ELSEVIER

Data & Knowledge Engineering 36 (2001) 185–210

DATA & KNOWLEDGE ENGINEERING

www.elsevier.com/locate/datak

Selecting and materializing horizontally partitioned warehouse views

C.I. Ezeife *

School of Computer Science, University of Windsor, Windsor, Ontario, Canada N9B 3P4

Received 17 May 1999; received in revised form 20 January 2000; accepted 19 April 2000

Abstract

Data warehouse views typically store large aggregate tables based on a subset of dimension attributes of the main data warehouse fact table. Aggregate views can be stored as 2^n subviews of a data cube with n attributes. Methods have been proposed for selecting only some of the data cube views to materialize in order to speed up query response time, accommodate storage space constraint and reduce warehouse maintenance cost. This paper proposes a method for selecting and materializing views, which selects and horizontally fragments a view, recomputes the size of the stored partitioned view while deciding further views to select. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Data warehouse; Views; Fragmentation; Performance benefit

1. Introduction

Decision support systems (DSS) used by business executives require analyzing snapshots of departmental databases over several periods of time. Departmental databases of the same organization (e.g., a bank) may be stored on different computer hardware and software platforms (e.g., checking database is stored on an Apple Macintosh PC system while the savings account database is stored on an IBM PC-based system). Differences may also exist in the underlying database management system used in each department (relational, object-oriented or straight flat files are all possibilities). The representation of database entities may be different as the checking database may represent a male bank customer named "John Andrew" as customer with customerid "0001" and gender "01" while the savings database may represent him as customer with customerid "c0001" and sex "M". To make the ongoing discussion easier to conceptualize, Fig. 1 shows two simple source databases for a bank. Both databases are relational but the first source database is used to handle savings accounts functions, which accepts customers' deposits and withdrawals to and from this account. The second source database handles checking accounts functions and is

* Tel.: +1-519-253-3000 ext. 3012.
E-mail address: cezeife@cs.uwindsor.ca (C.I. Ezeife).
Web: http://www.cs.uwindsor.ca/users/c/cezeife

0169-023X/01/$ - see front matter © 2001 Elsevier Science B.V. All rights reserved.
PII: S0169-023X(00)00044-6

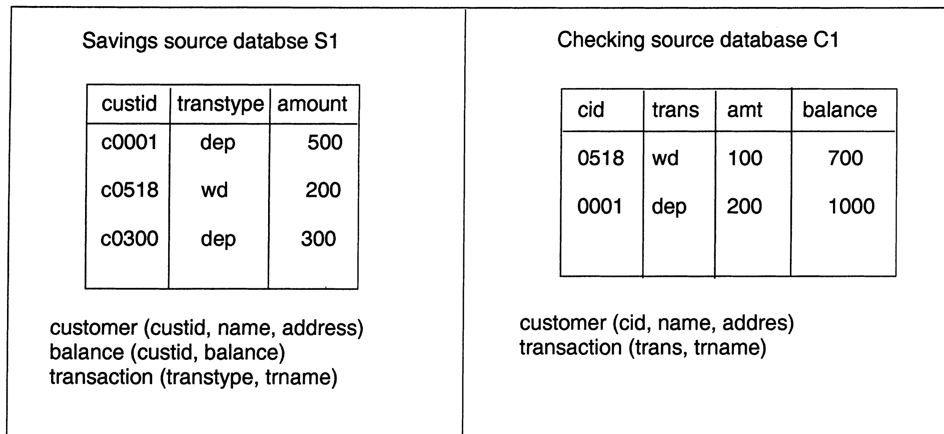

Fig. 1. Two banking source databases.

used to accept customers' deposits and withdrawals of cash to and from the checking account. The savings and checking databases record transactions (deposit or withdrawal) by customers on the appropriate account database every minute of the day. These function-oriented databases undergo frequent updates as a result of customers' transactions and the need to keep the account balances up-to-date and consistent. Simple queries adequately answered by these source databases include:

- What is customer c0001's savings account balance?
- Accept a deposit of $700 into customer 0518's account.
- How many accounts does John Andrew have?

If these are the only types of queries desired by the bank to support both customer and managerial services, then data warehouses are not needed because conventional databases would be adequate. The truth is that business management needs to find ways to gain competitive advantage and make more profit. These business goals could be achieved by learning about customers' interests and financial capabilities, learning which bank branches attract more customers, which customers across all accounts and branches bring in most cash to the bank and so on. To learn more about customers' financial strengths, business executives may be interested in posing queries that list the total daily balances at close of day, of all customers from across all branches in all accounts for over a period of a year. It becomes clear now that these types of complex queries cannot be answered with just the sample source databases alone because they both store only current data and previous account balances had been overwritten during transaction updates. Apart from the availability of historical recording of data, if we want to list the total balances of all customers from across all of the branches in all accounts, this query still requires that all the source databases in all branches and in all account departments be visited and some of the needed databases may not be available since they are being used by local transactions. Data warehousing systems provide a meaningful solution because they are used to store historical, integrated, "subject-oriented" and summarized data of an establishment [4,16,17].

A data warehousing system is a single data repository which integrates information from different data sources like relational databases, object-oriented databases, HTML files and others

[24]. A common relational data warehouse schema design is called the star schema. The star schema accommodates the main integrated data in a main warehouse table called the fact table. All attributes in the fact table apart from the main aggregation attributes are foreign keys. In addition to the fact table, the star schema includes dimension tables which define values for the dimension attributes (e.g., cname, ccity, cphone) functionally determined by the foreign keys in the fact table. The availability of such a historical, integrated, subject-oriented, non-volatile collection of data, called data warehousing [16] provides business decision makers with the tool to plan for the future. Front-ends to this core data warehouse include online analytical processing tools (OLAP), DSS, data mining applications and other custom-based querying applications [3]. Chaudhuri and Dayal [4] argue that data warehousing technologies have been successfully applied in many industries, including telecommunications, financial services, retails stores and health care. A simple warehouse fact table which integrates the checking and savings source databases presented as Fig. 1 is given below with its dimension tables. The C, A, R, T are acronyms standing for dimension attributes cid, acctcode, transtype and time-m, respectively, and these acronyms are used later to represent the table attributes.

Fact table is:
b-activity (cid, acctcode, transtype, time-m, Amount).
Dimension tables are:
customer (*cid*, cname, ccity, cphone),
account (*acctcode*, accttype, date-opened),
time (*time-m*, hour, day, month, year).

The fact table allows integration of all transactions in four types of accounts databases maintained by the bank, which are uniquely identified by the acctcode. The dimension hierarchies are defined from dimension tables and are used for roll-up and drill-down analysis as well as for describing foreign key attributes. A roll-up analysis progressively presents summarization from lowest level of detail to general level (e.g., computing total amount of money deposited every minute in a checking account, then total amount deposited every hour, then every day, month and year). On the other hand, a drill-down analysis computes an aggregate value first in most general level, then progressively presents it in more detailed levels.

