DynaMat: A Dynamic View Management System for Data Warehouses*

Yannis Kotidis
Department of Computer Science
University of Maryland
kotidis@cs.umd.edu

Nick Roussopoulos
Department of Computer Science
University of Maryland
nick@cs.umd.edu

Abstract

Pre-computation and materialization of views with aggregate functions is a common technique in Data Warehouses. Due to the complex structure of the warehouse and the different profiles of the users who submit queries, there is need for tools that will automate the selection and management of the materialized data. In this paper we present DynaMat, a system that dynamically materializes information at multiple levels of granularity in order to match the demand (workload) but also takes into account the maintenance restrictions for the warehouse, such as down time to update the views and space availability. DynaMat unifies the view selection and the view maintenance problems under a single framework using a novel "goodness" measure for the materialized views. DynaMat constantly monitors incoming queries and materializes the best set of views subject to the space constraints. During updates, DynaMat reconciles the current materialized view selection and refreshes the most beneficial subset of it within a given maintenance window. We compare DynaMat against a system that is given all queries in advance and the pre-computed optimal static view selection. The comparison is made based on a new metric, the Detailed Cost Savings Ratio introduced for quantifying the benefits of view materialization against incoming queries. These experiments show that DynaMat's dynamic view selection outperforms the optimal static view selection and thus, any sub-optimal static algorithm that has appeared in the literature.

1 Introduction

Materialized views represent a set of redundant entities in a data warehouse that are used to accelerate On-Line Analytical Processing (OLAP). A substantial effort of the academic community in the last years [HRU96, GHRU97, Gup97, BPT97, SDN98] has been for a given workload, to select an appropriate set of views that would provide the best performance benefits. The amount of redundancy added is controlled by the data warehouse administrator who specifies the space that is willing to allocate for the materialized data. Given this space restriction and, if available, some description of the workload, these algorithms return a suggested set of views that can be materialized for better performance.

This static selection of views however, contradicts the dynamic nature of decision support analysis. Especially for add-hoc queries where the expert user is looking for interesting trends in the data repository, the query pattern is difficult to predict. In addition, as the data and these trends are changing overtime, a static selection of views might very quickly become outdated. This means that the administrator should monitor the query pattern and periodically "re-calibrate" the materialized views by re-running these algorithms. This task for a large warehouse where many users with different profiles submit their queries is rather complicated and time consuming. Microsoft's [Aut] is a step towards automated management of system resources and shows that vendors have realized the need to simplify the life of the data warehouse administrator.

Another inherit drawback of the static view selection is that the system has no way of tuning a wrong selection, i.e use 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, in many cases, relatively small. "`Find the total volume of sales for the last 10 years`" is a fine example of that. Processing this query might take hours of scanning vast tables and aggregating, while the result is just an 8-byte float value that can be easily "cached" for future use. Moreover, during *roll−up* operations, when we access data at a progressively coarser granularity, future queries are likely to be totally computable out of the results of previous operations, without accessing the base tables at all. Thus, we expect a great amount of inter-dependency among a set of OLAP queries.

Furthermore, selecting a view set to materialize is just the tip of the iceberg. Clearly, query performance is

*This research was sponsored partially by NASA under grant NAG 5-2926, by NSA/Lucite under contract CG9815, by a gift from Advanced Communication Technology, Inc., and by the University of Maryland Institute for Advanced Computer Studies (UMIACS).

tremendously improved as more views are materialized. With the ratio $$/disk-volume constantly dropping, disk storage constraint is no longer the limiting factor in the view selection but the window to refresh the materialized set during updates. More materialization implies a larger maintenance window. This update window is the major data warehouse parameter, constraining over-materialization. Some view selection algorithms [Gup97, BPT97] 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. In [TS97] the authors 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 pre-computed results and maintenance time is also discussed in [DDJ+98]. However, none of these publications considers 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 philosophy starts with the premise that a result is a terrible thing to waste and that its generation cost should be amortized over multiple uses of the result. This philosophy goes back to our earlier work on caching of query results on the client's database $ADMS^+$ architecture [RK86, DR92], the work on prolonging their useful life through incremental updates [Rou91] and their re-use in the ADMS optimizer [CR94]. This philosophy is a major departure from the static paradigm of pre-selecting a set of views to be materialized and run all queries against this static set.

In this paper we present DynaMat, a system that dynamically materializes information at multiple levels of granularity in order to match the demand (workload) but also takes into account the maintenance restrictions for the warehouse, such as down time to update the views and space availability. DynaMat unifies the view selection and the view maintenance problems under a single framework using a novel "goodness" measure for the materialized views. DynaMat constantly monitors incoming queries and materializes the best set of views subject to the space constraints. During updates, DynaMat reconciles the current materialized view selection and refreshes the most beneficial subset of it within a given maintenance window. The critical performance issue is how fast we can incorporate the updates to the warehouse. Clearly if naive re-computation is assumed for refreshing materialized views, then the number of views will be minimum and this will lessen the value of DynaMat. On the other hand, efficient computation of these views using techniques like [AAD+96, HRU96, ZDN97, GMS93, GL95, JMS95, MQM97] and/or bulk incremental updates [RKR97] tremendously enhances the overall performance of the system. In DynaMat any of these techniques can be applied. In section 2.4.2 we propose a novel algorithm that based on the goodness measure, computes an update plan for the data stored in the system.

