The Opsis Project: Materialized Views for Data Warehouses and the Web

Nick Roussopoulos[1], Yannis Kotidis[2⋆], Alexandros Labrinidis[1], and Yannis Sismanis[1]

[1] Department of Computer Science
University of Maryland
College Park, MD 20742, USA
{nick,labrinid,isis}@cs.umd.edu
[2] AT&T Labs Research
180 Park Ave, P.O. Box 971,
Florham Park, NJ 07932, USA
kotidis@research.att.com

Abstract. The real world we live in is mostly perceived through an incredibly large collection of views generated by humans, machines, and other systems. This is the view reality. The Opsis project concentrates its efforts in dealing with the multifaceted form and complexity of data views including data projection views, aggregate views, summary views (synopses), point of view views, and finally web views. In particular, Opsis deals with the generation, the storage organization (Cubetrees), the efficient run-time management (Dynamat) of materialized views for Data Warehouse systems, and for web servers with dynamic content (WebViews).

1 Introduction

Most of the data stored and used today is in the form of *materialized views*, generated from several possibly distributed and loosely coupled source databases. These views are sorted and organized appropriately in order to rapidly answer various types of queries. The relational model is typically used to define each view and each definition serves a dual purpose, first as a specification technique and second as an execution plan for the derivation of the view data.

Materialized views are approximately 10 years younger than the relational model. Early papers that foresaw their importance include [26, 25, 7, 29, 34, 31, 30]. During this period, materialized views were considered by top relationalists as the "Pandora's box". It took another 6-7 years before it was realized how useful and versatile they were. Then, a flurry of papers rehashed the earlier results and almost brought the research on materialized views to extinction. But, materialized views were too important and research continues as of today. Relational views have several forms:

- *pure program:* an unmaterialized view is a program specification, "the intention", that generates data. Query modification [32] and compiled queries [2] were the first techniques exploiting views– their basic difference is that the first is used as a macro that does not get optimized until run-time, while the second stores optimized execution plans. Each time

⋆ Work performed while the author was with the Department of Computer Science, University of Maryland, College Park.

the view program is invoked, it generates (materializes) the data at a cost that is roughly the same for each invocation.

- *derived data:* a materialized view is "the extension" of the pure program form and has the characteristics of data like any other relational data. Thus, it can be further queried to build *views-on-views* or collectively grouped to build *super-views*. The derivation operations are attached to materialized views. These procedural attachments along with some "delta" relational algebra are used to perform incremental updates on the extension.
- *pure data:* when materialized views are converted to snapshots, the derivation procedure is detached and the views become pure data that is not maintainable (pure data is at the opposite end of the spectrum from pure program).
- *pure index:* view indexes [26] and ViewCaches [27] illustrate this flavor of views. Their extension has only pointers to the underlying data which are dereferenced when the values are needed. Like all indexing schemes, the importance of indexes lies in their organization, which facilitates easy manipulation of pointers and efficient single-pass dereferencing, and thus avoids thrashing.
- *hybrid data & index:* a partially materialized view [3] stores some attributes as data while the rest are referenced through pointers. This form combines data and indexes. B-trees, Join indexes [35], star-indexes and most of the other indexing schemes belong to this category, with appropriate schema mapping for translating pointers to record field values.
- *OLAP aggregate/indexing:* a data cube [10] is a set of materialized or indexed views [12, 23, 18]. They correspond to projections of the multi-dimensional space data to lesser dimensionality subspaces and store aggregate values in it. In this form, the data values are aggregated from a collection of underlying relation values. Summary tables and Star Schemas belong in this form as well.
- *WebViews:* HTML fragments or entire web pages that are automatically created from base data, typically stored in a DBMS [20]. Having a WebView materialized can potentially give significantly lower query response times, compared computing it on the fly. However, it may also lead to performance degradation, if the update workload is too high.

