Q_1, Q_2)$ where Q_1 is a data cube query.
base query $\leftarrow Q_1$, *slice query*$_1$ $\leftarrow Q_2$;
$i \leftarrow 0$, $Base_i \leftarrow \{Q_1\}$;
while no base query in $Base_i$ is materialized in cache **begin**
  $i \leftarrow i + 1$;
  for each $Q \in Base_i$ **begin**
    $Base_i \leftarrow Base_i \cup \{Q' |$ where Q' is the parent cube of Q $\}$
  end
end
if $Base_i$ *is empty set* **begin**
  $IA \leftarrow$ *raw data*;
  slice query$_2$ $\leftarrow Q$;
end
else begin
  Choose $b \in Base_i$ s.t. b is the data cube with the least estimated cost to evaluate Q using it;
  $IA \leftarrow b$;
  slice query$_2$ is the rewritten query of Q to use b;
end
return *base query*, IA, and *slice queries*;

Figure 4: **The *Split* algorithm**

However, if the result set of the base query is not materialized, the split algorithm will select an immediate alive ancestor with the least estimated cost that can be used to evaluate Q. If there is no immediate alive ancestor materialized or the estimated execution cost of the query using the selected data cube is more expensive than the cost of query execution using raw data, then the query is evaluated using raw data. For simplicity, we assume that the execution cost of Q on a data cube (or raw data) is proportional to the number of rows scanned in the data cube (or raw data). Note that the algorithm *split* terminates since the data cube derivability relation is a lattice structure. Once the algorithm finds an immediate alive ancestor (IA), then Q is rewritten as (IA, Q_2'), where Q_2' is another slice query which is evaluated on IA. As an example (Figure 3), if $DC1$ is not materialized, but $DC2$ is materialized in the cache, the result set of $Q1$ is returned by executing $Q1''$ using $DC2$ which is the IA of $DC1$.

3.2 Query Attachment Graph

Once the query Q is transformed by the split process into two queries, the *base query* or *its immediate alive* *ancestor* and the *slice query*, we can obtain the same result set of Q by executing the *slice query*. We maintain a dynamic graph, *query attachment graph*, in our metadata storage to represent these relationships among queries. The information contained in the graph is also used for query subsumption testing and for calculating *profit* value of each query. The profit value is used in the cache replacement and admission algorithms to determine which query result should be materialized in the cache (see Section 4 for definition of a query profit).

The query attachment graph $G = (V, E)$ is defined as follows. The vertex set V consists of two partitioned sets V_c and V_{nc}, and a special vertex *raw_data*. V_c denotes the set of queries previously evaluated in the system that have result sets materialized in the cache. V_{nc} denotes the set of queries previously evaluated in the system with result sets not materialized in the cache. The vertex *raw_data* represents the raw data base. Each vertex contains the query representation for the query subsumption testing, information for calculating the profit of the query, and a pointer to the query result, if the vertex is in V_c. Useless vertices and their incident links are removed by the garbage collector to keep the graph from growing too much (Section 4).

The edge set E also consists of two partitioned sets E_d and E_e. There is a link $(Q_1 \mapsto Q_2) \in E_d$, where Q_2 is a data cube query and $Q_2 \in V_c$ whenever Q_2 is the base query of Q_1. When both Q_1 and Q_2 $(Q_2 \in V_c)$ are data cube queries, there is a link $(Q_1 \mapsto Q_2) \in E_d$ whenever RS_2 is the immediate alive ancestor of RS_1. A link $(Q_1 \mapsto Raw\_data)$ means that Q_1 is evaluated using *raw data*. There is a link $(Q_1 \leftrightarrow Q_1') \in E_e$, where $Q_1 \in V_c$ and $Q_1' \in V_{nc}$ whenever Q_1 and Q_1' are *result-equivalent* queries, i.e., $RS_1 = RS_{1'}$.