The main benefit of DynaMat, is that it represents a complete self-tunable solution that relieves the 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 pre-computed optimal static view selection. These experiments show that the dynamic view selection outperforms the optimal static view selection and thus, any sub-optimal static algorithm proposed in the literature [HRU96, GHRU97, Gup97, BPT97].

The rest of the paper is organized as follows: Section 2 gives an overview of the system's architecture. Subsections 2.2 and 2.3 discuss how stored results are being reused for answering a new query, whereas in section 2.4 we address the maintenance problem for the stored data. Section 3 contains the experiments and in section 4 we draw the conclusions.

2 System overview

DynaMat is designed to operate as a complete view management system, tightly coupled with the rest of the data warehouse architecture. This means that DynaMat can co-exist and co-operate with caching architectures that operate at the client site like [DFJ+96, KB96]. Figure 1 depicts the architecture of the system. *View Pool* $\mathcal{V}$ is the information repository that is used for storing materialized results. We distinguish two operational phases of the system. The first one is the "on-line" during which DynaMat answers queries posed to the warehouse using the *Fragment Locator* to determine whether or not already materialized results can be efficiently used to answer the query. This decision is based upon a cost model that compares the cost of answering a query through the repository with the cost of running the same query against the warehouse. A *Directory Index* is maintained in order to support sub-linear search in $\mathcal{V}$ for finding candidate materialized results. This structure will be described in detail in the following sections. If the search fails to reveal an efficient way to use data stored in $\mathcal{V}$ for answering the query then the system follows the conventional approach where the warehouse infrastructure (fact table+indices) is queried. Either-way, after the result is computed and given to the user, it is tested by the *Admission Control Entity* which decides whether or not it is beneficial to store it in the Pool.

During the on-line phase, the goal of the system is to answer as many queries as possible from the pool, because most of them will be answered a lot faster from $\mathcal{V}$ than from the conventional methods. At the same time DynaMat will quickly adapt to new query patterns and efficiently utilize the system resources.

The second phase of DynaMat is the update phase, during which updates received from the data sources get stored in the warehouse and materialized results in the Pool get refreshed. In this paper we assume, but we are not restricted to, that the update phase is "off-line" and queries are not

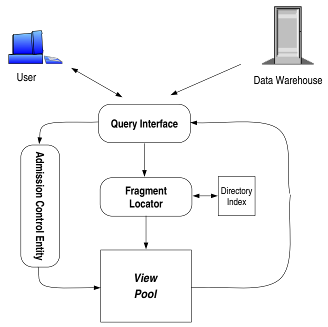

Figure 1: DynaMat's architecture

permitted during this phase. The maximum length of the update window W is specified by the administrator and would probably lead us to evict some of the data stored in the pool as not update-able within this time constraint.

2.1 View Pool organization

The View Pool utilizes a dedicated disk storage for managing materialized data. An important design parameter is the type of secondary storage organization that will be used. DynaMat can support any underling storage structure, as long as we can provide a cost model for querying and updating the views.

Traditionally summary data are stored as relational tables in most ROLAP implementations, e.g [BDD+98]. However, tables alone are not enough to guarantee reasonable query performance. Scanning a large summary table to locate an interesting subset of tuples can be wasteful and in some cases slower than running the query against the warehouse itself, if there are no additional indices to support random access to the data. Moreover, relational tables and traditional indexing schemes, are in most cases space wasteful and inadequate for efficiently supporting bulk incremental update operations. More eligible candidate structures include multidimensional arrays like chunked files [SS94, DRSN98] and also Cubetrees [RKR97]. Cubetrees are multidimensional data structures that provide both storage and indexing in a single organization. In [KR98] we have shown that Cubetrees, when used for storing summary data, provide extremely fast update rates, better overall query performance and better disk space utilization compared to relational tables and conventional indexes.

During the "on-line" phase of the warehouse, results from incoming queries are being added in the Pool. If the 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 results may not be update-able within the time constraint of W and thus, will be evicted from the pool. This is the update **time bound** case shown in Figure 2 with the size of the 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 pool size at the time of the updates.

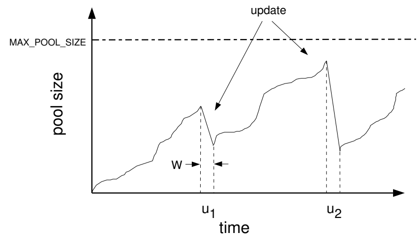

Figure 2: The **time bound** case

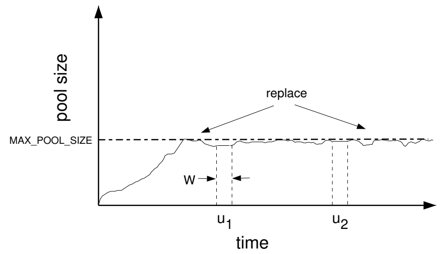

Figure 3: The **space bound** case

The **space bound** case is when the size of the pool is the constraining factor and not W. In this case, when the pool becomes full, we have to use some *replacement* policy. This can vary from simply not admitting more materialized results to the 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 3 shows the variations in the pool size in this case. Since we assumed a sufficiently large update window W, the stored results are always update-able and the actual content of the pool is now controlled by the replacement policy.

Depending on the workload, the disk space and the update window, the system will in some cases act as in time bound and in others as in space bound, or both. In such cases views are evicted from the pool, either because there is no more space or they can not be updated within the update window.