An n-dimensional data cube in relational OLAP is a table with 2^n subviews of the data cube. The aggregates of the warehouse example above can be represented by a four-dimensional data cube with the 16 subviews labeled *CART*, *CAR*, *CAT*, *CRT*, *ART*, *CA*, *CR*, *CT*, *AR*, *AT*, *RT*, *C*, *A*, *R*, *T*, ( ). The subview labeled CART for the aggregate "total amount", is asking for the the total amount of money involved in transactions by each customer for each account, for each transaction type every minute. The subview labeled ( ) computes the total amount of money in all transactions in the fact table and corresponds to the following SQL query:

```
Create View ( )-trans AS.
Select Sum (Amount) AS TotalAmt.
From b-activity.
```

This means that while view CART is asking for total amount of money group by all of C, A, R, and T, the view ( ) is asking for total amount of money but not grouping by any attributes. Storing the data cube table for fast query processing may not produce acceptable results if all 2^n views are huge and stored (materialized). There is also an increase in the view maintenance cost

when all these views are stored. Harinarayanan et al. [15] proposes a greedy algorithm for selecting a set of subviews of the data cube most beneficial to materialize in order to reduce the time needed to answer the queries given some storage space. Meredith and Khader [19] argue that an approach for improving warehouse performance is aggregate view partitioning and no formal algorithms have been presented on warehouse view selection through partitioning except an initial preliminary study of the problem discussed in [9].

1.1. Related work

Gray et al. [13] uses the data cube model to generalize the SQL groupby construct such that computation of a multidimensional level of an aggregate measure is allowed. With the data cube model, an extended SQL construct can be used to create an n-dimensional data cube table consisting of the 2^n subtables for an aggregate measure like total amount. Since dimension attributes on both warehouse fact and main cube aggregate views are foreign keys (e.g., cid, acctcode), 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. 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. Many works have proposed techniques for efficiently computing the data cube and these include [1,15,22]. Harinarayanan [15] 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 as defined in [15] is given in Fig. 2. The benefit of selecting view v into set S, $B(v,S)$ is computed as the sum of all benefits (B_w) of descendant views w of v. The benefit of a descendant view w is the difference between the cardinality of view v and the smallest view u already in S that can compute w. If this difference is negative, then $B_w = 0$. Gupta et al. [14] extend the greedy algorithm to select both views and indexes. Ezeife [11] defines a uniform scheme based on a comprehensive cost model for selecting both views and indexes. In [12], this uniform scheme is extended to handle dimension hierarchies. Both Ceri et al. [8] and Ozsu and Valduriez [20] present horizontal fragmentation ideas and schemes for relational databases based on simple predicates but with no query access frequencies taken into consideration. Horizontal fragmentation of a view or relation is the partitioning of the view (relation) based on the values of its attributes such that each fragment contains only a subset of the tuples in this view or relation. Work including [16,17,19,23] have all expressed the need for data partitioning schemes in the data warehouse aggregate materialization problem. Issues concerning maintenance of materialized views have been addressed by Blakeley et al. [2], Colby et al. [5,6] and Kotidis and Roussopoulos

Algorithm 1.1 *(Greedy algorithm)*

```
S= {cube top view}
for i = 1 to k do begin
      select the view v not in S such that
      benefit B(v,S) is maximized;
      S = S ∪ {v}
end
resulting S is the greedy selection
```

Fig. 2. The greedy algorithm.

[18]. However, these view maintenance solutions are mostly relevant to non-fragmented materialized views. Some view maintenance issues that are relevant to horizontally fragmented, materialized views are raised and addressed in [26,25].

1.2. Motivations and contributions

Before the advent of data warehousing systems, business departments and/or branches built and maintained each application database on many different hardware and software platforms. Each branch needed to write their separate extraction program for populating data from source to historical version. Analyzing records of transaction at different branches over a period of a year would entail writing an application capable of visiting all of the branches' historical databases. The shortfalls of this type of integration are:

1. Any change in the operational environment of one branch's database calls for a modification of the decision support application.
2. The branch databases may be running on different hardware and software platforms, making the decision support applications very complex and unsuccessful.
3. Visiting several databases in the course of answering a query is inefficient because source databases may be in use by local transactions.

Data warehousing has emerged as a means for integrating various source databases residing as relational databases, flat files, news wires, HTML document and knowledge bases, for decision support querying [24]. OLAP queries are complex and volume of data is large making query response time, maintenance cost and disk space utilization important warehousing issues. Carefully selecting and efficiently partitioning warehouse views for materializing would improve query response time, maintenance cost and disk space utilization.

This paper contributes by first presenting an algorithm that iteratively selects a data warehouse view most beneficial at each stage, then, fragments this view horizontally, recomputes the size of the partitioned view and future view selections are done with the newly computed size. Secondly, a fragment-advisor component which collects information needed for calculating the sizes of selected partitioned views is presented. Finally, the results of experiments from this approach are discussed.

The method being proposed here, materializes every selected warehouse view as a set of its horizontal fragments all kept at one database site. The issue of distributing these fragments to different database sites is not the focus of this work. The original view that this approach fragments is computed from a non-fragmented warehouse fact table or data cube view. This approach aims at reducing query response time by reducing the number of rows of a view that is visited in order to answer the query through the view's fragments. Similarly, the approach reduces maintenance cost of a view through its fragments since only one fragment of the view is accessed for any maintenance operation (delete, insert or update) [26].

1.3. Outline of the paper

Section 2 presents a working example of the proposed technique based on a simple banking warehousing system demonstrating view selection, fragmentation, querying and size-recomputation. Section 3 presents the warehouse system design architecture displaying all software and data components of the design. Section 4 presents the suite of algorithms in the system design

architecture, which implement the proposed view selection and materialization technique. Section 5 discusses experimental performance analysis while Section 6 presents conclusions.

2. An example

A working example is used in this section to show how (1) a cube aggregate view can be horizontally fragmented, (2) warehouse query access pattern to horizontal fragments of an already selected materialized view is used to recompute the size of the view and (3) a set of fragments of a materialized view can be selected as the best for answering a warehouse query. The formal algorithms involved in these three processes are presented later in Section 4.

Example 2.1. A banking data warehouse stores historical, integrated records on every bank transaction that bank customers have executed in all four bank accounts available (savings1 (S1), savings2 (S2), checking1 (C1) and checking2 (C2)) from across many branches. □

The data warehouse has the following fact and dimension tables:
b-activity (cid, acctcode, transtype, time-m, amount),
customer (*cid*, cname, ccity, cphone),
account (*acctcode*, 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 Fig. 3 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:

Q_1: Get the number of customers who have made more than two withdrawals in savings account S1 in any month.

cid	acctcode	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

Fig. 3. Sample warehouse fact table data (same as view CART).

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 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 query Q_1 is composed of the following attributes and predicates:

PA = Acctcode (A).
AA = Month (T), transtype (R), cid (C).
MA = Number of customers, Count (C).
Predicates : P_1: $A =$ "S1".

Q_2: Get the number of customers who have deposited some money in the morning minutes. The attributes and predicate from Q_2 are:

PA = Time-m (T).
AA = none.
MA = Number of customers or Count(C).
Predicates: $P_2 : T \leqslant 0720$.

Q_3: Find the total amount of dollars involved in each transaction type and account code between the minutes of 0720 and 0780 (lunch hour) every day. We have from this query:

PA = Time-m (T).
AA = Acctcode (A) and transtype (R).
MA = Total amount of dollars or Sum (Amount).
Predicates: $P_3 : T \geqslant 0720$ and $T < 0780$.

