Aggregate View Management in Data Warehouses

Yannis Kotidis

AT&T Labs

kotidis@research.att.com

Abstract

Materialized views and their potential have been recently rediscovered for the content of OLAP and data warehousing. A flurry of papers has been generated on how views can be used to accelerate ad-hoc computations over massive datasets. In this chapter we introduce and comment on some main-stream approaches for defining, computing, using and maintaining materialized views with aggregations in a large data warehouse.

1 Introduction

Data warehousing is a collection of decision support technologies that aim at enabling an enterprise to make better and faster decisions [CD97]. Finding the right information at the right time is a necessity for survival in today's competitive marketplace and this area has enjoyed an explosive growth in the past few years. Data warehousing products have been successfully deployed in data rich industries ranging from retail to financial services and telecommunications.

From a historical perspective, the data warehousing area is a byproduct of tremendous advances in information technology. During the last few decades there has been a increasing movement to computerize every possible business process. However, most of the applications developed over the years were mostly stand-alone efforts. As a result there was virtually no integration among different applications, even within the domain of a single enterprise. The situation was even more dramatic from the data point of view. The same data item could have inconsistent definitions, meanings and representations along different platforms.

Although relational databases were at first believed to provide direct access to a single unambiguous copy of the data, it was very soon realized that they couldn't support both operational and decision-support users. Operational workload consists of many concurrent short transactions that access and modify a few records at a time. In addition depending on the application it might have strict real-time requirements. The obvious examples are ATM and airline reservation systems. Such applications automate structured and repetitive tasks that require detailed up to date information. Decision support applications on the other hand use tactical information that answers "who" and "what" questions about past events [olaa]. This requires a stable view of the underlying data that can only be obtained using costly relational locking mechanisms, unless a second copy of the information is used. Furthermore, decision support often requires historical data that is usually unavailable in operational databases that only deal with recent data. Finally, operational databases are designed to reflect the operational semantics of their applications and to maximize transaction throughput. Decision support analysis on the other hand is typically ad-hoc, based on multidimensional business models and operations that require different data models and new access methods tailored for query intensive applications.

In the broadest sense, a data warehouse is a single, integrated informational store that provides stable, point-in-time data for decision support applications [BE97]. 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. Since data might be coming from different legacy systems within the organization, significant cleansing, transformation and reconciliation may be necessary before it is loaded in the repository. Data cleaning involves operations varying from data integrity tests and simple transformation tasks (e.g. inconsistent field lengths or descriptions) to more advanced tools that look for suspicious statistical patterns in the data.

2 Materialized Aggregate Views for Better Performance

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 software allows the data to be rendered across any dimension and at any level of aggregation. For example in a sales data warehouse customer, product, location and time of sale may be the dimensions of interest. In order to extract this information on the fly, the navigation tool interacts with the data warehouse, which provides the core data that is being analyzed. Examples of possible computations include among others:

- *multidimensional ratios* like "show me the contribution to weekly profit made by all items sold in NJ between May 1 and May 7".
- *quantiles* e.g. "show my top-5 selling products across the state of NJ".
- *statistical profiles* e.g. "show sales by store for all stores in the bottom 5% of sales".
- *comparisons* e.g. "sales in this fiscal period versus last period".

The main cost in terms of the time consumed of executing this type of queries is not only doing the actual arithmetic, but also of retrieving the data items (or "cells' in the multidimensional dialect) that affect the calculated functions. For a large data warehouse, executing queries with aggregations against the detailed records takes hours, simply because of the volume of records that are being accessed. As a result most implementations facilitate some form of pre-aggregation in order to support complex data-intensive queries in a interactive fashion. 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* [CD97]. A materialized view that contains highly consolidated information is typically much smaller than the base relations used to store all detailed records. As a result, querying the view instead of the base records offers several orders of magnitude faster query speeds.

Since materialized views promise high performance improvements over accessing detailed records 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 analyzes, and which

involve too much data to be calculated on the fly. For example in a sales data warehouse, a large majority of the queries may involve transactions over the last few weeks or months. Having these sales summarized in a view significantly speeds up many queries.

If query response time is the only concern, an eager policy of materializing all possible aggregations 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 dimensions that the dataset is analyzed on. Thus, materializing all possible views demands an enormous amount of pre-processing cost and disk volume to compute and store the aggregations. Furthermore, much like a cache, the views get dirty whenever the data warehouse tables are modified. For most organizations, the maintenance process is happening in a daily fashion, usually overnight. In order to correctly reflect the underlying data, the views have to be updated in a process that is known as view maintenance. Having many views materialized in the system lengthens the duration of the daily update process and reduces the warehouse "on-line" period, i.e. the portion of time the system is available for analysis.

Materialized views with aggregates introduce new challenges in the design and use of the data warehouse. These challenges include:

- identifying the views that we want to materialize.
- efficiently computing these views from the raw data
- exploiting the views to answer queries
- efficiently maintaining the views when new data is shipped to the data warehouse.

Before getting to see some of the approaches for dealing with these problems we will make a brief introduction on the star schema organization that is frequently used to model a relational data warehouse and the data cube operator that provides broadly accepted framework for describing materialized aggregate views.

3 A Relational Data Warehouse architecture

Information within the data warehouse is modeled in a multi-dimensional fashion. Formally, a dimension is a structural attribute that lists members of similar type [olaa]. The time dimension is such an example, common among most data warehouse applications. Dimension members allow us to pin-point the raw data for analysis. We can think of them as indices to a virtual multi-dimensional array of numeric attributes that are the target of the analysis. These attributes are often called *measures* and are used as input to the aggregation functions. Such measures include sales, revenue, inventory e.t.c.

Often dimension's members are organized on parent-child relationships, like for example all days, weeks, months, quarters and years along the time dimension. The resulting relations form a *hierarchy*. Dimension members along the hierarchy consolidate data of all the members which are their children. Hierarchies provide the means to encode our domain knowledge and address the data at different levels of aggregation. This is achieved through the *drill-down* and *roll-up* operators. By drilling-down on the aggregate data we get a more detailed view of the information. For example starting from the total sales per year, we drill-down and ask for a monthly breakdown of sales for the last year and then examine the actual transactions for a month with irregular volume of sales. Roll-up is the opposite operation where information is examined at progressively coarser granularity.

In a relational implementation, the data warehouse is usually organized in a specialized lay-out that is known as the *star schema* [Kim96]. In its simplest form there is a central table F called the fact table that contains the facts, i.e the transactional information that is of interest. Each record in F consists of two parts. The first part is a list of attributes that are foreign keys to the satellite dimension tables discussed bellow and identify a unique position for the record in the multidimensional space in which we analyze the data. The second part contains a list of measures that provide the numeric information on which analysis is being performed. For each dimension in the dataset there is a single table that contains information for the specific dimension. This table holds a primary key that is used to link dimension-specific attributes with the fact table.