2.2 Using MRFs as the basic logical unit of the pool

A multidimensional data warehouse (MDW) is a data 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 [Kim96] which, for each dimension it stores 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 keys $d_1, \ldots, d_n$. The Data Cube operator [GBLP96] 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). The lattice [HRU96] representation of the Data Cube in Figure 4 shows an example for three dimensions, namely a, b and c. Each node in the lattice represents a view that aggregates data over the attributes present in that node. For example (ab) in an aggregate view over the a and b grouping attributes.[1]

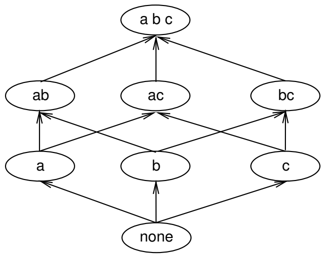

Figure 4: The Data Cube lattice for dimensions a, b and c

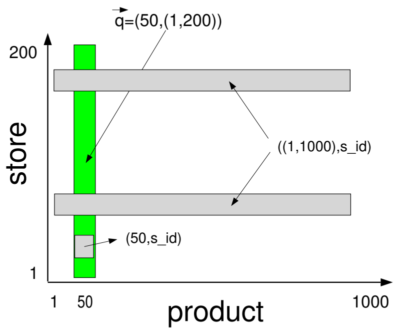

Figure 5: Querying stored MRFs

The lattice is frequently used by view selection algorithms [HRU96, GHRU97, SDN98] because it captures the computational dependencies among the elements of the Data Cube. Such dependencies are shown in Figure 4 as directed edges that connect two views, if the pointed view computes the other one. In Figure 4 we show only dependencies between adjacent views and not those in the transitive closure of this lattice. For example, view (a) can be computed from view (ab), while view (abc) can be used to derive any other view.

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

$$\vec{q} = \{R_1, R_2, \ldots, R_n\} \tag{1}$$

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

- 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 which denotes a dimension that is not present in the query.

[1] For simplicity in the notation, in this paper we do not consider the case where the 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 [HRU96] for the lattice.

For instance, suppose that $D = \{product, store\}$ is the set of dimensions in the MDW, with values $1 \leq product \leq 1000$ and $1 \leq store \leq 200$ respectively. The hyper-plane $\vec{q} = \{50, (1, 200)\}$ corresponds to the SQL query:

```
select product, store, aggregate_list
from F
where product=50
group by product, store
```

where *aggregate_list* is a list of aggregate functions (e.g sum,count). If the grouping was done on attributes different 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 called *slice queries* [GHRU97, BPT97, KR98]. We prefer the MR notation over the SQL description because it describes the workload in the Data Cube space independent of the actual schema of the MDW.

The same notation permits us to represent the materialized results of MR queries which we call Multidimensional Range Fragments (MRFs). DynaMat maps each SQL query to one, or more, MR queries. Given such a MR-query 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 is doesn't pay to check for combinations of materialized results for answering q. With extremely high probability, q is best computable out of a single fragment f or not computable at all. We will try to demonstrate this with the following example: Suppose that the previous query $\vec{q} = \{50, (1, 200)\}$ is given. If no single MRF in the pool computes q, then a stored MRF that partially computes q is of the form $\{50, s\_id\}$ or $\{(1, 1000), s\_id\}$, where s_id is some store value, see Figure 5. In order to answer q 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.

MRFs provide a slightly coarser grain of materialization if we compare them with a system that materializes views with arbitrary ranges for the attributes. However, if we allow fragments with arbitrary ranges to be stored in the 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, i.e 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. In the general case that k fragments compute some portion of the query there might be up to 2^k combinations that need to be checked for finding the most efficient way to answer the query. Having too many small fragments with possible overlapping sections which require additional filtering in the pool, results in poor performance not only during query execution but also during updates. In most cases, updating fewer, larger fragments of views (as in a MRF-pool) is preferable. We denote the number of fragments in the pool as $|\mathcal{V}|$. In section 2.4.2 we show that the overhead of computing an update plan for the stored data grows linearly with $|\mathcal{V}|^2$, making the MRF approach more scalable.

2.3 Answering queries using the Directory Index

As we described, when a MR-query q is posted to the data warehouse, we scan $\mathcal{V}$ for candidate fragments that answer q. Given a MRF f and a query q, f answers q iff for every non-empty range R_i of the query, the fragment stores exactly the same range and for every empty range $R_i = ()$ the fragment's corresponding range is either empty or spans the whole domain of dimension i[2]. We say in this case that hyper-plane $\vec{f}$ *covers* $\vec{q}$.

Instead of testing all stored fragments against the query, DynaMat uses a directory, the *Directory Index* (see Figure 1), to further prune the search space. This is actually a set of indices connected through the lattice shown in Figure 4. Each node has a dedicated index that is used to keep track of all fragments of the corresponding view that are stored in the pool. For each fragment f there is exactly one entry that contains the following info:

- Hyper-plane $\vec{f}$ of the fragment
- Statistics (e.g number of accesses, time of creation, last access)
- The *father* of f (explained below).

