A Uniform Approach for Selecting Views and Indexes in a Data Warehouse*

C.I. Ezeife
School of Computer Science
University of Windsor
Windsor, Ontario
Canada N9B 3P4
cezeife@cs.uwindsor.ca
Tel: (519) 253-3000 ext 3012; FAX: (519) 973-7093.

Abstract

Careful selection of aggregate views and some of their most used indexes to materialize in a data warehouse reduces the warehouse query response time as well as warehouse maintenance cost under some storage space constraint. Data Warehouses collect and store large amounts of integrated enterprise data from a number of independent data sources over a long period of time. Warehouse data are used for online analytical processing to assist management in making quick and competitive business decisions. Precomputing and storing summary tables (materialized views) reduces the amount of time needed to recompute these views across several source tables in order to answer complex warehouse queries. A data cube is an elegant way for representing aggregate information in a Warehouse and is an n-dimensional view with 2^n subviews. This paper presents a uniform technique for selecting the subviews of the data cube and their indexes to materialize in order to produce the best resultant benefit to the system in terms of query response time and maintenance cost while satisfying some storage space constraint.

Keywords: Warehouse, Views, Indexes, Fragmentation, Performance benefit

1 Introduction

Data Warehouses collect, store and integrate large amounts of data from various function oriented databases over a long period of time. Warehouse data are used in online analytical processing (OLAP) suitable for making quick and competitive business decisions (e.g., increasing market share, reducing costs and expenses and increasing revenues) [1]. Figure 1 gives a data warehouse architecture where a number of source databases (sdb) are integrated into an operational data store (ods) which contains integrated data for a shorter period of time without summary tables. Data in ods are populated into warehouse data. Warehouse data are accessible through executive information system (EIS) which includes complex queries requiring comparative analysis of aggregate information on millions of rows of data in warehouse tables. There are usually hundreds needed aggregation queries requiring hundreds of huge summary tables with millions of rows in the warehouse. The volume of data needed to store, places high demands on available storage space. As new transactional data at the data sources arrive (e.g., customers deposit or withdraw money from their accounts), stored summary tables need to be updated or refreshed amounting to higher maintenance cost being incurred for keeping these tables. A physical warehouse design tool assists in selecting and organizing the massive amount of data stored in the warehouse for better performance. The work proposed in this paper contributes to the development of a physical warehouse design tool.

The volume of data in the warehouse is kept manageable by storing data at various levels categorized largely by the age of the data. The size of the older and less frequently accessed data is reduced by increasing the granularity of data summarization (e.g., summarized by month and not by second). Thus, older data can belong to the lightly

*This research was supported by the Natural Science and Engineering Research Council (NSERC) of Canada under an operating grant (OGP-0194134) and a University of Windsor startup grant.

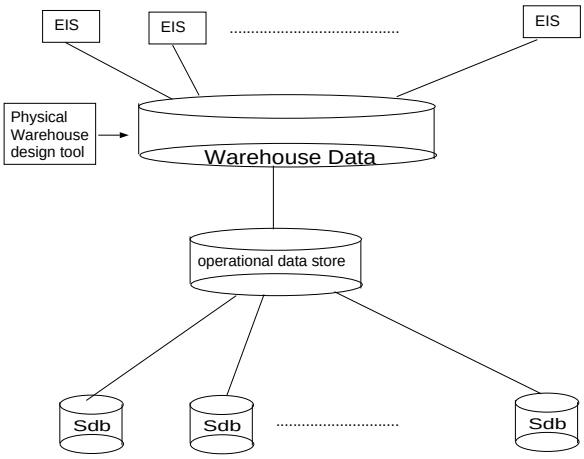

Figure 1: A Data Warehouse Architecture

or highly summarized data level on disk while the very old data (e.g., 20 years old) can be archived onto a low speed but high storage capacity medium like tape. Current detail level is the most vast in size because summarization is largely presented at the finest granularity (e.g., summarized by second) leading to high volume of data at this most frequently accessed level. Thus, data warehouse performance will be greatly improved upon if careful physical warehouse design is undertaken at this level.

