A Partition-Selection Scheme for Warehouse Aggregate Views

C.I. Ezeife*
School of Computer Science
University of Windsor
cezeife@cs.uwindsor.ca
http://www.cs.uwindsor.ca/users/c/cezeife

S.R Baksh
School of Computer Science
University of Windsor

Abstract

Data Warehouse views in relational OLAP (online analytical processing) are aggregation or summary tables which hold millions of tuples describing activities that occurred on various departmental source databases over long periods of time. An n-dimensional data cube is a multidimensional data model used to generate n different perspectives of the measure aggregates (e.g., SUM, MAX) of interest in a warehouse, and has 2^n subviews.

Due to storage space constraint and the need to speed up warehouse query response time, many algorithms have been proposed for selecting the best set of cube subviews to materialize. However, not much has been done on materializing fragmented views for both query response time and maintenance cost benefits. This paper presents an algorithm that first generates a set of horizontal fragments of each selected view and uses a modified version of the greedy algorithm to incorporate the partitioning of the parent view while selecting further views.

Keywords: Data Warehouse, Views, Fragmentation, Performance benefit

1 Introduction

Data warehouses are used to store historical, integrated, "subject-oriented" and summarized data of an organization [CD97]. Warehouse data support business decision making processes characterized by complex queries accessing millions of records while performing many scans, joins and aggregation. One approach used to improve warehouse query performance is pre-computation of aggregate views [GHRU97]. To make this discussion clearer, consider a simple banking system where there are two source databases, one for checking and the other for savings account. A simple warehouse fact table (b-activity) which keeps track of all activities (like deposit, withdrawal, etc.) going on in these two source databases can be constructed with the structure given below:
b-activity (cid (C), accttype (A), transtype (T), time (I), Amt (M)).

The attributes of the fact table are customer id, account type, transaction type, time in minute when the operation took place and the amount in dollars involved in this operation. With the rows of the fact table already stored, warehouse queries may be interested in the aggregate "SUM" by wanting the total amount of dollars deposited by each customer, each minute. They may want the total amount of dollars deposited in each account in the past two years grouped by minutes. Gray *et al.* in [GBLP96] defined the data cube model for generating aggregation of n-dimensions. While a 0-dimensional data cube produces a single aggregate value, a 1-dimensional data cube produces a column of aggregates in addition to the single aggregate value produced for all. An example of a 0-dimensional cube from our fact table b-activity is created using the following sql query:

```
Select Sum(Amt)
from b-activity;
```

An example of a 1-dimensional view which is part of a 2^n cube on the table is:

```
Select cid, Sum(Amt)
from b-activity
groupby cid;
```

In general, an n dimensional data cube has 2^n aggregate views. The n-dimensions represent the n different attributes on the fact table. With b-activity, a 4-dimensional cube with 16 subviews can be created. All the 2^4 subviews are obtained using different combinations of groupby attributes. The subviews from b-activity are: **CATI**, **CAT**, **CAI**, **CTI**, **ATI**, **CA**, **CT**, **CI**, **AT**, **AI**, **TI**, **C**, **A**, **T**, **I**, **{}**.

While the subview **CATI** represents the cube cell for the sql query "select cid, accttype, transtype, time-

*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 grant.

m, Sum(Amt) from b-activity groupby cid, accttype, transtype, time-m", the subview **C** stands for the query which groups by only cid. For optimal query response time performance, storing all 2^n cube views is required, however huge overheads may be incurred in computing and storing the data cube, maintaining or refreshing the cube subviews and as subviews grow in size storage space diminishes. [HRU96] proposes a greedy algorithm for selecting a set of views of the data cube to materialize or store in order to reduce the time needed to answer queries. [GHRU97] argues that commercial OLAP systems select views and indexes to materialize using a trial-and-error approach. Meredith and Khader in [MK96] argue that an approach for improving warehouse performance is aggregate view partitioning but no formal algorithms have been presented on warehouse view partitioning or selection through partitioning.

