Materialized Views in Oracle.

Conference Paper · January 1998

Source: DBLP

9 authors, including:

Karl Dias
Oracle Corporation
19 PUBLICATIONS **346** CITATIONS

SEE PROFILE

Andrew Witkowski
Oracle Corporation
23 PUBLICATIONS **263** CITATIONS

SEE PROFILE

Materialized Views In Oracle

Randall G. Bello, Karl Dias, Alan Downing, James Feenan, Jim Finnerty, William D. Norcott, Harry Sun, Andrew Witkowski, Mohamed Ziauddin

Oracle Corporation,
400 Oracle Parkway,
Redwood Shores, CA 94065

{rbello, kdias, adowning, jfeenan, jfinnert, wnorcott, hasun, awitkows, mziauddi}@us.oracle.com

Abstract

Oracle Materialized Views (MVs) are designed for data warehousing and replication. For data warehousing, MVs based on inner/outer equi-joins with optional aggregation, can be refreshed on transaction boundaries, on demand, or periodically. Refreshes are optimized for bulk loads and can use a multi-MV scheduler. MVs based on subqueries on remote tables support bi-directional replication. Optimization with MVs includes transparent query rewrite based on cost-based selection method. The ability to rewrite a large class of queries based on a small set of MVs is supported by using Dimensions (new Oracle object), losslessness of joins, functional dependency, column equivalence, join derivability, joinback and aggregate rollup.

1 Introduction

Oracle first introduced support for deferred incremental maintenance of single-table select-project snapshots in 1992. Recently, this capability has been expanded to include support for more classes of incrementally maintained Materialized Views (MV), query optimization, dependency management, bulk-mode refresh, and transaction-consistent refresh. Oracle 8.1 will support three classes of incrementally maintained MVs: a Materialized Join View (MJV), which is a materialization of a query with inner and outer equi-joins, a Materialized Aggregate View (MAV), which is an MJV with aggregation, and a Materialized Subquery View (MSV), which materializes EXISTS subqueries. In addition, Oracle allows creation of any other MV as defined by an arbitrary complex query; however, in this case only full refresh mode (i.e., complete recomputation) is supported. All MVs are also available for query optimization, where part of the query is replaced, transparently to the user, by pre-computed MV(s). Oracle MVs are part of an integrated solution for data warehousing that includes query rewrite, dimension support, and MV advisory functions.

Oracle MVs address such diverse areas as OLTP replication, data warehousing, distributed databases, and mobile disconnected clients. The flexibility required to support this areas is provided by timing, type, location, and rewrite attributes. The timing attributes are used to perform deferred MV maintenance on demand, on a transaction boundary, or on a periodic basis. The refresh type attribute specifies whether to recompute an MV from scratch or to incrementally refresh it considering only changes to the master tables since the last refresh. The location attribute specifies whether to maintain an MV on a local or remote site with respect to the master tables. In a data warehousing environment where update transactions are mostly bulk loads, efficient bulk incremental refresh methods are provided. MVs with remote tables may be refreshed periodically to support replication from an OLTP environment, or may be refreshed on demand to support mobile disconnected clients. The rewrite attribute determines whether the MV will participate in query optimization, where part of the query is replaced with the MVs pre-computed results.

Oracle stores each MV in a regular relational table. This enhances MV's flexibility because users can directly query them, put indexes on them for performance, partition them to improved scalability and maintainability, reorganize the table, etc. To support MVs created manually by users, an existing table can be registered as an MV for incremental maintenance, query rewrite, and dependency management. This is useful for data warehousing applications, which have pre-existing, manually maintained summary tables

Proceedings of the 24th VLDB Conference New York, USA, 1998

that are used for manual rewrite of queries and for off-line instantiation of large, remote MVs.

Query optimization with MVs is targeted towards OLAP, multi-dimensional analysis, and data warehousing domains where queries aggregate along complex dimensional and hierarchical relationships. The hierarchical and functional dependency relationships that are prevalent in these domains are captured by referential integrity (RI) constraints, primary key (PK) constraints, and by a new Dimension construct (D) that Oracle provides. All three relationships (RI, PK, and D) are used to rewrite a query using MVs.

