

OLAP Query Optimization in the Presence of Materialized Views

Thomas P. Nadeau (nadeau@aladdinsoftware.com)
Toby J. Teorey (teorey@eecs.umich.edu)
University of Michigan
Ann Arbor, Michigan, USA 48109-2122

Abstract

Existing research on query optimization in the presence of materialized views focus primarily on syntactic methods of finding the possible query rewrites utilizing the views available. Once the possible rewrites are determined, what appears to be the optimal query rewrite is converted into a physical plan which is then executed. The syntactic approach to finding query rewrites can be fundamentally improved by utilizing metadata gathered in the course of OLAP operations. We describe a metadata table that tracks information on each materialized view. Each row of the view metadata table contains information on one specific view, and each column is used for a different dimension in the fact table. The value of each dimension attribute in a given row of the metadata table contains the level of aggregation for that dimension in the corresponding view. The query optimizer can run a SQL query on the metadata table to determine quickly what views can be used to answer the query, and which of these possible data sources will likely give the quickest response. The role of this query optimization approach in the context of an overall plan for OLAP view estimation, selection, and maintenance is demonstrated.

1 Introduction

Data warehouses are commonly organized with one large central fact table, and many smaller dimension tables. This configuration is termed a star schema. Figure 1 illustrates with an example. The fact table is composed of two types of attributes: dimension attributes and measures. The dimension attributes in Figure 1 are *CustID*, *DateID* and *BindID*. The dimension attributes have foreign-key/primary-key relationships with the dimension tables. Together, the dimension attributes compose the primary key of the fact table. The dimension attributes are typically used in query “group by” expressions, or to join the fact table with the dimension tables. Notice that a dimension can also have a hierarchy. For example, the Figure 1 schema allows time to be grouped by *DateID*, *Month*, *Quarter* or *Year*. The fact table also contains measure attributes, the values to be aggregated. The measure attributes in Figure 1 are *Cost* and *Sell*.

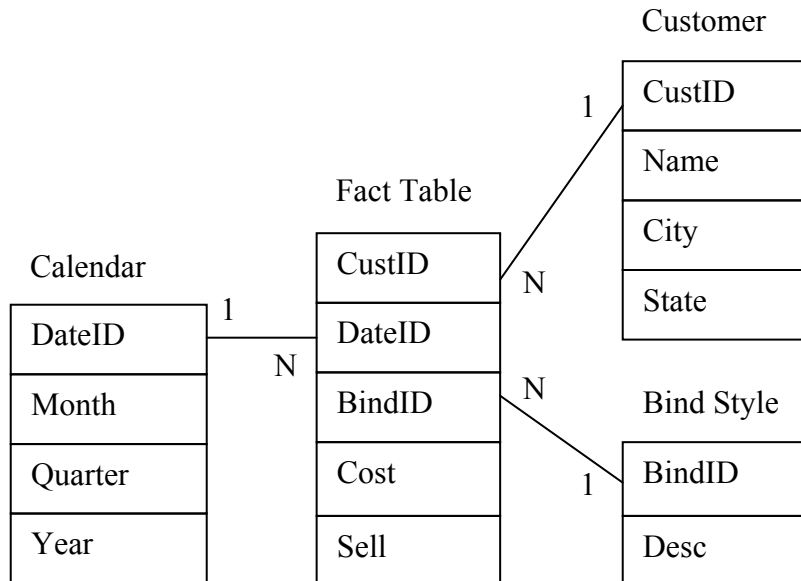


Figure 1 Example star schema for a data warehouse

A fact table in a data warehouse may contain many millions of rows, and the processing of a single query can require significant resources. To improve the quickness of response to queries, pre-aggregation is a useful OLAP strategy. Pre-aggregation requires the result to be saved to disk as materialized views, also known as automatic summary tables (ASTs). The number of possible views is exponential in the number of dimensions in the database. Faced with combinatorial explosion, limited disk space and limited update resources, an OLAP system must select a strategic set of beneficial views to materialize in order to achieve quick response to queries.

1.1 Top Level Architecture

The goal of OLAP is to give quick response to queries posed against a large repository of data. Figure 2 maps the approach taken here to the general problem of OLAP optimization. The larger problem of OLAP optimization is broken into four sub-problems: *View Size Estimation*, materialized *View Selection*, materialized *View Maintenance*, and *Query Optimization* with materialized views.