For our implementation we used R-trees based on the $\vec{f}$ hyper-planes to implement these indices. When a query q arrives, we scan using $\vec{q}$ all views in the lattice, that might contain materialized results f whose hyper-planes $\vec{f}$ *cover* $\vec{q}$. For example if $\vec{q} = \{(1, 1000), (), Smith\}$ is the query hyper-plane for dimensions $product$, $store$ and $customer$, then we first scan the R-tree index for view $(product, customer)$ using rectangle $\{(1, 1000), (Smith, Smith)\}$. Figure 6 depicts a snapshot of the corresponding R-tree for view $(product, customer)$ and the search rectangle. The shaded areas denote MRFs of that view that are materialized in the 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 $\{(1, 1000), (min_{store}, max_{store}), (Smith, Smith)\}$. If a fragment is found, we "collapse" the $store$ column and aggregate the measure(s) to compute the answer for q.

Based on the content of the pool $\mathcal{V}$, there are three possibilities. The first is that a stored fragment f matches exactly the definition of the query. In this case, f is retrieved

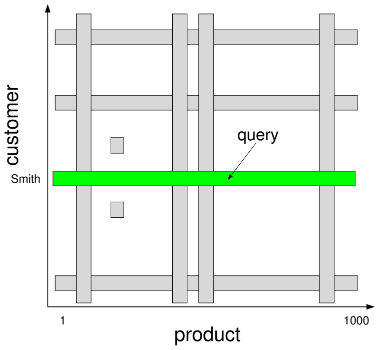

Figure 6: Directory for view $(product, customer)$

and returned to the user. If no exact match exists, assuming we are given a cost model for querying the fragments, we select the best candidate from the pool, to compute q. If view f is the materialized result of q, the fragment that was used to compute f is called the *father* of f and is denoted as $\acute{f}$. If however no fragment in $\mathcal{V}$ can answer q, the query is handled by the warehouse. In both cases the result is passed to the Admission Control Entity that checks if it can be stored in the pool.

As the number of MRFs stored in the pool is typically in the order of thousands, we can safely assume that in most cases the Directory Index will be memory resident. Our experiments validate this assumption and indicate that the look-up cost in this case is negligible. In cases where the index can not fit in memory, we can take advantage of the fact that the pool is reorganized with every update phase and use a packing algorithm [RL85] to keep the R-trees compact and optimized at all times.

2.4 Pool maintenance

For maintaining the MRF-pool, we need to derive a *goodness* measure for choosing which of the stored fragments we prefer. This measure is used in both the on-line and the update phases. Each time DynaMat reaches the space or time bounds we use the goodness for replacing MRFs. There can be many criteria to define such a goodness. 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 Index. Using this time-stamp as a goodness measure, results in an Least Recently Used (LRU) type of replacement in both cases.

- The frequency of access $freq(f)$ for the fragment:

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

  The frequency is computed using the statistics kept in the Directory Index and results in a Least Frequently used (LFU) replacement policy.

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

- The 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 hit by a query. An additional benefit of keeping larger results in the pool is that $|\mathcal{V}|$ gets smaller, resulting in faster look-ups using the Fragment Locator and less complexity while updating the 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:

$$goodness(f) = \frac{freq(f) * c(f)}{size(f)}$$

$c(f)$ is the cost of re-computing f for a future query. We used as an estimate of $c(f)$ the cost of re-computing the fragment from its father, which is computable in constant time. This metric is similar to the one used in [SSV96] for their cache replacement and admission policy. We refer to this case as the Smaller Penalty First (SPF).

In the remaining of this section we describe how the goodness measure is used to control the content of the pool.

2.4.1 Pool maintenance during queries

As long as there is enough space in the pool, results from incoming queries are always stored in $\mathcal{V}$. In cases where we hit the space constraint, we have to enforce a replacement policy. This decision is made by our `replace` algorithm using the $goodness$ measure of the fragments. The algorithm takes as input the current state of the 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 can not find candidate victims the search is aborted and the new result is denied storage in the pool. When a fragment f_{victim} is evicted the algorithm updates the $father$ pointer for all other fragments that point to f_{victim}. In section 2.4.2 we discuss the maintenance of the $father$ pointers.

2.4.2 Pool maintenance during updates

When the base relations (sources) are updated, the data stored in the MDW, and therefore the fragments in the 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 being computed by the views. Several methods have been proposed [AAD+96, HRU96, ZDN97] for fast (re)-computation of Data Cube aggregates. On the other hand, incremental maintenance algorithms have been presented [GMS93, GL95, JMS95, MQM97, RKR97] that handle grouping and aggregation queries.

For our framework, we assume that the sources provide the differentials 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 pool. In this scenario we also assume that all interesting aggregate functions that are computed are *self-maintainable* [MQM97] with respect to the updates that we have. This means that a new value for each function can be computed solely from the old value and from the changes to the base data.

Computing an initial update plan

Given a pool with $|\mathcal{V}|$ being in the order of thousands, our goal is to derive an *update plan* that allows us to refresh as many fragments as possible within a given update window W. Computing the deltas for each materialized result is unrealistic, especially if the deltas are not indexed somehow. In our initial experiments we found out that the time spent on querying the sources to get the correct deltas for each fragment is the dominant factor. For that reason our pool maintenance algorithm extracts, in a preprocessing step, all the necessary deltas and stores them in a separate view dV materialized as a Cubetree. This provides a efficient indexing structure for the deltas against multidimensional range queries. The overhead of loading a Cubetree with the deltas is practically negligible[3] 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 pool. These statistics are easy to maintain in the Directory Index. View dV includes all deltas within the hyper-plane:

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

For each fragment f in $\mathcal{V}$ we consider two alternative ways of doing the updates:

- We can query dV to get the updates 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 to get the deltas and the cost of updating f incrementally from the result.
- If the fragment was originally computed out of another result $\acute{f}$ we estimate the cost of recomputing f from its father $\acute{f}$, after $\acute{f}$ has been updated. The cost of computing f from its father is denoted as $UC_R(f)$ and includes the cost of running MR-query $\vec{f}$ against the fragment $\acute{f}$, plus the cost of materializing the result.