Figure 1 shows a star-schema organization for a data warehouse that analyzes sales data. The

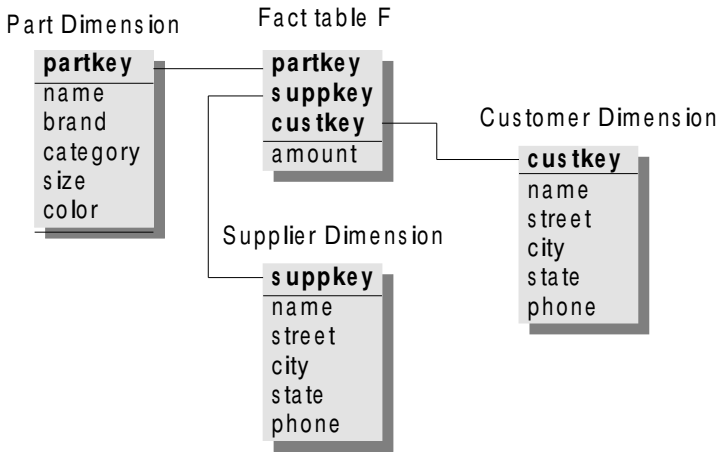

Figure 1: A Simple Star Schema Organization

data contains sales information for parts that are being purchased from different suppliers and sold to customers. In this simplified example each transaction records a sale to a customer. A record contains the part identifier *partkey*, the identifier of the supplier *suppkey* from which the part was ordered and also the identifier of the customer *custkey* who bought the part. There is also a numeric attribute *amount* that stores how much the customer paid for the product. The star schema provides a clean lay-out in which dimension specific attributes are hidden from the facts and are encapsulated in the dimension tables. For example the customer table contains all customer attributes, like name and contact info. Furthermore, it allows for easy schema modifications, fast loading of data and incremental updates.

The simplest form of analysis against the data is to aggregate the measure(s) on one or more selected dimensions. For example query "find the total sales per customer" is written in SQL:

```
q1: select custkey, sum(amount) as total_sales
    from F
    group by custkey
```

while query "find the total sales per customer and product" is translated to:

```
q2: select partkey,custkey, sum(amount) as total_sales
```

```
from F
group by partkey,custkey
```

More expressiveness can be achieved by combining facts with information stored in the dimension tables. For example query "find our customers in NJ along with their aggregated sales" is stated as:

q_3:
```
select customer.custkey, customer.name, sum(amount) as total_sales
from F, customer
where F.custkey = customer.custkey
and customer.state = 'NJ'
group by customer.custkey, customer.name
```

This type of queries consist of joins of the fact table with a number of dimension tables, with possible selections on some dimension attributes and the aggregation of one or more of the measures grouped by a subset of attributes from the dimension tables. These queries are known as Select-Project-Join (SPJ) queries.

The group-by operator in SQL is frequently used to compute such aggregations among ad-hoc groups. Gray et al in [GBLP96] introduced the data cube CUBE operator as a generalization of the traditional group by syntax. Their motivation was that certain types of data analysis are difficult to be expressed in SQL. Examples that are common in report writing are the histogram, cross-tabulation, roll-up, drill-down and sub-total constructs. In technical terms, the data cube is a redundant multidimensional projection of a relation. It computes all possible group by SQL operators among the dimensions. The data cube of our three dimensional example will be the union of all 7 aggregate queries that we can construct by grouping on different combinations of attributes *partkey*, *suppkey*, *custkey*, plus the value that is derived when we aggregate over all data:

```
select sum(amount) as total_sales
from F
```

In general for n dimensions, the data cube computes 2^n group bys, corresponding to all possible combinations of the selected dimensions. Each one of these group bys can be realized as a materialized view. In the following section we address the problem of efficiently computing these views.

4 Efficient Computation of Views with Aggregates

For the following discussion we assume that only aggregates based on the dimensions keys are of interest, like in queries q_1, q_2. However the results can be extended when groupings are performed on dimension members that are part of a hierarchy.

The straightforward way to compute all these views is to rewrite the CUBE query as a collection of 2^n SQL queries and execute them separately, like for instance queries q_1 and q_2 of the previous example. However, even for small number of dimensions, this naive computation of the views is unrealistic, because of the mere size of the raw data that needs to be accessed over and over again to compute the aggregates. Apart from the I/O overhead an independent computation of the aggregates does not realize possible overlap among the calculations that are needed for the views. For example after query q_2 is executed and the result is materialized as view u, we can rewrite the view that corresponds to query q_1 as:

```
create view v as
select custkey, sum(u.total_sales) as total_sales
from u
group by custkey
```

What allows this rewriting is a special property of the *sum()* function that allows aggregates to be further aggregated. Such functions allow the input set to be partitioned into disjoint sets that can be aggregated separately and later combined. Aggregate functions with this property are called *distributive* and are characterized from the fact that no state-information is needed for the function to summarize a sub-aggregation. Other distributive functions include *count()*, *min()* and *max()*.

Often functions of interest can be expressed by combining two or more distributive functions. The avg() function for example can be computed using sum() and count(). Functions that require a fixed size state information to describe a sub-aggregate are called algebraic and can be also optimized when computing the aggregates for multiple views in parallel. Other algebraic functions include center of mass, standard deviation, maxN() (N largest values) and minN(). However for functions like *median()* there is no bound on the size of state information that is needed to describe a sub-aggregate. Such functions are often referred to as *holistic* and void all optimizations that we describe bellow.

For non-holistic aggregate functions we can share I/O and computations among multiple views of the data cube. These gains are realized by exploiting well-defined dependencies among the views that are best depicted in the lattice [HRU96] of Figure 2, for our three dimensional example. Each node in the lattice represents a view that aggregates data over the attributes present in that node; for example node (*partkey*) is view v. The node labeled as *none* represents a view that computes a single super-aggregate over all input cells. For our sales dataset this will be the overall volume of sales for all records in the fact table. The lattice representation is based on a derived-from relationship, which defines a partial ordering $\preceq$ among the views. For two views v and u, $v \preceq u$ if and only if v can be computed from the tuples of u for any instance of the dataset. For example $(partkey) \preceq (partkey, custkey)$ but not vise-versa. In the lattice notation there is a downward path from u to v if and only if $v \preceq u$.

Typically group-bys are being computed either by sort-based or by hash-based methods as discussed in [Gra93]. [AAD+96] describes how these methods can be extended for multiple group-by computations, like the case of the data cube. Using the authors' notation there are four possible optimizations that can be incorporated in the computation of the aggregate views:

1. smallest-parent: this optimization implies that a view should be computed from the smallest previously computed view. For example view (*partkey*) can be computed from any of $(partkey, custkey)$, $(partkey, suppkey)$ and $(partkey, custkey, suppkey)$. If we have an estimate for the sizes of these views, we can pick the smallest one for computing (*partkey*).