The objective of physical warehouse design generally performed by the warehouse administrator is to achieve (1) minimum query response time, (2) minimum maintenance cost, and (3) maximum utilization of storage space. Since these are conflicting requirements, meeting all of them in one design is always a challenge and any technique to produce a balance is encouraging. One way to satisfy these three performance metrics for stored data is to predetermine the views and their indexes which produce the resultant minimum response time and maintenance cost given a specific space constraint. The access method provided for retrieving stored data also affects the retrieval speed of the data. B-tree and sequential access are fundamental access methods supported by many relational databases [1]. However, B-tree access is not adequate for handling very large tables whose transactions are not very well defined ahead of time. An optimizing technique requires redesigning the storage structures for data warehousing. Gray *et al.* in [4] introduced the idea of representing warehouse data with the data cube model. A data cube organizes a database aggregate value (e.g., total amount of dollars involved in banking transactions like updates or cash withdrawals) along n dimensions or n subjects (e.g., customer(C), time(I), transaction type(T)). A data cube view can be used to show the total amount of dollars involved in all daily transactions grouped by customer and the transaction type. This is just one of the eight subcubes (or subviews) of this 3-dimensional data cube. The equivalent SQL query for creating this subview of the data cube is:

```
CREATE VIEW CdT-trans(Customerid, Day,
     Transtype, TotalAmt) AS
SELECT Customerid, Day, Transtype,
SUM(Amt) AS TotalAmt
FROM fact-table, time_hierarchy
GROUP BY Customerid, Day, Transty
```

Here, there are three GROUP-BY attributes in this data cube and the attribute time is in seconds causing a join to be required for higher time dimension attribute of Day being asked for. A data cube with n possible group-by attributes has 2^n subcubes and so the size of the data cube can be very huge leading to space, response time and maintenance cost problems. Harinarayan in [6] used a greedy algorithm to select a set of views to materialize from a cube lattice. [5] extends this greedy algorithm to accommodate the problem of selecting views as well as their indexes to materialize in order to improve OLAP query performance.

This paper provides an alternative approach for selecting a set of subcubes of the data cube and a set of its indexes to achieve near optimal OLAP query performance. This work presents a flexible approach that accommodates a wide variety of physical features that may be present in a particular OLAP system by using a cost/benefit model which takes many performance metric into account. Specifically, the approach considers the top view of a data cube the main class object. The top view of the data cube is the view with all the group-by attributes which is called the base level view. Secondly, the scheme considers as this class's attributes all the 2^n subcubes of the data cube. Then, using the Warehouse view usage matrix, the warehouse view net benefit matrix and the warehouse application frequency matrix, the algorithm applies vertical clustering techniques similar to those used in [3] and [9] to iteratively define groups of attributes of the View class which cluster together based on their application usage frequency and their cumulative net benefit in the presence of some space constraint S. After creating vertical fragments of the view class with each fragment

containing a set of subviews, a fragment cleaning operation is performed to drop views in each fragment never accessed by applications and to drop one of each parent and child view pair contained in the same fragment. Thereafter, the total benefit of each cleaned fragment is computed and the fragment with the highest total benefit is selected as the one containing all subviews that should be materialized. Now, once a subcube of the data cube is selected by belonging to the selected fragment, to select its indexes, the subcube in turn is made the main class object while the list of its indexes are made its attributes, the appropriate three matrices of index usage, index net benefit and index application frequency are used to define fragments of indexes to materialize. In all cases, a selected and cleaned fragment may need to be refragmented if the available storage space is less than the total space requirement of the views in this fragment.

1.1 Motivating Example

Consider a simple banking information database which has information about customers (C), the type of transactions (T) they request from the bank either through the automated banking machine or through the counter and the time (I) they request for these transactions in seconds. This simple warehouse table called fact table in relational OLAP can be modeled as:
bank(Customerid (C), Time(I), Transtype(T), Amt)

This bank fact table holds transaction information from hundreds of branches and so has millions of rows. The fact table grows over time because it may contain records of transactions for up to ten years at the level of transactions done every second. Management can use this type of data to discover customer's interests and needs and direct their new investments towards satisfying most customer's needs thereby attracting more customers in the future. Examples of Warehouse queries that can be directed to the above fact table are:

1. Get the total amount of dollars involved in each transaction daily by each customer (CIT).
2. Get the total amount of dollars involved in all transactions by each customer last month (C).
3. Get the list of all customers who have deposited more than $1M in one month (I).