Q_4: Find the total amount of dollars deposited by each customer every minute in account C1. From this query, we obtain:

PA = Acctcode (A).
AA = Customer (C), Time (T).
MA = Total amount of dollars or Sum (Amount).
Predicates: P_4: $A =$ "C1".

The warehouse cube lattice is given as Fig. 4 while its dimension hierarchies are defined in Fig. 5.

In a typical warehouse environment, the warehouse administrator may be faced with the decision to select a number of aggregate views from both the main cube level and dimension level views. The dimension level views are obtained through direct product (combined cube lattice) of the 2^n cube views and the dimension hierarchies. When a query uses a dimension attribute like customer name, hour, day or year, a join of a cube level view with the dimension hierarchy of interest is necessary to answer the query. To cut down on the cost of such huge

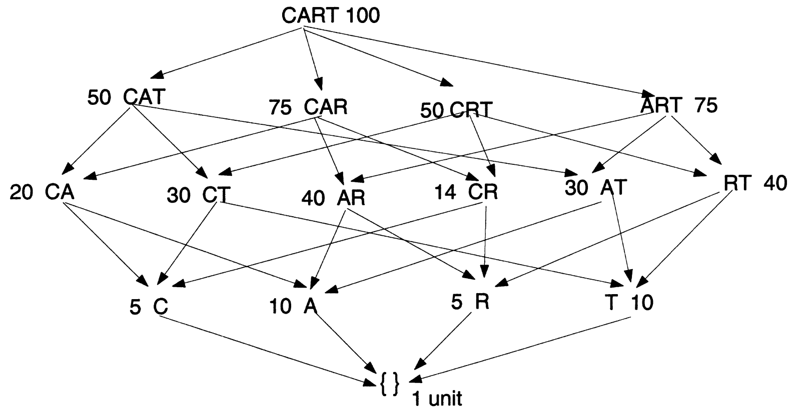

Fig. 4. The cube lattice of the CART warehouse.

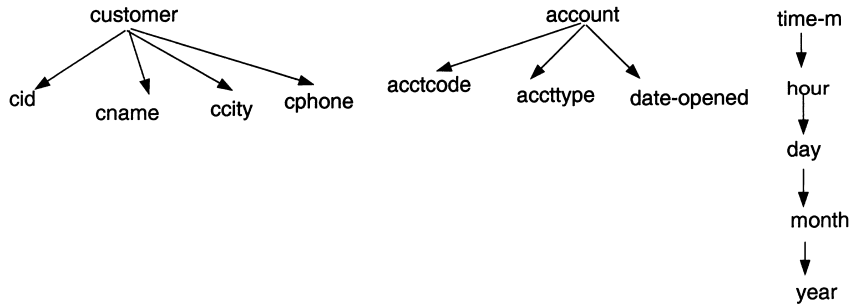

Fig. 5. Dimension hierarchies of the CART warehouse.

table joins, dimension level views may also be selected for materialization. Since the predicates from the queries have been defined, we attach some importance value (IP) to each predicate using the frequency of application access to them. The intuition here is that only some p most important predicates will be used in making decisions regarding fragmentation of a view. The IP or importance value of each predicate with respect to a view is obtained, by multiplying the cardinality of the predicate (number of rows from the view that are true for this predicate) when applied to the view by its application access frequency (the number of times the predicate is accessed by an application). If a predicate is accessed by more than one application, then 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–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 similar to view CART that, with respect to view CART, $|P_1| = 3$, $|P_2| = 6$, $|P_3| = 0$ and $|P_4| = 2$. Therefore, with respect to view CART, IP of $P_1 = 3 \times 100 = 300$, IP of $P_2 = 6 \times 40 = 240$, IP of $P_3 = 0 \times 20 = 0$, and IP of $P_4 = 120$. Note that in usual applications, a predicate may actually be accessed by more than one query or application accessing the same view.

Once the IP values for all predicates are defined, the current scheme simply selects p highest valued predicates, where p is defined by the warehouse administrator. Determining what constitutes an optimal number of fragments for each view would depend on a number of factors including the size of the view and the attributes in the selected predicates. Generally, p predicates would result in 2^p minterm fragments, some of which may be deleted to eliminate redundancies. Chakravarthy et al. [7] uses an enumerative algorithm to determine an optimal set of vertical fragments for a relation given a set of queries accessing its fragments and their access frequencies. The enumerative algorithm computes the performance of a number of vertical fragments of a relation as the value of an objective function called partition evaluator (PE). The PE measures the amount of local irrelevant and remote relevant accesses made to vertical fragments by queries. The enumerative algorithm proceeds by computing the PE value of the relation when it has only one fragment, then, it computes the PE value of the relation when it has two vertical fragments, then three and so on. The optimal number of vertical fragments most suitable for this relation given the same query workload come from the set of its vertical fragments which yields the lowest PE value. An extension of the PE measure for determining the performance of vertical fragments of object classes in a distributed object database system is also presented in [10].

This PE measure approach could be extended and used to obtain the number of fragments f of a view that yields best performance for a given set of queries. From the number of fragments, f, the optimal p predicate combinations is determined as $\log f / \log 2$ because p predicates create a maximum of 2^p fragments following the horizontal fragmentation process. This means that, given a list of all predicate combinations for fragmenting a view, the technique would first create all fragments for each p value of 1, 2, 3 predicates and so on. Then, it would calculate the PE value for each set of horizontal fragments arising from different p predicates. The optimal p number of predicates suitable for this environment is the one that yields the lowest PE value for its fragments. A PE value for a set of f horizontal fragments is an objective function that counts the total local irrelevant and remote relevant accesses made to fragments by queries while accessing each fragment of the view. Since for optimal fragmentation, it is desirable to minimize both the local irrelevant and remote relevant accesses to fragments, the set of fragments with lowest PE value yields best performance and the p predicates that created this set of fragments constitutes the optimal p predicates to choose for this warehouse view and queries setup.

However, since this computation is expensive, the warehouse administrator can simplify the technique by intuitively selecting p predicates based on the size of the view and the IP values of predicates. The bigger the table (view), the higher the p number of predicates that can be selected and since an p predicate would create a maximum of 2^p fragments of the view, an initial value of between two and five predicates for not more than 32 fragments of a view is recommended. Another factor that affects the choice of p is the number of predicates with high IP values. The more the number of predicates with high IP values, the higher the number p of them to be selected because this is an indication that many attributes are needed frequently by queries and this should be reflected in the partitioning process of the view.