1.1 Related Work

Gray *et al.* [GBLP96] came up with the concept of the data cube, a multidimensional representation of a set of aggregate measures. Since dimension attributes on both warehouse fact and main cube aggregate tables are foreign keys (e.g., cid, accttype), each foreign key attribute may have associated with it a dimension hierarchy specifying attributes for describing it or for allowing drill-down and roll-up analysis. Drill-down analysis computes a lower level detail summary from a higher level one (e.g., computing sum(Amt) group by hour after first computing sum(Amt) group by day) while roll-up analysis produces higher level detail summary from lower level detail one. An example of a dimension hierarchy for time where main cube has time recorded in minutes and the hierarchy table from finest attribute (or foreign key) to the coarsest, is minutes→hour→day→ month→year. [HRU96] expresses the dependencies between cube views and their dimension hierarchies using a lattice framework and defines the greedy algorithm for selecting the set of views to materialize. The greedy algorithm starts by selecting the top level view into the set S. Then, each of the cube views is checked for the one with the maximum benefit considering the views that are already in the set S. The benefit of a view, v not in the set S is computed as the sum of all positive differences between the sizes of the smallest parent u (in S) of each v's descendant view w, and v. Note that v is included in its descendant set with the above definition since this set represents all views that can be computed with the view v. Any negative difference in sizes of smallest parent of a descendant view of v and v is equated to zero. The benefit of all remaining views in the lattice are computed each time and the view with the highest benefit is included in the set S. Gupta *et al.* [GHRU97] extends the greedy algorithm to select both views and indexes. [Eze97b] defines a uniform scheme based on a comprehensive cost model for selecting both views and indexes. [Eze97a] extends this uniform scheme to handle dimension hierarchies. [OV91] presents horizontal fragmentation schemes for relational databases based on simple predicates and with no query access frequencies taken into consideration. [Inm96, MK96, TC96] have all expressed the need for data partitioning schemes in the data warehouse aggregate materialization problem.

1.2 Contributions

This paper contributes by presenting an algorithm for first horizontally fragmenting the data cube views starting with the base level view. The horizontal fragmentation scheme is different from that defined in [OV91] because it uses access frequencies of queries to attach importance to the predicates defined from the queries. These modifications are necessary because of the huge sizes of the warehouse tables as well as the complexity of the queries accessing the tables. Secondly, a modified version of the greedy algorithm which incorporates the benefits of fragmenting the parent views is used to select subsequent views to materialize. Every selected view is again checked for possible partitioning. This work's decomposition of warehouse queries into three main types of attributes prepares the ground for defining an algorithm which recommends the best set of view fragments for executing warehouse queries in the future. The performance benefits of the proposed schemes are demonstrated using elaborate examples.

1.3 Outline of the Paper

The balance of the paper is organized as follows. Section 2 presents motivating example based on a banking warehousing system which shows the activities that can occur on this database and how decisions can be made about partitioning of views using some sample queries. Section 3 presents formal discussion of the partitioning algorithm and section 4 presents the extended greedy algorithm for selecting the rest of the views to materialize. Section 5 discusses performance analysis while section 6 finally presents conclusions.

2 Motivating Example

This section gives an example which shows how a cube aggregate view can be horizontally fragmented although the formal presentation and discussion of the algorithm designed for this purpose is made in sections 3 and 4.

Example 2.1 Consider a banking data warehouse with historical records of every transaction customers have asked for in all four types of accounts available (savings1(S1), savings2(S2), checking1 (C1) and checking2 (C2)) from across many branches. ■

cid	accttype	transtype	time-m	Amount
C0001	S1	dep	199603210003	200
C0518	C1	wd	199603210100	500
C1000	C2	wd	199603210200	300
C0001	S1	dep	199603221300	500
C0518	S2	dep	199603230600	300
C0411	S2	dep	199603230600	400
C1000	C2	wd	199603231000	100
C0300	S1	dep	199603231200	300
C0411	C2	dep	199603240500	400
C0001	C1	wd	199603241100	600

Figure 1: Sample Warehouse Fact table data

The data warehouse has the following fact and dimension tables:
b-activity (cid(C), accttype(A), transtype(T), time-m(I), amount(M))
customer (cid, cname, ccity, cphone)
account (accttype, date-opened)
time (time-m, hour, day, month, year).
The domain of cid is c0001, c0002, .., c1000. The domain of transtype is deposit (dep), withdrawal (wd), transfer, billpay and balance display. A sample fact table data is given in Figure 1 for only 10 tuples although this table holds millions of rows typically.