The system computes the costs for the two[4] alternatives and picks the minimum one, denoted as $UC(f)$ for each

[3] Cubetree's loading rate is about 12GB/hour in a Ultra 60 with a single SCSI drive.

[4] A third alternative, is to recompute each fragment from the sources. This case is not considered here, because the incremental approach is

fragment. Obviously, this plan is not always the best one. There is always the possibility that another result f_1 has been added in the pool after f was materialized. Since the selection of the father of f was done before f_1 was around, as explained in section 2.3, the above plan does not consider recomputing f from f_1. An eager maintenance policy of the father pointers would be to refine them whenever necessary, e.g 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 `makeFeasible` algorithm that is 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 the initial update plan, which directs the most cost-effective way to update each one of the fragments using one of the two alternatives, i.e incrementally from dV or by re-computation from another fragment.

Computing a feasible update plan for a given window

The total update cost of the 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 select a portion of $\mathcal{V}$ that will not be materialized in the new updated version of the 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, then the real reduction in the update cost of the pool is less than $UC(f)$, since the update cost of f_{child} increases. For the lazy approach for 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. If $UC^{new}(f_{child})$ is the new update cost for f_{child} then the potential *update delta*, i.e the reduction in $UC(\mathcal{V})$, if we evict fragment f is:

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

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 hit the time constraint, we evict more fragments from the 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 pool reduces the total update cost because all their children are efficiently updated from them. For the remaining fragments we use the goodness measure to select candidates for eviction until the remaining set is update-able within the given window W. If the `goodness` function is computable in constant time, the cost for k evictions is $O(k|\mathcal{V}|)$. In the extreme case where W is too small that only a few fragments can be updated this leads to an $O(|\mathcal{V}|^2)$ total cost for computing a feasible update plan. However, in many cases just a small fraction of the stored results will be discarded resulting in close to $O(|\mathcal{V}|)$ complexity.

expected to be faster. However, for sources that do not provide their differentials during updates, we can consider using this option.

3 Experiments

The comparison and analysis of the different aspects of the system made in this section is based on a prototype that we have developed for DynaMat. This prototype implements the algorithms and different policies that we present in this paper as well as the Fragment Locator and the Directory Index, but not the pool architecture. For the latter we used the estimator of the Cubetree Datablade [ACT97] developed for the Informix Universal Server for computing the cost of querying and updating the fragments.

We have created a random MR-query generator that is tuned to provide different statistical properties for the generated query sets. 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 [SSV96] 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:

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

where c_i is the cost of execution 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 references to that query. This metric is also used in [DRSN98] for their experiments. Because query costs vary widely, CSR is more appropriate metric than the common hit ratio: $\frac{\sum_i h_i}{\sum_i r_i}$. However, a drawback in the above definition for our case, is that it doesn't capture the different ways that a query q_i might "hit" the Pool. In the best scenario, q_i exactly matches a fragment in $\mathcal{V}$. In this case the savings is defined as c_i, where c_i is the cost of answering the query at the MDW. However, in cases where another result is used for answering q_i the actual savings depend on how "close" this materialized result is to the answer that we want to produce. If c_f is cost of querying the best such fragment f for answering q_i, the savings in this case is $c_i - c_f$.[5] To capture all cases we define the savings provided by the pool $\mathcal{V}$ for a query instance q_i as:

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

using the above formula we define the Detailed Cost Saving

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

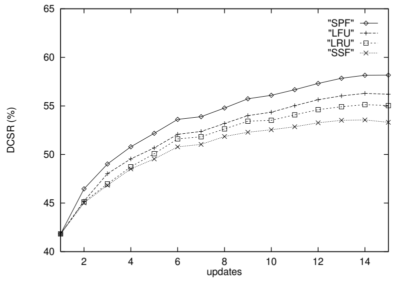

Figure 7: The **time bound** case, first 15x1500 queries

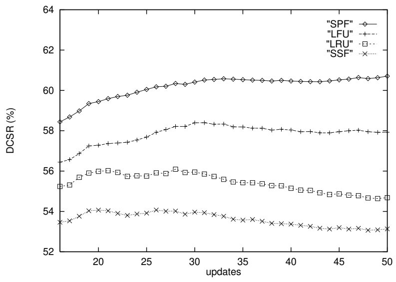

Figure 8: The **time bound** case, remaining 35x1500 queries

Ratio as:

$$DCSR = \frac{\sum_i s_i}{\sum_i c_i}$$

DCSR provides a more accurate measure than CSR for OLAP queries. CSR uses a "binary" definition of a hit: a query hits the 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 will return a savings of 10,000 for the "hit", while DCSR will credit the system will only 500 units based on the previous formula. DCSR captures the different levels of 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: Subsection 3.1 makes a direct comparison of the different ways to define the goodness as described in 2.4. Subsection 3.2 compares the performance of DynaMat against a system that uses the optimal static view selection policy. All experiments were ran using an Ultra SPARC 60 with 128MB of main memory.