2. cache-results: this optimization favors in-memory results of a previous computation to derive

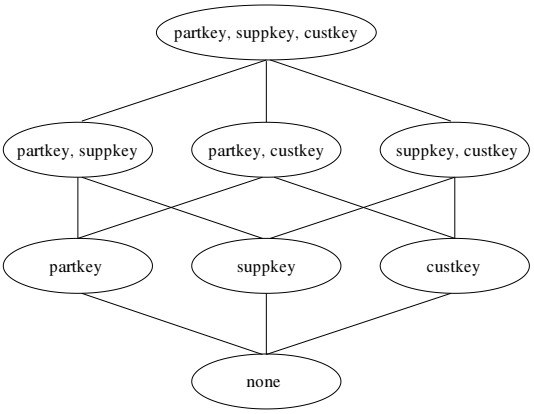

Figure 2: The Data Cube Lattice for three Dimensions

other views and thus reduce disk I/O.

3. amortize-scans: when data has to be read from disk, the cost of I/O should be amortized by computing as many views as possible. For instance if view $(partkey, suppkey)$ was previously stored on disk, we may compute all of $(partkey)$, $(suppkey)$, $(none)$ in one scan over $(partkey, suppkey)$.

4. share sorts and partitions: this optimizations are specific to the algorithm used for computing the views. When sort-based algorithms are employed we can share the cost of sorting among multiple views, or in some cases exploit partially matching sort orders. For example if the records of view $(partkey, suppkey)$ are sorted on low-$partkey$, low-$suppkey$, i.e. first on $partkey$ and then on $suppkey$ for tuples with the same part identifier, we can then compute view $(partkey)$ with no additional sorts in the following manner: we first project out the $suppkey$ column. This will give as a sorted list (with duplicates) of $partkey$ values along with partially aggregated measures. We then simply merge these sub-aggregates for every distinct $partkey$ value. Partially matching sort orders are also useful. If for example view $(partkey, suppkey, custkey)$ is sorted on attribute order, we can compute $(partkey, custkey)$ by partitioning on the first attribute and independently sorting on $custkey$. Similar optimizations can be performed when the aggregates are computed using hash-based algorithms by sharing the partitioning cost among multiple views.

For datasets that are much larger than the available main memory these optimizations are often contradictory. For instance we might decide to sort a memory resident view instead of using one that has a desirable sort order but is stored on disk. Comparing the sort based techniques against hash-based computations of the aggregates the author of [AAD+96] conclude that sorting works better for sparse datasets, while hashing methods gain from decreasing levels of sparseness. In practice performance strongly depends on the amount of memory that is available to hold intermediate results and auxiliary data structures (e.g hash tables) and the distribution of values of the dataset. Real-world data however, for many application domains, are often very large and sparse. Ross and Srivastava in [RS97] argue that none of the previous algorithms is very efficient for sparse datasets, especially when the base relation is much larger than main memory. For these cases, they propose an algorithm that follows a divide-and-conquer strategy to split the problem into several simpler computations of sub-cubes that are then computed by multiple in-memory sorts. Other techniques for computing multiple aggregate views in parallel can be found at [BR99, LRS99, MK99].

These algorithms work well for Relational OLAP (ROLAP) systems that store their data in conventional relational tables. For Multidimensional OLAP (MOLAP) systems, that store their data in sparse multidimensional arrays rather than in tables, Zhao et al [ZDN97] proposed a different array-based algorithm. For the synthetic and rather dense datasets that they used for their experiments the authors showed that the MOLAP approach can be significantly faster that the ROLAP table-based algorithms. The benefit comes from the fact that the array representation allows direct access to individual cells. However, multidimensional arrays have limitations when dealing with sparse high dimensional datasets.

4.1 Estimating the Size of the Views

An unexpected behavior of multidimensional aggregations is that the amount of possible results is many times larger than the detailed data. Figure 3 shows an input record of the fact table of Figure 1 in the three dimensional space defined by the keys: *partkey*, *suppkey*, *custkey*. Each possible aggregation across these dimensions can be seen as a projection of this point in the sub-

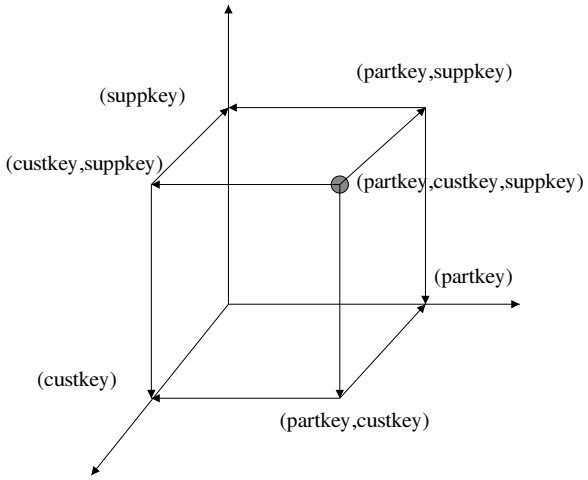

Figure 3: Multi-dimensional Projections of an Input Cell

space of the participating dimensions. For example all group by $(partkey, custkey)$ aggregates will be projected in the corresponding 2-dimensional plane in this space. Some of these projections will be overlapped with projections of other input points, however each input record may introduce as many as seven new aggregates in the worst case. Subsequently, data in each of these sub-spaces will be much denser and voluminous than the input data, which is expected to be fairly sparse for many real datasets [Col96].

Before getting to compute all or some of the views of the data cube, we need an estimate of the disk space required accommodating the derived aggregates. This information comes handy for tuning the algorithms used to compute the aggregates to decide on hashing values, sizes of disk runs for sorting e.t.c. The size of the views depends on the number of dimensions involved and the distribution of values along their domains. For simplicity we make the assumption that all dimensions have the same domain size D. For a view with i dimensions the maximum number of ways that values from these dimensions can be combined and return an aggregate is D^i, which is the volume of an i-dimensional array of length D per side. Since each input record generates at most one new entry in this array, the size of the view is also bounded by the size of the fact table F, denoted as $|F|$. Therefore, a crude upper-bound for the size of the view is $min(D^i, |F|)$. Figure 4 plots the size of the views for an eight dimensional uniform dataset where D is 50 and