The time the transaction took place is recorded as year/month/day/minute. Since in a day there are 1440 minutes (24 * 60), the last four digits of time is used to represent both minute and hour. Some warehouse queries on this table are:

- Q1: Get the number of customers who have made more than 2 withdrawals in savings account S1 in any month.
  For the purposes of our design, we decompose every warehouse query into three attribute components namely (1) Partition attributes (PA), (2) analysis attributes (AA) and (3) measure attributes (MA). Partition attributes are the attributes involved in the "where clause" of the SQL version of the query. Analysis attributes are those involved in the "group-by" clause and measure attributes are aggregates of interest. The first step in our approach is to be able to define simple predicates using the partition attributes. Simple predicates are of the form "PA (relational operator) Value". Thus, for each query we first identify the PA, AA and MA, then, from the PA we identify the set of simple predicates. With this approach, the following attributes and predicate have been identified from the query Q1.
  PA = Accttype (A)
  AA = Month (I), transtype (T)
  MA = Number of customers, Count(C)
  Predicates: P_1: A = "S1"

- Q2: Get all customers who have deposited some money in the morning minutes. The attributes and predicate from Q2 are:
  PA = Time-m (I)
  AA = none
  MA = Number of customers or Count(C)
  Predicates: P_2: $I \leq 0720$

- Q3: Find the total amount of dollars involved in each transaction type and account type between the minutes of 0720 and 0780 (lunch hour). We have from this query:
  PA = Time-m (I)
  AA = Accttype (A) and transtype (T)
  MA = Total amount of dollars or Sum(Amount)
  Predicates: P_3: $I \geq 0720$ AND $I < 0780$

- Q4: Find the total amount of dollars deposited by each customer every minute in account C1. From this query, we obtain:
  PA = Accttype (A)
  AA = Customer (C), Time (I)
  MA = total amount of dollars or Sum(Amount)
  Predicates: P_4: A = "C1"

Since we have defined the predicates from the queries, we attach some importance value (IP) to each predicate using the frequency of application access to them. The intuition here is that only some n most important predicates will be used in making decision about fragmentation of a view. We obtain the IP or importance value of each predicate by multiplying the cardinality of the predicate (number of rows from the fact table that are true for this predicate) by application access frequency (the number of times the predicate is accessed by an application). If a predicate is accessed by more than one application, the sum of these access products from all applications accessing the predicate will make the predicate's IP value. With our example warehouse, assume the queries Q_1 to Q_4 access the warehouse at the following frequencies respectively: 100, 40, 20 and 60 times. We can then see, reading from our sample fact table that $|P_1| = 3$, $|P_2| = 6$, $|P_3| = 0$ and $|P_4| = 2$. Therefore, IP of $P_1 = 3 * 100 = 300$, IP of $P_2 = 6 * 40 = 240$, IP of $P_3 = 0 * 20 = 0$, and IP of $P_4 = 120$. Note that in usual applications a predicate may actually be accessed by more than one query or application.

Once the IP values for all predicates are defined, the current scheme simply selects n highest valued predicates where n is defined by the warehouse administrator. With an n value of 2, predicates P_1 and P_2 are selected. The next step is to create minterm predicates using the selected simple predicates. A minterm predicate is a conjunction of all selected predicates with each

predicate appearing in either its natural(positive) form or negated form. Thus, with the two selected simple predicates, the following minterm predicates are generated:
$M_1 = P_1 \wedge P_2 \Longrightarrow A ='' S1'' \wedge T \leq 0720$
$M_2 = \neg P_1 \wedge P_2 \Longrightarrow A \neq'' S1'' \wedge T \leq 0720$
$M_3 = P_1 \wedge \neg P_2 \Longrightarrow A ='' S1'' \wedge T > 0720$
$M_4 = \neg P_1 \wedge \neg P_2 \Longrightarrow A \neq'' S1'' \wedge T > 0720$