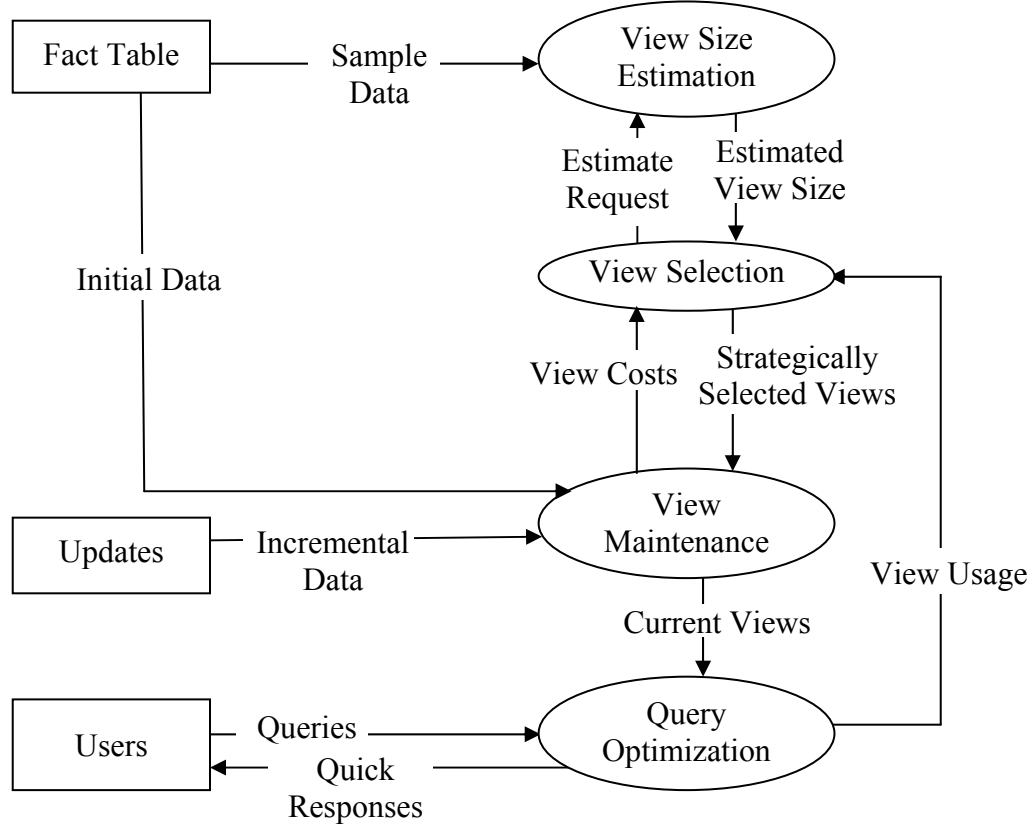


Figure 2 Architecture for OLAP optimization

The OLAP optimization process feeds *Sample Data* from the *Fact Table* into *View Size Estimation*. *View Selection* makes an *Estimate Request* for the view size of each new view it encounters while exploring promising regions of the search space. *View Size Estimation* queries the *Sample Data*, examines and models the distribution. The distribution observed in the sample is used to estimate the expected number of rows in the view for the full dataset. The *Estimated View Size* is passed to *View Selection*, which uses the estimates to evaluate the relative benefits of materializing the various views under consideration. *View Selection* picks *Strategically Selected Views* for materialization with the goal of minimizing total query costs. *View Maintenance* builds the original views from the *Initial Data* from the *Fact Table*, and maintains the views as *Incremental Data* arrives from *Updates*. *View Maintenance* sends statistics on *View Costs* back to *View Selection*, allowing costly views to be discarded dynamically, matching the update load. *View Maintenance* offers *Current Views* for use by *Query Optimization*. *Query Optimization* must consider which of the *Current Views* can be utilized to most efficiently answer *Queries* from *Users*, giving *Quick Responses* to the *Users*. *View Usage* feeds back into *View Selection*, allowing the system to be dynamic, adapting and improving over time, matching the query workload.

1.2 The Exponential Explosion of Views

The central issue challenging the design of OLAP systems is the exponential explosion of possible views as the number of dimensions increases. Let us begin by calculating the number of possible views for the example illustrated in Figure 1, and then we will establish a general equation. Figure 1 has three dimensions: *Calendar*, *Customer* and *BindStyle*. The *Calendar* dimension has four levels of hierarchy explicitly established in the *Calendar* table: *DateID*, *Month*, *Quarter* and *Year*. A user can choose to group data by any one of these levels. The user also has the option of grouping data across all time, which is implicitly a fifth hierarchical level. Thus the *Calendar* dimension has five levels of hierarchy; likewise, if the *Customer* table attributes *CustID*, *City* and *State* form a hierarchy (consider *Name* to be a descriptive field), then the *Customer* dimension has a four level hierarchy including the "All" level. The user may choose whether to group by *BindStyle* or not. Thus there are two choices along the *BindStyle* dimension. The choice of aggregation level along each dimension is orthogonal to the other dimensions. The number of possible views is the product of the hierarchical levels along each dimension. The number of possible views for the example in Figure 1 is $5 \times 4 \times 2 = 40$. Let d be the number of dimensions in a data warehouse. Let h_i be the number of hierarchical levels in dimension i . The general equation for calculating the number of possible views is given by Equation 1.