3.1 Comparison of different goodness policies

In this set of experiments we compare the DCSR under the four different goodness policies LRU, LFU, SFF and SPF. We used a synthetically generated dataset that models supermarket 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 where applied was estimated to be about 708GB. We generated 50 query sets with 1,500 MR-queries each, that were ran 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 data 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.

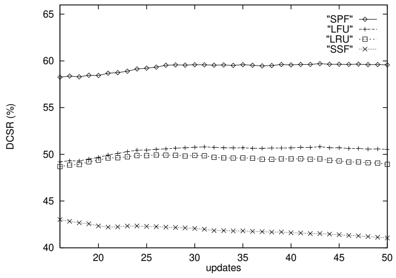

Figure 9: The **space bound** case

For the first experiment we tested the time-bound case. The size of the pool was chosen large enough to guarantee no replacement during queries and the time allowed for updating the fragments was set to 2% of W_{Data_Cube}, where W_{Data_Cube} is the estimated time to update the full Data Cube. For a more clear view we plot in Figure 7 the DCSR overtime for the first 15 sets of queries, starting with an empty 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 pool. This shows that simply by storing and reusing computed results from previous queries, we cut down the cost of accessing the MDW to 58.6%. Figure 8 shows how DCSR changes for the remaining queries. All four policies quickly increase their savings, by refining the content of the pool while doing updates, up to a point where all curves flatten out. At all times, 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 due to the on-the-fly materialization was about the same in all cases ($\simeq$19.8MB/query). For the winner SPF policy the average time spend on searching the Directory Index was negligible (about 0.4msecs/query). Computing a feasible update plan took on the average 37msecs, and 51msecs in the worst case. The number of MRFs stored in the pool by the end of the last update phase was 206.

Figure 9 depicts DCSR overtime in the space-bound case for the last 35 sets of queries, calculated at the beginning

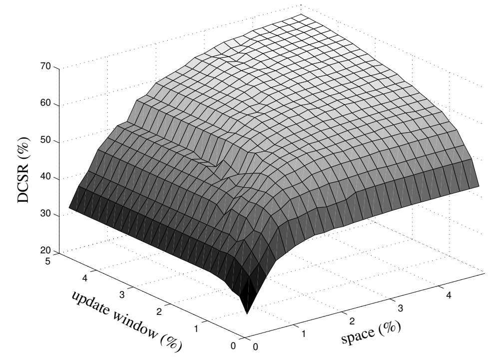

Figure 10: The **space & time bound** case

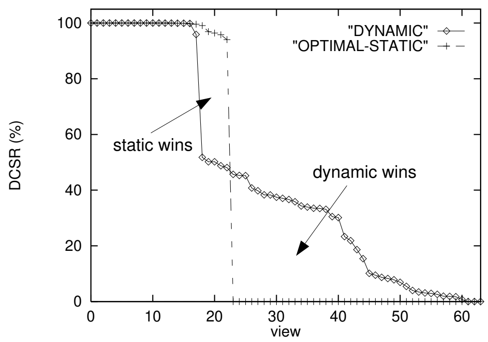

Figure 11: DCSR per view for uniform queries on the views

of each update phase. In this experiment there was no time restriction for doing the updates, and the space that was allocated for the pool was set to 14GB, i.e 2% of the full Data Cube size. In this case, the content of the pool is managed by the `replace` algorithm, as the limited size of the pool 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 on the Directory Index, was less that 3msecs per query. Computing the initial update plan for the updates, as explained in section 2.4.2, took 10msecs on the average. Since there was no time restriction and thus, the plan was always feasible, there was no additional overhead for refining this plan. The final number of fragments in the 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 pool from 0.2% up to 5% of the full Data Cube size, both in 0.2% intervals. Figure 10 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.

3.2 Comparison with the optimal static view selection

In the experiments in the previous section we saw that the SPF policy provides the best goodness definition for a 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 [HRU96, GHRU97, Gup97, BPT97] that considers only fully materialized views. Instead of comparing each one of these algorithms with our approach, we implemented SOLVE, a module that given a set of queries, the space and time restrictions, it 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 non trivial cases this exhaustive search is not feasible even for small values of n.

We used SOLVE to compute the optimal static view selection for a six-dimensional subset of our supermarket dataset, with 20 million tuples in the fact table. There were 40 update phases, with 100 thousand new tuples being added in the fact table each time. The time window 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 in the 6-dimensional Data Cube lattice. This lack of locality of the queries represents the worst-case scenario for the dynamic case that needs to adapt on-the-fly to the incoming query pattern. For the static view selection this was not an issue, because SOLVE was given all queries in advance. The optimal set returned, after 3 days of computations in an Ultra SPARC 60, includes 23 out of the 64 full-views in the 6-dimensional Data Cube. The combined size of these views when stored as Cubetrees in the disk is 281MB (1.6% of the full Data Cube). For the most strict and unfavorable comparison for the dynamic case, we set the size of the pool to the same number. Since the dynamic system started with an empty 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 measure 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

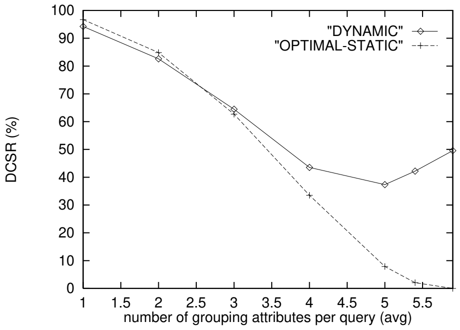

Figure 12: Dynamic vs Optimal-Static selection varying the average number of grouping attributes per query