Materialized views are similar to tables generated by SQL queries which are stored to answer these warehouse queries quickly. A query like (3) above may request customer name, phone number and account balance. These dimension data about the subjects in the data base are not directly derivable from the fact table. Attributes in the fact table are foreign keys which functionally determine the dimension attributes stored in dimension tables. Thus, the star schema used by most warehouse designs includes a set of dimension tables in addition to the fact table. Our example data warehouse has the following dimension tables:

```
 customer(customerid, name, type,
numberacct, totbalance)
 transaction(transtype, name, interest)
 timeinfo(time, minute, hour, day,
week, month, year)
```

The following SQL query is used to materialize the view needed by question (2) above.

```
CREATE VIEW CmT-bank(Customerid, Month,
Totalamt)
AS SELECT bank.Customerid, timeinfo.month,
SUM(bank.Amt) AS Totalamt
FROM bank, timeinfo
where bank.Time = timeinfo.Time AND
timeinfo.month = "last month"
GROUP BY Customerid.
```

This view is one of the 8 possible subcubes of a warehouse data cube with GROUP BY attributes Customerid, Time and Transtype. The GROUP-BY attributes of the data cube are used to present different dimensions of the measure aggregate value *total amount* involved in transactions. All of the subviews of this data cube are listed below and are based on grouping of attributes. We use similar sizes (number of rows) for our subcubes as is used in [6] for a different example because this makes it easier to later compare results from the two approaches. The data cube in our example database has the three attributes: customerid, time in second, and transaction type. All possible views given in terms of the GROUP-BY attributes are:

1. customerid, time, transaction-type (CIT) (assume size of this table is 6 million rows)
2. customerid, time (CI) (6 million rows)
3. customerid, transaction-type (CT) (0.8 million rows)
4. time, transaction-type (IT) (6 million rows)
5. customerid (C) (0.2 million)

6. time (I) (0.01 million)

7. transaction-type (T) (0.1 million)

8. none (ALL) (1)

The cube lattice of these eight views is given as Figure 2.

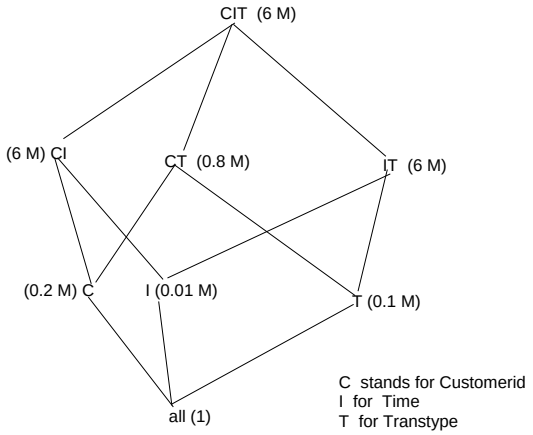

Figure 2: The Cube Lattice for the banking Warehouse

The "none" view, V_8, stands for the query: Select SUM(bank.Amt) from bank. This means there are no GROUP-BY attributes needed to produce the "none" view. Each of these subcubes is also considered a cell of the data cube and the domain of each dimension attribute includes the special value "ALL" [4].

Indexes (e.g., B-tree) could also be used to speed up query processing. Many indexes could be needed and built on some subcubes. [6] argues that indexes are useful in reducing query costs and that a data cube with n GROUP-BY attributes has associated with it about 3n! possible indexes, 2n! of which are fat indexes. The V_3 above on attributes customerid (C) and transaction-type (T) can have four possible indexes as follows: $I_{c,\ t}$, I_{ct}, I_{tc} since the order of the attributes in the index matter. The fat indexes have more than one key combination used as index. This means that our 3-dimensional data cube has 24 possible indexes and 8 subcubes and we are interested in identifying a subset of these 32 huge tables to materialize.

1.2 Related Work

Gray *et al.* [4] from Microsoft overcomes one limitation of the SQL GROUP-BY operator, which is that it does not allow functions for their attributes (e.g., ... GROUP BY SUM(Amt) is not a valid SQL construct). Since many OLAP applications would like to display many dimensions (or perspectives) of aggregate functions, they define the data cube as N-dimensional generalization of the SQL GROUP-BY operator. Then, Harinarayan *et al.* in [6] developed a lattice framework for expressing the dependencies between subviews of the data cube and present a greedy algorithm that works the lattice to select the set of views to materialize in order to maximize some cost benefit, given a specific storage constraint. The first work that considers the set of indexes to materialize in order to maximize benefit is in [5]. This work shows that materializing a set of cube indexes results in further performance gain. It observes that commercial OLAP systems select indexes to materialize using a trial-and-error approach. It further emphasizes the need to develop formal techniques for selecting cube indexes to materialize in a warehouse. Their algorithm computes and selects the sets of pairs of (view and its indexes) producing the maximum benefit per unit space. This selection is done using an extended version of the greedy algorithm. Our paper presents another flexible approach for selecting both views and their indexes which is based on a more sophisticated cost model and net benefit matrix for each independent view or index. The net benefit function can be extended to include many different factors that suit different OLAP requirements. An alternative approach also provides a basis for comparative analysis of existing approaches and following logical analysis of the scheme proposed in this paper, we believe the approach to be competitive.