Query optimization is provided for both MJVs and MAVs and applies to queries where joins have inner-, semi-, anti-, and left-outer-join semantics. Additionally, with MAVs, opportunities to rewrite using aggregation along hierarchical relationships are fully exploited. To avoid degrading query performance, a cost-based method is used to determine if a query should be rewritten at all.

2 Refresh algorithm for MJV

The deltas for the incremental maintenance of MVs are obtained from two sources: row-DML logs and direct-loader logs. Each log is associated with a table that has one or more MVs defined on it. Row-DML logs record changes made to the individual rows of a table via DML. The log can be specified to store images of a set of columns of a changed row, a vector indicating all its changed columns, the rowid, the type of DML (insert/ update/delete), and a timestamp. The log is designed to be shared by many MVs that can be refreshed independently. A row's timestamp is used when an MV is refreshed to determine whether the row needs to be applied to the MV. The timestamp is also used to purge the logs of entries that are no longer needed. MV logs are suitable for OLTP applications even though they tax each modified row with a small overhead.

In Oracle, a direct load appends new data as a physically contiguous range of rows; consequently, the new rows can be compactly logged as a contiguous range of rowids. This approach enables Oracle to efficiently determine the pre-update state of the table as well as the new rows. The direct loader's low-overhead approach to logging is required for data warehousing where loads are massive and frequent. The challenge in incremental MV maintenance is to correctly refresh MVs from both row-DML logs and direct-loader logs as they interact. For example, rows that are updated after they have been loaded may appear in both logs. In addition, a refresh is optimized for the more common case of one of the logs being empty.

Oracle supports deferred incremental maintenance of MJVs using a memoryless refresh by applying the entire set of changes to a table at one time in bulk operations, without considering the order in which the changes actually occurred. We illustrate the refresh algorithms using examples that consider MVs with two tables; the algorithms are easily generalized to any number of tables. Consider two tables R and S and assume that the materialized view M consists of an equijoin between R and S (i.e., $M = R><S$.) Let $R' = R + \Delta R$ and $S' = S + \Delta S$ denote the new, after update, versions of R and S respectively. In the equations below "+" and "-" have the same semantics as that of set union and difference respectively. Note that they are not commutative and they evaluate from left to right. If ΔM represents the changes to be applied to M, then it is easy to show that

$$\text{Q1} \quad \Delta M = R >< \Delta S + \Delta R >< S + \Delta R >< \Delta S$$
$$= \Delta R >< S' + R >< \Delta S$$

If the row-DML logs are not empty, recovery of the pre-update states (i.e., R and S) can be expensive to compute. Therefore the previous equation is expressed using only the deltas and the post-update states, R' and S':

$$\text{Q2} \quad \Delta M = R' >< \Delta S - \Delta R >< \Delta S + \Delta R >< S'$$

For an outer join between R and S, $R \rightarrow S$, this equation changes slightly to $\Delta M = R' >< \Delta S - \Delta S >< \Delta R + \Delta R \rightarrow S'$.

Q2 is the basis for our incremental refresh of MJVs. The first term in Q2, $R' >< \Delta S$, is responsible for changes caused by ΔS. ΔS is further divided into three sets: $\Delta S = \{D\} + \{I\} + \{U\}$, where $\{D\}$, $\{I\}$, $\{U\}$ is the set of rows deleted, inserted and updated respectively. In many situations, the algorithm represents an update as a delete followed by an insert and includes it in $\{D\}$ and $\{I\}$ respectively. When this occurs, the memoryless refresh algorithm operates in two phases, which use Oracle's parallel DML when possible:

1. Delete Phase. Delete all rows in M whose rows have their $S.rowid$ in $\{D\}$ of ΔS. For an outer join, $R \rightarrow S$, we set columns of M that reference S to null (instead of deleting them).
2. Insert Phase. Insert the result of $R' >< \Delta S$ into M. This set will compute the effect of newly inserted rows as well as updated rows. The rows considered are those in $\{I\}$. For an outer join, $R \rightarrow S$, set columns of M that reference S to their values from S (instead of inserting these rows).