$$\text{Possible views} = \prod_{i=1}^d h_i \quad (1)$$

If we express Equation 1 in different terms, the problem of exponential explosion becomes more apparent. Let g be the geometric mean of the number of hierarchical levels in the dimensions. Then Equation 1 becomes Equation 2.

$$\text{Possible views} = g^d \quad (2)$$

As dimensionality increases linearly, the number of possible views explodes exponentially. If $g = 5$ and $d = 5$, there are $5^5 = 3,125$ possible views. If we make $d = 10$, then there are $5^{10} = 9,765,625$ possible views. OLAP users need the freedom to scale up the dimensionality of their data warehouses. Clearly we can not create and maintain all possible views as dimensionality increases. The design of OLAP systems must deliver quick response while maintaining a system within the resource limitations.

1.3 OLAP Architecture Components

View Size Estimation

OLAP systems selectively materialize strategic views with high benefits in order to achieve quick response to queries, while remaining within the resource limits of the computer system. The size of a view affects how much disk space is required to store the view. More importantly, the size of the view determines how much disk input/output will be consumed when querying and maintaining the view. Calculating the exact size of a given view requires actually calculating the view from the base data. Reading the base

data and calculating the view is the majority of the work involved in actually materializing the view. Since the object of view materialization is to conserve resources, it becomes necessary to estimate the size of the views under consideration for materialization.

The consistency of the estimates produced by the view size estimation algorithm affects the quality of the set of views selected for materialization. The goal of view size estimation is to design a view size estimator with greater consistency than previous algorithms. Achieving greater consistency results in better choices by the view selection algorithm [8].

View Selection

The number of possible views is exponential relative to the number of dimensions, as demonstrated above. Most prior art has focused on static view selection algorithms with polynomial time and space complexity relative to the number of possible views. However, since the number of possible views is exponential relative to the number of dimensions, these algorithms presented in prior art are exponential relative to the number of dimensions. A view selection algorithm that picks a strategically beneficial set of views for materialization with polynomial time and space complexity relative to the number of dimensions is required to meet the demands of OLAP users. Any algorithm that considers every possible view is exponentially complex. We developed the Polynomial Greedy Algorithm [9], a greedy view selection algorithm that considers only views in promising subsets of all possible views. The promising subsets are nominated into a candidate set. The size of the candidate set is polynomial relative to the number of dimensions. The result is a view selection algorithm with polynomial time and space complexity relative to the number of dimensions. This advancement facilitates the scalability of OLAP systems into higher dimensionality.

View Maintenance

Once a view is selected for materialization, it must be computed and stored to disk. When the base data is updated, the aggregated view must also be updated to maintain consistency between views. The original view materialization and the incremental updates are both considered as view maintenance. The efficiency of view maintenance is greatly affected by the data structures implementing the view. OLAP systems are multidimensional, and fact tables contain large numbers of rows. The access methods implementing the OLAP system must meet the challenges of high dimensionality in combination with the large row counts [10].

Query Optimization

When a query is posed to an OLAP system, there may be multiple materialized views available which could be utilized to compute the result. For example, one view may contain data aggregated by month and another view may contain the same data aggregated by quarter. If the user wishes to see the data aggregated by year, either of the two existing materialized views can be utilized. With the possibility of answering queries from alternative sources, the optimization issue arises as to which source is the most

efficient for the given query. Most existing research focuses on syntactic approaches (see Section 5). The possible query translations are carried out, alternative query costs are estimated, and what appears to be the best choice is carried out. The approach presented here queries a metadata table containing information on the materialized views to determine the best view to query against, and then translates the original SQL query to utilize the best view.

Section 2 discusses how materialized views can be described in a useful way as metadata to aid with the query optimization process. Section 3 illustrates our query optimization process with an example. We analyze the computational complexity of our query optimization strategy in Section 4, along with a discussion of future experiments to be done to verify the methodology. Section 5 summarizes prior research in OLAP query optimization, and Section 6 is the conclusion.

2 Materialized View Metadata Used for Query Optimization

Materialized views aggregated from a fact table can be uniquely identified at the aggregation level for each dimension. Given a hierarchy along a dimension, let 0 represent no aggregation, 1 represent the first level of aggregation, and so on. For example, if the Calendar dimension has a hierarchy consisting of DateID, Month, Quarter, Year and All (i.e. complete aggregation), then DateID is level 0, Month is level 1, Quarter is level 2, Year is level 3 and All is level 4. If a dimension does not explicitly have a hierarchy, then level 0 means no aggregation, and level 1 means "All". The scales so defined along each dimension define a coordinate system for uniquely identifying each view in a product graph. Figure 3 illustrates a product graph in two dimensions. Product graphs are a generalization of the hypercube lattice structure introduced by Harinarayan, Rajaraman and Ullman [6] where dimensions may have associated hierarchies. The top node, labeled $(0,0)$ in Figure 3, represents the fact table. Each node represents a view with aggregation levels as indicated by the coordinate. The relationships descending the product graph indicate aggregation relationships, as in the hypercube lattice. A view can be aggregated from any materialized ancestor view. The five shaded nodes in Figure 3 indicate that these views have been materialized.