Continuing with the example, if an p value of 2 is used, 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) or negated form. Thus, with the two selected simple predicates, the following minterm predicates are generated:

$$M_1 = P_1 \wedge P_2 \Longleftrightarrow A = \text{“}S1\text{”} \wedge T \leqslant 0720,$$

$$M_2 = \neg P_1 \wedge P_2 \Longleftrightarrow A \neq \text{“}S1\text{”} \wedge T \leqslant 0720,$$

$$M_3 = P_1 \wedge \neg P_2 \Longleftrightarrow A = \text{“}S1\text{”} \wedge T > 0720,$$

$$M_4 = \neg P_1 \wedge \neg P_2 \Longleftrightarrow A \neq \text{“}S1\text{”} \wedge T > 0720.$$

Since all minterms are meaningful with respect to domains of applications, they form fragments of any view submitted 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 needs to be accounted for in only one fragment. The fragmentation scheme presented in this paper, uses both the access frequency and selectivity of a predicate to determine its importance. For fragmentation of a view, a set of p important predicates from user queries are used. A set of predicates is complete when both its natural and negated forms are used in defining its minterm predicates [20] as is the case with the p predicates employed in this work. Predicates used for fragmentation are also required to be minimal or relevant, and selecting only p most important valued predicates for fragmentation purposes in this work guarantees that only predicates relevant to the view in question are used in the fragmentation process. Thus, fragments created are complete because they come from a complete and minimal set of predicates. The fragments are also non-overlapping because the defined minterms are mutually exclusive.

The example above further serves to demonstrate 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 top level view CART in Fig. 3 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 10 tuples in the table. For partitioning views other than the top level view, all simple predicates are reapplied 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. For example, the predicates P_1–P_4, when reapplied to view CR (shown as Fig. 6) for the purposes of partitioning view CR, yields a cardinality of zero for all these predicates. This means that only predicates that are based on attributes C which is cid, and R (transtype) would yield non-zero

cid	transtype	Amount
C0001	dep	700
C0001	wd	600
C0300	dep	300
C0411	dep	800
C0518	dep	300
C0518	wd	700
C1000	wd	400

Fig. 6. View CR created from view CART.

cardinalities with respect to this view CR and these predicates would be among the selected ones if they have high enough IP values.

The first view selected is the top level view following the greedy approach [15]. Thus, view CART is selected and four horizontal fragments of this view are defined by our approach as explained next.

A main advantage of breaking a table or view into its horizontal fragments is drastic reduction in query response time because the average number of rows visited by queries through its horizontal fragment(s) is lower. recomputing the size of a selected fragmented view, requires computation of the average number of rows of this view accessed by queries through fragments. This average number of rows of the view accessed by all queries through fragments is computed as the sum of the products of the total number of rows in all fragments accessed by each query and the query access frequency, divided by the sum of access frequencies of all queries accessing this view. Thus, with our example warehouse and queries, Fig. 7 shows the fragments of the view CART, number of rows, frequency and total number of rows accessed by each of the queries Q_1–Q_4.

Once the new size of a selected fragmented view is computed using the method discussed in previous paragraph, future selection of other views are accomplished with the greedy algorithm but using the newly computed size of the view. Every selected view is in turn horizontally fragmented. Clearly, this reduces the average number of rows visited by queries and improves on query response time if queries are accurately directed to fragments that can adequately address their needs. This paper also provides an algorithm component called fragment-advisor, which is responsible for recommending the set of fragments of a selected and fragmented warehouse view that best answers a given warehouse query. The fragment-advisor selects all views from the materialized view set that can answer the query using queries' group-by attributes, measure attributes and analysis attributes. The view with the least total number of rows in all needed fragments is the best. The fragments of each view are read in as their minterm predicates. A fragment of a view is marked as needed by a query, if the conjunction of all predicates of the query is a subset of the minterm predicate of the fragment. For example, with view CART of the running example, Fig. 8 shows the minterm predicates of its four fragments with their sizes. The predicates for each query and the minterms they are part of are given in Fig. 9. A query predicate is a subset of every minterm predicate that it needs to completely answer it.

Query (1)	predicates (2)	Fragments accessed (3)	Number of rows from fragments (4)	Access frequency (5)	Product (6) = (4) * (5)
Q1	P1	F1, F3	3	100	300
Q2	P2	F1, F2	6	40	240
Q3	P3	F3, F4	4	20	80
Q4	P4	F2, F4	7	60	140
Totals				220	760

New Size of View = (6)/(5) = 760/220 = 3.45 or ceiling (3.45) = 4

Fig. 7. Fragments of view CART accessed by queries.

Materialized view	Fragment	Minterm Predicates	Number of rows
CART	F1	A = "S1" ^ (T <= 0720)	1
	F2	A = ~"S1"^(T <= 0720)	5
	F3	A = "S1"^ ~(T<= 0720)	2
	F4	A = ~"S1"^~(T<=0720)	2

Fig. 8. Minterms of fragments of view CART.

Query	Predicate (P)	Minterm fragments P is a subset of these minterms
Q1	P1: A ="S1"	F1, F3
Q2	P2: T<=0720	F1, F2
Q3	P3: T >= 0720 And T < 0780	F3, F4
Q4	P4: A = "C1"	F2, F4

Fig. 9. Minterm fragments that query predicates are part of.

3. Warehouse system design architecture

The objective of this section is to present a block architecture of the warehouse design technique being proposed in this paper. The block architecture which highlights the software and data components of the design is given as Fig. 10. Some definitions relevant for understanding the algorithms in Section 4 are also presented in this section.

From Fig. 10, it can be seen that the original data warehouse is made up of fact and dimension tables in addition to all aggregate or summary tables. However, this work introduces two major algorithms namely:

1. The selection–partition algorithm which selects only the best views to materialize while horizontally fragmenting every selected view and also recomputing its size. The information needed for recomputation of sizes of selected views include fragments of each view needed by warehouse queries. The second algorithm called the fragment-advisor is responsible for determining which fragments of a warehouse data cube view best answers any given warehouse query. Thus, obtaining the total number of rows of tables searched in order to answer a number of warehouse queries entails getting the sum of rows in all fragments of views needed by these queries. From this total, the average number of rows of the view visited by queries is computed as the total number of rows divided by the total number of accesses made by all queries accessing the view. The output of this scheme is a smaller warehouse with fewer summary tables.
2. The fragment-advisor algorithm interfaces between warehouse querying or access tools and the warehouse. The purpose of this algorithm is to predetermine which fragments of a view are needed to answer a warehouse query.

The overall output of warehouse queries is the same except that results of queries are turned out faster. It is important that the fragment-advisor algorithm does not carry much execution time overhead since this algorithm cannot be run off-line as the selection–partition scheme could. However, the fragment-advisor is a polynomial time algorithm and does not add huge overhead.

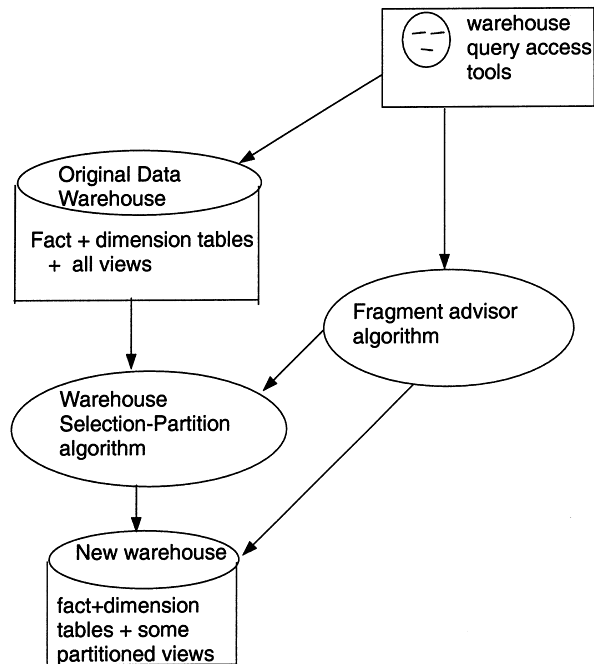

Fig. 10. A warehouse system design architecture.