Each of these forms is used by some component of a relational system. Having a unified view of all forms of relational views is important in recognizing commonalities, re-using implementation techniques, and discovering potential uses not yet exploited. The Opsis project has focused on developing storage and update techniques for all forms of materialized views. We have been particularly careful with the efficient implementation and scalability of these methods. We have architected, designed, implemented, and tested giant-scale materialized view engines for the demands of todays abundance of connectivity and data collection.

This paper is organized as follows. In the next section we describe the Cubetree Data Model, a storage abstraction for the data cube, and also present a compact representation for it using packed R-trees [24]. In Section 3, we present our algorithm for bulk incremental updates of the data cube. Section 4 has a brief outline of *DynaMat*, a view management system that materializes results from incoming aggregate queries as views and exploits them for future reuse. In Section 5 we explore the materialization policies for WebViews and present results from experiments on an industrial-strength prototype. Section 6 discusses the Quality of Service and Quality of Data considerations for WebViews. Finally, we conclude in Section 7.

2 A Storage Abstraction for OLAP Aggregate Views

Consider the relation $R(A, B, C, Q)$ where A, B, and C are the *grouping attributes* that we would like to compute the cube for the *measure attribute* Q. We represent the grouping attributes A, B, and C on the three axes of AxBxC and then map each tuple $T(a, b, c, q)$ of R using the values a,b,c for coordinates and the value q as the content of the data point $T(a, b, c)$. We now project all the data points on all subspaces of AxBxC and aggregate their content. We assume that each domain of R has been extended to include a special value (zero in this example) on which we do the projections. A projection on a subspace D^K with dimension $K <= N$, where N is the number of grouping attributes, represents the *group by* of all those attributes that correspond to D^K. The aggregate values of D^K are stored in the intersection points between D^K and the orthogonal $(N - K)$-dimensional hyper-planes that correspond to the remaining dimensions not included in D^K. For example, the projection planes $P1$,$P2$,... parallel to plane BxC shown in Figure 1, correspond to group by A and their aggregated values are stored in the content of their intersection point with axis A. The origin $O(0, 0, ..., 0)$ is used to store the (super)-aggregate value obtained by no grouping at all. We call this the *Cubetree Data Model* (CDM).

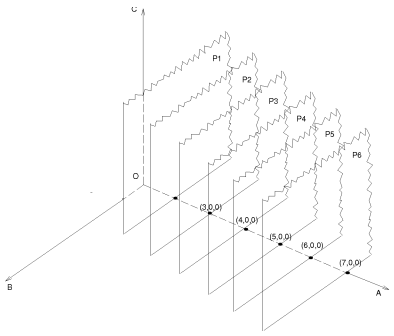


Fig. 1. group by A projections

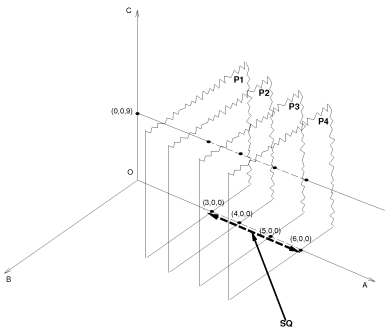


Fig. 2. Querying the Cube

In CDM, we map cube and relational queries into multi-dimensional range queries. For example, a query to find all the group by A values for A between 3 and 6 would be formulated as a range query $[(3, 0, 0) < A < (6, 0, 0)]$ shown by the bold-dashed line SQ in Figure 2. If now we would like to find out the percent contribution (multidimensional ratio) of $C = 9$ to these group by A values, we obtain the intersection points of line $C = 9$ with planes $P1$, $P2$, etc. and the content of them is divided by the corresponding aggregates on A.

Clearly, different combinations of relational, 1-dimensional or multi-dimensional storage structures can be used to realize the CDM. For example, the whole CDM can be realized by just a conventional relational storage [10] with no indexing capability for the cube. Another possibility, would be to realize CDM by an R-tree [14], or a combination of relational structures, R-trees and B-trees [5]. Since most of the indexing techniques are hierarchical, without loss of generality, we assume that the CDM is a tree-like (forest-like) structure that we refer to as the *Cubetree* of R.