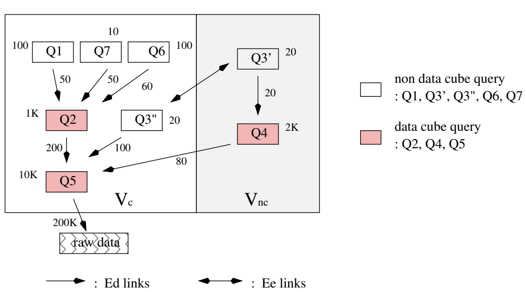

Figure 5: **A query attachment graph**

Figure 5 illustrates an example of a query attachment graph. In the figure, Q_1, Q_3'', Q_3', Q_6, and Q_7 are non data cube queries, while Q_2, Q_4, and Q_5 are data cube queries. Links $(Q_1 \mapsto Q_2)$, $(Q_6 \mapsto Q_2)$, $(Q_7 \mapsto Q_2)$ and $(Q_3'' \mapsto Q_5)$ show that Q_2 is the base query of Q_1, Q_6, and Q_7, so is Q_5 of Q_3''. That is, Q_1, Q_6, and Q_7 are evaluated on the data cubes RS_2, so is Q_3'' on RS_5. A link $(Q_2 \mapsto Q_5)$ means that Q_5 is the immediate alive ancestor query of Q_2. All the queries linked to Q_2 would be evaluated on RS_5 if the result of Q_2 is dropped out of the cache. A link $(Q_5 \mapsto raw\_data)$ shows that Q_5 is evaluated on *raw_data*.

Consider a query Q that is first transformed into (Q_4, Q_3'). Subsequently, query Q is transformed into (Q_5, Q_3'').

Query Q is evaluated through a slice query Q_3'' and using the materialized data cube query Q_5, since the base query Q_4 has not been materialized in the cache. In this case, we need to maintain the links incident to Q_4, since when the profit of Q_4 is high enough to move Q_4 to the V_c, all profit values of the queries adjacent to Q_4 need to be modified. If Q_4 moves to V_c, Q_3'' will have a link to Q_4 since Q_3' has a link $\mapsto$ to Q_4 and $(Q_3' \leftrightarrow Q_3'')$ indicates that Q_3' and Q_3'' are result-equivalent queries. Then the vertex Q_3' and its adjacent links will be removed in the graph.

One thing to note is that the query attachment graph can be constructed independently of the *split* algorithm. The graph itself represents the relations among queries in terms of their subsumption (or derivability). Other query transformations and subsumption algorithms [1] which relate queries to each other can be applied to the construction of the query attachment graph.

4 Dynamic Query Materialization

4.1 Problem Statement

In order to minimize the query response time, the cost incurred by execution of queries should be minimized. Caching query result sets reduces the query execution costs. To achieve a maximal reduction of query execution costs, preference should be given to caching of frequently referenced query results that are small, yet the underlying queries have high execution costs. However, the cached result sets must be re-freshed when the base tables are updated. Therefore, the cache manager needs to trade-off the savings that result from caching of query results and the costs of maintaining them. We first formalize the problem, show that finding the optimal solution is NP-hard and propose a greedy algorithm to solve the problem.

Let Q be the set of all queries q_i and RS the set of all materialized query results RS_j, with corresponding

[1]Our subsumption algorithm is ommitted in this paper. Interested readers may find it in [10].

index sets I and J ($J \subseteq I$), respectively, such that $q_i \in Q$ iff $i \in I$ and $RS_j \in RS$ iff $j \in J$. Then, the optimal query materialization algorithm should select the right set of materialized query results $RS_j \in RS$ in order to minimize the sum of the total execution cost of all queries and the total update cost for keeping the materialized result sets in the cache:

$$\textbf{min} \left\{ \sum_{i,j: q_i \in Q, RS_j \in RS} (c_{ij} \cdot r_{ij}) + \sum_{j: RS_j \in RS} (c_j^* \cdot u_j) \right\} \quad (1)$$

[2] is satisfied, subject to the constraint

$$\sum_{j: RS_j \in RS} s_j \leq S \quad (2)$$

where c_i is the execution cost of q_i using raw database,
c_{ij} is the execution cost of q_i using RS_j,
r_{ij} is the reference rate s.t. q_i is executed using RS_j,
u_j is the update rate of RS_j upon the changes of its base relations,
c_j^* is the update cost of RS_j,
s_j is the size of RS_j and S is the cache size.