3.1. Definitions

Definition 3.1. 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 3.2. An analysis attribute AA_{ij} from a user query Q_i represents the subjects or groupby attributes of interest to the application.

Definition 3.3. 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 3.4. A measure attribute MA_{ij} in a user query Q_i is an aggregation attribute of interest to the application.

Definition 3.5. All Attributes of a query (A_q) is the union of all the AA, PA and the non-aggregate part of MA.

Definition 3.6. Definition 3.7 All Attributes of a view (A_v) is the set of all groupby attributes that define the view v. For example, A_v of view CAR is $C,\ A,\ R$.

Definition 3.7. Possible view set (P_v) for answering a query is the set of views among the materialized views that can each adequately answer the query.

4. Formal warehouse design algorithms

Section 4 makes a formal presentation of both the selection–partition and fragment-advisor algorithms shown in the system architecture. While Section 4.1 discusses components of the selection–partition scheme, Section 4.2 discusses an example execution of this scheme, Section 4.3 presents size-recomputation scheme and Section 4.4 discusses the fragment-advisor scheme.

4.1. The selection–partition scheme

The selection–partition scheme is an algorithm which selects n best views to materialize using a modified version of the greedy algorithm presented in [15]. This algorithm begins by first selecting the top level view from the data cube lattice. Then, every selected view is horizontally fragmented using the function *view-partition*. Next, the size of the selected, horizontally fragmented view is recomputed using the function *size-recompute*. The new size of the view is used when computing the benefits of all views relative to already selected, partitioned views.

Input to the process is a set of warehouse queries and their frequencies 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: Find simple predicates from each user query Q_i, using the partition attributes PA_{ij}. The simple predicates are defined from the partition attributes as $\mathrm{PA}_{ij}\theta$ value, where θ is a logical operator from the set $\{=, <, >, \neq, \leqslant, \geqslant\}$ and value is from domain of partition attribute (PA_{ij}), the jth partition attribute of the ith query.

Step 2: Define the relative IP of each predicate P_{ik} of query Q_i. The importance of each predicate is obtained by adding up the product of the application frequency and the cardinality of this predicate on the view for every application that accesses the predicate. The formula for obtaining the importance of a predicate is given as

$$\sum_{q \in Q_i | \mathrm{access}(Q_i, P_{ik}) = 1} (\text{access frequency of } (q) * |P_{ik}|),$$

where $|P_{ik}|$ is the number of rows or tuples in the partition of the view defined by predicate P_{ik}, while $\mathrm{access}(Q_i, P_{ik}) = 1$ means that the query Q_i accesses the predicate P_{ik} a number of times similar to the given access frequency of Q_i.

Step 3: Select the p most important predicates. The current scheme selects p highest valued predicates. This step can be refined or optimized in the future to let the scheme decide what p 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 [20]. Thus, given an aggregate view V_i with the set of selected simple predicates $\mathrm{Pr}_i = \{P_{i1}, P_{i2}, \ldots, P_{ik}\}$ 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

Algorithm 4.1 *(Selection_Partition - Selects n views to partition and materialize)*

```
Algorithm Selection-Partition()
Input:      number of views to select (n), cube lattice with view sizes,
        set of queries {Q_m}, access frequency of queries {AF_m}.
Output: Set of fragments of n selected views V_i {V_i^j}
begin
SetV = {cube top view}
Selected = cube top view
numberselected = 1
while numberselected <= n do
        begin
                View-Partition(Selected)
                Size-Recompute(Selected)
                for each subview (s) ∉ SetV views do
                begin
                        Compute the greedy benefit of keeping s using new sizes
                        of partitioned parent views
                end
                SetV = SetV ∪ s with maximum benefit
                Selected = s
                numberselected = numberselected + 1
        end // of while numberselected //
end // of Selection-Partition //
```

Fig. 11. The selection–partitioning algorithm.

$$M_{ij} = \bigwedge_{p_{ik} \in \mathrm{Pr}_i} P_{ik}^*, \quad 1 \leqslant k \leqslant m, \quad 1 \leqslant j \leqslant z,$$

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 domains 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 selection–partition is given as Fig. 11. The selection–partition algorithm calls the function that partitions a selected view which is formally presented as algorithm view-partition in Fig. 12. The function for size-recomputation also called by algorithm selection–partition is discussed formally in a subsection after an example of view selection and partitioning. Once a view is selected into *SetV*, partitioned and its size recomputed, the benefit of all views in the lattice not yet members of the set SetV are recalculated using the new parent sizes. The process continues until the needed n views have been selected into SetV. The benefit of a view is defined the same way as the benefit used by the greedy algorithm except that the new size of the parent view is applied. During each iteration, the view with maximum benefit is selected.

4.2. An example selection with the selection–partition scheme

Assume we want to select three views from the example CART data warehouse in Section 2, the cube lattice with view sizes is shown as Fig. 13.

The original sizes of the views are shown beside the views while the recomputed size is shown instead if the original size of the view is cancelled. When the size of a view is inside a circle, it