1.3 Motivation and Contributions

Enterprise database has evolved in various forms over the years and reside in source databases as relational databases, flat files, news wires, HTML document, and knowledge bases. Data warehousing provides means for integrating these source databases for decision support querying [10]. Since OLAP queries are complex and volume of data is large, there is need to balance the time-space tradeoff in order to make this system usable. Carefully selecting and defining a set of views and their indexes to materialize contributes to finding this needed balance between storage space, maintenance cost and query response time. This paper contributes by defining some concrete and low level cost/benefit model for computing the benefits of keeping each view or index (both memory and CPU benefits), the processing cost required

for refreshing and maintaining the view or index. The net benefit of a view is computed as its total benefit less its processing cost. The paper further defines and establishes a relationship between net benefit matrix, application access frequency matrix and the view usage matrix. These three matrices form input to an extension of the grouping algorithm similar to those used in vertically fragmenting attributes of classes, or relations for distributed database systems [3] and [9]. An example to illustrate the working of this scheme is given and a more rigid performance analysis will be undertaken in the future. This work is unique and provides some flexibility to accommodate varying physical factors (e.g., whether warehouse tables are stored on cache memory or random access memory).

1.4 Outline of the Paper

The balance of the paper is organized as follows. Section 2 introduces the cost/benefit model we use in the paper. Section 3 describes the matrices needed for the grouping while section 4 presents the schemes for grouping both views and indexes. Section 5 finally presents conclusions and future work.

2 The Cost/Benefit Model

In this section we discuss and present the assumptions surrounding our cost/benefit model which is used to compute the net benefit (NB) the system derives by materializing a view (V) or an index (I). The net benefit is the same as the cost benefit that the system achieves by materializing a view or index. The cost benefit that a system derives from a view or index if it is materialized is obtained from (1) query response time benefit (QRB) and (2) refreshing or maintenance cost benefit (MCB). The query response time gain of a view is the time gain achieved while using main memory to satisfy this query's request plus the CPU time gained assuming this view is pre-stored. Thus, the query response time benefit has the two components (i) Memory usage benefit and (ii) CPU time benefit. The maintenance cost of a stored view is the time spent refreshing a stored view or index and the disk usage cost incurred for storing the view. The maintenance cost for a view or index that is not stored is the negative of its maintenance cost value assuming it is stored. Thus, since in calculating the net benefit of a view, we are assuming the view is to be stored, we use the maintenance cost benefit of a stored view. The maintenance cost of a stored view consists of (i) disk usage cost and (ii) refreshing cost. Figure 3 gives a pictorial representation of a more detailed summary while further detailed description of the cost model is presented in this section. The equation for the net benefit for a materialized view V is summarized as follows.

$$(\mathrm{NB})_v = (\mathrm{QRB})_v + (\mathrm{MCB})_v$$

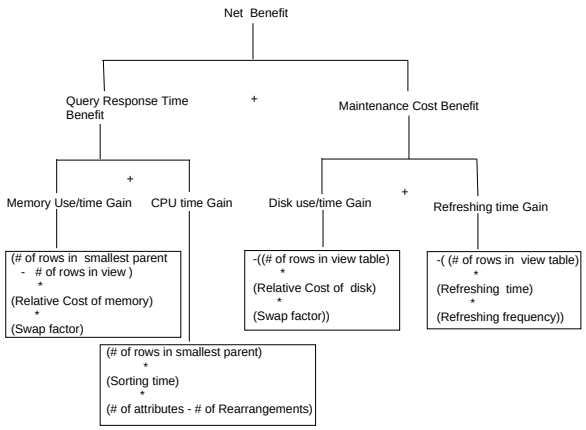

Figure 3: The Net Benefit For a Materialized View

2.1 Query Response Time Benefit

The gain in response time obtained by materializing a view or index constitutes the query response time benefit. If a view/index is stored, there is no need to generate this aggregate table from the fact table or its parent view, thus this frees the main memory for other processes and data leading to a memory usage gain.

Memory Use/time Gain (MMG)

The amount of memory required by a very huge table which needs to be swapped in and out of memory in page units adds to both the processing cost and processing time because of disk I/O overhead. Thus, MMG is the memory processing time benefit for a materialized view which is taken as the number of rows of table that would have been processed if this view were not stored. If this view were not stored, it is computed from its smallest immediate parent and we include the number of times the table is swapped in and out of memory as the swap factor. The relative speed of this memory using its relative cost derived from the memory hierarchy counts as its relative cost. From the memory hierarchy, we assign the following relative costs to different memory types:

Memory Type	Relative Cost
CPU register	5
Cache register	4
RAM memory	3
Disk storage	2
Tape storage	1

The Main Memory time Gain (MMG) for a view is computed as:
$(MMG)_v$ = (Number of rows in its smallest immediate parent - Number of rows in view) * (Relative Cost of Memory) * (Swap factor).

