A Case for Dynamic View Management

YANNIS KOTIDIS
AT&T Labs—Research
and
NICK ROUSSOPOULOS
University of Maryland

Materialized aggregate views represent a set of redundant entities in a data warehouse that are frequently used to accelerate On-Line Analytical Processing (OLAP). Due to the complex structure of the data warehouse and the different profiles of the users who submit queries, there is need for tools that will automate and ease the view selection and management processes. In this article we present DynaMat, a system that manages dynamic collections of materialized aggregate views in a data warehouse. At query time, DynaMat utilizes a dedicated disk space for storing computed aggregates that are further engaged for answering new queries. Queries are executed independently or can be bundled within a multiquery expression. In the latter case, we present an execution mechanism that exploits dependencies among the queries and the materialized set to further optimize their execution. During updates, DynaMat reconciles the current materialized view selection and refreshes the most beneficial subset of it within a given maintenance window. We show how to derive an efficient update plan with respect to the available maintenance window, the different update policies for the views and the dependencies that exist among them.

Categories and Subject Descriptors: H.2.7 [**Database Management**]: Database Administration—*data warehouse and repository*; H.4.m [**Information Systems Applications**]: Miscellaneous

General Terms: Management

Additional Key Words and Phrases: Data cube, data warehousing, materialized views, OLAP

1. INTRODUCTION

The ability to participate or react quickly and decisively in today's competitive marketplace is critical to the success of organizations. Recent advances

An extended abstract of a preliminary version of this article was presented at the ACM SIGMOD International Conference on Management of Data, Philadelphia, Pennsylvania, 1999.
Authors' addresses: Y. Kotidis, AT&T Labs—Research, 180 Park Avenue, P.O. Box 971, Florham Park, NJ 07932-0000; N. Roussopoulos, Department of Computer Science, University of Maryland, College Park, MD 20742.

in information technology have made available overwhelming amounts of information. One of the more well-known examples is WalMart's dataset that captures point-of-sale transactions from over 2,900 stores in six countries with an estimate size of more than 100 terabytes. Telecom companies provide another example of data rich industries. The Network Services Research Center at AT&T Labs has collected over 60 terabytes of data from various sources (call-detail, IP-traffic, etc.) with the number expected to grow to over 200 terabytes within two years. Being able to manage and analyze these tremendous volumes of enterprise-data offers a strategic advantage in the marketplace.

The data warehousing technology and tools provide an integral part of any decision support system. In the broadest sense, a data warehouse is a single, integrated informational store that provides stable, point-in-time data for decision support applications. Unlike traditional database systems that automate day-to-day operations, a data warehouse provides an environment in which an organization can evaluate and analyze its enterprise data over time.

To ensure easy access to the information, most data warehouses adopt a multidimensional approach for representing the data. The origins of this practice go back to PC spreadsheet programs that were extensively used by business analysts. More advanced multidimensional access is now achieved through interfaces that provide *On-Line Analytical Processing* (OLAP), which involves interactive access to a wide variety of possible views of the information. OLAP queries compute key performance metrics that enable the enterprise to better understand its businesses. Examples of OLAP involve the computation of *multidimensional ratios* (e.g., "Show me the contribution to weekly profit made by all items sold in Maryland between May 1 and May 7"), *comparisons* (e.g., "Show sales in this fiscal period, broken down into monthly intervals, versus last period"), and computing *quantiles* and *statistical profiles* (e.g., "Show sales by store for all locations in the bottom 10% of sales").

In most cases, the main cost in terms of the time consumed of executing this type of queries is not doing the actual arithmetic, but of retrieving the data items that affect the calculated functions. For a large dataset, executing queries with aggregations against the detailed transaction records is prohibitively expensive, simply because of the volume of records that are being accessed. As a result, data warehouses facilitate some form of preaggregation in order to support complex data-intensive queries. In relational databases, materialized derived relations (views) have long been proposed to speed up query processing. In the data warehouse, these views store redundant, aggregated information and are commonly referred to as *summary tables* [Chaudhuri and Dayal 1997]. A materialized view contains consolidated information and is typically much smaller than the base relations used to store all detailed records. As a result, querying the view instead of the base relations offers several orders of magnitude faster query speeds.

Since materialized views promise high performance improvements for analytical queries, they are a valuable component in the design of a data warehouse. They might, for example, include high-level consolidations, which are bound to be needed for reports or ad-hoc analyses, and which involve too much data to be aggregated on the fly. If query response time is the only concern, an eager policy of materializing all possible aggregates that might be requested will yield an excellent effect on performance since each query will require a minimum amount of data movement and on the fly calculations. Unfortunately, this plan is not viable, as the number of possible aggregate views is exponential in the number of attributes (dimensions) that the dataset in analyzed on. Furthermore, much like a cache, the views get dirty whenever the data warehouse tables are modified. Thus, one should also take into account the view maintenance overhead, which is payable each time new data is shipped to the data warehouse and its base tables get refreshed.

In this article, we present DynaMat [Kotidis and Roussopoulos 1999], a system that manages dynamic collections of materialized views in a data warehouse. At query time, DynaMat utilizes a dedicated disk space (called the *View Pool*) for storing computed aggregates (called *view fragments*) that are further engaged for answering new queries. The View Pool is efficiently organized through a network of indexes connected in a lattice (hyper-cube) topology that permits fast access to the fragments based on a partial order that we impose among the views. This organization allows DynaMat to manipulate results of different levels of aggregation and exploit them to answer queries that don't have an exact match stored in the View Pool. During updates, the materialized set is reconciled and the most beneficial subset of it is refreshed within a given maintenance window. A critical performance issue is how fast we can incorporate the updates to the views. Efficient computation of the views using techniques like Gupta et al. [1993], Griffin and Libkin [1995], Jagadish et al. [1995], Agrawal et al. [1996], Harinarayan et al. [1996], Mumick et al. [1997], and Zhao et al. [1997] and/or bulk incremental updates [O'Neil et al. 1996; Roussopoulos et al. 1997; Jermaine et al. 1999] enhances the overall performance of the system. In DynaMat, any of these techniques can be applied. Our update algorithm identifies dependencies among the views in order to share the refresh cost among multiple aggregates and considers both incremental and recomputation techniques.

We believe that the main contribution of DynaMat is the idea of continuous view management that results in better utilization of the available system resources. DynaMat offers a self-tunable solution that relieves the data warehouse administrator from having to monitor and calibrate the system constantly. In our experiments, we compare DynaMat against a system that is given all queries in advance and the precomputed optimal static selection of views for them. These experiments show that dynamic view management outperforms the optimal static system and thus any suboptimal static algorithm proposed in the literature. The reason is that DynaMat, at any point in time, utilizes both the available disk space and maintenance window, while a static system is bound to fully utilize one of them.

The rest of the article is organized as follows: in Section 2, we discuss some implications of the view selection problem and make a case for a dynamic approach. Section 3 gives an overview of the basic functionality of DynaMat and presents an abstract interface that allows easy integration into existing systems. In Section 4, we describe how the View Pool is organized to handle efficiently both ad-hoc and precompiled queries. In Section 5, we show how to efficiently update the View Pool, within a given maintenance window. Section 6 contains the experiments while, in Section 7, we comment on related work. Finally, in Section 8, we draw the conclusions.

2. A CASE FOR DYNAMIC VIEW MANAGEMENT

Disk space and creation/maintenance overhead will not allow us to materialize all interesting aggregate views in the data warehouse. The view selection problem consists of finding those views that minimize query response time [Roussopoulos 1982; Harinarayan et al. 1996; Baralis et al. 1997; Gupta 1997; Gupta et al. 1997; Shukla et al. 1998; Smith et al. 1998; Karloff and Mihail 1999] under a resource constraint, typically disk space. Most view selection algorithms also take into account the query workload (e.g., by using frequency counts for each view) and compute a set of views that fit in the available disk space and best optimize query performance.

This static selection of views, however, contradicts the dynamic nature of decision support analysis. In many cases, users submit their queries interactively, that is, they do not have a precompiled set of queries in mind, but rather they are making up their queries on the way, based on the feedback they get from the system. This type of analysis often results in querying the dataset in surprising ways that are not best supported from the materialized views selected by the previous algorithms. Furthermore, as query patterns and data trends change over time and as the data warehouse is evolving with respect to new business requirements that continuously emerge, even the most fine-tuned selection of views that we might have obtained at some point will very quickly become outdated. This means that the data warehouse administrator should monitor the query pattern and periodically "recalibrate" the materialized views by rerunning these algorithms. This task for a complex data warehouse with many users of different profiles is rather complicated and time consuming. In addition, the maintenance window, the disk space restrictions and other important operational parameters of the data warehouse may also change. For example, an unexpected large volume of updates will throw the selected set of views as not updateable unless some of these views are discarded.

Another inherent drawback of a static view selection is that the system has no way of tuning a wrong selection by reusing results of queries that couldn't be answered by the materialized set. Notice that, although OLAP queries take an enormous amount of disk I/O and CPU processing time to be completed, their output is often quite small. Query "*Show the total volume of sales for the last 5 years*" is a fine example of that. Processing this query might take hours of scanning and aggregating vast tables of detailed records, while the result is a single value that can be easily "cached" for future reuse. Moreover, during *rollup* operations [Gray et al. 1996], the data is examined at a progressively coarser granularity and future queries are likely to be computable from previous results without accessing the base tables at all.

We believe that selecting a view set to materialize is just the tip of the iceberg. Clearly, query performance is improved as more views are materialized.[1] With the cost of disk volume constantly dropping, disk storage constraint may no longer be the limiting factor in view selection but the time window to refresh the materialized set after updating the detailed records. More materialization implies a larger refresh time. This update window is the major data warehouse parameter, constraining overmaterialization. Some view selection algorithms [Gupta 1997; Baralis et al. 1997] take into account the maintenance cost of the views and try to minimize both query-response time and the maintenance overhead under a given space restriction. Theodoratos and Sellis [1997] define the data warehouse configuration problem as a state-space optimization problem, where the maintenance cost of the views needs to be minimized, while all the queries can be answered by the selected views. The trade-off between space of precomputed results and maintenance time is also discussed in Do et al. [1998]. However, they do not consider the dynamic nature of the view selection problem, nor they propose a solution that can adapt on the fly to changes in the workload.

Our premise is that a result is a terrible thing to waste and that its generation cost should be amortized over multiple uses of the result. Our main motivation comes from earlier work on caching query results in the *ADMS*$\pm$ architecture [Roussopoulos and Kang 1986; Delis and Roussopoulos 1992], the work on prolonging their useful life through incremental updates [Roussopoulos 1991] and their reuse in the ADMS optimizer [Chen and Roussopoulos 1994]. This is a major departure from the static paradigm of preselecting a set of views to be materialized and running all queries against this static set.

3. FUNCTIONALITY OF DYNAMAT AND API CALLS

We envision DynaMat as a self-tunable view management system, tightly coupled with the rest of the data warehouse architecture. DynaMat manages a dedicated disk space that we call *View Pool* ($\mathcal{V}$), in which previously computed aggregates are stored. We use the term, *view fragments* or simply *fragments*, when referring to the data that is materialized in $\mathcal{V}$. For faster access to these fragments and bookkeeping, a fragment lookup directory (called *Directory*) is maintained, as shown in Figure 1.

We distinguish two operational phases of the system. The first is the "online" phase during which DynaMat answers queries posed through a *queryPool*(q) *API call*. A description of the queries accepted by this interface is in Section 4. DynaMat determines whether or not stored fragments can be exploited to answer a new query by comparing the cost of answering the query from a fragment against the cost of running the same query against the detailed records in the data warehouse. Both costs are estimated by probing the query optimizer. Our model exploits dependencies among materialized aggregates of different levels of aggregation. This means that a more detailed aggregate is used to answer

[1]There are some exceptions due to the effect of thrashing in memory buffers, that we ignore in this discussion.