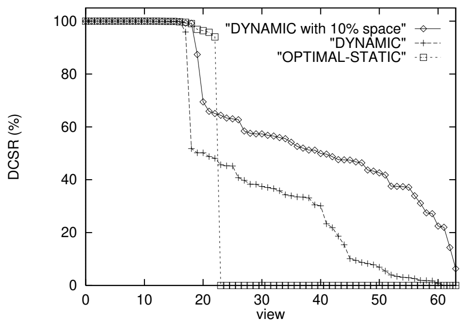

Figure 13: DCSR per view for space = 10%

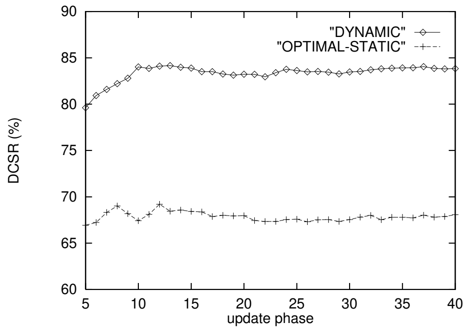

Figure 14: Dynamic vs Optimal-Static selection for drill-down/roll-up queries

system was 108.11MB and the average write-back I/O cost 2.18MB. For the optimal static selection the average I/O per query is 112.94MB and no write-back, without counting the overhead of materializing the statically selected views for the first time.

For a more clear 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 11. 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 rest 41 views its savings drops to zero. DynaMat on the other hand provides substantial savings for almost all the views. On the right hand side of the graph are the larger views of the Data Cube. Since most results from queries on these views are too big to fit in the pool, even DynaMat's performance decreases because they can not be materialized in the shared disk space.

Figure 12 depicts the performance of both systems for a non-uniform 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,[6] it favors accesses on views from the upper levels of the Data Cube lattice, which views are bigger in size and need larger update window. These views, because of the space and time constraints are not in the static optimal selection. On the other hand, the dynamic approach materializes 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, almost all queries in the workload access the single top-level six-dimensional view of the lattice. DynaMat adapts nicely to such workload and allocates most of the pool space to MRFs of that view. That explains the performance of DynaMat going up at the right hand side of the graph.

The 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 for todays standards. We ran two more experiments with pool size 5% (878MB) and 10% (1.7GB) 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. Figure 13 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 the larger views of the Data Cube.

In the previous experiment the queries that we ran were selected uniformly from all 64 views in the Data Cube lattice. This is the worst case scenario for DynaMat which gains a lot more from locality of follow-up queries. Often in OLAP, users do drill-downs or roll-ups, where starting from a computed result, they refine their queries and ask for a more or less detailed view of the data respectively. DynaMat can enormously benefit from the roll-up queries because these queries are always computable from results that were previously added in the pool. To simulate such a workload we tuned our query-generator to provide 40 sets of 500 queries each with the following properties: 40% of the times a user asks a query for a randomly selected view from the Cube, 30% of the times the user performs a roll-up

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

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 bound and we re-computed the optimal static selection for the new queries. Figure 14 depicts DCSR for this workload. Compared to the previous example, DynaMat further increases its savings (83.84%) by taking advantage of the locality of the roll-up queries.

4 Conclusions

In this paper we presented DynaMat, a view management system that dynamically materializes results from incoming queries and exploits them for future reuse. DynaMat unifies view selection and view maintenance under a single framework that takes into account both the time and space constraints of the system. 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 static paradigm that considers only full views for materialization, MRFs provide a finer and more appropriate granularity of materialization. The operational and maintenance cost of the MRFs, which includes any directory look-up operations during the online mode and the derivation of a feasible update plan during updates, remains practically negligible, in the order of milliseconds.

We compared DynaMat against a system that is given all queries in advance and the pre-computed optimal static view selection. These experiments indicate that DynaMat outperforms the optimal static selection and thus any sub-optimal view selection algorithm that has appeared in the literature. Another important result that validates the importance of DynaMat, is that just 1-2% of the Data Cube space and 1-2% of the update window for the full Data Cube are sufficient for substantial performance improvements.

However, the most important feature of DynaMat is that it represents a complete self-tunable system that dynamically adjusts to new patterns in the workload. DynaMat relieves the warehouse administrator from having to monitor and calibrate the system constantly regardless of the skewness of the data and/or of the queries. 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.

5 Acknowledgments

We would like to thank Kostas Stathatos and Alexandros Labrinidis for their helpful comments and suggestions.

References

[AAD+96] S. Agrawal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Sarawagi. On the Computation of Multidimensional Aggregates. In *Proc. of VLDB*, pages 506–521, Bombay, India, August 1996.

[ACT97] ACT Inc. The Cubetree Datablade. http://www.act-us.com, August 1997.

[Aut] AutoAdmin Project, Database Group, Microsoft Research.

[BDD+98] R. G. Bello, K. Dias, A. Downing, J. Feenan, J. Finnerty, W. D. Norcott, H. Sun, A. Witkowski, and M. Ziauddin. Materialized Views In Oracle. In *Proc. of VLDB*, pages 659–664, New York City, New York, August 1998.