After applying ΔS, the memoryless algorithm to M is applied to table R. In the delete phase for this step, all the rows that were inserted as a result of $\Delta R >< \Delta S$ in the insert phase for table S would be deleted. This prevents the result from being counted twice. The delete phase accounts for the undo term, $\Delta R >< \Delta S$, in Q2. The insert phase will insert all rows resulting from $\Delta R >< S'$.

3 Basic Refresh Algorithms for MAVs

When only bulk inserts occur, then both the pre-update state and the delta rows of all tables can be efficiently recovered; therefore, equation Q1 applies, and ΔM can be computed using at most one term for each detail table. If there are referential integrity constraints or dimensional relationships (see Section 5) defined on the tables, then some terms are guaranteed to be empty, and are not computed explicitly.

Without reading the logs, Oracle can detect whether a log can be ignored, allowing simpler or fewer refresh operations. Note that each log is related to a master table and all MVs on that master table refer to the table's log. Thus, Oracle can determine whether a table has been modified since the last refresh of an MV by comparing the commit time of the last update to the log (or master table) with the last refresh time recorded for the MV.

For MAVs that join multiple tables, a common case in data warehousing, each term in Q1 must be aggregated into a *delta summary* before it can be merged with ΔM. Each delta summary is merged, in turn, with the contents of the MAV.

Because data warehousing typically requires that many MAVs be refreshed within a fixed refresh window, the Oracle RDBMS performs global optimizations that minimize the overall refresh time of a set of MAVs. It supports MAV-based refresh, which considers the relationships between MAVs and schedules them such that an MAV may be refreshed via parallel DML using the contents of another MAV rather than from master tables. This optimization improves refresh performance considerably by eliminating the cost of joining and aggregating over master tables. In addition, Oracle employs load-balancing scheduling algorithms that allow concurrent refresh of multiple MAVs. Moreover, Oracle supports a "refresh_dependent" functionality that refreshes only the MVs that require refresh after changes to one or more of their master tables.

The need to refresh a plurality of multi-table MAVs within a fixed refresh window, in a production environment requires recoverability features in addition to performance optimizations. In this case, rather than a single long-running transaction, there is a series of checkpointed transactions. Therefore, during a refresh of a large number of MAVs, those that completed do not need to be restarted in the event of system failure. Only the MAVs that did not complete successfully will need to be re-executed, a property that improves recovery time in the event of system failure.

In the special case when a MAV contains a single master table (and no joins), and when it does not contain a MIN or MAX function, Oracle can incrementally maintain the MAV in the presence of both direct loads and row DML operations. The refresh of a single-table MAV is performed using one of two techniques: *self-maintenance* or the *memoryless refresh* algorithm. Self-maintenance uses the row-DML logs to update the MAV without referencing the master tables, but it cannot be used unless the direct load log is empty. Because the size of the aggregate view in most cases is orders of magnitudes less than that of the master table, fewer rows than log entries need to be modified in the MAV. It is necessary to log the "before" and "after" values of the columns being aggregated in the row-DML logs to use this scheme. The memoryless algorithm is always safe to use. It consists of an insert and a delete phase for the table in the MAV similar to that mentioned in Section 2.

4 Materialized Views with Subqueries

Oracle 8.0 supports materialized subquery views where each join between tables is expressed by a correlated EXISTS subquery. For incremental maintenance, the join inside each level of the subquery is required to be based on a unique key of the table at that level. Predicates that are functions of the table in each level of the subquery are allowed as conjunctions. The refresh of an MSV is analogous to MJV refresh because the correlated subquery is converted to a join internally.