Let C_{RS_j} be $\sum_{i,j: q_i \in Q, RS_j \in RS} (c_{ij} \cdot r_{ij}) + \sum_{j: RS_j \in RS} (c_j^* \cdot u_j)$. Then C_{RS_j} is the total cost associated with RS_j, i.e., the total execution cost of all queries using RS_j plus the total maintenance cost of query result RS_j. Then, the problem is the same as finding out a set J to minimize $\sum_{j \in J} C_{RS_j}$ subject to the space constraint $\sum_{j \in J} s_j \leq S$. This problem can be reduced to the 0/1 Knapsack problem. Consequently, there is no efficient solution for this problem. Similar objective functions to (1) are found in [14, 1, 7].

In following sections we present a dynamic query materialization algorithm as a greedy solution for this problem. Our algorithm makes use of a profit metric, which considers for each query result set the cost savings incurred by its materialization, its average rate of reference, its average rate of update, its updating cost, and its size.

4.2 Profit Model

The result sets of queries are materialized dynamically based on the potential cost gain associated with each result set. This cost gain should include not only cost savings associated with executing the attached queries on the corresponding result set, but also update cost of the result set when its base relations are updated.

[2] Actual cost may include the searching cost c_s to find out a RS_j for a given Q_i. In our model, this cost is negligible, so it is not included here (See Section 3.1).

Let Q_i, Q_j, and Q_k be linked as $Q_i \mapsto Q_j \mapsto Q_k$. Then *benefit*$_{ijk}$ is defined as c_{ik} - c_{ij}, where c_{ij} is the cost of executing Q_i using RS_j and c_{ik} is the cost of executing Q_i using RS_k. If there is no Q_i s.t. $i \neq j$ and $Q_j \mapsto Q_k$ (i.e., $i = j$), the *benefit*$_{jjk}$ becomes the cost executing Q_j using RS_k, since $c_{jj} = 0$. Intuitively, the *benefit*$_{ijk}$ is the cost saving using RS_j instead of RS_k for the executing query Q_i. If we do not know the exact cost c_{ik}, we estimate c_{ik} using a simple linear cost model, i.e., $c_{ik} = c_{ij} \cdot \frac{|RS_k|}{|RS_j|}$. For example, in Figure 5, the numbers next to the query represents $|RS|$, and the numbers next to the link $Q_i \mapsto Q_j$ represents c_{ij}. Then, *benefit*$_{1,2,5}$ is $c_{1,5} - c_{1,2} = 50 \cdot \frac{10K}{1K} - 50 = 500 - 50 = 450$, *benefit*$_{1,1,2}$ is $c_{1,2} - c_{1,1} = 50 - 0 = 50$, and *benefit*$_{5,5,raw}$ is $200K$. Similarly, if the query execution cost using its adjacent vertex where the query vertex is in V_{nc} is not available, it can be estimated using a linear cost model. For example, $c_{3',4}$ is estimated as $c_{3",5} \cdot \frac{|RS_5|}{|RS_4|} = 20$, since we know $Q_{3'}$ is result-equivalent to $Q_{3''}$ by the link $Q_{3'} \leftrightarrow Q_{3''}$.

By using the benefit associated with the query Q_j, the profit value of result set of Q_j is defined as:

$$\text{profit}(RS_j) = \frac{\sum_{i \in I, k \in K} \{\lambda_i \cdot \textit{benefit}_{ijk}\} - \mu_j \cdot c_j^*}{s_j} \quad (3)$$

where I and K are the set of i and k s.t. $Q_i \mapsto Q_j$, $Q_j \mapsto Q_k \in E_d$.
λ_i is the estimated reference rate of Q_i,
μ_j is the estimated update rate of RS_j,
c_j^* is the update cost of RS_j
s_j is the size of the RS_j.