Algorithm 4.2 *(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^m from query Q_m
       Pr_i = Pr_i ∪ Pr^m
       // Now define the importance (IP) of these new predicates //
       for each Pr_m^j in Pr^m
              IP of Pr_m^j = IP of Pr_m^j + (AF_m * |Pr_m^j|)
end // of for m //
Select p predicates with highest IP values from Pr_i
//Generate minterm predicates as follows. //
// Note that k marks which predicate. //
// Initialize. //
numberofminterms = 1
numberofpredicates = 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 numberofminterms
              mcopy_i = m_i
       end
       for i = 1 to numberofminterms
              m_i = mcopy_i ∧ P_k
       end
// Now conjunct the negative form of predicate to new minterms //
       for i = 1 to numberofminterms
       begin
              actualmintermnumbernow = numberofminterms + 1
              m_actualmintermnumbernow = mcopy_i ∧ ¬P_k
       end
       numberofminterms = actualmintermnumbernow
       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}
```

Fig. 12. The view partitioning algorithm.

indicates that this view is selected, fragmented and the size-recomputed but the size remains the same. One reason for a recomputation of a view size yielding the same value as its original size is the process of fragmenting the view results in only one fragment. Fig. 14 shows the process of selecting the three views with the greedy algorithm using the partitioned and recomputed sizes of the selected views. Recall that from Section 3, the recomputed size of CART is 4 out of 10 or 40

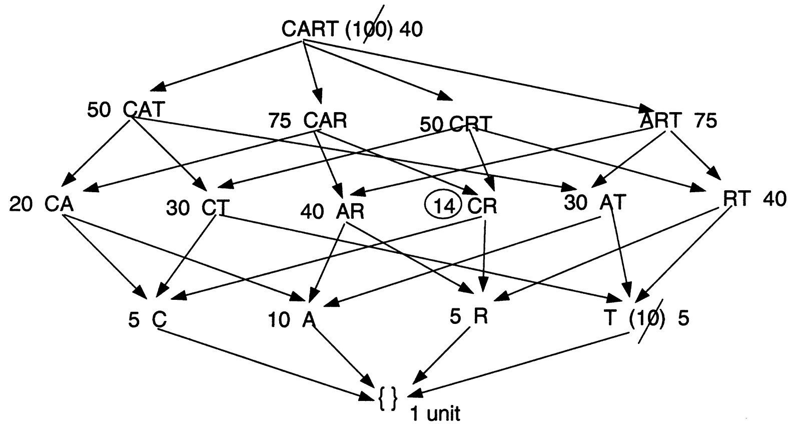

Fig. 13. An example cube lattice with sizes recomputed.

Views	Choice1	Choice2
CAT	0	0
CAR	0	0
CRT	0	0
ART	0	0
CA	20*4=80	(20*3)+(0)= 60
CT	10*4=40	(10*3)+(0)= 30
AR	0*4=0	0
CR	26*4=104 *	
AT	10*4=40	(10*3)+(0)=30
RT	0*4=0	0
C	35*2=70	9*2=18
A	30*2=60	4*2=8
R	35*2=70	9*2=18
T	30*2=60	30*2=60 *
{}	39	13

Fig. 14. Example view selection using partitioned parent views.

out of 100. The view CART is the top level view from the cube lattice and is selected into SetV. [1] Then, the recomputed size of CART is used to choose the next view from the lattice by computing the benefits of each view. The benefit of a view, v not in the set SetV is computed as the sum of all positive differences between the sizes of the smallest parent view u (in SetV) of each v's descendant view w, and v. Thus, for the view CR, the smallest parent in SetV is CART with size 40. The descendants of CR are four views (CR, C, R and ( )) including itself. Thus, the benefit of CR is computed as $4(40 - 14)$ since the size of CR is 14. The view CR with the highest benefit is selected this first time and included in SetV.

Next, we partition the selected view CR by first computing the IP of the predicates. The IP value of a query predicate is the product of the cardinality of this predicate on the selected data cube view that answers the query, and the query access frequency. A given query can be answered

[1] The asterisk beside a view in Fig. 14 during any selection iteration, indicates that this view is the chosen one.

by one or more warehouse cube views. This implies that predicates from one query can yield non-zero IP values on any view that computes it and thus, can compete in the choice of the most important p predicates for partitioning the view. However, before a view is horizontally partitioned, it has been selected by the greedy algorithm using the newly computed sizes of its parents. This means that only those queries that this selected view can compute will have their predicates competing for the best p predicates to be used to partition the view. Since the cardinalities of all four predicates when applied to view CR is 0, their IP values are also 0. Thus, no predicates are selected for defining horizontal fragments of this view and no fragments are defined making the size of CR the same. Since the second selected view *CR* cannot be partitioned, the next step is to find the benefits of all views again which leads to selection of view T with highest benefit of 60. For example, in computing the benefit of view CA, this time when both view CART and CR are already in the SetV, both descendant views CA, C and A can be computed with CA which has a lower size yielding a total benefit of $3(40 - 20)$, but descendant view ( ) can be computed with view CR of lower size and thus contributes a benefit of 0 to CA. Note that the selected view in this case could also have been CA since it also has a benefit that is equal to the highest. Checking the IP values selects $P_2 : T \leqslant 0720$ as the only predicate to be used for fragmenting view T. Thus, the two fragments defined for view T are $T \leqslant 0720$ and $T > 0720$. The recomputed size of this view is 5.

4.3. The size-recomputation scheme

The algorithm for recomputing the size of a view basically recomputes the new size of a view as the average number of rows of the view accessed by all queries through its fragments, taking into account all query access frequencies. The formula that implements the size-recomputation is given below and the algorithm size-recompute is given as Fig. 15

Algorithm 4.3 *(Size-Recompute - Computes the new size of a selected partitioned parent view)*

Algorithm Size-Recompute(V_i)
input: set of queries $\{Q_m\}$ and their predicates ; access frequencies of queries $\{AF_m\}$, set of fragments of F_m of V_i
Output: New size of view V_i

```
begin
    querytotalaccess = 0
    totalaccessfrequency = 0
    for each q ∈ Q_m
    begin
        for each f ∈ F_m do
            begin
            querytotalaccess = (|q| on f * AF_q) + querytotalaccess
            if |q| on f ≠ 0
            then
            totalaccessfrequency = totalaccessfrequency + AF_q
        end // of for each f //
    end // of for each q //
    Newsize of V_i = querytotalaccess/totalaccessfrequency
end
```

Fig. 15. The view size-recomputation algorithm.

$$\text{New view size} = \sum_{\text{query}\, q_i | \text{access}(q_i, v) = 1} \frac{\sum_{\text{frag } f_m | \text{access}(q_i, f_m) = 1} (|f_m| \ * \ AF_{q_i})}{\sum AF_{q_i}}.$$

The measure of new size of a view is defined in terms of the average number of rows of the view accessed by all queries, because this measure most accurately reflects the number of rows of the view actually accessed by a query taking into account all the access frequencies of queries and only those fragments of the view accessed. An alternative measure is to use the maximum number of rows of the view accessed by any one query and this may hide the utility of accessing only fragments and not the entire view. On the other hand, the minimum number of rows accessed by any one query could be used at the risk of exaggerating the gain of the approach. The algorithm accepts as its input data, the set of warehouse queries. Each query's input also includes its predicates and fragments of the view in question it accesses. Fig. 7 from Section 2 illustrates this procedure. From the given input data, the total number of rows each query accesses from all its fragments is computed. The row count information can easily be obtained with an SQL query that selects all rows from each fragment. For example, from Fig. 7, query Q_1 has predicate P_1 and accesses fragments F_1 and F_3 and we can obtain the total number of rows accessed by Q_1 through these two fragments with the SQL query:

```
Select *
from CART_F1
where P1

Union

Select *
from CART_F3
where P1
```

The product of the total number of rows accessed and the access frequency of the query is computed. The sum of all accesses made by all queries accessing the view is obtained and divided by the total of access frequencies of all these queries. The ceiling of this value gives the new size of the view.

4.4. The fragment-advisor algorithm

The fragment-advisor algorithm takes a warehouse query and from its predicates, it recommends the fragments of a view that produces fastest response time for the query. This algorithm performs run-time analysis that makes maximal use of already statically defined and materialized view fragments. For applications with frequent radical changes in access patterns of queries, it will be even more beneficial to provide techniques for determining when a set of horizontal fragments of a materialized view no longer represents optimal set due to changes in queries and their frequencies and to trigger a dynamic refragmentation of the view. However, the work presented here performs only static fragmentation of views and assumes that sufficient changes will call for a static refragmentation of the views.

The sequence of steps to execute in order to find the set of fragments of a view which best answer a query are discussed next. Input to the scheme are PA, AA, MA and set of predicates Pr_q

of the query as well as the set of materialized views V with their fragments. Each fragment is input as the conjunctive minterm predicate that defines it and the size of the fragment which is the number of rows of this view for which the minterm predicate is true. The steps in the scheme are:

Step 1: Define the set of possible views P that can be used to adequately answer this query. To get the possible view set, we first define all attributes needed by the query by concatenating the query's PA, AA and MA sets. Thus, $A_q = \mathrm{PA}\|\mathrm{AA}\|\mathrm{MA}$. Then, for every view, v, in the set of materialized views, if the set of all attributes of the query A_q is a subset of the set of all attributes of the view A_v, v is made a member of the possible view set. In other words, if all attributes that play a role in the given query are concatenated, and this set of query attributes happens to be a subset of all attributes that define any given view, then, this view can be used to compute the query. For example, give the query "Get all customers who have deposited some money in the morning minutes", the all query attributes, A_q is the concatenation of PA (T), AA (none) and MA (C from Count(C)). This means that A_q is CT, and a listing of all data cube views from this particular warehouse setup that form superset of CT is CART, CAT, CRT and CT. However, only those views in this set that have been selected for materialization and already horizontally partitioned are in this query's possible view set. From the example selection and partition of Section 2, only

Algorithm 4.4 *(Fragment advisor - Recommends a set of fragments of a view for a query)*