A distinguishing characteristic of MSVs and of Oracle 7's single-table select-project MVs is bi-directional replication. Oracle allows the master tables for the MSV as well as the MSV itself to be updated. The updates to the MSV are then incorporated back into the master tables.

A typical use of MSVs is mobile sales force automation, where we use them for bi-directional replication between a high-end database at the corporate repository (master site) and thousands of low-end databases on laptops (remote sites). The salesmen at remote sites receive replicas of their portion of the master via MSVs. They update their MSVs while disconnected from the master site. The master site performs its own updates, which most likely are reconciliations of MSVs from other salesmen. Updates to MSVs are propagated either synchronously or asynchronously to the master, where conflicting updates are resolved using Oracle's symmetric replication [DDDEHJJLSSS94] [S95]. Furthermore, refresh groups allow multiple MVs to be consistently refreshed in a single transaction so that referential integrity relationships are maintained among multiple MVs.

5 Dimensions

Oracle 8, release 8.1, introduces the concept of a dimension that captures hierarchical (1:n parent-child) and

attribute (1:1 functional dependency) relationships in the database schema. A dimension may be thought of as a directed graph with each edge representing a hierarchical relationship and each node representing a level of aggregation. A hierarchy is a path through this graph.

For example, a simple *Time* dimension may contain two hierarchies: *date→month→quarter→year* and *date→week*, with arcs drawn from the child level to the parent level. Each arc in this graph has the property that a given value of the child is associated with exactly one value of the parent. For example, each month must be contained in exactly one quarter; consequently, a sum of sales by month can be rolled up to a sum of sales by quarter. A dimension may also contain attribute relationships between a hierarchy level and its functionally dependent columns. For example, if a time dimension table also contains a *monthName* column, then the attribute relationship would be *month→monthName*. The attribute relationships enable the Oracle optimizer to determine when an MV can be used to satisfy a query that references the dependent attribute columns that are not present in the MV. Note that the hierarchical and attribute relationships both represent functional dependencies

Dimensions can be defined using normalized and denormalized dimension tables. If the columns of a parent level and child level are in different relations, then the arc between them specifies a 1:n join relationship that can be enforced by an RI constraint. Hierarchical relationship and some of the attribute relationships in a denormalized table cannot be represented using RI and PK constraints. Specifying them in a dimension enables the Oracle optimizer to greatly expand the class of queries that can be rewritten.

6 Query Rewrite Concepts

The Oracle optimizer utilizes information about lossless joins, functional dependencies, column equivalence, and join derivability to rewrite a large class of queries with a small set of MVs. Let R><S (R->S) denote an inner (left outer) join between relations R and S. A join R><S is lossless if it preserves all tuples of R. RI, PK, and D constraints are used to discover inner joins that are lossless. A left outer join R->S naturally preserves all tuples of R so it is lossless. Observe that the concept of losslessness is asymmetric. Based on losslessness, Oracle optimizer rewrites a query even if an MV contains non-overlapping joins which is a powerful capability. For example, in and a multi-dimensional star schema an MAV may store n-dimensional aggregates using joins to n dimension tables. A query requesting k-dimensional aggregates (k<n) can be rewritten using this MAV provided non overlapping joins are lossless.

In addition to potentially eliminating tuples in R, another effect of non-overlapping join is the duplication of R tuples unless each R tuple joins with at most one S tuple. Such duplication effect can be compensated by using DISTINCT clause on MJV, or scaling down the aggregates in an MAV by using scale factors (duplicate counts of join key in S). The scale factor can be either precomputed in MAV or computed on-the-fly by joining MAV back to S.

The hierarchical and attribute relationships stored in a dimension represent the functional dependencies between column data. The functional dependency is also inferred from PK constraints wherein a primary key functionally determines every other column in a table. Functional dependency information is used in determining valid aggregate rollups and valid joinbacks. For example, if *city→state* then it is valid to rollup sum of sales by city to sum of sales by state. If city is a primary key in *cities* table then *city→cityName*. If city is stored in an MV but not *cityName*, and a if query references *cityName* then it is valid to rewrite the query using a joinback from MV to *cities* table. The column equivalences based on equi-joins are utilized to determine if R.x in an MV is equivalent to S.x in a query to avoid unnecessary joinbacks.