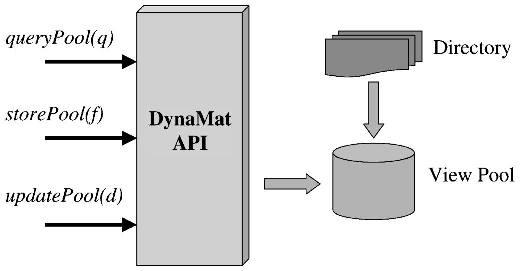


Fig. 1. DynaMat Overview.

queries of coarser granularity like, for instance, when computing the monthly sales out of the available daily aggregates. In the general case, more than one fragments might be used for computing the answer to a new query. Such fragments are quickly discovered using the Directory structure that is described in detail in Section 4.1.

For a dynamic system, an important functionality is the admission of new aggregates in the View Pool. Whenever a new fragment f is computed as a result of a query, f is passed back to DynaMat through the *storePool*(f) call. In a traditional caching system, a new result is always offered storage in the cache in order to exploit spatial—temporal locality in the user access pattern. For most applications, this is desirable if all cached objects have the same cost. However, this is not the case for materialized aggregates whose recomputation cost varies dramatically. Scheuermann et al. [1996] introduced an admission schema specific for data warehousing workload. In DynaMat, we extend these techniques, as described in Section 5.3.

The second phase of DynaMat is the update phase, during which updates received from the data sources get stored in the data warehouse and the fragments in the View Pool get refreshed. In DynaMat, we assume that the update phase is "offline" and queries are not permitted during maintenance. The maximum length W of the update process is specified by the administrator. Updates are introduced through an *updatePool*(d) function call, which defines a data source d that holds the update increment. This can be a log-file, a regular table, or a virtual view over multiple tables. Different update policies can be implemented, depending on the types of updates, the properties of the data sources and the aggregate functions that are computed by the fragments. From DynaMat's point of view, the goal is to select and update the most useful fragments within the update time constraint. Notice that this is not equivalent to updating as many fragments as possible, although often both yield the same result.

4. AGGREGATE AWARE VIEW POOL ORGANIZATION

A *multidimensional data warehouse* (MDW) is a repository in which data is organized along a set of dimensions $\mathcal{D} = \{d_1, d_2, \ldots, d_n\}$. A possible way to design a MDW is the star-schema [Kimball 1996] in which, for each dimension, there is a *dimension table* D_i that has d_i as its primary key and also uses a *fact table* F that correlates the information stored in these tables through the

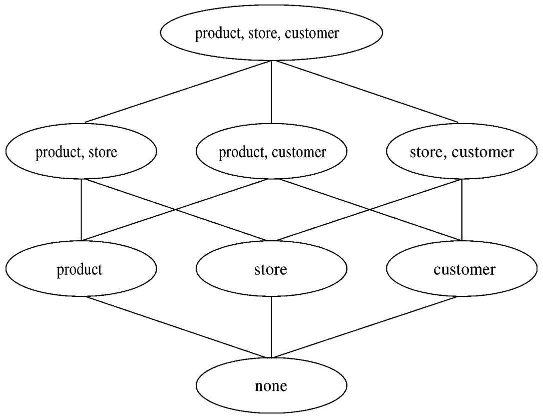


Fig. 2. The data cube lattice for $D = \{product, store, customer\}$.

foreign keys $d_1, \ldots, d_n$. The data cube operator [Gray et al. 1996] performs the computation of one or more aggregate functions for all possible combinations of grouping attributes, which are actually attributes selected from the dimension tables D_i.

As a running example, we use a simplified basket-dataset in which customers are buying products from various stores. This schema has three dimensions namely: *customer*, *product*, *store* and a single numeric measure *sales* that stores the amount of dollars spent in each transaction. A relational implementation uses the following fact table for organizing atomic transactions:

$$F(product, store, customer, sales).$$

Figure 2 shows a lattice representation [Harinarayan et al. 1996] of the data cube for $D = \{product, store, customer\}$. Each node in the lattice represents a view that aggregates data over the attributes present in that node; for example, $(product, store)$ is an aggregate view over the *product* and *store* grouping attributes. For simplicity in the notation, in this presentation we do not consider the case where grouping is done over attributes other than the dimension keys d_i. However, our framework is still applicable in the presence of more grouping attributes and hierarchies, using the extensions of Harinarayan et al. [1996] for the lattice.

The lattice is frequently used by view selection algorithms [Harinarayan et al. 1996; Gupta et al. 1997; Shukla et al. 1998] because it captures the computational dependencies among the views of the data cube. In Figure 2, we show only dependencies between adjacent views and not those in the transitive closure of this lattice. For example, view $(product)$ can be computed from view $(product, store)$, while view $(product, store, customer)$ can be used to derive any other view.

In this context, we assume that the data warehouse workload is a collection of *Multidimensional Range queries* (MR-queries) each of which can be visualized as a *hyperplane* in the data cube space using an n-dimensional "vector" $\vec{q}$:

$$\vec{q} = \{r_1, r_2, \ldots, r_n\}, \tag{1}$$

where r_i is a range in dimension's d_i domain. We restrict each range to be one of the following:

—a full range: $r_i = (min_{d_i}, max_{d_i})$, where min_{d_i} and max_{d_i} are the minimum and maximum values for key d_i.

—a single value for d_i

—an empty range that denotes a dimension that is not present in the query.

Example 4.1. Let *product* and *store* be integer keys in the range: $1 \leq$ *product* ≤ 1000 and $1 \leq$ *store* ≤ 200. Vector $\vec{q} = \{50, (1, 200), ()\}$ describes the following SQL query:[2]

```
select product, store, sum(sales)
from F
where product = 50
group by product, store
```

The empty range for the customer dimension denotes that the measure is aggregated over all values of this dimension.

If, in this example, the grouping was done on attributes other than the dimension keys, then the actual SQL description would include joins between some dimension tables and the fact table. This type of queries are often called *slice queries* [Gupta et al. 1997; Baralis et al. 1997; Kotidis and Roussopoulos 1998]. We prefer the multidimensional notation over the SQL description because it describes the workload in the data cube space independently of the actual implementation of the MDW.

The same notation permits us to represent the materialized results of MR-queries, which we call *multidimensional range fragments* (*MRFs*). Given a MR-query q and a cost model for accessing the stored MRFs, we want to find the "best" subset of them in $\mathcal{V}$ to answer q. Based on the definition of MRFs, we argue that it doesn't pay to try to combine multiple materialized results to answer q. With high probability, q is best computable out of a single fragment f or not computable at all. We illustrate this with the following example:

Example 4.2. Consider the previous query $\vec{q} = \{50, (1, 200), ()\}$. Figure 3 describes the content of the View Pool in the 2-dimensional subspace of (*product*, *store*) view. The light-gray areas represent results that are stored in the Pool in the form of MRFs while the dark-gray region depicts the values requested by the query. In this example, no single MRF in the View Pool contains all values in q. A stored fragment that partially computes the result is of the form $\{50, s\_id\}$ or $\{(1, 1000), s\_id\}$, where s_id is some store value. In order to answer the query, there should be at least one such fragment for all values of s_id between 1 and 200. Even if such a combination exists, it is highly unlikely that querying 200 different fragments to get the complete result provides a cost-effective way to answer the query.

[2]The *sum*() function is picked as a representative of an interesting aggregation of the measure.

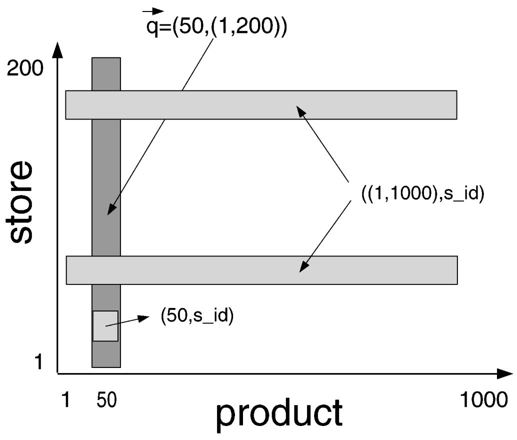


Fig. 3. Querying stored MRFs.

MRFs provide a slightly coarser grain of materialization if we compare them with views that allow arbitrary ranges for the attributes. However, if we allow such fragments to be stored in the View Pool, then the probability that a single stored fragment can solely be used to answer a new query is rather low, especially if most of the materialized results are small, that is, they correspond to small areas in the n-dimensional space. This means that we will need to use combinations of stored fragments and perform costly duplicate eliminations to compute an answer for a given query. Even though the multidimensional description of the fragments allows us to check for query containment [Yang and Larson 1987; Levy et al. 1995; Abiteboul and Duschka 1998; Kolaitis and Vardi 1998] quickly, in the general case, where many fragments overlap some portion of the query, there is a large number of combinations that need to be checked for finding the most efficient way to answer the query. A possible way to avoid this overhead is to organize the View Pool in a chunk-based schema as described in Deshpande et al. [1998]. This method partitions the cube space into equally sized n-dimensional "chunks" that conceptually define the finer block of data addressable by the system. A user query is also transformed to a set of chunks that need to be satisfied from the View Pool. Although this approach works well for managing the View Pool, it has the drawback that, in order to efficiently compute missing chunks in the data warehouse, the base tables have to be organized in a chunk-based format too. This requirement makes the approach impractical for commercial systems. In DynaMat, on the other hand, we do not make any assumptions on the structure of the data warehouse or even on how fragments get actually stored in the View Pool.

When managing the View Pool, an important consideration is the maintenance cost of the fragments. In most cases, updating fewer, larger fragments of views (as in a MRF-pool) is preferable as opposed to updating many smaller ones. We denote the number of fragments in the View Pool as $|\mathcal{V}|$. In Section 5.4, we show that the complexity of computing an update plan for the stored data is quadratic in $|\mathcal{V}|$, making the MRF approach more scalable.

4.1 Dynamic Query Processing

When a MR-query q is posted through the *queryPool*() API call, we search for candidate fragments that can be used to answer q. The following is easy to prove:

LEMMA 4.3. *Given a MRF f and a MR-query q, f answers q iff for every nonempty range r_i of the query, the fragment stores an equal or extended range and for every empty range $r_i = ()$ the fragment's corresponding range is either empty or spans the whole domain of dimension i.*[3]

DynaMat maintains a look-up structure that we call the *Directory* (see Figure 1), for finding candidate fragments for a new query. The Directory is designed to prune the search space down to a subset of potentially useful fragments from the View Pool. It is actually a set of smaller indices connected in the lattice topology of Figure 2. Each node has a dedicated index that keeps track of all fragments of the corresponding view that are stored in the View Pool. In our initial designs we indexed the fragments as objects in a k-dimensional ($k \leq n$) subspace, using R-trees. Each fragment f has exactly one entry that contains the following info:

—A hyperplane $\vec{f}_{index}$ with all $k \leq n$ nonempty ranges r_i in the fragment's vector $\vec{f} = \{r_1, \ldots, r_n\}$. For example, the result f of query $\vec{q} = \{50, (1, 200), ()\}$ has an entry $\vec{f}_{index} = \{(50, 50), (1, 200)\}$. $\vec{f}_{index}$ is used as a key in one of the R-trees.

—Statistics (e.g., number of accesses, time of creation, time of last access)

—The *father* of f (explained below).