```
Algorithm Fragment-advisor(V_i)
input:    set of queries {Q_m} as PA, AA, MA, and their predicates ;
          set of fragments F_m of materialized views V_i, minterms of each fragment and its cardinality.
Output:   A set of fragments F_m of a view
begin
Aq = PA||AA||MA
// Define the possible view set as follows: //
Possibleviewset = {}
for each view v in V do
      begin
      if attribute set A_q ⊆ A_v
            then Possibleviewset = Possibleviewset ∪ v
      end
Smallest-total-row = 0
for each view v in Possibleviewset do
      begin
            Needed-Fragments of v = {}
            Number-of-rows for v = 0
            for each fragment F of v do
            begin
            if the conjunction of all predicates Pr of query ⊆ minterm of F
                  then Need-Fragments of v = Needed-Fragments of v ∪ F
                  Number-of-rows for v = Number-of-rows for v + |F|
            end
            if Smallest-total-row > Number-of-rows for v
                  then Smallest-total-row = Number-of-rows for v
            whichview = v
      end
Recommended = Need-Fragments of v
end end
```

Fig. 16. The fragment-advisor algorithm.

the views CART, CR and T are materialized, making the possible view set of the query above only CART.

Step 2: Once we have defined the possible view set, the next step is to determine which of these competing views should be selected to answer the query. Intuitively, the chosen view is the one that requires scanning of fewest rows in order to answer the query. The number of rows of a view scanned can be determined as the sum of the cardinalities of all its fragments that need to be visited in order to answer the query. This means that the view which requires only some of its fragments to answer the query and with lowest total number of rows for answering the query, is the selected view and fragments. Thus, the scheme selects a view $v_j \in P$ and the set F_{ij} of fragments of v_j such that the total number of rows in all its fragments needed by the query is the minimum. The formal algorithmic definition of this solution is given as Fig. 16.

5. Experimental performance analysis

An experiment was conducted using a banking warehouse database stored on an SGI Oracle. The experiment was conducted on a general purpose computer system in the university called SGI Challenge XL. The SGI Challenge XL has 16 R4400 processors with 12 processors at 150 MHz, 2 processors at 200 MHz and 2 processors at 250 MHz. An enterprise Oracle DBMS version 7.1 runs on this computer system. A data warehouse fact table with a million tuples was created for generating the 16 needed data cube views. However, this version of Oracle did not allow creation of some views that require grouping by more than two attributes from the one million tuple table (e.g., views CART, CAR, CAT and CRT). Faced with this limitation, we reduced the size of the experimental fact table to 590 tuples to enable creation of all 16 data cube views to be selected and partitioned for the experiment. The warehouse fact table has the following structure corresponding to four dimension attributes (CART):