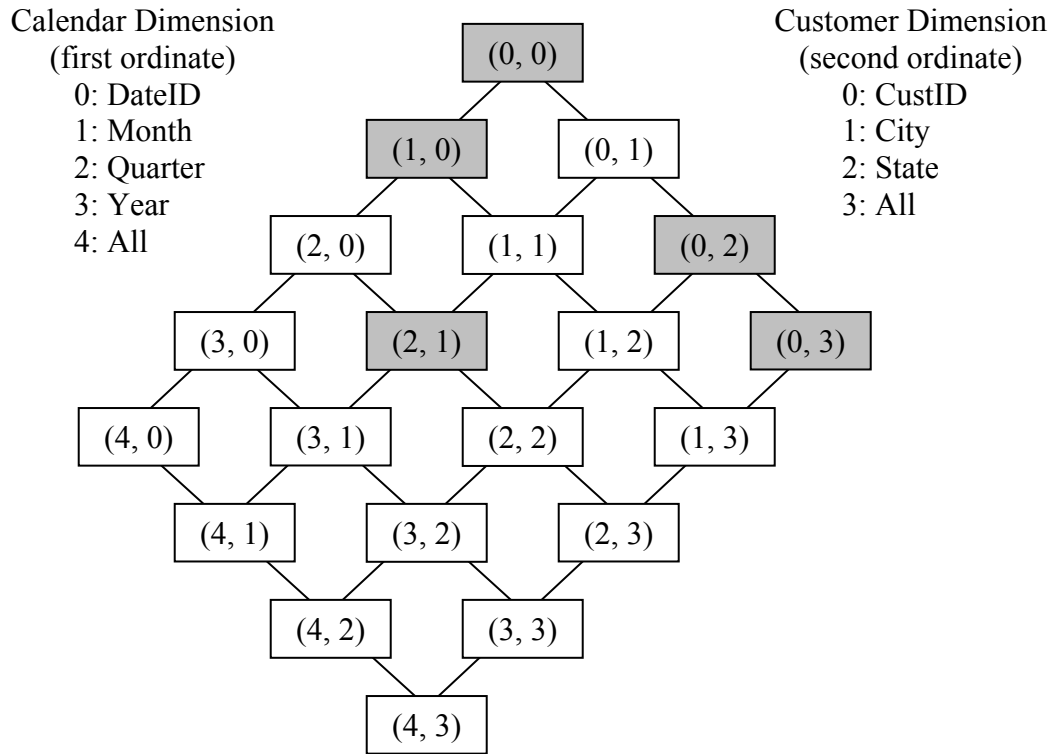


Figure 3 Product graph labeled with aggregation level coordinates

The coordinate system defined by the aggregation levels forms the basis for organizing the metadata for tracking the materialized views. Table 1 displays the metadata for the materialized views shaded in Figure 3. The two dimensions labeled *Calendar* and *Customer* form the composite key. The *Blocks* column tracks the actual number of blocks in each materialized view. The *ViewID* column is used to identify the associated materialized view. The implementation stores materialized views as tables where the value of the *ViewID* forms part of the table name. For example, the row with *ViewID* = 3 contains information on the aggregated view that is materialized as table AST3 (short for automatic summary table 3).

Dimensions		Blocks	ViewID
Calendar	Customer		
0	0	10,000,000	1
0	2	50,000	3
0	3	1,000	5
1	0	300,000	2
2	1	10,000	4

Table 1 Example materialized view metadata

Observe the general pattern in the coordinates of the views in the product graph with regard to ancestor relationships. Let $\text{Value}(V, d)$ represent a function that returns the aggregation level for view V along dimension d . For any two views V_i and V_j , such that $V_i \neq V_j$, V_i is an ancestor of V_j if and only if for every dimension d of the composite key, $\text{Value}(V_i, d) \leq \text{Value}(V_j, d)$. This pattern in the keys can be utilized to identify ancestors of a given view by querying the metadata. Identifying ancestor views is important because a query posed against a view that has not been materialized can be answered from any materialized ancestor of that view. The semantics of the product graph are captured by the metadata, permitting the OLAP system to search semantically for the best materialized ancestor view by querying the metadata table.