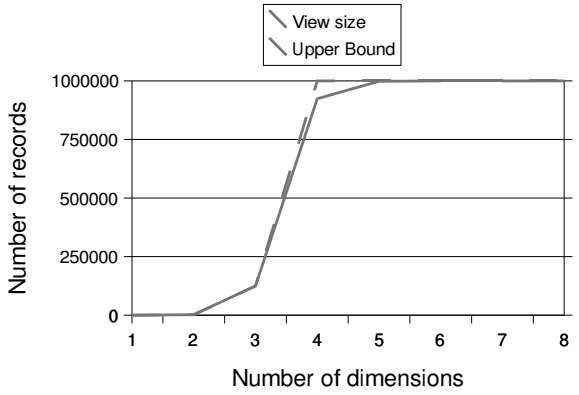

Figure 4: View Sizes for Uniform Distribution

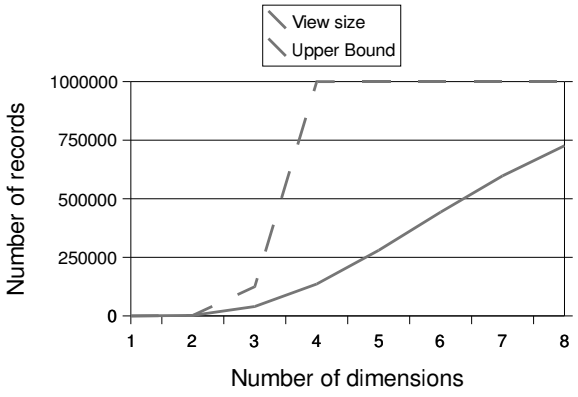

Figure 5: View Sizes for 80-20 Self-similar Distribution

the fact table contains 1 million records. The dotted line represents the previous upper bound. In real-datasets the result is strongly affected by any skewness observed. In Figure 5 we depict another dataset, in which the values along each dimension are following an independent 80-20 self-similar distribution [GSE+94]: 20% of each dimension's domain was present in 80% of the input records. The resulting views are now much smaller that the previous uniform case, as aggregates fall into existing computed cells.

For estimating the size of the views we can take a random sample of the dataset, compute the views on that and linearly scale up the estimate to the whole dataset. Unfortunately this estimate is very crude as scaling produces a biased estimator for the size of the views. Another problem with sampling is that even if a small fraction of the base data is accessed, it is still relatively expensive, as every sampled tuple may require one page access, depending on the strategy used. Shukla, Deshpande et al in [SDNR96] propose another strategy for estimating the size of the views using a probabilistic counting algorithm [FM85] that counts the number of distinct elements observed in a multi-set. The algorithm makes a single pass over the fact table F and produces an estimate using a fixed amount of memory. It maintains a bit vector $B[0 \ldots L-1]$, where L is a parameter, depending on the available memory. It also requires a hashing function $h()$ that uniformly distributes the input values from F into range $[0 \ldots 2^L - 1]$. For each tuple x read from F, we set bit $B[i]$, where i is the position of the least significant bit in the binary representation of

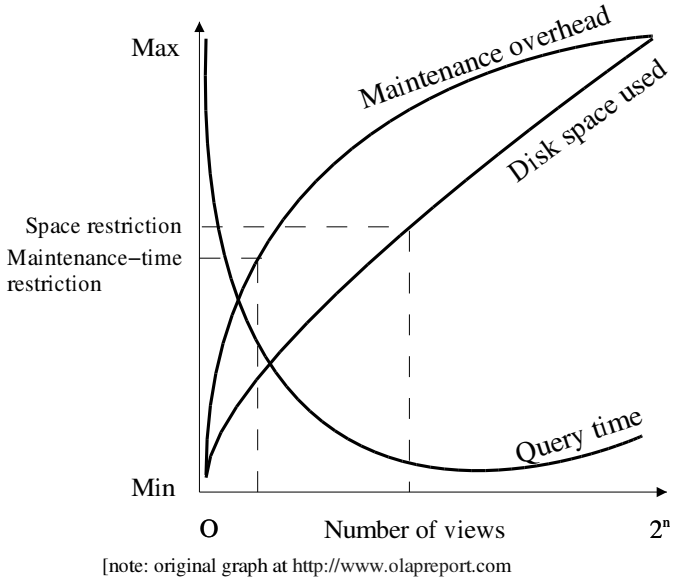

Figure 6: View Selection Tradeoffs

$h(x)$. For a perfectly uniform hashing function $h()$, $B[i]$ is set with probability $\frac{n}{2^{i+1}}$. Therefore, we can use the position of the leftmost zero bit in B to estimate the number of distinct elements in the input. This estimate is typically within a factor of 2 from the actual value, however precision can be improved using multiple hashing functions and bitmaps.

5 View Selection for Data Warehouses

For n dimensions there are 2^n different views that we can materialize and use for future queries. Even for a small number of dimensions the possible number of views will typically allow just a small fraction of them to be materialized. Picking the right subset of them is a non-trivial task because of the dependencies among the views that are depicted in the lattice of Figure 2. Based on the $\preceq$ partial order, a materialized view may be used to answer queries on other views too. Thus, that we may decide to materialize an infrequently used view if it allows us to answer queries on many other views with acceptable overhead. Furthermore, we may also include a view that is of no interest at query time but allows fast updates on other views as discussed in section 8.

View selection is usually modeled as the problem of finding those views that minimize query response time under some resource constraint. The resource can be the available disk space, or the time that we can spend on maintaining the views when the underlying data gets refreshed, or both. Extensive materialization will lead to diminishing query response gains and unacceptable disk-space and maintenance-time overhead. Figure 6 shows query response, disk space and maintenance cost as more views get materialized. The graphs imply that there is an optimum amount of views that deliver query response that is fairly close to what we could have achieved with a full materialization, but with much smaller resource consumption. It is interesting that adding extra views, will in some cases deliver worst query performance than a more conservative partial materialization [olab]. The reason is that as database size increases a smaller subset of the views will remain in memory buffers. In such cases it might be prudent to maintain fewer views and do aggregations on the fly, when required.

Karloff and Mihail in [KM99] show that no polynomial time, in the number of views approximation (with respect to query response time), algorithm exists for the view selection problem unless $P \neq NP$. This means that every polynomial-time algorithm will output solutions with query response arbitrary worst compared to the optimal selection. Thus, most studies focus on special cases of practical significance based on heuristics that guide the selection process.

Roussopoulos in [Rou82] first explored the problem of selecting a set of materialized views (with no aggregations) for answering queries under the presence of updates and a global space constraint. Haranarayan, Rajaraman and Ullman in [HRU96] model the view selection problem using the lattice framework. They use the benefit that we gain by materializing a view as a heuristic for choosing among the candidate views. Intuitively the benefit of a view depicts how materializing the view improves the cost of evaluating other non-materialized views including itself. More formally if $\mathcal{V}$ is a set of views that are already materialized, then the benefit $B(v, \mathcal{V})$ of adding a new view v in $\mathcal{V}$ is defined as follows: for each view $u \preceq v$ let $C_{\mathcal{V}}(u)$ be the cost of computing u from $\mathcal{V}$ and $C_v(u)$ the cost of computing u from v, after v is materialized. Then the benefit of v with respect to $\mathcal{V}$ is given from the formula:

$$B(v, \mathcal{V}) = \sum_{u:u \preceq v \wedge C_v(u) < C_{\mathcal{V}}(u)} (C_{\mathcal{V}}(u) - C_v(u))$$

Set $\mathcal{V}$ initially contains the top view from the lattice as there is no other view that can be used to answer queries to that view. Then, a greedy algorithm adds more views to $\mathcal{V}$ until a disk space constraint is fulfilled. The benefit metric can be easily modified to take into account possible knowledge of the query pattern on the views: we simply multiply the costs $C_{v/\mathcal{V}}(u)$ with the expected frequency rate of queries on that view.