3 Bulk Incremental Updates

Random record-at-a-time insertions are not only very slow because of the continuous reorganization of the space, but also destroy data clustering in all multidimensional indexing schemes. *Packed R-trees*, introduced in [24], avoid these problems by first sorting the objects in some desirable order and then *bulk loading* the R-tree from the sorted file and *packing* the nodes to capacity. This *sort-pack* method achieves excellent clustering and significantly reduces the *overlap* and *dead space* (i.e. space that contains no data points).

The proposed *bulk incremental update* computation is split into a *sort phase* where an update increment dR of relation R is sorted, and a *merge-pack phase* where the old Cubetree is packed together with the updates: $cubetree(R \cup dR) =$ Merge-Pack$(cubetree(R),$ sort$(dR))$. Sorting could be the dominant cost factor in the above incremental computation, but it can be parallelized and/or confined to a quantity that can be controlled by appropriate schedules for refreshing the cube. Note that dR contains any combination of relation insertions, deletions, and updates. For aggregate functions that are *Self Maintainable* [21] like *count()* and *sum()*, they are all equivalent because they all correspond to a write of all projection points with their content adjusted by appropriate arithmetic expressions.

We assume that a tuple in dR has the following structure: $< v_1, v_2, \ldots, v_N, q >$, where v_j, denotes the value on dimension j, j= $1, \ldots, N$ and q is the *measure* attribute. The generalization with more measure attributes is straightforward. During the sorting phase, we read dR and create a sorted run dg_i for each group by in the Cube. The format of a tuple in dg_i is: $< v_1^i, v_2^i, \ldots, v_{m_i}^i, agr_1, \ldots, agr_k >$, where $v_j^i, j = 1, \ldots, m_i$, denotes the values of each dimension in that group by. For a specified point in this m_i-dimensional space $agr_1, \ldots, agr_k$ hold the aggregated values. In order to be able to merge dg_i with the existing aggregates that are stored within the Cubetrees, the data within each update increment dg_i are being sorted in the same order as the data within the Cubetrees. For example, if all points of group by ABC are stored in the $A \rightarrow B \rightarrow C$ order, then the same sort order is being used for the new projections from the deltas.

During the second *merge-pack phase* the old Cubetrees are packed together with the updates. For each Cubetree r, all increments dg_i that are stored in that Cubetree are opened and merge-packed with r to create a new instance of the index.

3.1 Creation/Maintenance Measurements for a Grocery Demo Dataset

We used a synthetically generated grocery demo dataset that models supermarket transactions. The data warehouse is organized according to the star schema [17] organization. There is a single *fact table* `sales` that includes 12 dimension attributes and two real (4-byte) measure attributes, namely `revenue` and `cost`. We pre-computed several aggregate views and stored them within Cubetrees. These views aggregate data over attributes chosen from the dataset and compute the *sum()* aggregate for both measures.[1]

We ran the experiment on a 360MHz Ultra SPARC 60 with two SCSI 18GB Seagate Cheetah hard drives. For sorting the data, we used a simple quick-sort utility. There are numerous optimizations that we could have exploited for speeding up this phase, see [1, 22]. Table 1 shows

[1] A more detailed description of the dataset as well as an online demo are available at http://opsis.umiacs.umd.edu:8080

the time for the initial load of the Cubetrees and the time for each bulk-incremental update with a year's and five month's worth of data. The corresponding sizes for each update increment dR are also given.