3 The Query Optimization Process

A query posed by a user has a view that is natural for answering the query. The natural data source is determined by the level of aggregation in the user's query for each dimension. For example, given the product graph illustrated in Figure 3, if a user queries for total sales grouped by *Month* and *State*, the natural data source is the view labeled $(1, 2)$. If the view $(1, 2)$ is materialized, the user query can be answered from that view directly with no need for further aggregation. However, if the view $(1, 2)$ has not been materialized, as is the case in Figure 3, the query can still be answered by aggregating from any materialized ancestor of $(1, 2)$. The user query must be rewritten to utilize the materialized ancestor that appears to offer the fastest response. The metadata facilitates this process. The activities of the query optimization process are outlined in Figure 4.

We will walk through the process with the running example. If the fact table is named simply *FactTable*, and the measure to be aggregated is named *Sell*, then the user SQL query is:

```

Select Calendar.Month, Customer.State, Sum(FactTable.Sell)
From FactTable, Calendar, Customer
Where Calendar.DateID = FactTable.DateID
And Customer.CustID = FactTable.CustID
Group By Calendar.Month, Customer.State;

```

(3)

The first activity is to *Parse the User Query*. The parsing process determines that *Calendar.Month* represents aggregation level 1 on the *Calendar* dimension, and *Customer.State* represents aggregation level 2 along the *Customer* dimension. Thus the optimizer can *Determine the Natural View* for answering the user query is the view $(1, 2)$. The optimizer can then *Build a Metadata Query to Locate the Natural View*. The SQL code for our example is shown in Query 4.

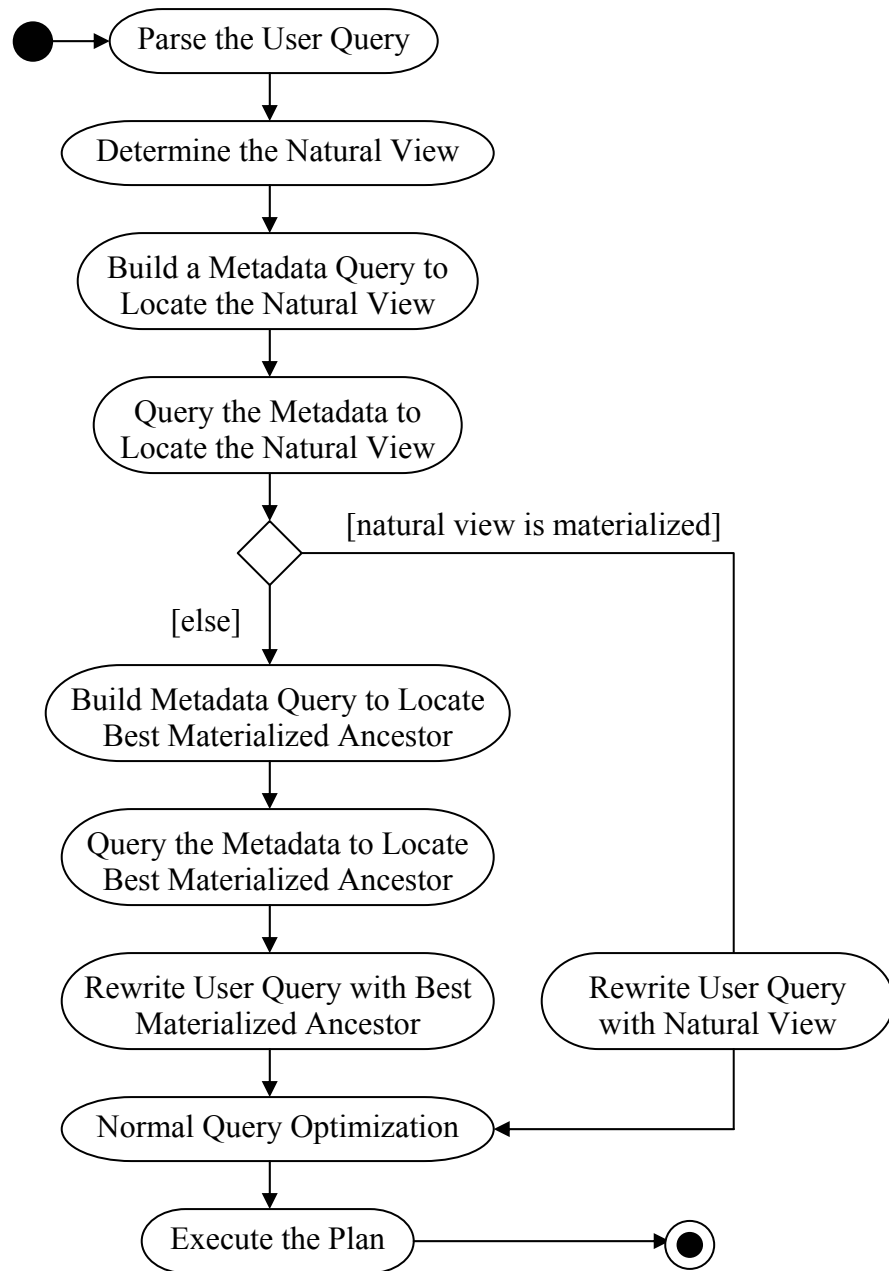


Figure 4 Activity diagram of query optimization with materialized views

```

Select *
From Metadata
Where Metadata.Calendar = 1
And Metadata.Customer = 2;