When a query q arrives, we scan the corresponding directory for each view that contains a superset of the dimensions present in the query. For each index, we create an appropriate search hyperrectangle that is used to retrieve all fragments f that answer q. This is illustrated in the following example:

Example 4.4. Let $\vec{q} = \{(1, 1000), (), Smith\}$ be the query of interest. The set of dimensions with nonempty ranges in q is $D_q = \{product, customer\}$. We therefore need to scan the View Pool for fragments of views $(product, customer)$ and $(product, store, customer)$. For the first view, we search the corresponding 2-dimensional R-tree index using rectangle $\{(1, 1000), (Smith, Smith)\}$. Figure 4 depicts a snapshot of this R-tree as well as the search area. The shaded areas denote MRFs of that view that are materialized in the View Pool. Since no fragment is found, based on the dependencies defined in the lattice, we also check view $(product, store, customer)$ for candidate fragments. For this view, we "expand" the undefined in q *store* dimension and search the corresponding R-tree using rectangle $\{(1, 1000), (1, 200), (Smith, Smith)\}$. To answer the query from a fragment of this view we need to "collapse" the store column and aggregate the measure(s) over the *product*, *customer* dimensions.

[3]In the latter case, we have to perform an additional aggregation to compute the result, as will be explained.

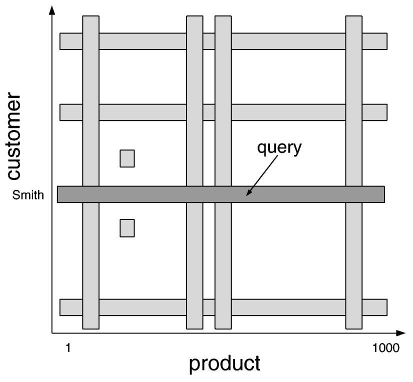


Fig. 4. Directory for view (*product*, *customer*).

If a stored fragment f exactly matches the query (i.e., $\vec{f} \equiv \vec{q}$), it is retrieved and returned to the user. If no exact match exists, we select the "best" fragment f from the View Pool that answers q, assuming that at least one can be found. Let function $c(q, f)$ denote the expected cost of querying a candidate fragment f for the values that are requested by q. If, for example, the fragments are stored as flat files, we estimate $c(q, f) = size(f)$, that is, the size of the fragment. Another possibility is to implement the fragments as relational tables with additional indices. In that case, $c(q, f)$ is determined by probing the optimizer. We also evaluate the expected cost of running the query at the data warehouse denoted as $c(q, DW)$. If a fragment f is found that can answer the query more effectively than the data warehouse, it is retrieved and used to compute the query. Let f_q be the materialized result of q. The fragment that was used to compute f_q is called the *father* of f_q and is denoted as $\acute{f_q}$. If no fragment answers q with cost less than $c(q, DW)$, the query is computed from the base tables of the data warehouse. In both cases, the result f_q is passed back to DynaMat through the *storePool*() API call and considered for admission in the View Pool.

As the necessary information stored for each fragment is just a few bytes long and the number of MRFs in the View Pool is in the order of thousands, we can safely assume that in all cases the Directory will be memory resident. Our experiments validate this assumption and indicate that the look-up cost in this case is negligible. For our current prototype, we are using linked lists for storing the fragment information for each node of the Directory, as we found them to perform sufficiently. For all of our experiments, the look-up time for a new query was in the order of microseconds, showing that the lattice organization of the Directory performs well.

4.2 Optimizing Precompiled Queries

In many cases, OLAP-style analysis gives rise to simultaneous related queries. Examples can be found in report-generating applications where a precompiled set of aggregates needs to be computed out of the raw data. Similarly many data-mining applications have an a-priori knowledge of the queries that they want to be executed. Another interesting example is when users interact with the data warehouse by sending batches of queries, disconnect for a while, and, after a reasonable amount of time, come back to find the results. These cases present a challenge for DynaMat to gain by optimizing the execution of these queries as a unit.

For better formulating the problem, in this section we treat the View Pool as a large disk-resident cache of materialized fragments (MRFs). Given a stream of precompiled queries that we want to execute, the optimization problem that we define consists of finding: (1) the best way to answer the queries from the View Pool and (2) the best execution order along with a replacement and materialization policy for the results of the queries, given a limited disk space.

4.2.1 *An Extended Multiquery Interface.* For the discussion in this section, the *queryPool*() interface call of DynaMat is extended to allow users to express multiple, possibly related, queries within a single Multi-Fragment Expression (MFX). An MFX is simply a multiset of MR-queries against the View Pool:

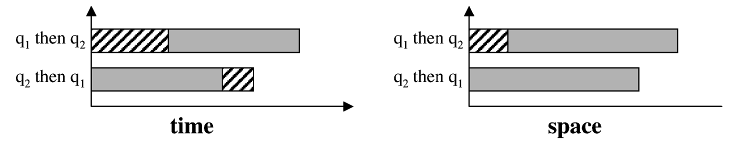
$$MFX = \{q_1, q_2, \ldots, q_k\}.$$

We assume that the user implies no order in the execution of the queries q_i. Similarly, we make no assumption that queries within the same MFX are necessarily related. Intuitively, a MFX provides the means to express multiple queries within the same unit. Compared to a sequential or concurrent execution of the queries $q_1, \ldots, q_k$, the MFX interface allows DynaMat to better explore possible dependencies among them as is illustrated in the following example:

Example 4.5. Assume the following two queries on our basket-dataset:

q_1:
```
select product, store, sum(sales)
from F
group by product, store
```
q_2:
```
select product, store, customer, sum(sales)
from F
group by product, store, customer
```

Both queries are expected to be costly to execute, since they both lack of predicates that would reduce the portion of base data that has to be accessed to compute the aggregates. Suppose the user issues query q_1 before q_2. If they are executed in a First-Come-First-Served manner, each would require a full scan of the base table F to be computed and a large volume of disk space to hold the result in the View Pool. If, on the other hand, the system had the knowledge that q_2 follows query q_1 it could postpone execution of q_1 until q_2 has been processed and then use the result of q_2 from the View Pool for answering query q_1. In this way, table F will only be scanned once. In addition, assuming a tight space bound that doesn't permit both results from q_1 and q_2 to be stored in the View Pool, the execution order $q_1 \rightarrow q_2$ will unnecessarily materialize q_1 in the View Pool and then probably replace it with the larger result q_2, as shown in



q_1 then q_2: Compute q_1 from MDW, Materialize q_1, Compute q_2 from MDW, Materialize q_2 (replace q1)

q_2 then q_1: Compute q_2 from MDW, Materialize q_2, Compute q_1 from q_2

Fig. 5. Order of execution and time/space tradeoffs.

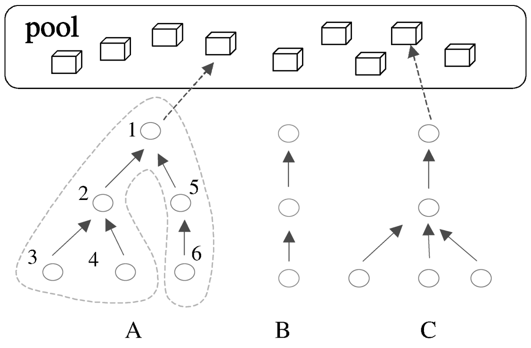


Fig. 6. Computed FDTs for a sample MFX expression.

Figure 5. The alternative execution order $q_2 \to q_1$ on the other hand, avoids this extra materialization.

4.2.2 *Processing an MFX.* Processing an MFX in DynaMat is a two-step approach. In the first, we identify dependencies among the queries in the expression and the materialized fragments in the View Pool. For query q_i in the MFX, we compute the following costs:

(1) the cost $c(q_i, f_j)$ of answering q_i from the result f_j of another query q_j $(j \neq i)$ in the expression, assuming this result has been materialized as a fragment f_j. If more than one query can be used, we pick the one with the smallest cost.
(2) the cost $c(q_i, f_{pool})$ of computing the query from the best candidate f_{pool} in the View Pool as described in Section 4.1.
(3) the cost $c(q_i, DW)$ of executing the query at the data warehouse.

For each query, the alternative with the smallest estimated cost is picked. If a query is to be answered from a fragment in the View Pool, we create a pointer from the query to that fragment. Similarly, if another query's result is to be used, we create a pointer from q_i to q_j. These pointers encode the computed dependencies among the queries and the fragments in the View Pool. Figure 6 shows these dependencies for a sample MFX. The small cubes in the figure represent materialized fragments in the View Pool, while the queries are depicted as circles. The arrows between the queries and the fragments denote the computed pointers and show the most cost-effective way to execute the queries.

This visualization results in a forest of inverted trees that we call *Fragment Dependency Trees* (FDTs). Each one of these trees links queries from the MFX that are related. Some FDTs are linked to a fragment in the View Pool meaning that the root query of the FDT can be answered from the View Pool. The remaining set of trees contain related queries that are not computable from any fragment in the View Pool.

After this phase is completed, we have encoded, in the resulting trees, the best way to answer each query in the MFX. The second phase of processing the MFX is to actually execute and compute the given queries. Each one of the queries is executed individually, taking into account the space restrictions of the system as described in Section 5.3, starting with FDTs whose roots are linked to some fragment in the View Pool. The intuition is that we don't want to postpone the execution of these queries because the pointed fragments might get evicted as new results are constantly introduced in the View Pool. Within a single FDT, queries are executed in a depth-first fashion. This is demonstrated in Figure 6 where the execution order of the queries in the leftmost FDT is shown. The next FDT to execute from that figure is C, while FDT B is executed last since none of its queries uses the View Pool. For that FDT, we want to make sure that the root of the tree remains in the View Pool until the whole tree is processed. DynaMat pins the result of the root-query in the View Pool, assuming of course that its result has actually been admitted, see Section 5.3. This guarantees that all other queries in the same tree will be handled without accessing the data warehouse because they all can be answered from the materialized result of the root query. For FDTs whose root query is linked to a fragment in the View Pool, this precaution step is not necessary because that fragment (or any of its parents) can be used to answer every query in the tree in case that the root or an intermediate node gets evicted from the View Pool during execution.

5. MANAGEMENT OF THE VIEW POOL

5.1 The Time and Space Constraints

During the "online" phase of the data warehouse, results from incoming queries are added to the View Pool. If the View Pool had unlimited disk space, the size of the materialized data would grow monotonically overtime. During an "update" phase u_i, some of the materialized fragments may not be updateable within the time constraint W and thus will be evicted from the View Pool. This is the update *time bound* case shown in Figure 7 with the size of the View Pool increasing between the two update phases u_1 and u_2. The two local minimums correspond to the amount of materialized data that can be updated within W and the local maximums to the View Pool size at the time of the updates.

The *space bound* case is when the size of the View Pool is the constraining factor and not W. In this case, when the View Pool becomes full, we have to use a *replacement* policy. This can vary from simply not admitting more results in the View Pool, to known techniques like LRU, FIFO, etc., or to using heuristics for deciding whether or not a new result is more beneficial for the system than an older one. Figure 8 shows the variations in the View Pool size in this case.

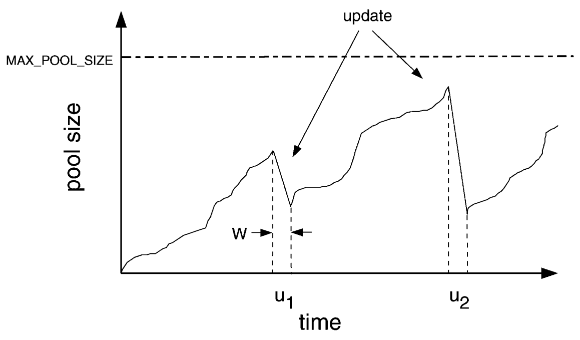


Fig. 7. The Time Bound case.