The greedy algorithm picks a set of views $\mathcal{V}$ with at least 63% the benefit of an optimal solution. More formally if B_{greedy} is the benefit of k views chosen by the greedy algorithm and B_{opt} is the benefit of an optimal set of k views then $B_{greedy}/B_{opt} \geq 1 - \frac{1}{e}$. Notice that only the benefit of the solution is bound to be relatively close to that of the optimal selection and there is no guarantee on the query response times for the chosen views. A drawback of this algorithm is that it is, in most cases, impractically slow. Each greedy step is quadratic to the number of views and this yields an $O(2^{3d})$ worst case running cost. Shukla, Deshpande and Naughton recently showed [SDN98] that a simple algorithm that picks views according to their size achieves the same benefit bound, for many practical cases.

A follow-up research [GHRU97] investigates the combined view and index selection problem under a given space constraint. For materialized views that are very large, having them pre-computed and stored in summary tables is wasteful, unless we index them in order to support fast access to individual records. The authors present a family of algorithms of increasing time complexity that consider also indices (B-trees) for the selected views.

Smith et al in [SLCJ98] propose a method of decomposing the views into a hierarchy of view elements that correspond to partial and residual aggregations. Their algorithm picks a non-redundant set of such elements and minimizes query response time. If extra space is available a second algorithm releases the non-redundancy requirement and focuses on selecting a set of elements that minimizes query processing for a target storage bound. The main drawback of this approach is the extremely high complexity of the decomposition step, which makes the algorithms impractical even for a limited number of dimensions/views.

Selecting a view set to materialize and possibly some indices on them, is just the tip of the iceberg. Clearly, query performance is improved as more views are materialized. With the cost of disks constantly dropping, disk storage constraint is no longer the limiting factor in the view selection but the time to refresh the materialized views during updates. More materialization implies a larger maintenance window. This update window is the major data warehouse parameter, constraining over-materialization, as seen in Figure 6. The work we discussed so far ignores the maintenance cost of the views. Gupta in [Gup97] provides a theoretical-framework for the general view selection problem and presents polynomial-time algorithms for some special cases, which lower-bound the benefit of the optimal solution. In particular, he considers the case where both query response time and the maintenance cost is to be minimized for a bounded space. This framework is extended in [Gup99] to address the problem of selecting views to materialize under the constraint of a given amount of total maintenance time. Baralis, Paraboschi and Teniente in [BPT97] and Yang, Karlapalem and Li in [YKL97] present various algorithms for minimizing the response time and the maintenance overhead without any resource constraint. Labio, Quass and Adelberg in [LQA97] use an A^* search, similar to that of [Rou82] to pick the best set of views when only the maintenance cost is to be minimized. Finally, Theodoratos and Sellis in [TS97] 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. They propose a genetic algorithm that gradually refines a sub-optimal selection by making local configuration changes.

6 Dynamic Caching of Views

The idea behind view selection is that a fairly small number of views may provide substantial performance boost for many complex analytical queries. In many cases however users submit their queries interactively, i.e they do not have a predetermined 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 data in surprising ways that are not best supported from the materialized views selected by the previous algorithms. In addition decision support queries

typically return relatively small results containing few interesting aggregates. Query: "find the total sales for the last 5 years in all stores in NJ" is a fine example of that. Processing this query requires scanning and aggregating lots of detailed records, while the result is just a single value.

Furthermore, as users query patterns and data trends change overtime 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 selected set of views should be monitored and re-calibrated if query performance is not satisfactory. This task for a complex data warehouse where many users with different profiles submit their queries is rather complicated and time consuming. In addition, the maintenance window, the disk space restrictions and other important operational parameters of the system may also change. For example, an unexpected large volume of daily updates will throw the selected set of views as not update-able unless some of these views are discarded.

An observation from Figure 6 is that the selection process is guided with respect to two constrains: the available disk space to store the aggregates and the required maintenance window when the views get refreshed. When optimizing for both space and maintenance-time it is likely that the selection will fully utilize only one of them. In Figure 6 the maintenance-time constrain is stricter and does not allow us to add extra views, even-though we can afford the disk space. At query time this extra space is waisted even-though it could be used to temporary stage other aggregates.

To summarize our discussion the following postulates are made:

- ad-hoc analysis is in many cases unpredictable. It might be hard to find a set of views that fits all users.
- query results are often relatively small, as they contain aggregated data
- data is relatively static with only infrequent updates that are happening in predetermined intervals
- a static selection of views can not fully utilize the disk-space and maintenance-time restrictions of the system

These observations suggest that a query result caching architecture is particularly well suited for a data warehouse environment. The cache manager utilizes a dedicated disk space for storing computed aggregates that are further engaged for answering new queries. The problem differs from traditional caching in the following aspects:

- cached aggregates have different sizes and computation costs. An aggregate query may yield a result as big as the top view of the lattice and as small as a single aggregate. Furthermore, it is far more expensive to recompute results of a high level of aggregation since they require scanning and processing lots of detailed records. This implies that LRU or LFU replacement policies are probably not well suited for managing the cache.

- cached results are often not independent. We have already discussed dependencies of materialized aggregates in the lattice framework of Figure 2. Drill-down and roll-up queries that are common in OLAP analysis tend to fill the cache with results of different aggregation levels. An effective caching architecture should understand and exploit the dependencies among the aggregates. For instance a more detailed cached result may be used to answer a coarser aggregate query in the future.

- cached results get dirty when the underlying data is modified. Traditional caching typically invalidates dirty objects when data changes at the sources. However, this practice is not efficient for disk resident materialized aggregates with potentially large re-computation overhead. Assuming updates are happening in a periodic fashion we need algorithms that will efficiently maintain the cache with respect to the changes and the dependencies among the cached results. If the whole cache can not be updated within the allowed maintenance period we would have to choose among the cached results and discard some of the aggregates.

- testing whether the cache content can be used to answer a query can be hard as discussed in the next section. Thus, we need practical implementations that will restrict the form of queries admitted in the cache to allow for fast look-ups at query time.

A framework for caching and reusing results in relational database systems has been presented in [Sel88]. The WATCHMAN cache manager, introduced in [SSV96] uses replacement and admis-

sion techniques tuned for analytical workload and is used in the dynamic caching system of [SSV99]. The cache manager dynamically maintains the content of the cache by deciding whether a newly computed query result should be admitted and if so which already cached result should be evicted to free some space. The admission and replacement algorithms are based on a profit metric, which considers for each result the average rate of reference, its size and its re-computation cost. [KR99] extends this metric to take into account the content of the cache and the maintenance cost of the result when the base data is updated. The intuition is that if two results are cached as views v and u and both have fairly large re-computation costs but $v \preceq u$ then v should probably get a smaller profit score, since it can always be re-computed from the cache using u without accessing the detailed records. Similarly, we might want to credit u with higher profit value based on the number of cached results that can be recomputed from u or be maintained from u in the hybrid maintenance scheme discussed in section 8.