Both estimated reference rate and update rate are smoothed by averaging the last K references and update times to RS_j. Smoothing of the reference and update rates helps to prevent flooding the cache with useless result sets in presence of workload with transient phases [9, 10]. The estimated reference rate λ_i and update rate μ_i are defined as

$$\lambda_i = \frac{K}{t - t_K}, \qquad \mu_i = \frac{K}{t - t_{u_k}} \quad (4)$$

where t is the current time, t_K is the time of the K-th most recent reference, and t_{u_k} is the Kth most recent timestamp that its base relation is updated. Both t_K and t_{u_k} are the oldest values in their sliding window for reference and update timestamps. Unlike other statistical methods for signal smoothing such as moving average, the above method enables aging of reference rate of non-accessed query result by incorporating the current time t into (4).

A simple way to update a materialized result set RS_j is by rebuilding it from the scratch. In such a case, it holds that $c_j^* = c_{jk}$. However, the update cost can be reduced using any of the incremental update methods [15]. Although the choice of a method to maintain materialized sets is orthogonal to our approach, we chose in our model the former mostly for its simplicity.

4.3 Cache Replacement and Admission Algorithms

The goal of the cache replacement algorithm is to maintain the "right" set of query result sets for a given cache size in order to minimize the cost defined in (1). In order to achieve this goal, our cache replacement algorithm LNC-R (Least Normalized Cost Replacement), initially developed in [9], sorts all query results whose information are held in the cache in ascending order of profit as defined in (3) and selects some of these sorted query results as the candidates for eviction when space needs to be freed. Our cache manager maintains the query attachment graph to enable the profit value of each result set of the query to be calculated irrespective of whether or not the query result is materialized in the cache.

Our cache replacement with admission algorithm LNC-RA allows a query result to be admitted into the cache only if it improves the overall profit. Let C be a set of replacement candidates for result set RS_i. LNC-RA decides to cache RS_i only if RS_i has a higher profit than all the query results in C. Thus, RS_i is cached only if the following inequality holds:

$$\text{profit}(RS_i) > \text{profit}(C) \tag{5}$$

where the profit of C (profit(C)) is defined as

$$\frac{\sum_{RS_j \in C, i \in I, k \in K}\{\lambda_i \cdot \mathit{benefit}_{ijk}\} - \sum_{RS_j \in C} \mu_j \cdot c_j^*}{\sum_{RS_j \in C} s_j} \tag{6}$$

where all the indices and notations are defined in as (3).

If there is no information about the past reference and update rate of the new query result, for example, the query result is retrieved for the first time, then our cache manager makes its decision based on available information and the profits of both the new query result and candidates are estimated by setting unavailable information to default values. Whenever the profit of a query Q_i, whose result set is not materialized, but is represented in the query attachment graph, is smaller than the least profit among all cached results, the node Q_i is removed and its incident links are also removed or adjusted. This means that the cache manager does not maintain retained reference information of Q_i any longer, since the query result RS_i would immediately become a victim for replacement if they are cached, and therefore retaining reference information of Q_i does not lead to the performance improvement. More details of our cache replacement and admission algorithm may be found in [9].

In Figure 6, we show a dynamic query materialization algorithm using query split, query attachment graph, and our cache replacement algorithm based on profit model presented so far.

```
Transform Q into (Q1, Q2); Q1 is base query
If Q1 ∈ Vc, then Q2 is slice query.
  Search for a query result which has a link ↦ to
    Q1 and subsumes Q2.
  If a subsuming query is found in Vc
    Execute Q2 on the subsuming query result.
  Else not found
    Execute Q2 on RS_Q1.
    Add Q2 into Vnc
    Add (Q2 ↦ Q1) into Ed
Else  // Q1 has not been materialized.
  Let Q11 be immediate alive ancestor and Q'2 is slice query.
  Add Q1, Q2 into Vnc
  Add (Q1 ↦ Q2) into Ed
  Search for a query result which has a link ↦ to Q11
    and subsumes Q'2.
  If a subsuming query is found in Vc
    Execute Q'2 on the subsuming query result.
  Else not found
    Execute Q'2 on RS_Q11.
    Add Q'2 to Vnc
    Add (Q2 ↔ Q'2) into Ee.
    Add (Q'2 ↦ Q11) into Ed
Run LNC-RA (cache replacement and admission algorithm).
```