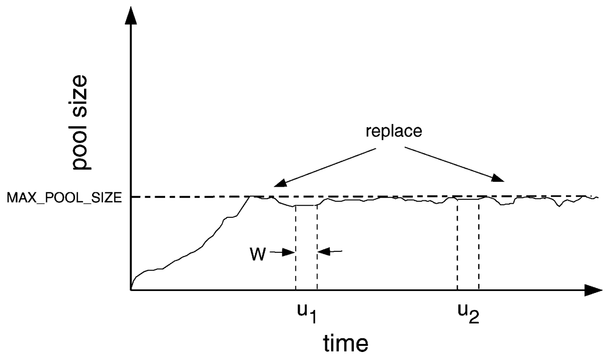


Fig. 8. The Space Bound case.

Since we assumed a sufficiently large update window W, the stored fragments are always updateable and the content of the View Pool is now controlled by the replacement policy.

Depending on the workload, the disk space and the update window, the system may in some cases behave as time or space bound, or both. In such cases, fragments are evicted from the View Pool, either because there is no more space or they can not be updated within the update window, as shown in Figure 9.

5.2 Defining a Goodness Metric

In the previous section, we saw that fragments are evicted from the View Pool either because there is not enough space or because they can not be maintained within the given update window, or both. Management of the fragments in these cases can be modeled as a caching problem, in which the cache resides not in main memory but in the disk. An important difference with traditional caching is that the fragments do not have the same size or the same recomputation costs. This is demonstrated through the following example:

Example 5.1. We assume that V contains only two fragments shown in Figure 10 that are the results of queries q_1 and q_2 of Example 4.5. Fragment f_1 is a higher level aggregation on dimensions `product` and `customer` only and

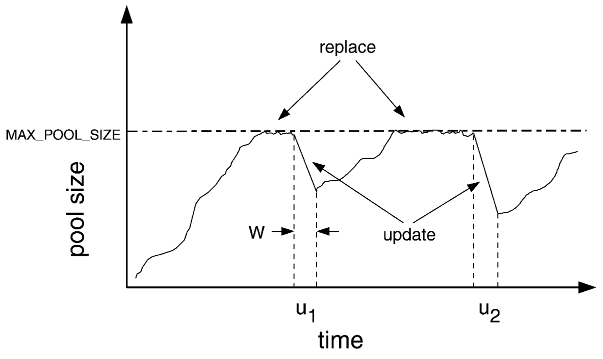


Fig. 9. The Space & Time Bound case.

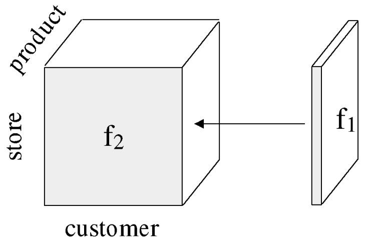


Fig. 10. Father-child relationship among two fragments.

therefore requires less space compared to f_2. What makes this example interesting is that the two fragments are not independent. The *father* pointer between them denotes that as long as fragment f_2 is in the View Pool, f_1 can be recomputed from it without accessing the detailed records. As a result, we expect the recomputation cost of f_1, in case it gets evicted and query q_1 is seen again, to be relatively small. However, if f_2 gets evicted, now q_1 is only computable through f_1 and the recomputation cost of f_1 gets increased. Because of such dependencies among the fragments, their recomputation costs change over time and the management unit of the View Pool should trace and record these changes.

In order to manage the View Pool we derive a *goodness* metric for choosing which of the stored fragments we prefer. This metric is used both in the online and the offline phases. Each time DynaMat reaches the space or time bounds, we use the goodness metric for replacing MRFs. There can be many criteria to define this metric. Among those we tested, the following four showed the best results:

—*The time that the fragment was last accessed by the system to handle a query*:

$$goodness(f) = t_{last\_access}(f).$$

This information is kept in the Directory. Using this time-stamp as a goodness value, results in a Least Recently Used (LRU) type of replacement.

—*The frequency $freq(f)$ of accessing the fragment at query time*:

$$goodness(f) = freq(f).$$

$freq(f)$ is computed using the statistics kept in the Directory and results in a Least Frequently Used (LFU) replacement policy.

—*The raw size*, $size(f)$, *of the result, measured in disk pages*:

$$goodness(f) = size(f).$$

The intuition behind this approach is that larger fragments are more likely to be useful for a query. An additional benefit of keeping larger results in the View Pool is that $|\mathcal{V}|$ gets smaller, resulting in faster look-ups using the Directory and less overhead while updating the View Pool. We refer to this case as the Smaller Fragment First (SFF) replacement policy.

—*The expected penalty rate of recomputing the fragment*, *if it is evicted*, *normalized by its actual size*:

$$goodness(f) = \frac{accesses(f) * c(f)}{staleness(f) * size(f)}.$$

$accesses(f)$ is the total number of accesses to the fragment at query time, $staleness(f)$ is the time since the last access to f and $c(f)$ is the cost of recomputing f for a future query if it gets evicted. We are using as an estimate of $c(f)$ the cost of recomputing the fragment from its father, which is computable in constant time. This metric looks similar to the one used by Scheuermann et al. [1996] for their cache replacement and admission policy. An important difference is that we estimate the cost $c(f)$ dynamically based on the content of the View Pool (i.e., the current father of the fragment). In Scheuermann et al. [1996], the cost assigned to a cached result is static and corresponds to the cost of rerunning the query at the data warehouse. This may lead to assigning large recomputation costs to fragments that can be easily recomputed from the data in the View Pool, as in Figure 10. We refer to this case as the Smaller Penalty First (SPF).

In the remaining part of this section, we describe how the goodness metric is used to control the content of the View Pool.

5.3 On-Line Management at Query Time

As long as there is enough space in the View Pool, results from incoming queries are always stored in $\mathcal{V}$. When the View Pool gets full, we use the *replace* algorithm of Figure 11. The algorithm takes as input the current state of the View Pool $\mathcal{V}$, the new computed result f and the space restriction S. A stored fragment is considered for eviction only if its goodness is less than that of the new result. At a first step, a set $F_{evicted}$ of such fragments with the smaller goodness values is constructed. If, during this process, we cannot find candidate victims the search is aborted and the new result is denied storage in the View Pool.

```
replace($\mathcal{V}$,$f$,$S$) {
  $F_{evicted} \leftarrow \emptyset$;
  while ($size(\mathcal{V} \cup f) - size(F_{evicted}) > S$) {
      pick $f_{victim}$: ((goodness($f_{victim}$)==minimum) &&
                    (goodness($f_{victim}$)<= goodness($f$)))
      if no such fragment abort;
      $F_{evicted} \leftarrow F_{evicted} + \{f_{victim}\}$;
  }
  for each $f_{victim}$ in $F_{evicted}$ {
    for each $f_1$: (father($f_1$)==$f_{victim}$)
      set father($f_1$) = father($f_{victim}$); //Forward father ptrs
    $\mathcal{V} \leftarrow \mathcal{V} - \{f_{victim}\}$; //evict fragment
  }
  $\mathcal{V} \leftarrow \mathcal{V} + \{f\}$; //insert new result
}
```

Fig. 11. The `replace` algorithm.

When a fragment f_{victim} is evicted the algorithm updates the *father* pointer of all fragments that point to f_{victim}. In Section 5.4, we discuss the maintenance of the *father* pointers.

5.4 Off-Line Management During Updates

When the data sources are updated, all data stored in the data warehouse and therefore the fragments in the View Pool, have to be updated too. Different update policies can be implemented, depending on the types of updates, the properties of the data sources and the aggregate functions that are computed by the views. Several methods have been proposed [Agrawal et al. 1996; Deshpande et al. 1996; Sarawagi et al. 1996; Ross and Srivastava 1997; Zhao et al. 1997] for fast (re)-computation of data cube aggregates. On the other hand, incremental maintenance algorithms have been presented [Gupta et al. 1993; Griffin and Libkin 1995; Jagadish et al. 1995; Mumick et al. 1997; Roussopoulos et al. 1997] that handle grouping and aggregation queries.

For our framework, we assume that the sources provide the differentials (called deltas) of the base data, or at least the log files are available. If this is the case, then an incremental update policy can be used to refresh the View Pool. In this scenario, we assume that all the aggregate functions that we compute are *self-maintainable* [Mumick et al. 1997] with respect to the updates that we have. This means that the new value for each function can be computed from the old value and the deltas, thus allowing incremental updates.

5.4.1 *Computing an Initial Update Plan.* Given a View Pool with several thousand fragments, our goal is to derive an *update plan* for refreshing the most "important" of them within a selected update window W. Retrieving the appropriate records from the deltas for each fragment is unrealistic if the deltas are not indexed somehow. In initial tests, we saw that the time spent on identifying the necessary deltas for each fragment is the dominant factor.

For this reason, we extract, in a preprocessing step, all the necessary deltas and store them in a separate view dV materialized as a Cubetree [Kotidis and Roussopoulos 1998], which provides efficient indexing for the deltas against multidimensional range queries. The overhead of loading a Cubetree with the deltas is practically negligible[4] compared to the benefit of having the deltas fully indexed. Assume that low_{d_i} and hi_{d_i} are the minimum and maximum values for dimension d_i that are stored in all fragments in the View Pool. These statistics are easily computed from the Directory. View dV includes all deltas within the hyperplane:

$$\vec{dV} = \{(low_{d_1}, hi_{d_1}), \ldots, (low_{d_n}, hi_{d_n})\}.$$

For each fragment f in $\mathcal{V}$, there are three alternative ways of applying the updates:

—*Incremental updates from dV*: We query dV to get the records that are necessary for refreshing f and then update the fragment incrementally. We denote the cost of this operation as $UC_I(f)$. It consists of the cost of running the MR-query $\vec{f}$ against dV: $c(f, dV)$ and the cost of updating f incrementally from this result.

—*Recompute from another fragment*: If the fragment was originally computed out of another result $\acute{f}$ (i.e., its father), we estimate the cost of recomputing f from $\acute{f}$, after $\acute{f}$ is updated. The cost of computing f from its father is denoted as $UC_R(f)$ and includes the cost $c(f, \acute{f})$ of running MR-query $\vec{f}$ against $\acute{f}$, plus the cost of materializing the result.

—*Recompute from the data warehouse*: We can recompute the fragment from the updated data warehouse. The cost in this case is $c(f, DW)$ plus the cost of materializing the updated result.

The third alternative is, in most cases, much slower that the other two and is not further considered in the analysis. However, for sources that do not provide their differentials during updates or for aggregate functions that are not self-maintenable with respect to the deltas, we should consider this option. The system computes the costs for the first two alternatives and picks the minimum one, denoted as $UC(f)$ for each fragment. Obviously, this plan is not necessarily the best one. There is always the possibility that another result f_1 has been added in the View Pool after f was materialized. Since the selection of the father of f was done before f_1 was around, as explained in Section 4.1, this plan does not consider recomputing f from f_1. An eager maintenance policy of the father pointers would be to refine them whenever necessary, for example, set $father(f) = f_1$, if it is more cost effective to compute f from f_1 than from its current father $\acute{f}$. We have decided to be sloppy and not refine the father pointers based on experiments that showed negligible differences between the lazy and the eager policy. The noticeable benefit is that the lazy approach reduces the worst-case complexity of the `replace` and the `refinePlan` algorithm that is

[4]We have achieved loading rates that exceed 33GB/hour using a SUN Ultra 60 workstation with two Seagate Cheetah disks.