Testing whether a cached result of an arbitrary aggregation query can be used to process a new query is NP-hard [YL95] in general. To overcome this difficulty dynamic caching systems restrict the form of queries that are maintained in the cache. [KR99] brakes an incoming query into a set of *multidimensional query fragments*. These are SPJ-queries where each selection predicate is of the form *attribute = literal*. The cache manager uses a network of multi-dimensional indices organized in a lattice topology for locating cached results that can answer a new query. [SSV99] uses a broader class of *canonical* queries. These are aggregation queries of the form:

```
select selection_list, aggr_list
from table_list
where join_condition
and select_condition
group by groupby_list
```

The *join_condition* is a list of equality predicates among attributes of the fact and the dimension tables connected by AND. Predicates in *select_condition* involve only single attributes and literals connected with one of the $<, >, \leq, \geq, =, \neq, between$ operators. Such a query is transform

into queries q_1 and q_2 where q_1 contains no selection clause and q_2 returns the same result as q when evaluated over the result set of q_1. q_1 is called the base query and if no joins are present it corresponds to one of the nodes of the data cube lattice.[1].

[DRSN98] introduces a different caching architecture where data are organized in the form of *chunks*. These chunks corresponds to partitions created when a uniform multidimensional grid is imposed on-top of the dataset. Answering a query translates into finding the appropriate chunks that contain the requires aggregates. Two potential drawbacks of this approach is that chunking might not work well on skewed datasets and also that it requires the data warehouse tables to be stored using a specialized *chunked file organization* [SS94].

7 Answering Queries Using Views

In many cases, queries on the data warehouse can be answered using the materialized aggregate views without accessing the detailed records. Given a query q and a view v, checking if all records of q are stored in v (along with possible uninteresting tuples) is reduced to the query containment problem and is well known to be NP-hard [AD98, KV98, KMT98]. For the special case of SPJ-views there are algorithms [LY85, YL87, CKPS95, LMSS95, SDJL96] that can be used to optimize the execution of user queries against the views.

The data cube views have a very restricted form that makes the problem somehow easier, since they contain no joins and selections but only groupings over dimension's keys. The most common way to use such a view is to roll up by grouping on additional columns and add possibly some selection filters. Even in this case we should make sure that the aggregate functions used in the view can be safelly rolled up for answering the query. For example a view that computes both *count()* and *avg()* can be rolled up for computing *sum()* for a query but for other statistical functions rolling up the aggregates may not be possible.

In the general case the views that have been materialized in the data warehouse may contain selections and joins between the fact and some of the dimension tables. For example assume that

[1] the lattice notation is extended when joins are allowed between the fact and the dimension tables as described in [HRU96]

our star schema of Figure 1 has been extended to include the time dimension and a materialized view v stores the total sales by quarter for each part:

```
v: create view v as
   select part.partkey, time.quarter, time.year, sum(sales) as total_sales
   from F, part, time
   where F.partkey = part.partkey and F.timekey = time.timekey
   group by part.partkey, time.quarter, time.year
```

Assume now a query q that requests the total sales per part for a specific category of parts (e.g. "auto-parts") for the year 2000:

```
q: select part.partkey, sum(sales) as total_sales
   from F, part, time
   where F.partkey = part.partkey and F.timekey = time.timekey
   and part.category = ''auto-parts'' and time.year=2000
   group by part.partkey
```

in the presence of v the query can be rewritten as following:

```
q(v): select v.partkey,sum(v.total_sales) as total_sales
      from v, part,
      where v.partkey = part.partkey
      and v.year = 2000
      and part.category = ''auto-parts''
      group by v.partkey
```

In this rewriting we avoid joining with the fact and the time tables since the view contains the necessary information to roll up from quarters to years, however we still have to do a join with the dimension table for parts to get the category attribute. Such a join is called a joinback [BDD+98] and is made possible because we know that the *partkey* attribute in the view is a primary key

on that table and thus no information is lost when using the view. We would like to point out here that any rewriting functionality should be integrated within the query optimizer that knows about integrity constrains, value distributions and possible indexes on the views and the tables. For example it might be faster to query the fact table through an appropriate index than rewriting a query to use a large unindexed materialized view.

The hierarchical and functional relationships of attributes stored in the dimension tables should be taken into account when doing roll ups or joinbacks. For the previous example we used the fact that *partkey* is the primary key for the part dimension to infer the missing category attribute from the part table. Functional dependency information is also necessary when rolling up aggregates. For example if city uniquely identifies a state in the customer table it is valid to roll up sum of sales by city to sum of sales by state. Sometimes domain knowledge is necessary to correctly interpret the results. For example if we do not sell to all cities in NJ, then the previous aggregates will refer to a subset of cities in the NJ. Similarly if a part belongs to multiple categories (this requires some modifications in our schema) then rolling up the sum of sales per category to compute the overall sales gives an incorrect result.

8 Updating the Views

As changes are made to the base tables of the data warehouse, the materialized aggregate views must also be updated to reflect the new state of the detailed records. The data warehouse tables are themselves views of multiple external data sources that periodically ship their updates to the repository. Changes are not made immediately but are deferred and applied in large batches during a down-time period. This not only allows the data warehouse to provide a consistent snapshot during analysis but makes maintenance more efficient using bulk load and update techniques. In the database literature there is an abundance of work related to view maintenance[2]. Frequently, only a small part of the view changes in response to changes in the base relations, or similarly few of the changes affect the view. In such cases it might be faster to compute only the changes

[2] refer to [GM95] for a survey

partkey	total_sales	max_sale
100	28	4
101	11	5
102	5	5
103	17	8
104	22	8

Figure 7: Materialized View v on *partkey*

partkey	suppkey	custkey	amount
102	...	...	7
100	...	...	2
103	...	...	1
102	...	...	3
106	...	...	9
102	...	...	5

Figure 8: F^+: insertion in the fact table

in the view to update its materialization. This is called incremental view maintenance and uses a delta paradigm to represent changes of the base relations that are then used to update the view(s). This implies that we have access to the set of changes that have happened to the base data. Unfortunately this is not always the case, especially for views over distributed data sources that often don't use relational databases. Furthermore, even if the data sources are willing to report changes, querying them during maintenance may be prohibitively expensive [QGMW96].

For our discussion, we assume that all aggregate views are updated with respect to changes in the fact table, after the fact table itself has been updated. Different policies are implemented, depending on the types of updates and the properties of the aggregate functions that are computed by the views. One can always recompute the aggregate views from scratch, using techniques described in section 4, every time the fact and/or the dimension tables are modified. This approach ultimately leads to unacceptable performance as the size of the data warehouse increases over time.