This applies to all views/indexes except the main parent base level view/index and the (ALL) view at the lowest level in the lattice. This is because it is beneficial to always store the main parent view since every other subview is derivable from this view. Secondly, the (ALL) view if needed by any warehouse query has high benefits because it has a negligible size of one row. The MMG for this special class of views (MMG)s is given below:
$(MMG)_s$ = (Number of rows in main parent view) * (Relative Cost of Memory) * (Swap factor).

For example, in our banking data cube, the CIT has 6 million rows and is the same size as the main fact table, thus, we can use its size in place of the fact table's size when ever necessary. The subview CI, with size 6 million from the cube lattice (Figure 2) is directly derivable from its immediate parent CIT, the time gained with respect to memory usage if we materialize this view (CI) in addition to its parent (CIT) is none because the two tables are the same size and are swapped into memory equal number of times. This scheme intends to always store the main parent view. $(MMG)_{ci}$ = (6M - 6M) * 3 * 1 = 0. $(MMG)_{ci}$ for the parent view CIT is (6M * 3 * 1) = 18M.

The CPU Time Benefit (CPB)

The CPU time benefit of a stored view is equivalent to the CPU time which is gained by materializing a view. We reason about this time gain by considering what happens if this view is not materialized. If view is not materialized, its smallest immediate parent is used to generate the view and this may require a resorting of the parent view. For simplicity, we leave the sorting time, swap factor, refreshing time and refreshing frequency at 1 to demonstrate the idea. Number of times a parent view needs to be sorted is accommodated by assuming the maximum number of rearrangements is the same as the number of attributes in the cube, while a stored view has no more rearrangements.

The CPU time benefit for a view (CPB) is computed with the following equation:
$(CPB)_v$ = (Number of rows in its smallest immediate parent) * (Sorting time) * (Number of attributes - Number of needed rearrangements).

The above equation applies to all views/indexes except the main parent base level view and the (ALL) view with the following special CPU benefit $(CPB)_s$.
$(CPB)_s$ = (Number of rows in the main view) * (Sorting time) * (Number of attributes - Number of Rearrangements).

For example, for the view CI, $(CPB)_{ci}$ = (6M * 1 * 3) = 18M and the CPB for the main parent view CIT is = (6M * 1 * 3) = 18M .

2.2 The Maintenance Cost Benefit (MCB)

The cost of maintaining a view is obtained from its disk usage cost (number of its rows and relative cost of disk), the time it takes to load it in memory (its swap factor) and the refreshing cost of the view (cost of updating the view). The MCB of a view is the sum of its disk usage time gain and refreshing time gain. It is the negative of the total cost acquired by the system in order to maintain the view. The exception are the main parent view and the (ALL) view which have maintenance cost benefits of zero because the main parent view is needed to compute every other non-materialized view while the ALL view has only one row and does not take up any meaningful space. The MCB for all views and indexes can be computed with the following formulae.
$(MCB)_v = (\text{Disk time Gain})_v + (\text{Refreshing time})_v$.
$(MCB)_s = 0$

Disk Use/Time Gain (DTG)

This is obtained from the disk space cost (number of rows in view * its relative cost) and its processing cost (swap factor). The negative of these costs represents the disk time benefit for the view. The DTG for all views/indexes except the main parent view and the ALL view is given below:
$(DTG)_v$ = -((Number of rows in view) * (Relative cost of disk) * (Swap factor)).

Our example CI view has a DTG of (- (6M * 2 * 1)) = -12M.

Refreshing Time Gain (RTG)

This is the negative of the cost incurred for refreshing or updating the stored view/index and it is derived with the following formula: $(RTB)_v$ =

- ((Number of rows in view) * (Refreshing time) * (Refreshing frequency)).

For the example view CI, RTB is -((6M * 1 * 1)) = - 6M. Thus, CI has an MCB of (-12M - 6M) = -18M and since it has a QRB of 18M, its net benefit (NB) = 18M - 18M = 0. The net benefit for the parent view (CIT) is (36M - 0) = 36M. The refreshing time is estimated from previous running of the system and is the average time for updating a view with only one row. The refreshing frequency stands for the number of times a stored view is updated in a day.