Join derivability allows us to recompute a join in a query from a join in an MV. With an left outer join in MV it is possible to recompute inner join in a query by filtering anti-join rows, recompute semi-join by eliminating duplicate rows, and recompute anti-join by filtering theinner join rows. With inner join in an MV it is possible to recompute semi-join in a query by eliminating duplicate rows. The join derivability support of Oracle optimizer allows queries with IN and EXISTS subqueries (semi-joins) to be rewritten using MVs with inner or left outer joins, and queries with NOT IN and NOT EXISTS subqueries (anti-joins) to be rewritten using MVs with left outer joins.

7 General Rewrite Algorithm

Oracle performs query rewrite by comparing the join graphs of a query block (QB) and a candidate MV. The two graphs should intersect but non-overlapping subgraph is allowed in QB, in MV, or both. The algorithm is recursively applied to each QB of a query, and is attempted both before and after view flattening and subquery transformation. Subquery transformations include the conversion of IN or EXISTS subqueries to semi-joins, and NOT IN and NOT EXISTS subqueries into anti-joins, which enable highly complex queries that are common in OLAP and multi-dimensional analysis to be rewritten. Applying rewrite before view flattening allows MVs that are defined using views to be used. Simple view name matching between a QB and an MV is used which enables

the use of arbitrary complexity underneath the views.

The algorithm is divided into two phases: eligibility and transformation. The eligibility phase determines whether rewrite is possible, determines how to join an MV to non-overlapping relations in a QB, and determines what additional join or filtering conditions are required, if any, upon rewrite. If aggregation is present in a QB, the eligibility algorithm also determines whether the QB aggregates are computable from MAV aggregates.

The transformation phase replaces overlapping relations of a QB with an MV, and synthesizes additional joins and selection predicates as necessary to recover QB from MV. If aggregation is present in a QB, additional transformations may be required to compute the aggregated outputs of QB from the aggregated outputs of MAV.

The following eligibility checks are performed before a QB is rewritten:

1) Join Compatibility Check: The join graph G(M) of an MV is compared with the join graph G(Q) of a QB and three join subgraphs are identified. The *intersection subgraph* G(I) represents the overlapped region between G(M) and G(Q), so G(I) = G(M) $\cap$ G(Q), the *delta subgraph* $\Delta G(Q)$ represents the part of G(Q) that is not in G(I), so ΔG(Q) = G(Q) - G(I), and the *delta subgraph* $\Delta G(M)$ represents the part of G(M) that is not in G(I), so ΔG(M) = G(M) - G(I). G(Q) can be recovered by joining ΔG(Q) to MV when all the joins in ΔG(M) are lossless and the joins in G(I) are of the same type between MV and QB. A transformation is needed if some joins in G(I) are not of the same type but are compatible. For example, if G(M) = S←L→O and G(G) = L><O><C, then ΔG(M) = S, G(I) = L→O, ΔG(Q) = C, and if S←L is lossless then QB can be rewritten as MV><C with a filter added to exclude the anti-join rows of L→O. If MV is an MJV that contains rowid or primary keys of O, the filter "O.rowid is not null" or "O.pk is not null" is added to the rewritten QB.

2) Data Sufficiency Check: All columns of matching relations in QB other than the join and aggregate columns should be either equal to or functionally determined by columns in MV. For example, if QB contains reference to column *cityName* of relation *geography* that is functionally determined by the *cityId* column in MV then *cityName* can be recovered by joining MV to *geography* using *cityId*. If join key *cityId* in geography is not known to be unique then MV is joined to a derived table that selects distinct *cityId* values along with other needed columns from geography. Column equivalence based on equi-joins is used in avoiding redundant joinbacks. If intersecting relations *R* and *S* are equi-joined on *R.x=S.x*, and MV selects column *R.x* while QB references *S.x*, then *R.x* is substituted for *S.x* during rewrite.