Transactions	Ins+Upds	Total Records	Sort Time	Pack Time	Cubetrees (GB)	Packing rate
1/1/90-12/31/97	127,702,708	127,702,708	3h 04m 30s	5m 51s	2.92	29.95 GB/h
1/1/98-12/31/98	22,468,605	143,216,789	13m 52s	7m:54s	3.41	25.95 GB/h
1/1/99-5/31/99	26,027,692	160,587,143	10m 11s	8m 53s	3.83	25.88 GB/h

Table 1. Initial Bulk-load and Bulk-incremental Updates

For sorting and packing we utilized both Seagate Cheetah disks i.e. reading the input from one disk and packing the Cubetrees in the other. The size of the views for the initial creation was 2.92GB and the packing phase was completed in less than 6 minutes. This corresponds to a packing rate (speed) of 29.95GB/h or 8.52MB/sec, roughly 68% of the raw serial disk write rate. The remaining bandwidth is lost due to the necessary processing of the input. The second and third lines of the table show the performance during bulk-incremental updates. The effective disk packing rate that we got for updates was slightly slower, at about 26GB/h. This is because we only used two disks, storing the input data (updates) in the first and the Cubetrees in the second. Therefore, during updates both the old and the new-version of the Cubetrees were on the same disk sharing its I/O bandwidth.

4 Dynamic Management of Aggregate Views

Disk space and creation/maintenance overhead will not allow us to materialize all interesting group-bys of the data cube. The view selection problem [26, 15, 4, 33, 16] consists of finding those group-bys that minimize query response time under a resource constraint (typically disk space) and store them as materialized views.

This static selection of views however, contradicts the dynamic nature of decision support analysis. Especially for ad-hoc queries where the expert user is looking for interesting trends in the dataset, the query pattern is difficult to predict. Furthermore, as 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. In addition, the maintenance window, the disk space restrictions and other important functional parameters of the system 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.

Another inherit drawback of a static view selection scheme is that the system has no way of tuning a wrong selection by re-using results of queries that couldn't be answered by the materialized set. Notice that although OLAP queries take an enormous amount of disk I/O and CPU processing time to be completed, their output is often quite small as they summarize the underlying data. Moreover, during *roll−up* operations [10] the data is examined at a progressively coarser granularity and future queries are likely to be computable from previous results without accessing the base tables at all.

In [19] we introduced DynaMat, a dynamic view management system for data warehouses. Our work has been motivated by earlier research on caching and resusing query results in relational database systems [31, 9]. DynaMat manages a dedicated disk space that we call the *View*

Pool, in which previously computed aggregates are stored. There are two distinct modes of operation. The first is the "on-line" mode during which user queries are allowed. DynaMat determines whether or not aggregates (views) stored in the View Pool can be used in a cost-effective manner to answer a new query, in comparison to running the same query against the detailed records in the data warehouse. This is achieved by probing the query-optimizer and getting an estimate of the execution cost of the query at the data warehouse. Whenever a new query result is computed, DynaMat uses an admission/replacement strategy that exploits spatio-temporal locality in the user access pattern, but also takes into account the computational dependencies of the stored query results.

Periodically, updates received from the data sources get shipped to the data warehouse and the View Pool gets refreshed. During updates, DynaMat switches to an "off-line" mode during which queries are not permitted. The maximum length of the update phase is specified by the administrator. Different update policies are implemented, depending on the types of updates, the properties of the data sources and the aggregate functions that are computed by the query results. From DynaMat's point of view the goal is to select and update the most useful fragments within the update time constraint. Notice that this is not equivalent to updating as many fragments as possible, although often both yield similar results.

5 WebView Materialization

WebViews are HTML fragments that are automatically created from based data, which are typically stored in a DBMS. For example, a search at an online bookstore for books by a particular author returns a WebView that is generated dynamically; a query on a cinema server generates a WebView that lists the current playing times for a particular movie; a request for the current sports scores at a newspaper site returns a WebView which is generated on the fly. Except for generating web pages as a result of a specific query, WebViews can also be used to produce multiple versions (*views*) of the same data (for example, translating the contents of a web page in multiple languages), and to support *multiple web devices*, especially browsers with limited display or bandwidth capabilities, such as cellular phones or networked PDAs.

Although there are a few web servers that support arbitrary queries on their base data, most web applications "publish" a relatively small set of *predefined* or *parameterized* WebViews, which are to be generated automatically through DBMS queries. A weather web server, for example, would most probably report current weather information and forecast for an area based on a ZIP code, or a city/state combination. Given that weather web pages can be very popular and that the update rate for weather information is not high, materializing such WebViews would most likely improve performance.