Figure 6: **The pseudo algorithm of dynamic cube materialization**

When a new query Q arrives at the cache manager, the cache manager first checks the type of the query. If Q is not a canonical query, then Q is processed in the same way as in [9]. If Q is a canonical query, the split process is performed. This procedure returns the *base query* or its *immediate alive ancestor*(IA), and the *slice queries*. Then the subsumption testing is invoked. If there is any materialized query result which subsumes the slice query, the result of the slice query is answered using the subsuming query result.

If the cost of slice query, c_2, is estimated more costly than the cost of Q, c_1, then ($Q \mapsto \mathit{raw\_data}$) is inserted into E_d. That is, Q is evaluated from the raw data. If $c_2 \leq c_1$ and the *base query* exists in the cache space, Q is evaluated from the *base query* and (*slice query* $\mapsto$ *base query*) is inserted into E_d. If $c_2 \leq c_1$ and a *base query* does not exist but IA exists in the cache space, Q is evaluated from the IA and (*slice query* $\mapsto$ IA) is inserted into E_d. In this case, we also need to maintain the relationship between *base query* and another *slice query'* which could use *base query* if the result set of *base query* is materialized in the cache. In this manner, if the profit value of *base query* increases later on, the *base query* will be moved to the cache space. Therefore, corresponding edges (*slice query'* $\mapsto$ *base query*) and (*base query* $\mapsto$ *IA*) should be inserted into E_d, and (*slice query'* $\leftrightarrow$ *slice query*) should be inserted into E_e.

Once all the links are associated with the given query Q in the query attachment graph, the algorithm calls the cache replacement algorithm to determine which result sets should be materialized and which result sets should be replaced using its admission algorithm.

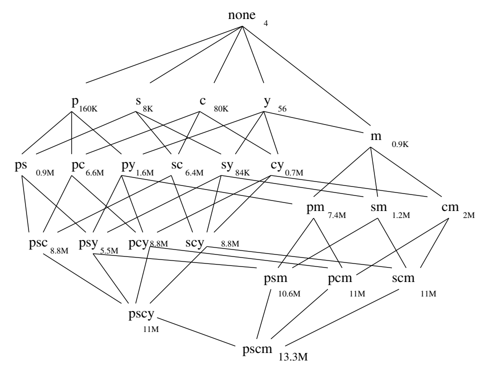

Figure 8: **A Lattice structure of data cubes on a TPC-D database**

5 Experimental Evaluation

5.1 Experimental Setup

We evaluated the performance of our dynamic query materialization algorithm on a TPC-D benchmark using an Oracle 7 database on HP-UX. The size of database is 100MB excluding index space. We chose such a small size of the database to be able to run a sufficiently large number of queries (20K) so that the cache achieved a steady state. We consider a subset of the TPC-D database schema which corresponds to a star schema (Figure 7). The four dimensions we are interested in are *partkey*(p), *suppkey*(s), *custkey*(c), and time as *year*(y) and *month*(m). The measure of interest in the *order* fact table is the total *quantity*. The lattice structure of the data cubes on these four dimensions is illustrated in Figure 8, and the number indicates the sizes of the corresponding data cube.

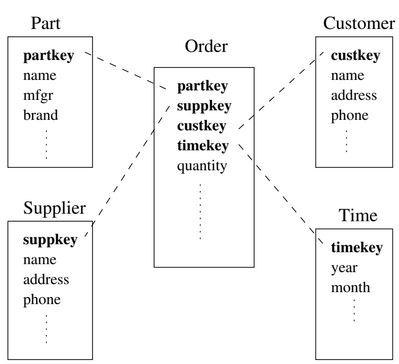

Figure 7: **A TPC-D based star schema**

We run 20K TPC-D-like canonical queries. Each query is generated to have a randomly selected number of dimensions and a randomly selected attribute values for the predicates appearing in the corresponding *select-condition*. All queries are distinct, i.e. there were not two identical queries in the entire workload.