Recently we have seen incremental update algorithms [GMS93, GL95, JMS95, Qua96, MQM97, RKR97, KR98] that handle views with aggregations. These algorithms avoid full re-computation of the views by using appropriate maintenance expressions in response to changes in the dataset. Often such changes involve only insertions, however the update process should be able to handle both insertions and deletions. The delta paradigm divides these changes into two sets F^+ and F^-. Set F^+ contains all new tuples inserted in the fact table, while F^- all deleted records. Updates (modifications) are expressed in this framework as a deletion followed by an insertion. For a materialized view v our goal is to create an expression that will correctly reflect changes F^+

partkey	total_sales	max_sale
100	2	2
102	15	7
103	1	1
106	9	9

Figure 9: Summary delta table $delta_v^+$

partkey	total_sales	max_sale
100	30	4
101	11	5
102	20	7
103	18	8
104	22	8
106	9	9

Figure 10: Updated View

and F^- to v. This may or may not be possible depending on the aggregate functions that are computed by the view and the type of changes that we want to apply. All distributive functions like *count()*, *sum()*, *max()* can be refreshed incrementally when only insertions are allowed. For example assume that the following view is materialized in the data warehouse:

v: `create view` v `as`
`select partkey,sum(amount) as total_sales, max(amount) as max_sale`
`from` F
`group by partkey`

The view computes the maximum and total sales per part from the transactions stored in the fact table F. Figures 7,8 provide a snapshot for this view along with a set of insertions F^+ that we want to apply. Both *sum()* and *max()* allow further aggregation based on the new data. Based on this property we can create a summary delta table [MQM97, RKR97] for this view as follows:

$delta_v^+$: `create view` $delta_v^+$ `as`
`select partkey,sum(amount) as total_sales, max(amount) as max_sale`
`from` F^+
`group by partkey`

$delta_v^+$ is shown in Figure 9 and applies the definition of the view on the new records only. We can now *merge* the old snapshot of the view with the newly computed aggregates in the following way:

1. if for a record in $delta_v^+$ there is no a record in v with the same *partkey* value we copy the record to the view. This is the case for the new entry with *partkey*=106.

2. if there is already an entry in v, for the *sum()* function we add the two aggregates and update the value stored in v, while for the *max()* function we keep the maximum of the two.

Figure 10 shows the refreshed view after changes in F^+ have been applied. In order to implement the merge procedure we can open a cursor in $delta_v^+$ and check each record against the view. For this process to work efficiently there should be an index on the primary key *partkey* of the view. However, for large batches of insertions checking the index over and over again will yield a substantial overhead. If on the other hand group bys are computed using a sort-based algorithm we could take advantage of matching sort orders (like the case of Figures 7,9) and simply merge the aggregates by sequentially scanning the records of both tables [RKR97].

In the presence of multiple views we can use optimizations similar to those employed for computing the views at the fist place. For example we can compute $delta_v^+$ from the summary-delta table $delta_u^+$ of another view u if $v \preceq u$ and the delta table of u is smaller that F^+. Alternatively, if u has already been updated, we can re-compute view v from u and avoid the overhead of materializing its summary-delta table. This implies a hybrid maintenance scheme, in which some of the views are updated incrementally from the deltas, others are recomputed from F and others are re-computed from other views. The decision is based on a cost-based optimization that takes into account the time that we can spend in maintaining the views, the disk space available for temporal results and the data structures that we use for storing and indexing the views and the fact table. [KR99] describes such a hybrid update algorithm.

Table 8 summarizes the alternative ways that view v can be maintained with respect to changes in F^+. We would like to stretch out that incremental updates are not a panacea for the view maintenance problem. Complex maintenance expressions over unindexed data/deltas may result to thrashing and void any benefits from incremental computations of the aggregate functions as shown in [KR98].

Incremental and hybrid update algorithms can be used for algebraic functions too, however their efficiency is questionable if the function requires significant state information to describe a

policy	*description*
re-computation	recompute from F
incremental	incrementally from $delta_v^+$
hybrid	update $u : v \preceq u$, recompute v from u

Table 1: Update policies for aggregate views

sub-aggregate. For example *avg()* can be maintained using partial *sum()* and *count()* information, but for *maxN()* the implementation will be very inefficient.

If we try to extend the delta-paradigm for deletions we will find that the results do not hold for all distributive functions. For instance if F^- contains just a single record with *partkey*=104 and *amount*=8 this tells us that the entry with the maximum sales for this part is deleted from the fact table. However, we have no way of knowing what the new maximum sale for this part is without looking back at the fact table. For the *sum()* function the situation is slightly better. Each deleted record can be expressed as an insertion where the value of the measure is negated. This yields correct output from the merging step if we remove from v records whose *sum()* aggregate is zero, as they correspond to entries that contained some value but their individual records are now deleted from F. Thus, we can treat zero as a special value that indicates when all tuples in a group have been deleted. One however should restrain from using such deductions because they are highly application depended and can lead to unpredictable results. In order to consistently maintain the *sum()* aggregate in the presence of deletions we can include a *count()* function in the view definition. When *count()* becomes zero, we can safely deduct that the record can be removed from the view.

The authors of [MQM97] define a function as self-maintenable if its new value can be computed solely from the old value and the changes to the base records with respect to new updates. Functions can be self-maintenable with respect to insertions F^+, deletions F^- or both. All distributive functions are by definition self-maintenable with respect to insertions and all self-maintenable functions must be distributive. Adding extra information can make a distributive function like

sum() self-maintenable with respect to deletions but this is not always the case. For instance if $count() > 0$ we still need to look in F to find the new value for *max()* and *min()* after the maximum or the minimum value is deleted.

9 Final Comments

Materialized views and their implications have been recently rediscovered for the content of OLAP and data warehousing. A flurry of papers has been generated on how views can be used to accelerate ad-hoc computations over massive datasets. Picking and materializing the right set of views, with respect to the workload and the disk space and maintenance time constraints has also received considerable attention. Recent approaches seem to agree that their use is best exemplified in a dynamic environment, in which a materialized set of views is reconciled with each new query. Substantial effort has also been given for optimizing their computation and maintenance tasks. From the commercial side materialized views are eventually getting the attention they deserve in products like Oracle 8.1, IBM DB2 [ZCL+00] and HP Intelligent Warehouse. Materialized views with their versatility and potential introduce new challenges with interesting research and engineering questions. Taking a leap from the centralized data warehouse model and moving in a distributed and possibly mobile world, we are faced with new challenges in view management. Thus, the excitement about materialized views and their applications is expected to continue for the foreseeable future.

References

[AAD+96] S. Agrawal, R. Agrawal, P.M. Deshpande, A. Gupta, J.F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the Computation of Multidimensional Aggregates. In *Proceedings of the 22nd VLDB conference*, pages 506–521, Bombay, India, August 1996.

[AD98] S. Abiteboul and O. M. Duschka. Complexity of Answering Queries Using Materialized Views. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART*

Symposium on Principles of Database Systems, pages 254–263, Seattle, Washington, June 1998.