Since all minterms are meaningful with respect to the sample data, they form fragments of the base level view or any view we submit as input. Fragmenting the top level view may affect the selection of other views to be materialized and this information is included in the greedy algorithm while making future view selection as discussed further in section 4. The fragments produced are non-overlapping, complete and minimal in the sense that every tuple in the original view need to be accounted for in only one fragment. The example above demonstrates that created fragments are non-overlapping, complete and minimal because every tuple belongs to only one fragment, every tuple can be found in some fragment, and every fragment is relevant. Using the fact table b-activity in Figure 1 and selecting the tuples of minterms M_1 from this table, we find that only tuple with cid "C0001" is selected, thus $|M_1|$ is 1, $|M_2|$ is 5, $|M_3|$ is 2 while $|M_4|$ is 2 giving back the total of ten tuples in the table. For partitioning views other than the top level view, all simple predicates are again applied and the use of the cardinality of each simple predicate on the particular view being fragmented serves the purpose of selecting out those simple predicates that are not very relevant to this view.

3 Data Cube View Partitioning Algorithm

This section first presents some formal definitions of the terms to be used in presenting the algorithms, then, formally presents an algorithm for partitioning any data cube view, given warehouse queries and their access frequencies.

3.1 Definitions

Definition 3.1 *Primary horizontal fragmentation* is the partitioning of a relation based on the values of its attributes such that each fragment contains only a subset of the tuples in this relation. ■

Definition 3.2 *A user query accessing* an aggregate view V_i is made up of a set of analysis attributes AA, a set of partitioning attributes PA and a set of measure attributes MA. ■

Definition 3.3 *An analysis attribute AA_{ij}* from a user query q_i represents the subjects or groupby attributes of interest to the application. ■

Definition 3.4 *A partition attribute PA_{ij}* in a user query q_i defines the subset of records found in view V_k which is relevant to the application. It is the attribute in the "where clause". A set of simple predicates is usually derived from each partitioning attribute. ■

Definition 3.5 *A measure attribute MA_{ij}* in a user query Q_i is an aggregation attribute of interest to the application. ■

3.2 View-Partitioning Scheme

This section presents the steps involved in horizontally fragmenting a view of the data cube. Input to the process is a set of warehouse queries and their frequency of access to the warehouse per unit of time, say, daily. The other input is the warehouse view to partition. The steps involved in partitioning a view are given below:

- Step 1: Define simple predicates from each user query using the partition attributes. The simple predicates are defined from the partition attributes as $PA_{ij}\ \theta\ Value$ where θ is a relational operator from the set {=, <,>, ≠, ≤, ≥} and Value is from domain of partition attribute (PA_{ij}).

- Step 2: Define the relative importance value (IP) of each predicate Pr_i. The importance of each predicate is defined by adding up the product of the application frequency and the cardinality of this predicate for every application that accesses the predicate. The formula for obtaining the importance of a predicate is given as:
$\forall_{Q_m|access(Q_m,P_k)=1} \sum_{m=1}^{numberofQ's}$ (access frequency of $Q_m * |P_k|$)
where $|P_k|$ is the number of rows or tuples in the partition of the view defined by predicate $|P_k|$, while access(Q_m, P_k)=1 means that the query Q_m accesses the predicate P_k a number of times defined by access frequency of Q_m.

- Step 3: Select the n most important predicates. The current scheme selects n highest valued predicates. This step can be refined or optimized in the future to let the scheme decide what n produces best outcome.

- Step 4: Generate horizontal fragments of the view by defining minterm predicates. A minterm predicate is a conjunction of simple predicates in either their natural or negated forms [OV91]. Thus, given an aggregate view V_i with the set of selected simple predicates $Pr_i = \{P_{i1}, P_{i2}, \ldots, P_{ik}\}$ from queries accessing it, the following set of minterm predicates $M_i = \{M_{i1}, M_{i2}, \ldots, M_{iz}\}$ can be defined on it. Each M_{ij} in M_i is defined as:
$M_{ij} = \bigwedge_{p_{ik} \in Pr_i} P^*_{ik}$, $1 \leq k \leq m$,$1 \leq j \leq z$
where $P^*_{ik} = P_{ik}$ or $P^*_{ik} = \neg P_{ik}$.
Having both the natural and negated forms of each

predicate ensures completeness of views. In other words, partitioning the view should not cause loss of any tuples. Some minterm fragments may not be feasible from the meaning of data and their implications, and those have to be deleted from the minterm set. The objective is to generate only a complete and minimal set of minterms.