We measured the performance of different cache managers using a *cost saving ratio*(CSR) defined as:

$$CSR = \frac{\sum_{ij}(c_i \cdot r_i - c_{ij} \cdot h_{ij} - c_j^* \cdot u_j)}{\sum_i c_i \cdot r_i} \qquad (7)$$

where c_i is the execution cost of query q_i using the raw database, c_{ij} is the execution cost of query q_i using DC_j on which is the selected materialized cube by the algorithm to execute q_i, r_i is the total number of references to query q_i, h_{ij} is the number of times that q_i is evaluated using DC_j, c_j^* is the update cost of DC_j, and u_j is the total number of reorganizations performed to reflect the updates of DC_j. The query and update execution cost were expressed in number of buffer reads (which included both buffer hits and misses). Query response time is typically proportional to the number of rows (or buffer blocks) processed [7]. It is also important that such a cost metric abstracts from the specifics of our hardware configuration.

5.2 Performance Comparisons

We compared the performance of our dynamic query materialization algorithm (LNC-RA) with other published algorithms, Harinarayan et al's approach [7] (*Static-1*), its modified version (*Static-2*) and Yang et al's approach [14] (*YKL97*).

Both Static-1 [7] and Static-2 are static algorithms that use greedy heuristics to select n data cubes with the largest *benefit* values, where n is a predetermined number. The *benefit* of a cube x is the sum of the cost differences between the evaluation cost of its descendants ys using x and the evaluation cost of ys using another view known as the cheapest way of evaluating y during each selection steps, $1, \ldots, n$. We argue that it may be impractical to fix the number of materialized cubes in most caching system where the cache size is fixed. We modified Static-1 into Static-2 which considers the remaining size of cache instead of the number of materialized cubes. While Static-1 does not select any more data cubes when the next data cube with the largest benefit exceeds the remaining size of cache space, Static-2 selects other data cubes with next largest benefit when cache has enough room for them. For example, if data cubes a, b, c have 100, 50, 10 as their benefits (sizes), and the remaining cache size is 80, Static-1 does not choose any cube to be cached, because the size of a with the largest benefit is over 80, while Static-2 materializes b and c.

YKL97 [14] is also a greedy heuristic that uses an extended *benefit* model. The model calculates benefit by considering the query reference rate as well as the update cost associated with cube updates. A similar algorithm is shown in [1]. The method may be ragarded as a static approach since it requires that the reference and update rates of future queries are known in advance. However, since this was not practically feasible, we estimate the query frequencies and the update frequencies based on past references and updates during some intervals, respectively. Then we run their cube selection algorithm in a regular manner to reorganize the set of materialized cubes in the cache.

The cost saving ratios of all algorithms are compared in Figures 9 and 10. Queries are randomly distributed in Figure 9 and distributed with Zipf (α, β) distribution in Figure 10, where the skew is applied to the base cubes. In other words α % of the total queries use β % of the data cubes as their base queries. We use 30 and 70 as α and β, respectively. The minimum cache size required is 15% of database size, since *Static-1* always materializes the lower bound cube, *pscm*, in our experiment. Therefore, we considered cache sizes 15%, 30%, and 45% of database size without the loss of generality.

We found that our dynamic cube materialization provides consistently better performance over all other competitors. All caching algorithms indicate that their CSRs are positive, more over than about 30% in both figures. It means that materializing data cubes for query evaluation is indeed useful for the cost saving no matter which algorithm is used. However, on average LNC-RA performs 54% and 37% better than Static-1 and Static-2, respectively in Figure 9. In Figure 10 where the query distribution is skewed, LNC-RA performs even better; on average 94% and 70% better than Static-1 and Static-2, respectively. It is because LNC-RA keeps the materializing data cubes dynamically in the cache so that the selected data cubes help the query response time while the query reference rates are changing. LNC-RA also outperforms YKL97 in both experiments. On average, LNC-RA outperforms 41% in Figure 9 and 77% better in Figure 10. This shows that the benefit of considering the result set size in cache replacement is also important factor to determine the profit of data cubes.