3 Matrices For the Algorithm

The objective of the proposed scheme is to define non-overlapping clusters of cube subviews in one step, and to run the same algorithm for each selected view and its indexes to again generate non-overlapping fragments of indexes. The purpose of this section is to define the four main matrices used to generate these fragments. Since intuitively we should aim at obtaining the fragment (set of views/indexes) that gives the highest cumulative net benefit taking into account the frequency of application query access to the entire warehouse and the disk storage space available, the four relevant matrices which constitute input to our scheme are:

1. View/Index Usage Matrix (VUM): This matrix indicates for each application Q_i (specified as matrix row label) and each view/index V_j/I_j (specified as matrix column label) whether Use(Q_i, V_j) = 0 or 1. Use(Q_i, V_j) is defined as 1 if application Q_i accesses V_j/I_j directly or V_j/I_j is the smallest immediate parent of the needed view/index. Otherwise Use(Q_i, V_j) = 0. In order to compute the VUM, assign a usage of 1 to every subview, V_j that directly answers a query Q_i. Also assign a usage of 1 to its smallest parent view, V_{sm}. This represents the fact that V_j could be computed with V_{sm} if V_j is not materialized. Using the three query examples in section 1.1, we arrive at the following VUM shown as Figure 4. In this VUM, query 1 accesses view 1, query 2 accesses views 3 and 5, while query 3 accesses views 4 and 6.

2. The Application frequency matrix (AFM): is a one column matrix which measures the number of accesses made by each of the applications Q_i to the warehouse. This information is gathered from earlier system requirements analysis or from previous trial-and-error running of the system. Assume the three warehouse queries given above in section 1.1 access the warehouse at the following frequencies 20, 10 and 30 respectively. Then, the application frequency matrix (AFM) is as given as Figure 5.

	V1	V2	V3	V4	V5	V6	V7	V8
Q1	1	0	0	0	0	0	0	0
Q2	0	0	1	0	1	0	0	0
Q3	0	0	0	1	0	1	0	0

Figure 4: The View Usage Matrix for the Example Warehouse

Access Frequency	
Q1	20
Q2	10
Q3	30

Figure 5: The Application Frequency Matrix

3. The Net Benefit Matrix (NBM): is a one row matrix which gives the net benefit of each of the view/index as computed with the formulae in section 2 above. The NBM for our example views V_1 to V_8 is given in Figure 6. From the NBM, the net benefit for materializing V_1 is 36, that for V_2 is 0, while V_4 has a net benefit of -6.

V1	V2	V3	V4	V5	V6	V7	V8
36	0	14	-6	3	30	3	36

Figure 6: The Net Benefit matrix for the Warehouse Views

4. The fourth important matrix which is finally fragmented to obtain the needed grouping with the bond energy algorithm is the View/index affinity matrix (VAM). The view/index affinity matrix measures the number of accesses made to view/index pairs by all applications accessing the warehouse while considering their cumulative net benefits. The affinity matrix is

an (n x n) matrix for all n views/indexes. The view affinity value for a view pair measures the bond between two views according to how they are accessed by applications and what cumulative net benefits they bring to the system. The view affinity value for a view pair is computed by getting the sum of the product of their cumulative net benefit and the application frequency of each application accessing them together. The formal definition of the $(\text{VAM})_{j,k}$ which stands for the view affinity matrix for view V_j and view V_k is given below. This includes the net benefit for keeping any pair of views together given the many different access patterns of applications.

$(\text{VAM})_{j,k}$
$= \sum_{i|(use(Q_1,V_j)=1) \wedge use(Q_i,V_k)=1)} ((AFM)_{i,1} * ((NBM)_{1,j} + (NBM)_{1,k}))$.

The view affinity matrix (VAM) generated for our example is given in Figure 7. Examples of how these VAM elements are computed follow: $(\text{VAM})_{1,1}$ = (AFM for Q_1) * ((NBM for V_1) + (NBM for V_1)) = (20 * 72) = 1440 and $(\text{VAM})_{3,5}$ = (AFM for Q_2) * ((NBM for V_3) + (NBM for V_5)) = (10 * (14 + 3)) = 170. Q_2 is used in calculating $(\text{VAM})_{3,5}$ because the two views V_3 and V_5 are accessed by application Q_2.