Personalized WebViews [6] can also be considered for materialization, if first they are decomposed into a *hierarchy* of WebViews. Take for example a personalized newspaper. It can have a selection of news categories (only metro, international news), a localized weather forecast (for Washington, DC) and a horoscope page (for Scorpio). Although this particular combination might be unique or unpopular, if we decompose the page into four WebViews, one for metro news, one for international news, one for the weather and one for the horoscope, then these WebViews can be accessed frequently enough to merit materialization.

5.1 WebView Materialization Policies

We explore three materialization policies: virtual, materialized inside the DBMS and materialized at the web server. In the *virtual* policy, everything is computed on the fly. To produce a WebView we must query the DBMS and format the results in HTML. Since no views are cached, we only need to update the base tables, whenever there is an update.

In the *materialized inside the DBMS* policy, we save the results of the SQL query that is used to generate the WebView. To produce the WebView, we must access the stored results and format them in HTML. The main difference of WebView materialization from web caching is that, in the materialization case, the stored query results need to be kept up to date all the time. This leads to an immediate refresh of the materialized views inside the DBMS with every update to the base tables they are derived from.

Finally, in the *materialized at the web server* policy, in order to satisfy user requests we simply have to read the WebView from the disk, where a fresh version is expected to be stored. This means that on every update to one of the base tables that produce the WebView, we have to refresh the WebView (or recompute it, if it cannot be incrementally refreshed) and save it as a file for the web server to read.

5.2 The selection problem

The choice of materialization policy for each WebView has a big impact on the overall performance. For example, a WebView that is costly to compute and has very few updates, should be materialized to speed up access requests. On the other hand, a WebView that can be computed fast and has much more updates than accesses, should not be materialized, since materialization would mean more work than necessary. We define the WebView selection problem as follows:

> *For every WebView at the server, select the materialization strategy (virtual, materialized inside the DBMS, materialized at the web server), which* **minimizes the average query response time** *on the clients.*

We assume that there is no storage constraint, since storage means disk space (not main memory), and also WebViews are expected to be relatively small. The decision whether to materialize a WebView or not, is similar to the problem of selecting which views to materialize in a data warehouse [11, 13, 28], known as the *view selection problem*. There are two crucial differences. First of all, the multi-tiered architecture of typical database-backed web servers raises the question of *where* to materialize a WebView. Secondly, updates are performed *online* at web servers, as opposed to data warehouses which are usually off-line during updates.

5.3 Experiments

In [20] we considered the full spectrum of materialization choices for WebViews in a database-backed web server. We compared them analytically using a detailed cost model that accounts for both the inherent parallelism in multitasking systems and also for the fact that updates on the base data are to be done concurrently with the accesses. We have implemented all flavors of WebView materialization and run extensive experiments on an industrial strength prototype, based on the Apache web server and Informix, running on a SUN UltraSparc-5 with 320MB of memory and a 3.6GB hard disk. We used a pool of 22 SUN Ultra-1 workstations as clients. Due to space constraints we only present two of our experiments.

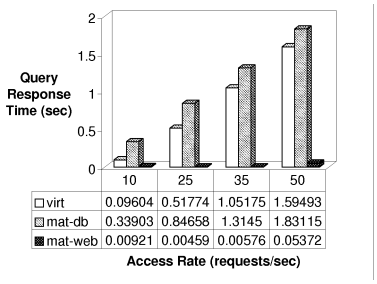


Fig. 3. Scaling up the access rate

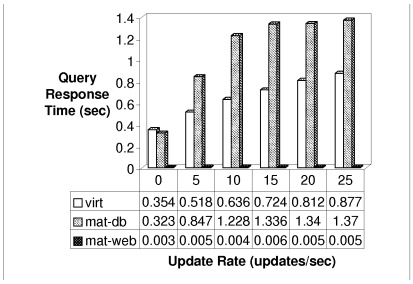


Fig. 4. Scaling up the update rate

In the first set of experiments, we varied the incoming access request rate (10-50 requests/sec) and measured the average query response time under the three different materialization policies: virtual (`virt`), materialized inside the DBMS (`mat-db`) and materialized at the web server (`mat-web`). A load of 50 requests/sec corresponds to a rather "heavy" web server load of 4.3 million hits per day for dynamically generated pages. The incoming update rate was 5 updates/sec for all experiments.