[BDD+98] Randall G. Bello, Karl Dias, Alan Downing, James Feenan Jr., William D. Norcott, Harry Sun, Andrew Witkowski, and Mohamed Ziauddin. Materialized Views in Oracle. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 659–664, New York City, New York, August 1998.

[BE97] R. Barquin and H. Edelstein, editors. *Building, Using and Managing the Data Warehouse*. The Data Warehousing Institute Series. Prentice Hall PTR, 1997.

[BPT97] E. Baralis, S. Paraboschi, and E. Teniente. Materialized View Selection in a Multidimensional Database. In *Proceedings of the 23th International Conference on VLDB*, pages 156–165, Athens, Greece, August 1997.

[BR99] Kevin S. Beyer and Raghu Ramakrishnan. Bottom-Up Computation of Sparse and Iceberg CUBEs. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 359–370, Philadephia, Pennsylvania, June 1999.

[CD97] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record*, 26(1), September 1997.

[CKPS95] S. Chaudhuri, R.i Krishnamurthy, S. Potamianos, and K. Shim. Optimizing Queries with Materialized Views. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 190–200, Taipei, Taiwan, March 1995.

[Col96] G. Colliat. OLAP, Relational and Multidimensional Database Systems. *SIGMOD Record*, 25(4):64–69, Sept 1996.

[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.

[FM85] P. Flajolet and G. N. Martin. Probabilistic Counting Algorithms for Database Applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[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 *Proceedings of the 12th ICDE Conference*, 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 Conference*, pages 328–339, San Jose, CA, May 1995.

[GM95] A. Gupta and I. Singh Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *Data Engineering Bulletin*, 18(2):3–18, 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.

[Gra93] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[GSE+94] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. Weiberger. Quickly Generating Billion-Record Synthetic Databases. In *Proc. of the ACM SIGMOD*, pages 243–252, Minneapolis, May 1994.

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

[Gup99] H. Gupta. Selection of Views to Materialize Under a Maintenance Cost Constraint. In *Proceedings of ICDT*, Jerusalam, Israel, January 1999.

[HRU96] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing Data Cubes Efficiently. In *Proceedings 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.

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

[KM99] H. J. Karloff and M. Mihail. On the Complexity of the View-Selection Problem. In *Proceedings of the 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 167–173, Philadelphia, Pennsylvania, May 1999.

[KMT98] P. G. Kolaitis, D. L. Martin, and M. N. Thakur. On the Complexity of the Containment Problem for Conjunctive Queries with Built-in Predicates. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 197–204, Seattle, Washington, June 1998.

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

[KR99] Yannis Kotidis and Nick Roussopoulos. DynaMat: A Dynamic View Management System for Data Warehouses. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 371–382, Philadelphia, Pennsylvania, June 1999.

[KV98] P. G. Kolaitis and M. Y. Vardi. Conjunctive-Query Containment and Constraint Satisfaction. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 205–213, Seattle, Washington, June 1998. ACM Press.

[LMSS95] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering Queries Using Views. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 95–104, San Jose, California, May 1995.

[LQA97] W. Labio, D. Quass, and B. Adelberg. Physical Database Design for Data Warehouses. In *Proceedings of ICDE*, pages 277–288, Birmingham, U.K., April 1997.

[LRS99] Jianzhong Li, Doron Rotem, and Jaideep Srivastava. Aggregation Algorithms for Very Large Compressed Data Warehouses. In *Proceedings of 25th International Conference on Very Large Data Bases*, pages 651–662, Edinburgh, Scotland, September 1999.

[LY85] P.-Å. Larson and H. Z. Yang. Computing Queries from Derived Relations. In *Proceedings of the 11th VLDB Conference*, pages 259–269, Stockholm, Sweden, 1985.

[MK99] Seigo Muto and Masaru Kitsuregawa. A Dynamic Load Balancing Strategy for Parallel Datacube Computation. In *DOLAP '99*, pages 67–72, Kansas City, Missouri, November 1999.

[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.

[olaa] The OLAP Council.
http://www.olapcouncil.org.

[olab] The OLAP Report.
http://www.olapreport.com.

[QGMW96] D. Quass, A. Gupta, I.S. Mumick, and J. Widom. Making Views Self-Maintainable for Data Warehousing. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems*, pages 158–169, Miami Beach, Florida, December 1996.

[Qua96] D. Quass. Maintenance Expressions for Views with Aggregation. In *Proceedings of VIEWS 96*, pages 110–118, Montral, Canada, June 1996.

[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.

[Rou82] N. Roussopoulos. View Indexing in Relational Databases. *ACM Transactions on Database Systems*, 7(2):258–290, June 1982.

[RS97] K.A. Ross and D. Srivastava. Fast Computation of Sparse Datacubes. In *Proceedings of the 23th VLDB Conference*, pages 116–125, Athens, Greece, Augoust 1997.

[SDJL96] D. Srivastava, S. Dar, H.V. Jagadish, and A. Y. Levy. Answering Queries with Aggregation Using Views. In *Proceedings of the 22th International Conference on Very Large Data Bases*, pages 318–329, Mumbai (Bombay), India, September 1996.

[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.

[SDNR96] A. Shukla, P.M. Deshpande, J.F. Naughton, and K. Ramasamy. Storage Estimation for Multidimensional Aggregates in the Presense of Hierarchies. In *Proc. of VLDB*, pages 522–531, Bombay, India, August 1996.

[Sel88] T. K. Sellis. Intelligent Caching and Indexing Techniques for Relational Database Systems. *Information Systems*, 13(2):175–185, 1988.

[SLCJ98] J. R. Smith, C. Li, V. Castelli, and A. Jhingran. Dynamic Assembly of Views in Data Cubes. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 274–283, Seattle, Washington, June 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.

[SSV99] Junho Shim, Peter Scheuermann, and Radek Vingralek. Dynamic Caching of Query Results for Decision Support Systems. In *SSDBM*, pages 254–263, Cleaveland, Ohio, July 1999.

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

[YKL97] J. Yang, K. Karlapalem, and Q. Li. Algorithms for Materialized View Design in Data Warehousing Environment. In *Proceedings of the 23th VLDB Conference*, pages 136–145, Athens, Greece, Augoust 1997.

[YL87] H. Z. Yang and Per-Åke Larson. Query Transformation for PSJ-Queries. In *Proceedings of 13th International Conference on Very Large Data Bases*, pages 245–254, Brighton, England, September 1987.

[YL95] Weipeng P. Yan and Per-Åke Larson. Eager Aggregation and Lazy Aggregation. In *Proceedings of 21th International Conference on Very Large Data Bases*, pages 345–357, Zurich, Switzerland, September 1995.

[ZCL+00] Markos Zaharioudakis, Roberta Cochrane, George Lapis, Hamid Pirahesh, and Monica Urata. Answering Complex SQL Queries Using Automatic Summary Tables. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 105–116, Dallas, Texas, May 2000.

[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.