The formal definition of the algorithm View-Partion is given as Figure 2.

4 Selecting Other Views Using a Partitioned Parent View

This section discusses the impact that partitioning of a parent view has on the selection of other cube views for materialization. We propose to modify the greedy algorithm of [HRU96] by including the benefits of partitioning a parent view. The process is illustrated by showing that a different set of views is recommended for materialization when the parent view is partitioned as is recommended using straight greedy algorithm with an example from [HRU96] given as Figure 3.

If view a is partitioned into 4 fragments using the algorithm from section 3. If we first use the simple cost model which uses only the total number of rows accessed, and we assume all views are accessed 10 times by queries which access only 2 of the fragments of about the same size 75% of the time, then, the average number of rows accessed in view (a) by a query is (100/2 * .75) + (100 * .25) = 62.5 rows. This makes for a 37.5% reduction in the average number of rows accessed by queries because of the partitioning. The next question is "which views will be selected next using the greedy algorithm?". Since the number of rows now accessed in view a by applications is 62.5 and not 100, our approach replaces the size of this parent view by 62.5 (approximately 63). Using this size the greedy scheme is applied to find the next two views to include in the set of views to be materialized, with the set S initially having only one element, the top-level view a. Figure 4 shows how the selections are made.

The greedy algorithm recomputes the benefit of selecting each remaining view in addition to the list already in S. To make each choice, the view with the highest benefit is selected and during the first choice as can be seen from the column labeled choice 1, the obvious choice is view e which has the maximum benefit of 99. The benefit of e is computed using the lattice of Figure 3 and the set S with one element, the top level view {a}. Since the view e has three views that can be answered using e (its two descendant views and itself), and the difference between size of e and that of its smallest parent view a already in S is 33 using the recomputed partitioned size of a which is 63, then the benefit of e is 33 * 3 to give 99. During the second

Algorithm 3.1 *(View_Partition - Generate a number of horizontal fragments of view V_i)*

Algorithm ViewPartition(V_i)
Input: View V_i, set of queries $\{Q_m\}$,
access frequency of queries $\{AF_m\}$.
Output: Set of fragments of V_i $\{V_i^j\}$
begin
Predicate Set $Pr_i = \{\}$
Initialize importance values of predicates to 0
for m= 1 to number of queries
    Get predicates Pr^k from query Q_m
    $Pr_i = Pr_i \cup Pr^k$
    // Now define the importance (IP) of //
    // these new predicates //
    for each Pr_j^k in Pr^k
    IP of Pr_j^k = IP of Pr_j^k + $(AF_m * |Pr_j^k|)$
end // of for m //
Select n predicates with highest IP values from Pr_i
//Generate minterm predicates as follows. //
// j marks which minterm and k marks which predicate. //
// Initialize. //
j = 1
k = 1
while k ¡= number of selected predicates
begin
//First conjunct the natural form of predicates to the already //
//existing minterms after making a copy of these minterms. //
  for i = 1 to j
    $mcopy_i = m_i$
  end
  for i = 1 to j
    $m_i = mcopy_i \wedge P_k$
  end
// Now conjunct the negative form of predicate to //
  // 2 new minterms //
  for i = 1 to j
  begin
    newj = newj + 1
    $m_{newj} = mcopy_i \wedge \neg P_k$
  end
  j = newj
  k = k + 1
end //of while //
// Delete non-feasible minterms from the list //
for c = 1 to number of minterm predicates
begin
    if any two Pr_k in M_c are contradictory
    delete M_c from the minterm predicate set.
end
end {ViewPartition}

Figure 2: The View Partitioning Algorithm

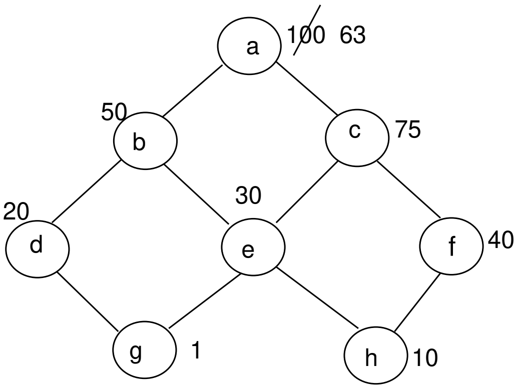

Figure 3: An Example Cube Lattice

	Choice 1	Choice 2
b	13 * 5 = 65	
c	0	
d	43 * 2 = 86	43 + 10=53
e	33 * 3 = 99	
f	23 * 2 = 46	23
g	62 * 1 = 62	29
h	53 * 1 = 53	20