```

(4)

The optimizer uses Query 4 to *Query the Metadata to Locate the Natural View*. If Query 4 returns a row, then *natural view is materialized* is true, and processing proceeds to *Rewrite User Query with Natural View*, and then to *Normal Query Optimization*. However, for our example, the natural view is not materialized, and no row is returned by Query 4 since there is no matching row in Table 1. Therefore the optimizer follows the *else* transition to the *Build Metadata Query to Locate Best Materialized Ancestor* activity. The optimizer builds Query 5. Note that in the case where no materialized aggregated view is usable, Query 5 will return the metadata row representing the fact table, since the metadata table includes a row representing the fact table. As a last resort, the user query can always be answered using the fact table as the data source.

```

Select *
From Metadata
Where Metadata.Calendar <= 1
And Metadata.Customer <= 2
Order By Blocks;

```

(5)

The optimizer uses Query 5 to *Query the Metadata to Locate Best Materialized Ancestor*, which returns the metadata shown in Table 2. The rows in Table 2 correspond to the materialized ancestors of the view (1, 2). The best materialized ancestor for answering Query 3 is the first view in Table 2, since the materialized ancestors have been ordered by size, and the smallest of these views is the quickest to read from disk.

Dimensions		Blocks	ViewID
Calendar	Customer		
0	2	50,000	3
1	0	300,000	2
0	0	10,000,000	1

Table 2 Example result of metadata query

The optimizer proceeds to *Rewrite User Query with Best Materialized Ancestor*. Query 3 is rewritten to aggregate the desired information from the materialized view (0, 2) instead of the fact table. The optimizer determines from the *ViewID* in the first row of the metadata in Table 2 that the view (0, 2) is stored in the table named "AST3". The schemas of automatic summary tables are defined by the OLAP system when the view is materialized. The names of the attributes are determined by the level of aggregation, and the names of the corresponding levels in the hierarchies. The schema of AST3 is shown in Figure 5 as an example.

AST3	
<u>DateID</u>	
<u>CustomerState</u>	
Sell	

Figure 5 Example schema of an automatic summary table

The Calendar dimension remains at the finest granularity, thus the attribute name remains *DateID*, as it is in the fact table. The Customer dimension is grouped by state. The corresponding attribute name *CustomerState* is derived by concatenating the dimension name with the level name. This systematic mapping from aggregation levels to attribute names allows the query optimizer to rewrite the user query appropriately. Query 6 shows the rewrite of Query 3.

```

Select Calendar.Month, AST3.CustomerState, Sum(AST3.Sell)
From AST3, Calendar
Where Calendar.DateID = AST3.DateID
Group By Calendar.Month, AST3.CustomerState;

```

(6)

The query optimizer proceeds to the *Normal Query Optimization* mechanisms utilized in today's database management systems. Lastly, the optimizer proceeds to *Execute the Plan* and the process is complete.

Overall, the approach outlined in Figure 4 is simple, and requires little processing to accomplish rewriting the user query to utilize what appears to be the best alternative of the materialized views. The query optimizer creates one or two simple queries against the metadata and then rewrites the user query appropriately. This is less effort than finding possible rewrites and then determining which looks to be the best plan, as is done in prior art.

This optimization process takes place within the OLAP system, and is invisible to the user. The user does not need to know the existence of the automatic summary tables, and their schemas. The user sees the process as submitting a query to the OLAP system, and then quickly receiving the desired results.

The metadata approach to query optimization with materialized views is easily implemented in conjunction with the normal query optimization processes utilized in commercial databases. The approach augments the normal query optimization activity, as can be seen in Figure 4. We implemented our approach in conjunction with a Microsoft SQL Server database, and utilized the approach to facilitate materializing views. After a set of views is selected for materialization, they can be materialized progressively, each view being materialized based on the smallest materialized ancestor. The activities of the process are identical to those shown in Figure 4, except that the user query is generated by the materialization process. The implementation did not require access to the source code for the SQL Server optimization process. No special purpose structures were required.

The efficiency of the metadata approach needs to be verified empirically. A comparison with the lattice index approach [5] would be informative. Implementing the lattice index requires the ability to obtain query time estimates from the optimizer of a database. This is problematic without access to the source code of a commercial database, or at least an API with the functionality to obtain query time estimates. We would propose this as future work, in conjunction with any database vendor interested in pursuing this matter further.