In Figure 3 we report the average query response time per WebView as they were measured at the web server. We immediately notice that the `mat-web` policy has average query response times that are consistently at least an order of magnitude (10 - 230 times) less than those of the `virt` or `mat-db` policies. Under the `mat-web` policy, servicing a request involves simply reading a file from disk, whereas, under the `virt` and `mat-db` policies, the system needs to compute a query at the DBMS for every request. Furthermore, since the web processes are "lighter" than the DBMS processes, the `mat-web` policy scales better than the other two.

In the second set of experiments, we varied the incoming update rate from 0 to 25 updates/sec, while the access request rate was set at 25 accesses/sec. In Figure 4 we plot the average query response times for this experiment under the three materialization policies. Our first observation is that the average query response time remains practically unchanged for the `mat-web` policy despite the updates, because the updates are performed in the background. The second observation is that the `virt` policy is performing 56% - 93% better than the `mat-db` policy in the presence of updates. This is explained by the fact that updates under the `mat-db` policy lead to extra work at the DBMS in order for the materialized views to be kept up to date.

6 Measuring the Quality of Web Servers

Caching of static web pages [8] is known to improve the *Quality of Service* (QoS) for user requests, since it improves the average query response time. For dynamic content however, web caching does not provide any freshness guarantees on the cached data. Servicing user requests fast is of paramount importance only if the data is fresh and correct, otherwise it may be more harmful than slow or even no data service. In general, when measuring the Quality of a system that uses materialized views, we need to evaluate both QoS and the *Quality of Data* (QoD), or how "fresh" the served data are. Web caching improves QoS dramatically, but completely ignores QoD of the cached data. On the other hand, when QoD is very important, web servers rely

on computing frequently changing web data on-demand . This achieves near-perfect QoD, but seriously impedes performance or leads to server melt-downs. WebView Materialization [20, 36] aims at bridging this gap, since it prolongs the QoS benefits of caching using amortization and incremental update algorithms on the cached data. This improves the QoD at a small degradation in QoS. In our work we try to provide the best trade-off between QoS and QoD based on the user/application requirements and the incoming workload.

7 Conclusions

In this paper we concentrated on the most important feature of the relational database model: *materialized views*. We focused on their usage on data warehousing and Web servers, with emphasis on updateability, performance, and scalability.

Specifically, we presented the Cubetree Data Model, a storage abstraction for the data cube. Cubetrees maintain the view records internally sorted (using packed R-trees) and allow bulk incremental updates through an efficient merge-packing algorithm. We briefly described DynaMat, a dynamic view management system that manages collections of materialized aggregate views based on user workload and available system resources (disk space, update cost). Finally, we explored the materialization policies for WebViews, presented experimental results from an industrial-strength prototype and discussed the Quality of Service and Quality of Data considerations for WebViews.