[BPT97] E. Baralis, S. Paraboschi, and E. Teniente. Materialized View Selection in a Multidimensional Database. In *Proc. of VLDB*, pages 156–165, Athens, Greece, August 1997.

[CR94] C.M. Chen and N. Roussopoulos. The implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching. In *Proc. of the 4th Intl. Conf. on Extending Database Technology*, pages 323–336, 1994.

[DDJ+98] L. Do, P. Drew, W. Jin, V. Junami, and D. V. Rossum. Issues in Developing Very Large Data Warehouses. In *Proceedings of the 24th VLDB Conference*, pages 633–636, New York City, New York, August 1998.

[DFJ+96] S. Dar, M.J. Franklin, B. Jonsson, D. Srivastava, and M. Tan. Semantic Data Caching and Replacement. In *Proc. of the 22th International Conference on VLDB*, pages 330–341, Bombay, India, September 1996.

[DR92] A. Delis and N. Roussopoulos. Performance and Scalability of Client-Server Database Architectures. In *Proc. of the 18th VLDB*, pages 610–623, Vancouver, Canada, 1992.

[DRSN98] P. M. Deshpande, K. Ramasamy, A. Shukla, and J.F. Naughton. Caching Multidimensional Queries Using Chunks. In *Proceedings of the ACM SIGMOD*, pages 259–270, Seattle, Washington, June 1998.

[GBLP96] J. Gray, A. Bosworth, A. Layman, and H. Piramish. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. In *Proc. of the 12th ICDE*, pages 152–159, New Orleans, February 1996. IEEE.

[GHRU97] H. Gupta, V. Harinarayan, A. Rajaraman, and J. Ullman. Index Selection for OLAP. In *Proceedings of ICDE*, pages 208–219, Burmingham, UK, April 1997.

[GL95] T. Griffin and L. Libkin. Incremental Maintenance of Views with Duplicates. In *Proceedings of the ACM SIGMOD*, pages 328–339, San Jose, CA, May 1995.

[GMS93] A. Gupta, I.S. Mumick, and V.S. Subrahmanian. Maintaining Views Incrementally. In *Proceedings of the ACM SIGMOD Conference*, pages 157–166, Washington, D.C., May 1993.

[Gup97] H. Gupta. Selections of Views to Materialize in a Data Warehouse. In *Proceedings of ICDT*, pages 98–112, Delphi, January 1997.

[HRU96] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing Data Cubes Efficiently. In *Proc. of ACM SIGMOD*, pages 205–216, Montreal, Canada, June 1996.

[JMS95] H. Jagadish, I. Mumick, and A. Silberschatz. View Maintenance Issues in the Chronicle Data Model. In *Proceedings of PODS*, pages 113–124, San Jose, CA, 1995.

[KB96] A.M. Keller and J. Basu. A Predicate-based Caching Scheme for Client-Server Database Architectures. *VLDB Journal*, 5(1), 1996.

[Kim96] R. Kimball. *The Data Warehouse Toolkit*. John Wiley & Sons, 1996.

[KR98] Y. Kotidis and N. Roussopoulos. An Alternative Storage Organization for ROLAP Aggregate Views Based on Cubetrees. In *Proceedings of the ACM SIGMOD Conference*, pages 249–258, Seattle, Washington, June 1998.

[MQM97] I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of Data Cubes and Summary Tables in a Warehouse. In *Proceedings of the ACM SIGMOD Conference*, pages 100–111, Tucson, Arizona, May 1997.

[RK86] N. Roussopoulos and H. Kang. Preliminary Design of ADMS$\pm$: A Workstation-Mainframe Integrated Architecture for Database Management Systems. In *Proc. of VLDB*, pages 355–364, Kyoto, Japan, August 1986.

[RKR97] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cubetree: Organization of and Bulk Incremental Updates on the Data Cube. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 89–99, Tucson, Arizona, May 1997.

[RL85] N. Roussopoulos and D. Leifker. Direct Spatial Search on Pictorial Databases Using Packed R-trees. In *Procs. of 1985 ACM SIGMOD*, pages 17–31, Austin, 1985.

[Rou91] N. Roussopoulos. The Incremental Access Method of View Cache: Concept, Algorithms, and Cost Analysis. *ACM–Transactions on Database Systems*, 16(3):535–563, September 1991.

[SDN98] A. Shukla, P.M. Deshpande, and J.F. Naughton. Materialized View Selection for Multidimensional Datasets. In *Proceedings of the 24th VLDB Conference*, pages 488–499, New York City, New York, August 1998.

[SS94] S. Sarawagi and M. Stonebraker. Efficient Organization of Large Multidimensional Arr ays. In *Proceedings of ICDE*, pages 328–336, Houston, Texas, 1994.

[SSV96] P. Scheuermann, J. Shim, and R. Vingralek. WATCHMAN: A Data Warehouse Intelligent Cache Manager. In *Proceedings of the 22th VLDB Conference*, pages 51–62, Bombay, India, September 1996.

[TS97] D. Theodoratos and T. Sellis. Data Warehouse Configuration. In *Proc. of the 23th International Conference on VLDB*, pages 126–135, Athens, Greece, August 1997.

[ZDN97] Y. Zhao, P.M. Deshpande, and J.F. Naughton. An Array-Based Algorithm for Simultaneous Multidimensional Aggregates. In *Proceedings of the ACM SIGMOD Conference*, pages 159–170, Tucson, Arizona, May 1997.