4 Complexity of Query Optimization

Complexity analyses of query optimization in the literature have focused on more general classes of problems than encountered in the context of OLAP systems. For example, Levy et al. [7] state that "If Q is a conjunctive query with built-in predicates and V are conjunctive views without built-in predicates, then the problem of determining whether there exists a rewriting of Q that uses V is NP-complete." Their analysis assumes that the views may be derived from multiple base tables. However, each materialized view in an OLAP system based on the hypercube lattice structure [6] or the product graph structure is derived from a single fact table. The rewrite problem is simpler in this context. Let us examine an approach similar to the metadata query approach presented in Section 2. Then we will analyze the complexity of the query rewrite process in the context of OLAP systems.

The prior art that appears to be closest to the metadata query approach is Goldstein and Larson [5]. They utilize an index structure called a lattice index. The structure of the lattice index is very closely related to the hypercube lattice [6], except that only the materialized views are represented, and the view size is not represented. Given a user query, their optimizer determines the required dimensions, scans the top nodes of the lattice index for views that are supersets of the required dimensions, and then recursively follows the edges downward for other nodes that may be supersets of the required dimensions. The nodes thus located represent views that can be utilized in answering the user query. Query rewrites are created for all such views. Then the optimizer analyzes the expected performance of each rewrite, and chooses the best alternative. The worst case complexity of this view matching is $O(d)$ where d is the number of dimensions, assuming the comparison time per dimension is some constant. The maximum number of views to be compared is $O(k)$ where k is the number of materialized views. Thus the worst case complexity for comparison processing is $O(dk)$. The worst case number of edges to be traversed occurs when the k nodes are arranged in two layers, each with $k/2$ nodes, and the top layer is fully connected with the bottom layer. If the required dimensions are a subset of each node in the top layer, then $O(k^2)$ edges are traversed. Thus the overall worst case complexity of finding the usable views with the lattice index approach is $O(dk + k^2)$. According to the experiments run by Goldstein and Larson [5], the typical complexity appears to be linear relative to k .

The worst case complexity for comparison processing for the metadata query approach is also $O(dk)$. However, there are no edges to be traversed as in the lattice index, so the view representations are never visited multiple times. Therefore the $O(k^2)$ edges traversed in the lattice index has no counterpart in the metadata query approach.

The metadata query approach as defined in Section 3 also requires a sort of the result set, which in the worst case takes $O(k \lg(k))$ time. Thus the worst case complexity of the metadata query approach as defined is $O(dk + k \lg(k))$. This is definitely an improvement over the worst case performance of the lattice index approach. The complexity of the metadata query approach could be improved slightly to $O(dk)$ by writing a scan procedure that would track the best view while passing through the table, thereby eliminating the need to sort the result set. The worst case complexity cannot be improved any further. The worst case performance for any matching algorithm must be at least $O(dk)$, since in the worst case the entire key for each of the k views must be scanned to determine if it is a match.

There are other advantages to the metadata query approach versus the lattice index. The metadata query approach currently handles hierarchies along the dimensions whereas the lattice index as it exists is restricted to data sets without hierarchies along dimensions. The metadata table does not require any special structure; it can be stored in a regular data base table. The metadata query approach does not create a rewrite for every matching view and then decide which to execute, rather it picks the view up front, and then does the rewrite. This saves time during the rewrite process.

5 Prior Research in OLAP Query Optimization

The traditional query optimization processes developed for data base management systems address the problem of how to quickly answer queries from base tables and the available indexes. Materialized views aggregated from the base tables enter as a prominent factor for consideration in OLAP systems. Often a query can be answered using any one of a variety of materialized views. The query optimization process must include steps to determine which view is most useful in answering a query, and the query posed against the base data must be translated into a query against the appropriate view.

Chaudhuri et al. expand the traditional query optimization algorithm common in commercial databases to take advantage of materialized views [2]. The approach can be broken down into three stages: Unfold the views in a query (this is done in the traditional algorithm), find alternative rewrites utilizing views, and select the best alternative.

Levy et al. address the problem of finding a minimal rewrite (i.e. the rewrite that contains the minimal number of literals) [7]. They prove that the minimal rewrite problem is NP-Complete. The approach to finding a good rewrite is carried out in two steps. First, containment mappings are found from views to the query. The appropriate view literals are added to the query. The second step is to remove redundant literals from the query. The net effect is to substitute the views appropriately.

Srivastava et al. present algorithms for rewriting queries to equivalent queries utilizing views [12]. The rewrite algorithms exploit syntax and semantics when producing equivalent queries. Previous algorithms focused only on syntax. Utilizing semantics allows for a more general application of the algorithms. The authors focus on queries with the SQL form select-from-where-groupby-having. Formalized preconditions and algorithms are presented for producing the rewrites.