Figure 4: Example of View Selection Using Partitioned Parent Views

selection, view d with the maximum benefit of 53 is selected. This benefit of 53 is obtained by finding the difference between the sizes of the smallest parent view of d in S and d, then adding this difference to the difference in sizes of the smallest parent view of every descendant view of d, already in S and d. This gives a benefit of 43 for the difference between sizes of parent view a of d and d; 10 for the difference between sizes of parent view e in S, of d's descendant view g and d. Thus, the views selected using our Partition-Selection scheme are: {a, e, d} while the views selected using the straight greedy approach are: {b, d, f}. This shows that if there is a high percentage of range queries or queries accessing only parts of a view, partitioning views produces some performance gain and causes a different set of views to be chosen.

4.1 The Partition-Greedy Scheme

The algorithm for selecting a set of views when its parent views are fragmented horizontally computes the total number of rows that are accessed on a view. This count is obtained as the sum of all rows in all fragments accessed, multiplied by the number of times the view is accessed. It follows that in the worst case, all fragments are visited in which case all rows of the view are visited. However, since fragments are non-overlapping, the total

Algorithm 4.1 *(PartitionGreedy - Select the next s views using partitioned parent views)*

Algorithm PartitionGreedy(V_i)
input: available disk space, views;
Output: a set of selected views, SetV
begin
  SetV = {partitioned top view}
  Compute the new size of partitioned view
  while space < limit do
  begin
    for each subview(s) not∈ SetV do
      Compute the benefit of keeping s using new sizes of partitioned parent views
    end
    SetV = SetV ∪ s with maximum benefit
    Now partition s using View-Partition algorithm defined in section 3.
    Compute the new size of partitioned s
  end
end

Figure 5: Greedy Algorithm with Partitioned Parent Views

number of rows visited by queries can only be less than the original number of rows in the view. Once the sizes of the selected partitioned views are obtained (that is average number of its rows visited by a query), the greedy algorithm is run the usual way to select the rest of the views and examples show that the views selected are generally different from the set that would be selected if parent views are not partitioned. The formal definition of the modified greedy algorithm is given as Algorithm PartitionGreedy in Figure 5.

5 Performance Analysis

Partitioning of stored views is beneficial only if there is evidence of some gain in system performance. Gain in system performance is revealed through reduced query response time and/or reduced maintenance cost. This section argues that partitioning a view leads to both reduced query response time and maintenance cost. The query response time is reduced because only a fraction f of all the rows in the view are accessed on the average by a query. In the worst case, f is 1 in which case all fragments are visited and all the rows in the view are accessed on the average by each query. In the best case, f is 0 meaning that no rows of the view are scanned on the average by a query. This suggests the need for a fragment-advisor algorithm whose responsibility is to recommend the best set of fragments of a view a query needs to visit. Future work incorporates a fragment-advisor algorithm which can easily be designed using a

% of Queries (X-axis) accessing 50% of view	%age reduction in number of (Y-axis) rows accessed (Gain in response time)
10	5
20	10
30	15
40	20
50	25
60	30
70	35
80	40
90	45
100	50

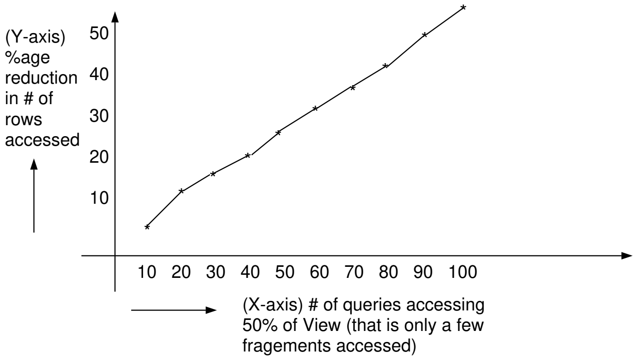

Figure 6: Relationship between Queries Accessing fragments of a View and Response Time Gain

combination of both partitioning, analyses and measure attributes of queries and the predicates to determine the best execution path.