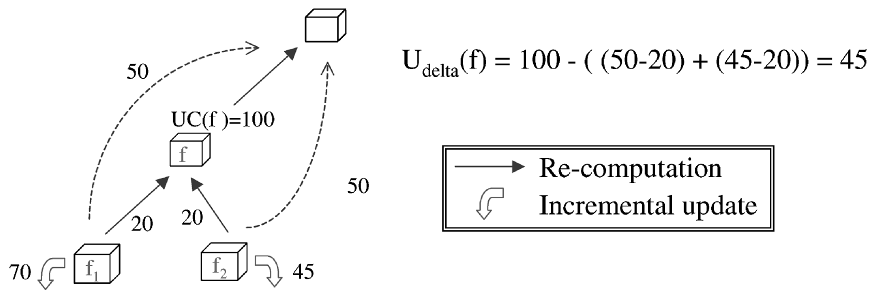


Fig. 12. Forwarding the *father* pointers.

discussed in the next section from $O(|\mathcal{V}|^3)$ down-to $O(|\mathcal{V}|^2)$, thus making the system able to scale for large number of fragments.

By the end of this phase, the system has computed an initial update plan, which directs the most cost-effective way to update each one of the fragments using one of the two alternatives, that is, incrementally from dV or by recomputation from another fragment.

5.4.2 *Computing a Time-bound Update Plan.* The total update cost of the View Pool is $UC(\mathcal{V}) = \sum_{f \in \mathcal{V}} UC(f)$. If this cost is greater than the given update window W, we have to discard a portion of $\mathcal{V}$ and not include it in the new updated version of the View Pool. Suppose that we choose to evict some fragment f. If f is the father of another fragment f_{child} that is to be recomputed from f in the initial update plan, then the reduction in the update cost of the View Pool is less than $UC(f)$, since the update cost of f_{child} increases. For the lazy approach of maintaining the father pointer we *forward* the *father* pointer for f_{child}: set $father(f_{child}) = father(f)$. We now have to check if recomputing f_{child} from $father(f)$ is still a better choice than incrementally updating f_{child} from dV. Let $UC^{new}(f_{child})$ and $UC^{old}(f_{child})$ be the new and old update cost for f_{child} based on this process. The reduction in the overall update cost of the View Pool if we evict fragment f is:

$$U_{delta}(f) = UC(f) - \sum_{\substack{f_{child} \in \mathcal{V}: \\ father(f_{child}) = f}} (UC^{new}(f_{child}) - UC^{old}(f_{child})). \qquad (2)$$

Example 5.2. Figure 12 shows how to apply formula (2) in a sample case where a fragment f with update cost 100 time units is evicted. This fragment has two children, namely f_1 and f_2 that are scheduled to be recomputed from f, since this is cheaper than updating them incrementally. The initial costs for f_1 and f_2 are:[5]

—*Incremental Updates from dV*: $UC_I(f_1) = 70$, $UC_I(f_2) = 45$
—*Recompute from another fragment*: $UC_R(f_1) = UC_R(f_2) = 20$

[5]The recomputation and incremental update costs are shown next to the arrows.

```
refinePlan($\mathcal{V}$,$W$, initial $UC(f)$) {
  //First remove non-updateable fragments
  for each $f$ in $\mathcal{V}$ : $(UC(f) > W)$ {
    for each $f_{child}$: (father($f_{child}$)==$f$) {
      set father($f_{child}$) = father($f$);
      update cost $UC(f_{child})$;
    }
    $\mathcal{V} \leftarrow \mathcal{V} - \{f\}$; //evict $f$
  }
  If $(UC(\mathcal{V}) <= W)$
    return $\mathcal{V}$;
  //Start evicting fragments according to the goodness measure
  While $(UC(\mathcal{V}) > W)$ {
      pick $f$:((goodness($f$)==minimum) && ($U_{delta}(f) > 0$));
      If no such $f$ pick $f$:((goodness($f$)==minimum) && ($U_{delta}(f)$ == 0));
      for each $f_{child}$: (father($f_{child}$)==$f$) {
        set father($f_{child}$) = father($f$);
        update cost $UC(f_{child})$;
      }
      $\mathcal{V} \leftarrow \mathcal{V} - \{f\}$;
  }
  return $\mathcal{V}$;
}
```

Fig. 13. The `refinePlan` algorithm.

If fragment f gets evicted, we forward the *father* pointers of both children and then test if recomputing them from their "grandfather" is still a better choice that the incremental approach. For f_1, this is true but not for child f_2 whose incremental cost is just 45 time units. Therefore, the total reduction of the update cost of the View Pool will not be 100 but just 45 time units since the update cost of f_1 and f_2 combined is increased by 55 time units.

If the initial plan is not feasible, we discard at a first step all fragments whose update cost $UC(f)$ is greater than the window W. If we still exceed the time constraint, we evict more fragments from the View Pool. In this process, there is no point in evicting fragments whose U_{delta} value is less or equal to zero. Having such fragments in the View Pool reduces the total update cost because all their children are efficiently updated from them. For the remaining fragments, we use the goodness metric to select candidates for eviction until the remaining set is updateable within the given window W.

The `refinePlan` algorithm in Figure 13 starts with the initial update plan and evicts fragments until the View Pool is updateable. At a first step, the algorithm discards all fragments whose estimated update cost is larger that W and forwards the *father* pointer of their children, as described above. Since a fragment has at most one father, the number of forwarding steps required for k_1 evictions is $O(k_1|\mathcal{V}|)$, in the worst case. If the remaining set is still not feasible, then, in the second loop, we start evicting results according to their goodness value. If this metric is computed in constant time, the cost for k_2 more evictions is $O(k_2 * |\mathcal{V}| * O(1)) + O(k_2|\mathcal{V}|)^6 = O(k_2|\mathcal{V}|)$. In the extreme case, where W is too small so that only a few fragments can be updated, this leads to an $O(|\mathcal{V}|^2)$ complexity for computing a feasible update plan. However, in many cases, just a small fraction of the stored results will be discarded resulting in pseudolinear execution.

For our simulation, we used a slightly more sophisticated implementation of the `refinePlan` algorithm. We are building a heap structure to order all fragments in the View Pool according to their goodness metric. This takes $O(|\mathcal{V}|)$ time. Using the heap, removing the fragment with the smallest goodness value is an $O(log|\mathcal{V}|)$ step. Notice that, even though the overall forwarding cost is $O(k|\mathcal{V}|)$ in the worst case, usually fragments with a small goodness value do not have children (this is an empirical observation). Thus, in many cases, the cost of forwarding the father pointers is practically negligible. This results in an "empirical" complexity of $O(|\mathcal{V}| + k\ log|\mathcal{V}|) \approx O(|\mathcal{V}|)$ for $k \ll |\mathcal{V}|$.

6. EXPERIMENTS

The comparison and analysis of the different aspects of the design made in this section are based on a simulation that we have developed for DynaMat. We implemented the algorithms and different policies that we present in this article as well as the Directory structure, but not the View Pool architecture. For the latter, we assumed that the fragments were stored in Cubetrees [Roussopoulos et al. 1997] and tuned the simulation accordingly using our Cubetree DataBlade [Kotidis and Roussopoulos 1998] for the Informix Universal Server. We also implemented a random MR-query generator for creating query sets with different statistical profiles.

A important issue for establishing a reasonable set of experiments was to derive the measures to base the comparisons upon. The *Cost Saving Ratio* (*CSR*) was defined in Scheuermann et al. [1996] as a measure of the percentage of the total cost of the queries saved due to hits in their cache system. This measure is defined as:

$$CSR = \frac{\sum_i c_i h_i}{\sum_i c_i r_i}, \tag{3}$$

where c_i is the execution cost of query q_i without using their cache, h_i is the number of times that the query was satisfied in the cache and r_i is the total number of calls to that query. This metric is also used in Deshpande et al. [1998] for their experiments. Because the costs of queries vary widely, CSR is a more appropriate metric than the common hit ratio: $\sum_i h_i / \sum_i r_i$. A potential limitation of CSR, for our case, is that it doesn't capture the different ways that a query q_i might "hit" the View Pool. In case the result of q_i is already materialized in $\mathcal{V}$, the savings is defined as $c_i = c(q_i, DW)$. When another stored result is used, the actual savings depend on how "close" this result is to the answer that we want to produce. If $c_f = c(q_i, f)$ is the cost of querying the best fragment f

6This portion of the formula depicts the cost of forwarding the father pointers.

for answering q_i, the savings in this case is $c_i - c_f$.[7] To capture all cases, we define the savings provided by the View Pool $\mathcal{V}$ for a query q_i as:

$$s_i = \begin{cases} 0 & \text{if } q_i \text{ can not be answered by } \mathcal{V} \\ c_i & \text{if there is an exact match for } q_i \text{ in } \mathcal{V} \\ c_i - c_f & \text{if } f \text{ from } \mathcal{V} \text{ was used to answer } q_i. \end{cases} \tag{4}$$

Using the above formula, we define the Detailed Cost Saving Ratio (DCSR) as:

$$DCSR = \frac{\sum_i s_i}{\sum_i c_i}. \tag{5}$$

DCSR provides a more accurate measure than CSR for OLAP queries. CSR is based on a "binary" definition of a hit: a query hits the View Pool or not. For instance, if a query is computed at the MDW with cost $c_i = 10{,}000$ and from some fragment f with cost $c_f = 9,500$, CSR returns a savings of 10,000 for the "hit", while DCSR credits the system will only 500 units based on the previous formula. DCSR captures the effectiveness of the materialized data against the incoming queries and describes better the performance of the system.

The rest of this section is organized as follows: Section 6.1 makes a direct comparison of the different choices for the goodness metric, described in 5.2. Section 6.2 compares DynaMat against a system that uses the optimal static view selection. Finally, in Section 6.3, we demonstrate that executing queries with MFXs further increases the performance of DynaMat. All experiments were ran using an 300 Mhz Ultra SPARC 60 with 128 MB of main memory running Solaris 2.6.

6.1 Comparison of Different Goodness Metrics

In this set of experiments, we compare the DCSR under the four goodness policies: LRU, LFU, SFF and SPF described in Section 5.2. We used a synthetically generated dataset that models super-market transactions, organized by the star schema. The MDW had 10 dimensions and a fact table containing 20 million tuples. We assumed 50 update phases during the measured life of the system. During each update phase, we generated 250,000 new tuples for the fact table that had to be propagated to the stored fragments. The size of the full data cube for this base data after all updates were applied was estimated to be around 708 GB. We generated 50 query sets with 1,500 MR-queries each that were executed between the updates. These queries were selected uniformly from all $2^{10} = 1,024$ different views in the data cube lattice. In order to simulate hot spots in the query pattern, the values asked by the queries for each dimension are following the 80–20 law: 80% of the times a query was accessing values from 20% of the dimension's domain. We also ran experiments for uniform and Gaussian distributions for the query values but are not presented here as they were similar to the 80–20% distribution.

[7] c_i and c_f do not include the cost to fetch the result which is payable even if an exact match is found.

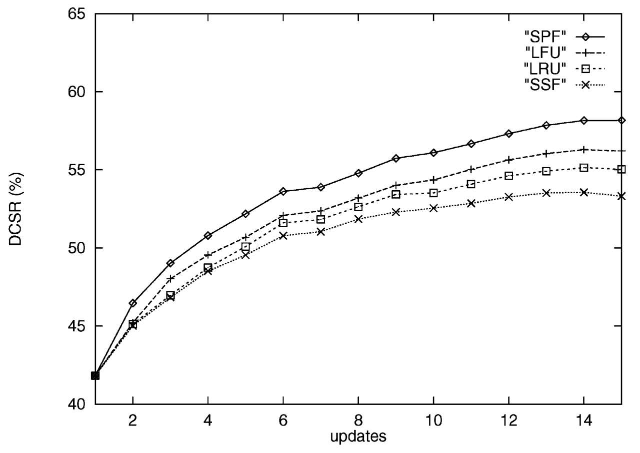


Fig. 14. First 15 × 1500 queries.