Cohen, Nutt and Serebrenik [4] focus on rewriting aggregate queries using views that may or may not have aggregate operators. The approach is a syntactic analysis. The paper addresses the case of disjunctive aggregate queries. For the cases discussed, they show that the existence of partial and complete rewritings is decidable, and analyze the complexity.

The MiniCon algorithm of Pottinger and Levy [11] seeks a query rewrite that can most completely answer the query posed with the data sources available. Query optimization is cited as a possible extension.

Goldstein and Larson tackle the problem of query optimization, scalable in the number of views available [5]. They contribute an efficient algorithm for view matching, and a data structure called a lattice index that speeds up the view matching process. When 1000 views are available, their query optimization takes about 0.15 seconds on their platform.

Afrati, Li and Ullman [1] achieve better efficiency for query optimization calculations. Their approach reduces the search space of query rewrites considered, while insuring the reduced search space includes a logical plan resulting in an optimal physical plan.

Most of the existing publications on query optimization focus primarily on syntactic methods of finding the possible query rewrites utilizing the views available. Once the possible rewrites are determined, that which appears to be the optimal query rewrite is converted into a physical plan and executed. The syntactic approach to finding query rewrites could be fundamentally improved by utilizing metadata gathered in the course of OLAP operations. The metadata can be queried to determine a good materialized view to facilitate a quick response. This amounts to searching the metadata semantically. Then the original query can be rewritten to query the chosen view instead of the fact table. After the query rewrite, the normal query optimization techniques can be applied.

6 Conclusion

Optimization in OLAP systems can be subdivided into four problems: view size estimation, selection of views for materialization, materialized view maintenance, and query optimization in the presence of materialized views. This paper briefly summarizes each of these four areas, how the processes interact, and how the query optimization phase can be implemented. This approach serves as a plan for the implementation of a commercial OLAP system with faster query response and greater scalability properties than previous systems. The intelligent use of metadata is utilized to improve query optimization in the presence of materialized views. The semantic search for the best data source is straightforward and reduces the number of query rewrites compared to previous approaches. This approach to OLAP optimization adds significant improvements to performance and creates opportunities for OLAP users to explore their data more quickly and more fully.

References

- [1] F.N. Afrati, C. Li, J.D. Ullman, "Generating Efficient Plans for Queries Using Views," in *SIGMOD 2001*, pp. 319-330, 2001.
- [2] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, K. Shim, "Optimizing Queries with Materialized Views," in *ICDE 1995*, pp 190-200, 1995.
- [3] E.F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377-387, June 1970.
- [4] S. Cohen, W. Nutt, A. Serebrenik, "Rewriting Aggregate Queries Using Views," in *PODS 1999*, pp 155-166, 1999.
- [5] J. Goldstein, Per-Ake Larson, "Optimizing Queries Using Materialized Views: A Practical, Scalable Solution," in *SIGMOD 2001*, pp. 331-342, 2001.
- [6] V. Harinarayan, A. Rajaraman, J.D. Ullman, "Implementing Data Cubes Efficiently," in *Proceedings of the 1996 ACM-SIGMOD Conference*, pp. 205 - 216, 1996.
- [7] A.Y. Levy, A.O. Mendelzon, Y. Sagiv, D. Srivastava, "Answering Queries Using Views," in *PODS 1995*, pp 95-104, 1995.
- [8] T.P. Nadeau, T.J. Teorey, "A Pareto Model for OLAP View Size Estimation," in *Proceedings of CASCON '01*, pp. 1-13, 2001. Also in *Information Systems Frontiers*, vol. 5, no. 2, pp. 137-147, Kluwer Academic Publishers, 2003.
- [9] T.P. Nadeau, T.J. Teorey, "Achieving Scalability in OLAP Materialized View Selection," in *Proceedings of DOLAP'02*, pp 28-34, 2002.
- [10] T.P. Nadeau, T.J. Teorey, "Paged Bin-Tree: An Efficient Multidimensional Index Structure," technical report – see <http://www.eecs.umich.edu/~teorey/cv.html>
- [11] R. Pottinger, A. Levy, "A Scalable Algorithm for Answering Queries Using Views," in *VLDB 2000*, pp 484-495, 2000.
- [12] D. Srivastava, S. Dar, H.V. Jagadish, A.Y. Levy, "Answering Queries with Aggregation Using Views," in *VLDB 1996*, pp 318-329, 1996.
- [13] T.J. Teorey, *Database Modeling & Design*, Third Edition, Morgan Kaufmann Publishers, 1999.
- [14] E. Thomsen, G. Spofford, D. Chase, *Microsoft OLAP Solutions*, John Wiley & Sons, 1999.