	V1	V2	V3	V4	V5	V6	V7	V8
V1	1440	0	0	0	0	0	0	0
V2	0	0	0	0	0	0	0	0
V3	0	0	280	0	170	0	0	0
V4	0	0	0	-360	0	720	0	0
V5	0	0	170	0	60	0	0	0
V6	0	0	0	720	0	1800	0	0
V7	0	0	0	0	0	0	0	0
V8	0	0	0	0	0	0	0	0

Figure 7: The View Affinity Matrix for the Example Warehouse

4 The View/Index Grouping Scheme

In this section, we present algorithms for selecting a set of views or indexes to materialize given their sizes, application query access frequency and pattern (the views they access), as well as the cube lattice which shows the relationship between the views/indexes. Other input to the scheme consist of the physical storage media, frequency for refreshing the stored views and the swap frequency to determine the average number of times a given view is swapped into memory in order to answer a warehouse query request.

4.1 The Selection Algorithm

Once the view/index affinity matrix is generated, using the view/index usage matrix, application frequency matrix and the view/index net benefit matrix, an earlier competitive grouping scheme called the *Bond Energy algorithm* (BEA) developed in [7] and analyzed in [2], is used to form n fragments of views/indexes. Fragments are formed based on their usage and the benefits the system derives for keeping them. The BEA accepts the view affinity matrix as input and permutes its rows and columns to generate a clustered view affinity matrix. The objective of the permutation is to maximize a certain global affinity measure. Given an (n x n) view affinity matrix, the BEA creates the clustered affinity matrix by initially placing the first two columns of the view affinity matrix in the first two columns of the clustered affinity matrix. Then, for each of the remaining columns of the view affinity matrix, it iteratively checks for the best position in the current clustered affinity matrix to place this column. It makes this decision by computing the contribution of each ordering in the clustered affinity matrix and picking the ordering with the highest contribution to the global affinity measure. The contribution of an ordering is obtained by adding the sum of products of corresponding elements of the two adjacent columns and subtracting the sum of the product of corresponding elements of the non-adjacent columns. For example, with columns 1 and 2 already in the clustered affinity matrix, to determine a position in the clustered affinity matrix to place column 3 of the view affinity matrix, it checks the contributions of the ordering 0–3–1, 1–3–2, and 2–3–4 where 1–3–2 means keep column 3 of view affinity matrix between columns 1 and 2 of clustered affinity matrix. The contribution of the ordering 1–3–2 is obtained from the expression: 2 x [(sum of the products of columns 1 and 3) + (sum of the products of columns 3 and 2) - (sum of the products of columns 1 and 2)]. Non-existent columns like 0 and 4 in this case, produce sum of products of 0. Once a column ordering has been determined for the clustered affinity matrix, the rows of the clustered affinity matrix are reordered to have the

same ordering as its columns. For example, if for a 4 by 4 clustered affinity matrix, the column ordering is determined as 4-3-1-2 which means column 4 of the view affinity matrix is now column 1 of the clustered affinity matrix, the row ordering operation basically brings up row 4 of the same matrix to row position 1 and the same rearrangement is undertaken for all rows. The clusters are formed in such a way that views/indexes with larger affinity values are collected together. Following the clustering, an earlier *Partition* algorithm described in [8] and [9] accepts the clustered affinity matrix and the view/index usage matrix to produce fragments of the views/indexes. The Partition algorithm finds sets of views/indexes mostly accessed by distinct sets of applications. The Partition algorithm aims at finding the best point along the diagonal of the clustered affinity matrix to split the views so as to minimize the incidents of sets of applications needing to cross-reference views/indexes in a different group. These two algorithms have long time applications in distributed database systems and can be applied meaningfully to the warehouse view/index selection problem to achieve desired results. After partitioning into two or iteratively n distinct fragments, we perform fragment cleaning operation before we pick the fragment with higher (or highest in the case of n fragments) cumulative net benefit. The fragment cleaning operation involves first dropping any view which is not used by any application as recorded in the application usage matrix information. Secondly, starting with the youngest child view in the fragment, we compare each view with its smallest parent to drop one of the two views which has lower net benefit. For control, every view is compared only once and this second step is used to eliminate redundancy without loss of benefits to the system. This fragment intuitively contains the set of views/indexes mostly accessed (and with highest cumulative benefits) together with the parent view/index. Thus the steps involved in identifying the set of views/indexes to materialize are summarized below:

- Step 1: Generate the three matrices, view/index usage matrix (VUM), application frequency matrix (AFM), and the net benefit matrix (NBM) using the techniques and formulae presented in Section 3.
- Step 2: Compute the view/index affinity matrix (VAM) as discussed in section 3.
- Step 3: With view/index affinity matrix, generate a clustered view/index affinity matrix by running the Bond Energy algorithm (BEA).
- Step 4: Run the Partition algorithm, on the clustered affinity matrix to generate non-overlapping fragments of views/indexes. Note that the techniques for defining matrices in steps 1 and 2 above are new contributions by this work..
- Step 5: Perform fragment cleaning to drop every view not accessed by a query and to drop which of a parent and child view pair has a higher net benefit.
- Step 6: Select the fragment with higher total net benefits.