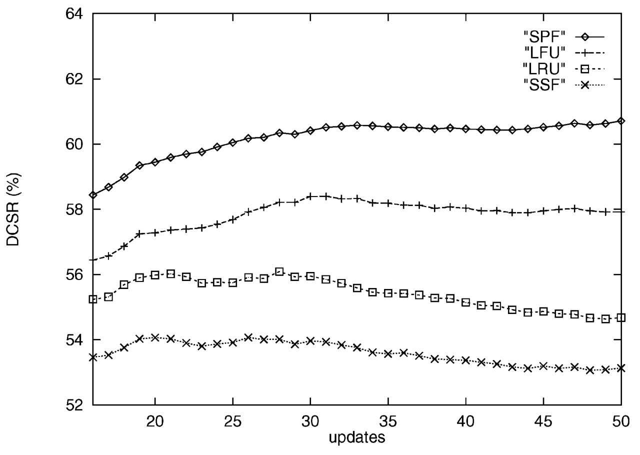


Fig. 15. Remaining 35 × 1500 queries.

For the first experiment, we tested the time-bound case. The size of the View Pool was chosen large enough to guarantee no replacement during queries and the time allowed for updating the fragments was set to 2% of the estimated time to update the full data cube, which we denote as W_{Data_Cube}. For a more clear view, we plot in Figure 14 the DCSR overtime for the first 15 sets of queries, starting with an empty View Pool. In the graph, we plot the cumulative value of DCSR at the beginning of each update phase, for all queries that happened up to that phase. The DCSR value reaches 41.4% at the end of the first query period of 1,500 queries that were executed against the initially empty View Pool. This means that simply by storing and reusing computed results from previous queries, we cut down the cost of accessing the MDW to 58.6%. Figure 15 shows how DCSR changes for the remaining queries. All four policies quickly increase their savings, by refining the content of the View Pool while doing updates,

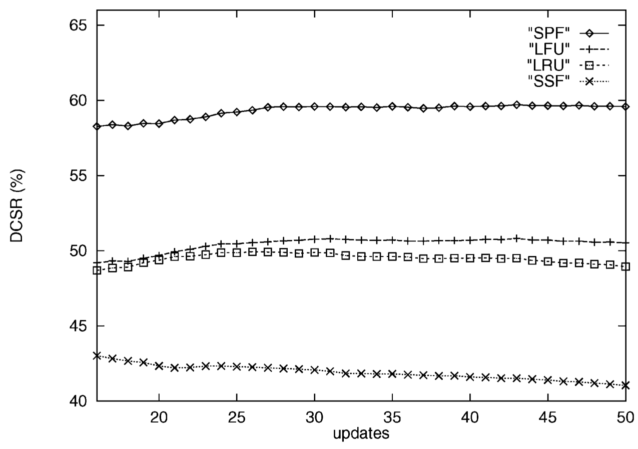


Fig. 16. The Space Bound case.

up to a point where all curves flatten out. At all times, the SPF policy is the winner with 60.71% savings for the whole run. The average I/O per query, was 94.84, 100.08, 106.18 and 109.09 *MB*/query for the SPF, LFU, LRU and SFF policies respectively. The average write-back I/O cost for materializing the fragments was about the same in all cases ($\simeq$19.8 *MB*/query). For the winner SPF policy, the average time spent on searching the Directory was negligible (about 0.4 ms/query). Computing a feasible update plan took on the average 37 ms, and 51 ms in the worst case. The number of MRFs stored in the View Pool by the end of the last update phase was 206.

Figure 16 depicts DCSR overtime in the space-bound case for the last 35 sets of queries, calculated at the beginning of each update phase. In this experiment, there was no time restriction for applying the updates, and the space that was allocated for the View Pool was set to 14 GB, that is, 2% of the full data cube size. The content of the View Pool is now managed by the *replace* algorithm, as the limited storage space results in frequent evictions during the on-line mode. Again, the SPF policy showed the best performance with a DCSR of 59.58%. For this policy, the average time spend on the *replace* algorithm, including any modifications in the Directory, was less that 3 ms per query. Computing the initial update plan for the updates, as explained in Section 5.4, took 10 ms on the average. Since there was no time restriction, this plan was always feasible. The final number of fragments in the View Pool was 692.

In a final experiment, we tested the four policies for the general case, where the system is both space and time bound. We varied the time window for the updates from 0.2% up to 5% of W_{Data_Cube} and the size of the View Pool from 0.2% up to 5% of the full data cube size, both in 0.2% intervals. Figure 17 shows the DCSR for each pair of time and space settings for the SPF policy, that outperformed the other three. We can see that even with limited resources DynaMat provides substantial savings. For example, with just 1.2% of disk space and 0.8% time window for the updates, we get over 50% savings compared to accessing the MDW.

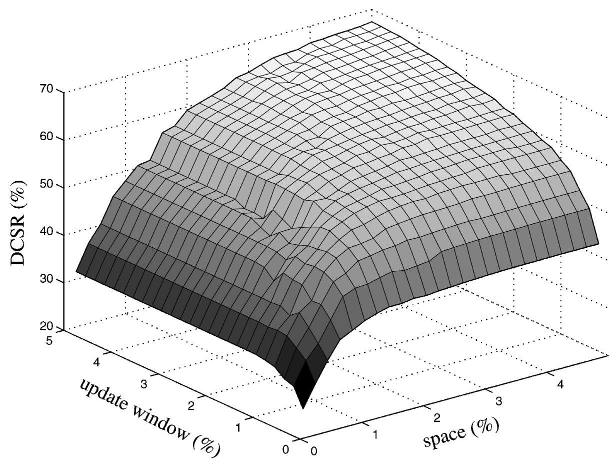


Fig. 17. The Space & Time Bound case.

6.2 Comparison with the Optimal Static View Selection

In the experiments of the previous section, we found the SPF policy to be superior for dynamic view (fragment) selection during both updates (time-bound case) and queries (space-bound case), or both. An important question, however, is how the system compares with a static view selection algorithm like Harinarayan et al. [1996], Gupta et al. [1997], Gupta [1997], and Baralis et al. [1997] that considers only fully materialized views. Instead of comparing each one of these algorithms with DynaMat, we implemented SOLVE, a module that, given a set of queries, the space and time restrictions, searches exhaustively all feasible view selections and returns the optimal one for these queries. For a data cube lattice with n dimensions and no hierarchies, there are 2^n different views. A static view selection, depending on the space and time bounds, contains some combination of these views. For for $n = 6$, the search space contains $2^{2^6} = 18,446,744,073,709,551,616$ possible combinations of the 64 views of the lattice. Obviously, some pruning can be applied. For example, if a set of views is found feasible, there is no need to check any of its subsets. Additional pruning of large views is possible depending on the space and time restrictions that are specified; however, for nontrivial cases, this exhaustive search is not feasible but for very small values of n.

We used SOLVE to compute the optimal static view selection for a 6-dimensional subset of our synthetic supermarket dataset, with 20 million tuples in the fact table. There were 40 update phases, with 100 thousand new tuples being inserted each time. The time window W for the updates was set to the estimated 2% of that of the full data cube (W_{Data_Cube}). We created 40 sets of 500 MR-queries each that were executed between the updates. These queries targeted uniformly the 64 different views of the 6-dimensional data cube lattice. This lack of query locality is bad for the dynamic case that needs to adapt to the incoming query pattern. For the static view selection, query locality is not an issue, because SOLVE was given all queries in advance. The optimal set

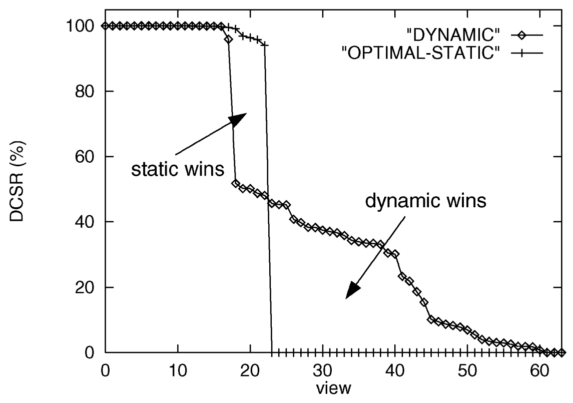


Fig. 18. DCSR per view for uniform queries on the views.

returned, after 3 days of computations in our hardware platform, includes 23 out of the 64 full-views in the 6-dimensional data cube. The combined size of these views in the disk is 281 MB (1.6% of the full data cube). For the most strict and unfavorable comparison for the dynamic case, we set the size of the View Pool to the same number. Since the dynamic system started with an empty View Pool, we used the first 10% of the queries as a training set and measured system's performance for the remaining 90%. We used the SPF policy to derive the goodness of the MRFs for the dynamic approach.

The measured cumulative DCSR for the two systems was about the same: 64.04% for the dynamic and 62.06% for the optimal static. The average I/O per query for the dynamic system was 108.11 MB and the average write-back I/O cost 2.18 MB. For the optimal static selection, the average I/O per query is 112.94 MB and no write-back, without counting the overhead of materializing the statically selected views for the first time.

For a clearer view on the performance differences between the static and the dynamic approach, we computed the DCSR per view and plotted them in decreasing order of savings in Figure 18. Notice that the x-axis labeling does not correspond to the same views for the two lines. The plot shows that the static view selection performs well for the 23 materialized views; however, for the remaining 41 views, its savings drops to zero. DynaMat, on the other hand, provides savings for almost all the views. The right-hand side of the graph contains the larger views of the data cube. Since most results from queries on these views are too big to fit in the View Pool, even DynaMat's performance decreases because they cannot be materialized in the shared disk volume.

Figure 19 depicts the performance of both systems for a nonuniform set of queries where the access to the views is skewed. The skewness is controlled by the number of grouping attributes in each query. As this number increases,[8] it favors accesses on views from the upper levels of the data cube lattice, because the views are bigger in size and need a larger update window. These views,

[8]Having three grouping attributes per query, on the average, corresponds to the previous uniform view selection.

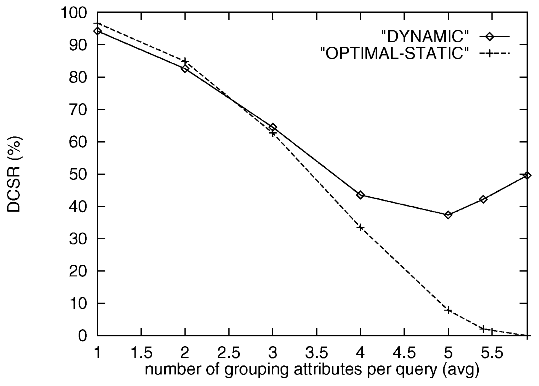


Fig. 19. Varying the average number of grouping attributes per query.

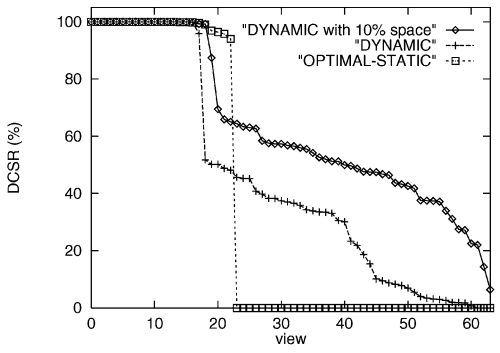


Fig. 20. Extra disk space (10%).

because of the space and time constraints, are not in the static optimal selection. On the other hand, the dynamic approach provides results whenever possible and, for this reason, it is more robust than the static selection, as the workload shifts to the larger views of the lattice. As the average number of grouping attributes per query reaches 6, most queries access the single top-level six-dimensional view of the lattice. DynaMat adapts nicely to this query pattern and allocates most of the View Pool space to MRFs of that view. That explains the performance of DynaMat going up at the right-hand side of the graph.