The partition-selection approach for materializing views is better for applications accessing mostly parts of the view. In the example in section 4, the assumption made is that there are 75% range queries accessing only 50% of the rows in the view through its fragments. We can show that the query response time gain obtained using this approach increases as the percentage of rows accessed in the view decreases (meaning fewer fragments scanned) and the percentage of queries accessing only parts (only fragments) of the view increases. The response time gain also increases as access frequencies of queries increase and the view sizes increase over time. With our example top level view a in Figure 3 of section 4, the gain in response time in terms of percentage reduction in number of rows accessed is shown in Figure 6 as percentage of queries accessing 50% of rows in the view increases.

On how partitioning of view leads to a reduction in cost, the argument is that if high percentage of source database updates are directed to portions of the view kept in fragments, only a few fragments will be scanned and updated leading to lower maintenance cost. Further proof of the gains of this approach can be accomplished through experimental analysis. This will enable collection of data on both query response time gain, maintenance cost gain with different query patterns, varying view sizes, varying selections and partitioning of the views using the algorithm and this is intended for future work.

6 Conclusion and Future Work

This paper investigated the problem of first reducing the percentage of total number of rows of huge warehouse views scanned by queries through horizontal fragmentation. An algorithm was defined that recommends the best set of fragments to keep for a stored view for both query response time and maintenance cost gains.

Secondly, the paper contributes by proposing a scheme for selecting other views to materialize using a partitioned cube parent view. This scheme is a modified version of the greedy algorithm which recomputes the size of the parent view as the actual average number of rows the view is accessed through fragments. Every selected view can again be fragmented.

Examples to demonstrate the workings of these algorithms were given. The benefits of these techniques were argued and more analysis can be produced through experiments. The ground works laid in the paper can be extended to include a fragment-advisor scheme which recommends the best set of fragments for a query and the partition-selection approach needs to work when dimension hierarchies and possibly indexes are taken into consideration. We are currently investigating these extensions as well as defining techniques to make this design more dynamic.

References

[CD97] Surajit Chaudhuri and Umeshwar Dayal. An Overview of Data Warehousing and OLAP Technology. In *Sigmod Record, Vol. 26, No. 1*, March 1997.

[Eze97a] C.I. Ezeife. A Uniform Approach For Selecting Views and Indexes in a Data Warehouse. In *Proceedings of the 1997 International Database Engineering and Applications Symposium, Montreal, Canada*, pages 151–160.
ieee publication, August 1997. Available at http://www.cs.uwindsor.ca/users/c/cezeife/.

[Eze97b] C.I. Ezeife. Accommodating Dimension Hierarchies in a Data Warehouse View/Indexes Selection Scheme. In W.G. Wojtkowski, Wita Wojtkowski, S. Wrycza, and J. Zupancic, editors, *Systems Development Methods for the Next Century*, pages 195–211. Plennum Publishing Corporation, New York, 1997. This is proceeding of isd conference 97 also available at http://www.cs.uwindsor.ca/users/c/cezeife/.

[GBLP96] 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.

[GHRU97] Himanshu Gupta, Venky Harinarayan, Anand Rajaraman, and Jeffrey Ullman. Index Selection for OLAP. In *International Conference on Data Engineering, Burmingham, U.K*, 1997.

[HRU96] Venky Harinarayan, Anand Rajaraman, and Jeffrey Ullman. Implementing Data Cubes Efficiently. In *ACM SIGMOD International Conference on Management of Data*, pages 205–216, June 1996.

[Inm96] W. H. Inmon. *Building the Data Warehouse.* John Wiley Symand[] Sons, Inc., second edition, 1996.

[MK96] M.E. Meredith and A. Khader. Divide and Aggregate: Designing Large Warehouses. *Database Programming and Design*, 9(6), June 1996.

[OV91] M.T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, 1991.

[TC96] Susan Tideman and Robert Chu. Building Efficient Data Warehouses: Understanding the Issues of Data Summarization and Partitioning. In *Proceedings of the Twenty-First Annual SAS Users Group International Conference, SUGI 21, Vol. 1, pp 520-527*, 1996.