3) Grouping Compatibility Check: If QB contains a GROUP BY clause then each grouping column of QB should match exactly with or be functionally dependent on a grouping column of a candidate MAV. Conversely, if the MAV groups by some columns which neither match nor functionally dependent, then aggregates in the MAV should be re-aggregated, i.e., rolled up when the QB is rewritten. Similarly, if QB grouping is found compatible based on functional dependency, then aggregates in the MAV should be rolled up. For example, if QB requests SUM(sales) by year, and the candidate MAV contains SUM(sales) by month, and further if it is known that month→year, then QB can be rewritten by rolling up SUM(sales) in MAV from the month to the year level.

4) Aggregate Computability Check: If a QB contains aggregates, then each aggregate in the QB must be computable from one or more aggregates in a candidate MAV. For example, SUM(x) in a QB is computable from COUNT(x) and AVG(x) in a MAV. If roll up of aggregates stored in the MAV is necessary then, certain types of aggregates require other auxiliary aggregates to be available. For example, AVG(x) can be rolled up only if COUNT(x) is also present. Aggregates with expressions are also supported. For example, SUM(a+b) in a QB is matched with either SUM(a+b) or SUM(b+a) in a MAV, and SUM(a) + SUM(b) in a QB is matched with SUM(a) and SUM(b) in MAV. Oracle supports rewrite of COUNT, COUNT(*), COUNT(DISTINCT), SUM, MIN, MAX, AVG, VARIANCE, and STDDEV (standard deviation) aggregates.

8 Heuristic and Cost Based Rewrite

An MV is defined on a set of relations, and there must be some intersection of this set with the set of relations in a QB for the MV to be a candidate for rewrite. In the case of a QB with aggregates, candidate MAVs are further restricted to contain all relations referenced in the aggregates of the QB. To identify a set of candidate MVs for a QB, a list of MVs is maintained for each relation. This list for relation R contains all the MVs that reference R as their master table. Using the intersection or union of MV lists that are attached to relations in a QB, candidate MVs are quickly identified.

If a QB contains aggregates, then rewrite is attempted first using an MAV. Whether or not the QB is rewritten, if joins still remain, rewrite is attempted using MJVs or MAVs. Rewrite is repeated as long as joins in the QB remain or no eligible MV is found. Because aggregation shrinks the data size, rewrite using a MAV is always tried first to obtain this benefit up front.

When attempting to a QB, it is possible to find more than one eligible MV. When this occurs, a heuristic called *query reduction factor* is used to identify the best choice in

a list of eligible MVs. The query reduction factor is the ratio of the sum of the cardinalities of matching relations in a QB to the cardinality of the MV. This metric is further refined when it is determined that the MV needs to be joined back to some matching relations in the QB to account for the reduction in benefit due to joinback.

After the entire query is rewritten using one or more MVs, it is optimized and its optimal cost is found. Because MVs in Oracle are maintained as normal tables with their own indexes and partitioning, the optimization will automatically include such table attributes, which will often cause further refinements. Next, the original version of the query is optimized and its optimal cost is found. The rewritten query is discarded if its optimal cost is found to be greater than the optimal cost of the original query.

9 Related Work

Many of the concepts discussed in this paper have been examined before. For example, incrementally maintained MVs have been in the literature for years including simple snapshots [LHMPW86], join indexes [BM90], bulk insert optimizations [MQM97], and aggregates [GM95] [GMS93]. Scheduling of multiple MVs has also been investigated [CM96]. Similarly, query rewrite using MVs has been extensively researched. Query rewrite using conjunctive MVs without grouping and aggregation is shown in [CKPS95] [LMSS95]. Rewrite based on syntactic transformation of a query where a subset of it matches with an MV is described in [GHQ95]. In [SDJL96] rewrite based on MVs with grouping and aggregation is shown but no meta information (functional dependency, constraints) is used. [CCHJJMSW98] describes rewrite that utilizes declared hierarchy rollup paths.