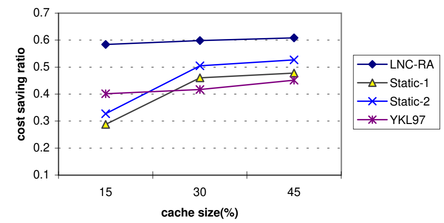

Figure 9: **Performance comparison on random queries**

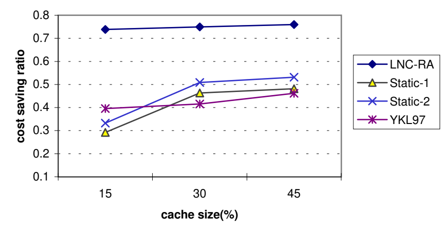

Figure 10: **Performance comparison on Zipf distributed queries**

6 Conclusion

We have presented a design of cache manager for the DSS environment. The cache manager employs a *query split* algorithm to efficiently find query results which subsume the submitted canonical form of the query. The query split algorithm takes into account that a given canonical query can be derived from either base query or its immediate alive ancestor.

The cache manager dynamically maintains the cache content by deciding whether a newly generated query result should be admitted to the cache and if so, which query results should be evicted from the cache to free space for the new query result. The decisions are explicitly based on the profit of each query result which consider both the benefit of cost saving by materializing the query result in the cache and the maintenance cost to keep the query result in the cache. The cache manager maintains the subsumption relationship between both materialized and non-materialized query results in the metadata store to improve the efficiency of cache replacement and admission algorithms. The experimental results show that our cache manager can improve performance of other systems on average by at least 37% and by as much as 70% on a skewed workload.

References

[1] E. Baralis, S. Paraboschi, and E. Teniente. Materialized view selection in a multidimensional database. In *Proceedings of the International Conference on Very Large Databases*, 1997.

[2] S. Chaudhuri, R. Krshnamurthy, S. Potamianos, and K. Shim. Optimizaing queries with materialized views. In *Proceedings of International Conference on Data Engineering*, 1985.

[3] C. Chen and N. Roussopoulos. The implementation and performance evaluation of the ADMS query optimizer: Integrating query result caching and matching. In *Proceedings of the International Conference on Extending Database Technology*, 1994.

[4] Incorporated Cognos. Cognos OLAP software white papers. Available at *http://www.cognos.com-/busintell/dw_roi/dw_white_paper.html*.

[5] P. Deshpande, K. Ramasamy, A. Shukla, and J. Naughton. Caching multidimensional queries using chunks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1998.

[6] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *Proceedings of the International Conference on Very Large Databases*, 1995.

[7] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1996.

[8] W. Inmon. *Building the Data Warehouse*. John Wiley and Sons, second edition, 1996.

[9] P. Scheuermann, J. Shim, and R. Vingralek. WATCHMAN: A data warehouse intelligent cache manager. In *Proceedings of the International Conference on Very Large Databases*, 1996.

[10] J. Shim. *An Intelligent Cache Manager in Data Warehousing Environment and its Application to the Web Caching*. PhD thesis, Northwestern University, 1998.

[11] D. Srivastava, S. Dar, H. Jagadish, and A. Levy. Answering queries with aggragates using views. In *Proceedings of the International Conference on Very Large Databases*, 1996.

[12] Transaction Processing Performance Council. *TPC Benchmark D*, 1995.

[13] W. P. Yan and P. A. Larson. Eager aggregation and lazy aggregation. In *Proceedings of the International Conference on Very Large Databases*, 1995.

[14] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. In *Proceedings of the International Conference on Very Large Databases*, 1997.

[15] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1995.