References

1. S. Agrawal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Sarawagi. On the Computation of Multidimensional Aggregates. In *Proc. of 22nd VLDB conference*, pages 506–521, Bombay, India, August 1996.
2. M.M. Astrahan et al. System R: Relational Approach to Database Management. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976.
3. Lars Baekgraard and Nick Roussopoulos. "Efficient Refreshment of Data Warehouse Views". Technical Report CS-TR-3642, Dept. of Computer Science, Univ of Maryland, May 1996.
4. E. Baralis, S. Paraboschi, and E. Teniente. Materialized View Selection in a Multidimensional Database. In *Proc. of the 23th VLDB Conference*, Athens, Greece, August 1997.
5. R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1(3):173–189, 1972.
6. Phil Bernstein et al. "The Asilomar Report on Database Research". *SIGMOD Record*, Dec. 1998.
7. José A. Blakeley, Per Åke Larson, and Frank Wm. Tompa. "Efficiently Updating Materialized Views". In *Proc. of the ACM SIGMOD Conference*, pages 61–71, Washington, DC, May 1986.
8. Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. "Web Caching and Zipf-like Distributions: Evidence and Implications". In *Proc. of INFOCOM'99*, New York, USA, March 1999.
9. 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.
10. J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. 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.
11. Ashish Gupta and Inderpal Singh Mumick. "Maintenance of Materialized Views: Problems, Techniques, and Applications". *Data Engineering Bulletin*, 18(2):3–18, June 1995.

12. H. Gupta, V. Harinarayan, A. Rajaraman, and J. Ullman. Index Selection for OLAP. In *Proceedings of ICDE*, pages 208–219, Burmingham, UK, April 1997.
13. Himanshu Gupta. "Selection of Views to Materialize in a Data Warehouse". In *Proc. of the 6th International Conference on Database Theory (ICDT '97)*, Delphi, Greece, January 1997.
14. A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, Boston, MA, June 1984.
15. V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing Data Cubes Efficiently. In *Proc. of ACM SIGMOD*, pages 205–216, Montreal, Canada, June 1996.
16. 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.
17. R. Kimball. *The Data Warehouse Toolkit*. John Wiley & Sons, 1996.
18. 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.
19. 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.
20. Alexandros Labrinidis and Nick Roussopoulos. "WebView Materialization". In *Proc. of the ACM SIGMOD Conference*, Dallas, Texas, USA, May 2000.
21. I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of Data Cubes and Summary Tables in a Warehouse. In *Proc. of the ACM SIGMOD Conference*, Tucson, Arizona, May 1997.
22. 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.
23. 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.
24. 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.
25. Nick Roussopoulos. "The Logical Access Path Schema of a Database". *IEEE Transactions on Software Engineering*, 8(6):563–573, November 1982.
26. Nick Roussopoulos. "View Indexing in Relational Databases". *ACM Transactions on Database Systems*, 7(2):258–290, June 1982.
27. Nick Roussopoulos. "An Incremental Access Method for ViewCache: Concept, Algorithms, and Cost Analysis". *ACM Transactions on Database Systems*, 16(3), September 1991.
28. Nick Roussopoulos. "Materialized Views and Data Warehouses". *SIGMOD Record*, March 1998.
29. Nick Roussopoulos and Hyunchul Kang. "Principles and Techniques in the Design of ADMS ±". *Computer*, pages 19–25, December 1986.
30. A. Segev and J. Park. Maintaining Materialized Views in Distributed Databases. In *Proceedings of the 5th International Conference on Data Engineering*, Los Angeles, CA, 1989.
31. Timos Sellis. "Intelligent caching and indexing techniques for relational database systems". *Information Systems*, 13(2), 1988.
32. Michael Stonebraker. "Implementation of Integrity Constraints and Views by Query Modification". In *Proc. of the ACM SIGMOD Conference*, San Jose, California, May 1975.
33. D. Theodoratos and T. Sellis. Data Warehouse Configuration. In *Proc. of the 23th International Conference on VLDB*, pages 126–135, Athens, Greece, August 1997.
34. F. Tompa and J. Blakeley. Maintaining Materialized Views Without Accessing Base Data. *Information Systems*, 13(4):393–406, 1988.
35. Patrick Valduriez. "Join Indices". *ACM Transactions on Database Systems*, 12(2), June 1987.
36. Khaled Yagoub, Daniela Florescu, Valerie Issarny, and Patrick Valduriez. "Caching Strategies for Data-Intensive Web Sites". In *Proc. of the 26th VLDB Conference*, Cairo, Egypt, Sep 2000.