The formal algorithm is given as follows:

Algorithm: View/Index Selection
Given: Application frequency matrix, View usage matrix, data cube lattice
Output: Two fragments of Views/indexes
BEGIN
Compute the Net Benefit matrix (as in section 3)
Compute the View affinity matrix (as in section 3)
Compute the clustered View affinity matrix (using BEA)
Partition the clustered view affinity matrix (Partition algorithm)
Perform fragment cleaning (explained in section 4)
{Select fragment Fi which has higher total net benefit.}
END

Submitting the matrices generated above for our example warehouse, to an already implemented BEA - Partition algorithm generates two initial view fragments as follows: Fragment 1= $\{V_1\}$, Fragment 2= $\{V_1, V_2, V_3, V_4, V_5, V_6, V_7, V_8\}$. Initial step of the fragment cleaning operation causes views 2, 7 and 8 not accessed by any application to be dropped. The view, V_6 has V_4 as its immediate parent but this parent view is dropped since V_6 has a higher net benefit of 36. Next, we consider V_5 with parent view V_3 and drop V_5 in favor of V_3. The fragments we are now left with after the cleaning operations are: Fragment 1= $\{V_1\}$, and Fragment 2= $\{V_1, V_3, V_6\}$. We find the total net benefit for each fragment and select the fragment with the higher total net benefit which in this case is fragment 2. Therefore, the selected views to materialize are: V_1, V_3, V_6 which are CIT, CT and I from the cube lattice. It is easy to observe that the warehouse queries (Q_1 to Q_3) are accessing views CIT, C and I and all the selected views can adequately address the concerns of these queries and are the most beneficial to materialize. If the total size of all selected views exceeds the available disk

space S, we keep dropping the view with the lowest net benefit until the total size is less or equal to the available disk space.

5 Conclusions and Future Work

In this paper, a general scheme for selecting views and indexes to materialize in a warehouse based on their application usage need and their total cumulative net benefit, is presented. A comprehensive cost model is developed which is able to accommodate many factors and can be adjusted to suit specific warehouse designs and needs without modifying the selection scheme.

It is our belief that this scheme is competitive as the time complexities for both the bond energy and the Partition algorithms are $O(n^2)$ for n views or indexes while computing the net benefit matrix has time complexity of O(n). The paper presents an alternative approach for selecting views which uniformly applies to selection of indexes. Contributions of the work include defining a rigorous cost/benefit model which considers low level physical factors that may impact on query response time, maintenance and storage costs of views. The cost model contains parameters whose values can be adjusted to suit the needs of different warehouse designs and presents a scheme for integrating the benefits obtained from each of the views/indexes with fragmentation schemes previously used in distributed database systems.

For future work, we intend to compare the performance of this scheme with that presented in [5] and include proof to show that it is practical, scalable, extendible and always provides a solution.

References

[1] Ramon Barquin and Herb Edelstein. *Planning and Designing the Data Warehouse*. Prentice Hall, 1997.

[2] S. Chakravarthy, J. Muthuraj, R. Varadarajan, and S. B. Navathe. An Objective Function for Vertically Partitioning Relations in Distributed Databases and its Analysis. *Distributed and Parallel Databases*, 2(1):183–207, 1993.

[3] C. I. Ezeife and Ken Barker. Distributed Object Based Design: Vertical Fragmentation of Classes. Submitted to Journal of DAPSs, 1996.

[4] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Proceedings of the 12th International Conference on Data Engineering*, pages 152–159, 1996.

[5] Himanshu Gupta, Venky Harinarayan, Anand Rajaraman, and Jeffrey Ullman. Index Selection for OLAP. In *International Conference on Data Engineering, Burmingham, U.K*, 1997.

[6] Venky Harinarayan, Anand Rajaraman, and Jeffrey Ullman. Implementing Data Cubes Efficiently. In *ACM SIGMOD International Conference on Management of Data*, June 1996.

[7] W.T. McCormick, P.J. Schweitzer, and T.W. White. Problem Decomposition and Data Reorganization by a Clustering Technique. *Operations Research*, 20(5), 1972.

[8] S.B. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical Partitioning Algorithms for Database Design. *ACM Transactions on Database Systems*, 9(4), 1984.

[9] M.T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1991.

[10] J. Widom. Research Problems in Data Warehousing. In *Proceedings of the 4th International Conference on Information and Knowledge Management (CIKM)*, November 1995.