The View Pool size in the above experiments was set to 1.6% of the full data cube as this was the actual size of the views used by the optimal static selection. This number, however, is rather small by today's standards. We ran two more experiments with View Pool size 5% (878 MB) and 10% (1.7 GB) of the full data cube size. The optimal static selection does not refine the selected views because of the update window constraint (2%). DynaMat, on the other hand, capitalizes the extra disk space and increases the DCSR from 64.04% to 68.34% and 78.22% for the 5% and 10% storage, respectively. Figure 20 depicts the computed DCSR per view for this case. As more disk space is available, DynaMat achieves even more savings by materializing more fragments from

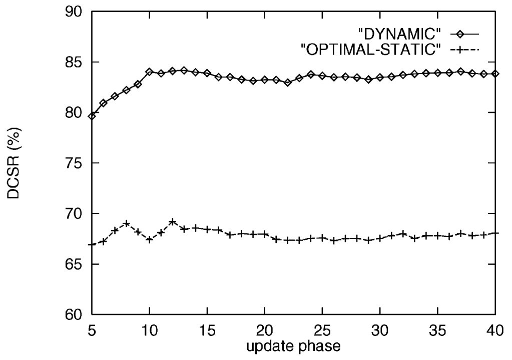


Fig. 21. Drill-down/roll-up queries.

the larger views of the data cube. This experiment demonstrates the important difference between the static and dynamic systems: because of the continuous management, DynaMat utilizes both the available time and space constraints while the static system typically utilizes only one of these two resources; the update time in this experiment.

In the previous experiment, the queries that we ran were selected uniformly from all 64 views in the data cube lattice. Often in OLAP, users do follow-up queries, such as drill-downs or roll-ups, where starting from a computed result, they refine their queries and ask for a more or less detailed aggregate view of the data, respectively. DynaMat enormously benefits from roll-up queries because they are always computable from results that were previously added in the View Pool. To simulate such a workload we created 40 sets of 500 queries each with the following properties: 40% of the times a user asks a query on a randomly selected view from the data cube, 30% of the times the user performs a roll-up operation on the last reported result, and 30% of the times the user performs a drill-down.

For this experiment, we used the previous set-up for the 2% and 10% time and space bounds and we recomputed the optimal static selection for the new queries. Figure 21 depicts DCSR for this workload. Compared to the previous example, DynaMat further increases its savings by taking advantage of the locality of the roll-up queries.

6.3 Experiments with Multi Fragment Expressions

For the first experiment, we used the previous 6-dimensional synthetic dataset and we generated a random set of 500 queries. For this run, we did not impose any space restriction for the View Pool and there were no updates since we wanted to concentrate on the execution of the queries only. The cumulative DCSR after we executed all queries starting with an empty View Pool was 70.54%. We then grouped the same queries using MFXs of various lengths. For the first MFX execution, we grouped every 10 consecutive queries into a single expression. This resulted in having 50 expressions with 10 queries each. Table I shows the measured DCSR values as the number of queries in the

Table I. Cumulative Savings vs MFX Size

MFX Size	1	10	20	50	100	200	500
DCSR	70.54	71.09	71.15	72.24	73.51	75.61	80.01 (optimal)

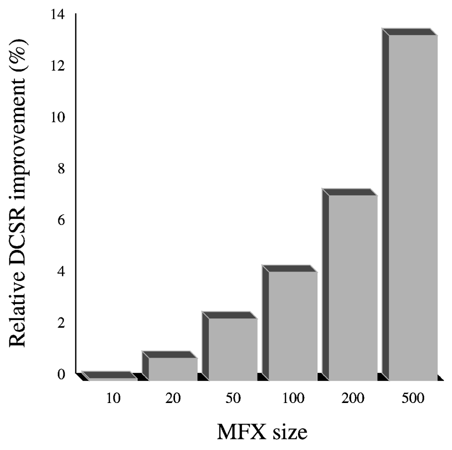


Fig. 22. MFX processing for uniform queries.

expressions increased up to 500. For the latter run, all queries were bundled into a single expression and issued to DynaMat as a unit. The DCSR value for this execution was 80.01% and corresponds to the maximum value reported. Because there were no space or update-time restrictions, this is the optimal savings that can be obtained by reordering their execution starting with an empty View Pool.

In Figure 22, we have computed the relative improvements in the measured DCSR over the starting point of 70.54% that corresponds to the original execution without using MFXs. The figure shows that, even for random uncorrelated queries, grouping them in larger sets always improves the performance.

As a more representative workload for OLAP, we also tested another set of 500 correlated queries, similar to the ones used in the last experiment of the previous section. Figure 23 shows the relative improvements in the measured DCSR by using MFXs, as the number of queries in each group increases from 10 to 500. This time because of the stronger correlation among the queries, even smaller groupings are adequate for getting higher improvements. The optimal execution of all queries in a single MFX provides in this case relative improvement of 17.3% over the single-query execution model. This comes from the ability of the system to exploit and gain from all types of correlations among the queries. On the contrary, the single-query execution model provides savings only for roll-up type of analysis due to hits in the View Pool as seen in Figure 21.

7. DISCUSSION

Dynamic management of materialized views can be formulated as a caching problem. Semantic caching of query results has been introduced in Dar et al.

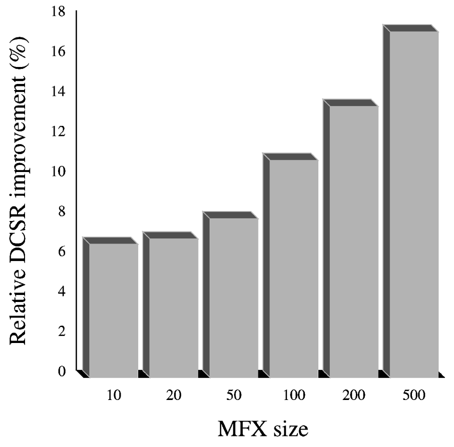


Fig. 23. MFX processing for drill-down/roll-up queries.

[1996] for Client—Server main memory architectures. The client maintains a semantic description of the cached data, which allows tuples that are not available in the cache to be requested using a "remainder" query. Semantic caching incorporates semantic notion of locality for cache replacement unlike page-level caching that chooses victims based on temporal and spatial locality. In a similar context, Keller and Basu in [1996] utilize a collection of constraint formulas to describe cached data regions.

An important difference of our design from traditional caching architectures is that the materialized OLAP aggregates have different computation costs. For instance, it is far more expensive to recompute results of a high level of aggregation (e.g., query q_1 query q_2 in Example 4.5) since they require scanning and processing lots of detailed records. A nice formulation of the problem of caching precomputed results in relational database systems is given by Sellis [1988b]. The WATCHMAN cache manager, introduced by Scheuermann et al. [1996], uses replacement and admission techniques specifically designed for data warehousing workload. Similar techniques are also described in Deshpande et al. [1998] where caching is introduced for data warehouses organized based on a specialized chunked file organization [Sarawagi and Stonebraker 1994]. With the exception of Deshpande and Naughton [2000], previous caching schemes fail to address and exploit interdependencies among materialized aggregates of different levels of aggregation. Drill-down and roll-up queries that are common in OLAP analysis tend to fill the cache with correlated aggregates of different aggregation levels. An effective caching architecture should understand and exploit the dependencies among these aggregates. In addition, all previous systems were introduced for main-memory caching. As a result, they do not address updates and the memory cache is simply invalidated each time the data warehouse is updated. This practice is not suitable for disk resident-materialized aggregates with potentially large recomputation overhead.

An underlying question has to do with the applicability of this line of research in real systems. After all, query result reuse has been discussed in various context (e.g., caching, materialized views, multiquery optimization) in the literature but little progress has been made in implementing these ideas in commercial systems. We believe that query-reuse is a viable technique in data warehousing environments for the following reasons:

—*Ad-hoc analysis is very unpredictable*. It is hard to derive a static set of views that fits all users. Thus, adaptive techniques that materialize frequent computations are necessary.

—*Data cube views have a rather restricted form that makes the problem somehow easier*. Query processing in DynaMat, described in Section 4.1, implements a simple rewriting policy based on a multidimensional description of the stored results. In the general case of SPJ-views, there are quite a few algorithms [Larson and Yang 1985; Yang and Larson 1987; Chaudhuri et al. 1995; Levy et al. 1995; Srivastava et al. 1996] that could be used to optimize the execution of user queries against the fragments. Even though such techniques are applicable for our case, we chose a cleaner engineering-oriented approach that in practice seems to perform well. Our query model allows for an abstract interface, shown in Figure 1, that is potentially easier to be integrated into existing systems.

—*Query results are often relatively small, as they contain aggregated data*. The potential saving when reusing computed aggregates justify the additional complexity of bookkeeping and managing these results.

—*Data is relatively static with only infrequent updates that are happening in predetermined intervals*. Thus, a stored result has a potentially long period of time to prove itself useful for future queries.

Interestingly, we find recent papers from database vendors [Bello et al. 1998; Zaharioudakis et al. 2000; Goldstein and Larson 2001] on the use of materialized views (that are, after all, query results) in query optimization. DB2, for example, has implemented the capability to define Automatic Summary Tables and Replicated Tables, which materialize frequently referenced computations redundantly to enhance query performance.

In Section 4.2, we extended DynaMat's API to handle multiple, possible related queries. The optimization problem that we addressed consists of finding the execution order and materialization/replacement plan that best utilizes the available materialized fragments of the View Pool according to the given disk space constraint. Our solution was based on an open-world assumption meaning that during the processing of an expression the View Pool is available for other users/queries. Under the assumption of an empty View Pool with an infinite disk space, the problem is very similar to a multiquery optimization problem. Global query optimization has long been studied [Sellis 1988a; Shim et al. 1994; Roy et al. 2000] for database applications. Recently, Zhao et al. [1998] have addressed the problem of optimizing multiple dimensional queries in the content of data warehousing. Their work focuses in processing *Multi-Dimensional Expressions* (*MDX*) defined in the OLE DB for OLAP standard by Microsoft [Microsoft]. MDX provide a framework in which a user can naturally ask several related OLAP queries in a single unit. The authors consider the evaluation of MDX expressions by a relational database system.

Compared to previous work in multiquery optimization MDX provide new opportunities of optimizations because of the more restricted nature of the OLAP queries involved.

8. CONCLUSIONS

In this article, we made a case for dynamic management of views in data warehouses. We presented DynaMat, a view management system that materializes results from incoming queries as views and exploits them for future reuse. DynaMat unifies view selection and view maintenance under a single framework that takes into account both the update time and space constraints of the data warehouse.

We have defined and used the Multidimensional Range Fragments (MRFs) as the basic logical unit of materialization. Our experiments show that, compared to the conventional paradigm that considers only full views, MRFs provide a finer and more appropriate granularity of materialization. The operational and maintenance cost of MRFs, which includes directory look-up operations at query time and the derivation of an efficient update plan during updates, remains practically negligible, in the order of milliseconds. At query time, we utilize a distributed directory structure that permits fast access to the stored fragments. Queries are executed independently, or can be bundled within a multiquery expression. In both cases, execution is optimized with respect to the materialized data and the available disk space.

We compared DynaMat against a system that is given all queries in advance and the precomputed optimal static view selection. These experiments indicate that DynaMat outperforms the optimal static selection and thus any suboptimal view selection algorithm that has appeared in the literature. Another important result that validates the importance of materialized views, is that just 1–2% of the space and 1–2% of the update window required for the full data cube are sufficient for substantial performance improvements.