Oracle has focused on a practical subset of the problem space that is believed to be of most use to its customers. While most of the literature has concentrated upon immediate-mode maintenance, Oracle's algorithms are based on deferred-mode maintenance. Also our algorithms include checkpointing of multiple-refreshes, a feature that enhances reliability. Query rewrite in Oracle utilizes as much meta information as possible including the hierarchical and attribute relationships declared in a new Oracle object called Dimension.

References

[BM90] J. A. Blakeley, N. L. Martin. Join Index, Materialized View, and Hybrid Hash-Join: A Performance Analysis *Proc. IEEE Int'l. Conf. on Data Eng.* Los Angeles, CA February 1990.

[CCHJJMSW98] L.S. Colby, R.L. Cole, E. Haslam, N. Jazayeri, G. Johnson, W.J. McKenna, L. Schumacher, D. Wilhite. Red Brick Vista: Aggregate Computation and Management. *Proc. of the 14th Int'l. Conf. on Data Eng.*, Orlando, FL, 1998.

[CKPS95] S. Chaudhuri, R. Krishnamurthy, Spyros Potamianos, K. Shim. Optimizing Queries with Materialized Views. *Proc. of Int'l. Conf. on Data Eng.*, 1995.

[CM96] L.S. Colby, I.S. Mumik, Staggered Maintenance of Multiple Views, *Proc. of the Workshop on materialized Views: Techniques and Applications,* Montreal, Canada, 1996

[DDDEHJJLSSS94] D. Daniels, L.B. Doo, A. Downing, C. Elsbernd, G. Hallmark, S. Jain, B. Jenkins, P. Lim, G. Smith, B. Souder, J. Stamos, Oracle's Symmetric Replication Technology and Implications for Application Design, in the *Proc. of ACM SIGMOD 1995, Int'l. Conf. on Mgmt. of Data,* Minneapolis, MN, 1994

[GHQ95] A. Gupta, V. Harinarayan, D. Quass. Aggregate-Query Processing in Data Warehousing Environments. *Proc. of the 21st VLDB Conf.*, Zurich, Switzerland, 1995

GMS93] A. Gupta, I.S. Mumick, V.S. Subrahmanian. Maintaining Views Incrementally. *Proc. of ACM SIGMOD 1993 Int'l. Conf. on Mgmt. of Data*, Washington, DC, 1993

[GM95] A. Gupta, I.S. Mumick, Maintenance of Materialized Views: Problems, Techniques, and Applications, *IEEE Data Eng. Bulletin, Special Issue on Materialized Views and Data Warehousing*, Vol 18, No. 2, 1995.

[LHMPW86] B. Lindsay, L. Haas, C. Mohan, H. Pirahesh, P. Wilms. A Snapshot Differential Refresh Algorithm. *Proc. of ACM SIGMOD 1995, Int'l. Conf. on Mgmt. of Data*, 1986.

[LMSS95] A.Y. Levy, A.O. Mendelzon, Y. Sagiv, D. Srivastava. Answering Queries using Views. *Proc. of the 14th Symposium on Principles of Database Systems (PODS)*, San Jose, CA, 1995

[MQM97] I.S. Mumick, D. Quass, B.S. Mumick. Maintenance of Data Cubes and Summary Tables in a Warehouse. *Proc. of ACM SIGMOD 1997, Int'l. Conf. on Mgmt. of Data,* 1997

[S95] G. Smith, "Oracle7 Symmetric Replication", Oracle White Paper, Part #A33128, Oracle Corporation, Redwood Shores, CA, 1995

[SDJL96] D. Srivastava, S. Dar, H.V. Jagadish, A.Y. Levy. Answering Queries with Aggregation Using Views. *Proc. of the 22nd VLDB Conf.*, Mumbai, India, 1996