However, the most important feature of DynaMat is that it represents a self-tunable solution that adapts to new query patterns and new time/space constraints. This way DynaMat relieves the data warehouse administrator from having to monitor and calibrate the system constantly. Even for cases that there is no specific pattern in the workload, like the uniform queries used for some of our experiments, DynaMat manages to pick a set of MRFs that outperforms the optimal static view selection. For more skewed query distributions, especially for workloads that include a lot of roll-up queries, the performance of DynaMat is even better.

REFERENCES

Abiteboul, S. and Duschka, O. M. 1998. Complexity of answering queries using materialized views. In *Proceedings of the 17th Annual ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Seattle, Wash., June). ACM, New York, pp. 254–263.

Agrawal, S., Agrawal, R., Deshpande, P., Gupta, A., Naughton, J., Ramakrishnan, R., and Sarawagi, S. 1996. On the computation of multidimensional aggregates. In *Proceedings of the 22nd VLDB conference* (Bombay, India, Aug.), pp. 506–521.

Baralis, E., Paraboschi, S., and Teniente, E. 1997. Materialized view selection in a multidimensional database. In *Proceedings of the 23th International Conference on VLDB* (Athens, Greece, Aug.), pp. 156–165.

Bello, R. G., Dias, K., Downing, A., Jr., J. F., Norcott, W. D., Sun, H., Witkowski, A., and Ziauddin, M. 1998. Materialized views in oracle. In *Proceedings of the 24rd International Conference on Very Large Data Bases* (New York, New York, Aug.), pp. 659–664.

Chaudhuri, S. and Dayal, U. 1997. An overview of data warehousing and OLAP technology. *SIGMOD Record*, *26*, 1 (Sept.).

Chaudhuri, S., Krishnamurthy, R., Potamianos, S., and Shim, K. 1995. Optimizing queries with materialized views. In *Proceedings of the 11th International Conference on Data Engineering* (Taipei, Taiwan, Mar.), pp. 190–200.

Chen, C. and Roussopoulos, N. 1994. The implementation and performance evaluation of the ADMS query optimizer: Integrating query result caching and matching. In *Proceedings of EDBT* (Cambridge, UK, Mar.), pp. 323–336.

Dar, S., Franklin, M., Jonsson, B., Srivastava, D., and Tan, M. 1996. Semantic data caching and replacement. In *Proceedings of the 22th International Conference on VLDB* (Bombay, India, Sept.), pp. 330–341.

Delis, A. and Roussopoulos, N. 1992. Performance and scalability of client-server database architectures. In *Proceedings of the 18th VLDB* (Vancouver, Canada), pp. 610–623.

Deshpande, P., Agrawal, S., Naughton, J., and Ramakrishnan, R. 1996. Computation of multidimensional aggregates. Tech. Rep. 1314, Univ. Wisconsin, Madison, Madison, Wis.

Deshpande, P. and Naughton, J. 2000. Aggregate aware caching for multi-dimensional queries. In *Proceeding of the 7th International Conference on Extending Database Technology* (Konstanz, Germany, Mar.), pp. 167–182.

Deshpande, P., Ramasamy, K., Shukla, A., and Naughton, J. 1998. Caching multidimensional queries using chunks. In *Proceedings of the ACM SIGMOD* (Seattle, Wash. June), pp. 259–270.

Do, L., Drew, P., Jin, W., Junami, V., and Rossum, D. V. 1998. Issues in developing very large data warehouses. In *Proceedings of the 24th VLDB Conference* (New York, N.Y., Aug.), pp. 633–636.

Goldstein, J. and Larson, P. 2001. Optimizing queries using materialized views: A practical, scalable solution. In *Proceedings of ACM SIGMOD International Conference on Management of Data* (Santa Barbara, Calif., May), ACM, New York.

Gray, J., Bosworth, A., Layman, A., and Piramish, H. 1996. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Proceedings of the 12th ICDE Conference* (New Orleans, La., Feb.), IEEE Computer Society Press, Los Alamitos, Calif., pp. 152–159.

Griffin, T. and Libkin, L. 1995. Incremental maintenance of views with duplicates. In *Proceedings of the ACM SIGMOD Conference* (San Jose, Calif., May). ACM, New York, pp. 328–339.

Gupta, A., Mumick, I., and Subrahmanian, V. 1993. Maintaining views incrementally. In *Proceedings of the ACM SIGMOD Conference* (Washington, D.C., May). ACM, New York, pp. 157–166.

Gupta, H. 1997. Selections of views to materialize in a data warehouse. In *Proceedings of ICDT* (Delphi, Greece, Jan.), pp. 98–112.

Gupta, H., Harinarayan, V., Rajaraman, A., and Ullman, J. 1997. Index selection for OLAP. In *Proceedings of ICDE* (Burmingham, U.K., Apr.), pp. 208–219.

Harinarayan, V., Rajaraman, A., and Ullman, J. 1996. Implementing data cubes efficiently. In *Proceedings of ACM SIGMOD* (Montreal, Ont., Canada, June). ACM, New York, pp. 205–216.

Jagadish, H., Mumick, I., and Silberschatz, A. 1995. View maintenance issues in the chronicle data model. In *Proceedings of PODS* (San Jose, Calif.). ACM, New York, pp. 113–124.

Jermaine, C., Datta, A., and Omiecinski, E. 1999. A novel index supporting high volume data warehouse insertions. In *Proceedings of 25th International Conference on Very Large Data Bases* (Edinburgh, Scotland, U.K., Sept.), pp. 235–246.

Karloff, H. J. and Mihail, M. 1999. On the complexity of the view-selection problem. In *Proceedings of the 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Philadelphia, Pa., May). ACM, New York, pp. 167–173.

Keller, A. and Basu, J. 1996. A predicate-based caching scheme for client-server database architectures. *VLDB J. 5*, 1, 35–47.

Kimball, R. 1996. *The Data Warehouse Toolkit*. J. Wiley, New York.

KOLAITIS, P. G. AND VARDI, M. Y. 1998. Conjunctive-query containment and constraint satisfaction. In *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Seattle, Wash., June). ACM, New York, pp. 205–213.

KOTIDIS, Y. AND ROUSSOPOULOS, N. 1998. An alternative storage organization for ROLAP aggregate views based on cubetrees. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Seattle, Wash., June). ACM, New York, pp. 249–258.

KOTIDIS, Y. AND ROUSSOPOULOS, N. 1999. DynaMat: A dynamic view management system for data warehouses. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Philadelphia, Pa., June). ACM, New York, pp. 371–382.

LARSON, P.-A. AND YANG, H. Z. 1985. Computing queries from derived relations. In *Proceedings of the 11th VLDB Conference* (Stockholm, Sweden), pp. 259–269.

LEVY, A. Y., MENDELZON, A. O., SAGIV, Y., AND SRIVASTAVA, D. 1995. Answering queries using views. In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (San Jose, Calif., May). ACM, New York, pp. 95–104.

MICROSOFT. OLE DB for OLAP. http://www.microsoft.com/data/olepdb.

MUMICK, I. S., QUASS, D., AND MUMICK, B. S. 1997. Maintenance of data cubes and summary tables in a warehouse. In *Proceedings of the ACM SIGMOD Conference* (Tucson, Az., May). ACM, New York, pp. 100–111.

O'NEIL, P. E., CHENG, E., GAWLICK, D., AND O'NEIL, E. J. 1996. The log-structured merge-tree (LSM-tree). *Acta Inf. 33*, 4, 351–385.

ROSS, K. AND SRIVASTAVA, D. 1997. Fast computation of sparse datacubes. In *Proceedings of the 23th VLDB Conference* (Athens, Greece, Aug.), pp. 116–125.

ROUSSOPOULOS, N. 1982. View indexing in relational databases. *ACM Trans. Datab. Syst. 7*, 2 (June), 258–290.

ROUSSOPOULOS, N. 1991. The incremental access method of view cache: Concept, algorithms, and cost analysis. *ACM Trans. Datab. Syst. 16*, 3 (Sept.), 535–563.

ROUSSOPOULOS, N. AND KANG, H. 1986. Preliminary design of ADMS±: A workstation-mainframe integrated architecture for database management systems. In *Proceedings of the 12th International Conference on VLDB* (Kyoto, Japan, Aug.), pp. 355–364.

ROUSSOPOULOS, N., KOTIDIS, Y., AND ROUSSOPOULOS, M. 1997. Cubetree: Organization of and bulk incremental updates on the data cube. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Tucson, Az., May), ACM, New York, pp. 89–99.

ROY, P., SESHADRI, S., SUDARSHAN, S., AND BHOBE, S. 2000. Efficient and extensible algorithms for multi query optimization. In *Proceedings of ACM SIGMOD International Conference on Management of Data* (Dallas, Tex., May). ACM, New York, pp. 249–260.

SARAWAGI, S., AGRAWAL, R., AND GUPTA, A. 1996. On computing the data cube. Tech. Rep. RJ10026. IBM Almaden Research Center, San Jose, Calif.

SARAWAGI, S. AND STONEBRAKER, M. 1994. Efficient organization of large multidimensional arrays. In *Proceedings of ICDE* (Houston, Tex.), pp. 328–336.

SCHEUERMANN, P., SHIM, J., AND VINGRALEK, R. 1996. WATCHMAN: A data warehouse intelligent cache manager. In *Proceedings of the 22th VLDB Conference* (Bombay, India, Sept.), pp. 51–62.

SELLIS, T. 1988a. Multiple-query optimization. *ACM Trans. Datab. Syst. 13*, 1, 23–52.

SELLIS, T. K. 1988b. Intelligent caching and indexing techniques for relational database systems. *Inf. Syst. 13*, 2, 175–185.

SHIM, K., SELLIS, T., AND NAU, D. 1994. Improvements on a heuristic algorithm for multiple-query optimization. *DKE 12*, 2, 197–222.

SHUKLA, A., DESHPANDE, P., AND NAUGHTON, J. F. 1998. Materialized view selection for multidimensional datasets. In *Proceedings of the 24th VLDB Conference* (New York, New York, Aug.), pp. 488–499.

SMITH, J. R., LI, C., CASTELLI, V., AND JHINGRAN, A. 1998. Dynamic assembly of views in data cubes. In *Proceedings of the Symposium on Principles of Database Systems* (*PODS*) (Seattle, Wash., June). ACM, New York, pp. 274–283.

SRIVASTAVA, D., DAR, S., JAGDISH, H., AND LEVY, A. Y. 1996. Answering queries with aggregation using views. In *Proceedings of the 22nd International Conference on Very Large Data Bases* (Mumbai (Bombay), India, Sept.), pp. 318–329.

THEODORATOS, D. AND SELLIS, T. 1997. Data warehouse configuration. In *Proceedings of the 23th International Conference on VLDB* (Athens, Greece, Aug.), pp. 126–135.

YANG, H. Z. AND LARSON, P.-Å. 1987. Query transformation for PSJ-queries. In *Proceedings of 13th International Conference on Very Large Data Bases* (Brighton, England, Sept.), pp. 245–254.

ZAHARIOUDAKIS, M., COCHRANE, R., LAPIS, G., PIRAHESH, H., AND URATA, M. 2000. Answering complex SQL queries using automatic summary tables. In *Proceedings of ACM SIGMOD International Conference on Management of Data* (Dallas, Tex., May). ACM, New York, pp. 105–116.

ZHAO, Y., DESHPANDE, P., AND NAUGHTON, J. 1997. An array-based algorithm for simultaneous multidimensional aggregates. In *Proceedings of the ACM SIGMOD Conference* (Tucson, Az., May). ACM, New York, pp. 159–170.

ZHAO, Y., DESHPANDE, P., NAUGHTON, J. F., AND SHUKLA, A. 1998. Simultaneous optimization and evaluation of multiple dimensiona queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Seattle, Wash., June). ACM, New York, pp. 271–282.

Received September 1999; revised July 2000, November 2000, and February 2001; accepted June 2001