```
cid number not null (customer ID),
acc character (15) (account code),
trans character (15) (transaction type),
time character (15) (transaction time),
amount number (transaction amount),
size of fact table : 590 rows.
```

From the 590 rows of the warehouse fact table, we defined 16 data cube views by running the SQL select instructions group by appropriate attributes to represent the views and the sizes of the views are given in Fig. 17. A variety of 20 queries are used for the experiment to cover many types of predicates. Fig. 18 describes the query workload used for the experiment. The queries are described in terms of their PA, AA, MA, predicates, cardinality of their predicates and their access frequencies. Several runs of the selection–partition scheme was performed at different system times and the average CPU times of the runs are reported.

The algorithms presented in Section 4 as well as the greedy algorithm were implemented. This scheme was run on the data cube with the sample query workload and compared with a run of the greedy algorithm on the same query workload. The four views selected by the two selection schemes are shown in Fig. 19. The selection–partition scheme also stores the horizontal fragments of selected views.

Views	Size
CART	590
CAT	590
CAR	191
CRT	590
ART	590
CA	40
CT	487
AR	20
CR	50
AT	590
RT	590
C	10
A	4
R	5
T	487
{}	1

Fig. 17. The warehouse view sizes.

Query	PA	AA	MA	Predicate	Pred cardinality	Frequency	IP value
Q1	A	C,R	sum(amt)	P1:"A=S1"	151	20	3020
Q2	A	C,R	sum(amt)	P2:"A=S2"	167	50	8350
Q3	R	A	count(C)	P3:R="wd"	112	40	4480
Q4	T	none	count(C)	P4:T<0480	59	30	1770
Q5	R	C,A	sum(amt)	P5:R="tr"	103	45	4635
Q6	R	C	sum(amt)	P6:R="dep"	160	60	9600
Q7	A	none	count(C)	P7:A="C1" OR "C2	272	25	6800
Q8	R	C	count(C)	P6:R="dep"	160	60	9600
Q9	A	C,R	sum(amt)	P8:A="C1"	122	55	6710
Q10	R	C	count(trans)	P6:R="dep"	160	35	5600
Q11	T	C, R	sum(amt)	P9:T<0480	59	70	4130
Q12	A	C,T	sum(amt)	P10:A="C1"	122	65	7930
Q13	R	C,T	sum(amt)	P6:R="dep"	160	50	8000
Q14	R	C	count(*)	P6:R="dep"	160	45	7200
Q15	R,A	C	sum(amt)	P11:R="dep" and A="S1"	100 38	50	5000
Q16	T	R,A	sum(amt)	P12:T<0780 and T>0720	38	45	1520
Q17	T,R	C,A	sum(amt)	P12	38	45	1710
Q18	R,A	T,C	count(C)	P13:R="wd" and A="S1"	22	75	1650
Q19	R	T,C	sum(amt)	P14:R="bpm"	112	55	6160
Q20	R,A	C	sum(amt)	P15:R="bpm" and A="S1"	29	25	725

Fig. 18. The experimental query workload.

Greedy algorithm	Selection-Partition Scheme
CART	CART
CAR	CAR
AR	AR
CT	CA

Fig. 19. Views selected by two approaches.

Query	Greedy algorithm	Selection-Partition Scheme
Q1	0.03	0.02
Q2	0.03	0.03
Q3	0.01	0.01
Q4	0.02	0.02
Q5	0.03	0.03
Q6	0.03	0.03
Q7	0.03	0.01
Q8	0.03	0.03
Q9	0.03	0.02
Q10	0.02	0.02
Q11	0.04	0.04
Q12	0.07	0.06
Q13	0.1	0.09
Q14	0.03	0.02
Q15	0.02	0.01
Q16	0.04	0.04
Q17	0.03	0.03
Q18	0.05	0.03
Q19	0.04	0.04
Q20	0.02	0.01
Total	0.7	0.59
Average	0.035	0.0295

Fig. 20. Comparative response times for two approaches.

We monitored and collected the CPU query response times of the 20 queries on the four views selected with greedy algorithm and compared them with the response times of same queries on the four selected, partitioned views obtained using the selection–partition scheme and Fig. 20 shows the response times in seconds while Fig. 21 shows the graphical representation of this comparison. This example represents a 16% improvement over the average response time achieved with the straight greedy algorithm.

Scaling the size of the warehouse by a factor of about one million (1 M) to represent more real life situation will increase the difference in the benefit of this approach. The size of data used for the experiment although reasonably representative is limited by resource constraints.

5.1. Complexities of the algorithms

If v represents the number of views in the cube lattice and m the number of views to select and partition, the computation time of the selection–partition scheme is $m * v *$ maximum (computation time of view-partion algorithm, computation time of size-recompute algorithm). Furthermore, let the number of queries accessing the system be q and let each query have a maximum of x predicates, p represents the number of predicates with highest IP values selected for fragmentation, and t the maximum possible number of minterm predicates. The computation time for view-partition is the maximum of xq and pt. If the maximum number of fragments possible in a view is f, then the computation time for size-recompute algorithm is fq. This means that the selection–partition scheme is of time complexity $O(xmvq + mvxt + mvfq)$. The time complexity of the straight greedy algorithm executed iteratively and in terms of the same variables is $O(mv)$. The

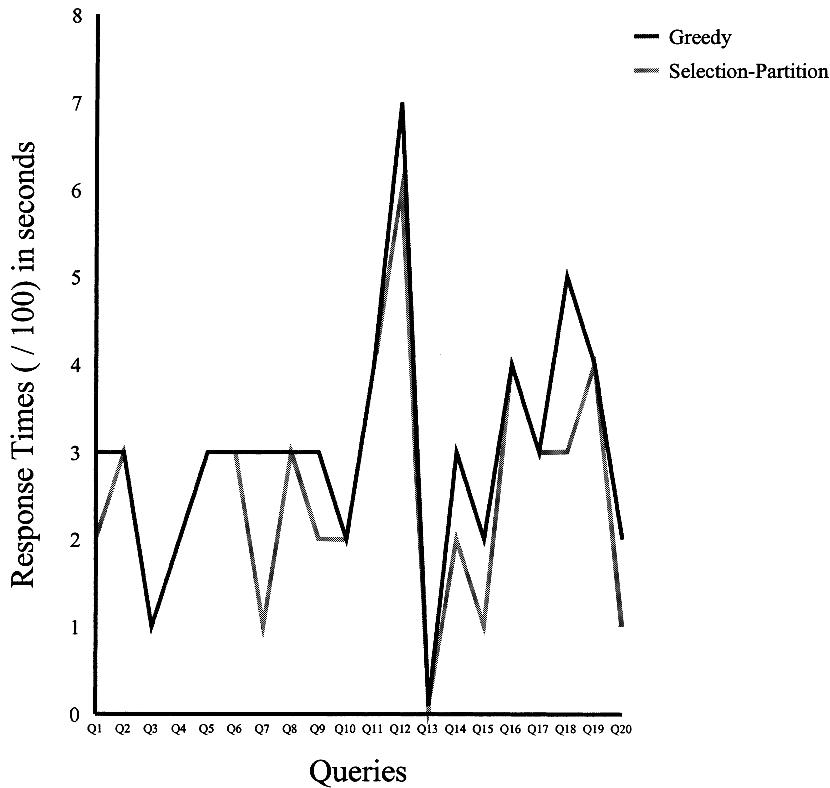

Fig. 21. Graph for comparing response times.

complexity of the fragment-advisor algorithm is $O(vf)$. These are polynomial time algorithms and the variables x, m and f are usually not large numbers. In running these programs the execution times we notice are negligible in agreement with these computation times.

5.2. Possible extensions to the basic model

The basic selection–partition scheme requires the database administrator to determine how many highest valued predicates to select for defining the horizontal fragments of selected views. The examples and experiments are run on mainly cube level views. Real life environment would require considering both dimension views and indexes as indexes can drastically reduce the number of page I/O operations [21].

Dimension views and indexes could be assigned higher type weights which are used to raise the IP values and thus increase their chances of being selected. A frequently used dimension view cuts down on the cost of joining huge tables and can also be used for drill-down and roll-up analysis. While a basic cube view could be assigned a type weight of 1, dimension view could be assigned an additional 0.5 weight for every dimension join attribute that is part of it. Only indexes of selected views are materialized and indexes should be defined for each partition of the view. It is possible to consider storing only indexes of frequently used partitions of the view. There is benefit in dynamically repartitioning views when application access patterns change sufficiently and future

work should provide this extension. Extending this approach to distributed warehouse environment where horizontal fragments of views are allocated to distributed sites is an interesting research issue that will benefit distributed applications.

On maintenance of partitioned-stored views, only relevant fragments of the view are consulted for updates, insertions and deletions. The fragment-advisor algorithm can serve to identify the fragments that need to be visited. A parallel view maintenance expression can be developed to run concurrently on fragments of a view.

The size of data and view could be increased in further experiments to observe changes in performance. The access frequencies, PA, AA and MA attributes of the test queries could be changed to collect more results and the experiments can be run on different warehouse setups.

6. Conclusions and future work

Partitioning of stored views leads to some improvement in system performance because of reduced query response time and maintenance cost since most queries will indeed scan fewer fragments than all, and in turn scan fewer rows than are stored in the original view. The query response time is reduced because only a fraction f of all 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.

This paper contributes by proposing an algorithm that selects and materializes warehouse views as their horizontal fragments, recomputes their sizes for better selection of future views. A fragment-advisor component is included which recommends the fragments of a view most suitable for answering a query. An experimental study comparing the results of this approach with those of the greedy algorithm is conducted and reported.

Future work in this direction should include accommodating indexes, defining maintenance expressions for partitioned, stored views, dynamic refragmentation of selected views when query access information change enough and as fragments are maintained to include new records.

Acknowledgements

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.

References

[1] S. Agrawal, R. Agrawal, P.M. Deshpande, A. Gupta, J.F. Naughton, R. Ramkrishna, S. Sarawagi, On the computation of multidimensional aggregates, in: Proceedings of the 22nd International VLDB Conference, 1996.
[2] J.A. Blakeley, P.A. Larson, F.W. Tompa, Efficiently updating materialized views, in: Proceedings of the ACM SIGMOD International Conference, May 1986, pp. 61–71.
[3] A. Berson, S. Smith, Data Warehousing, Data Mining and OLAP, McGraw-Hill, New York, 1997.
[4] S. Chaudhuri, U. Dayal, An overview of data warehousing and OLAP technology, in: Sigmod Record, vol. 26, 1, March 1997.
[5] L.S. Colby, A. Kawaguchi, D.F. Lieuwen, I.S. Mumick, K.A. Ross, Algorithms for deferred view maintenance, in: Proceedings of the ACM SIGMOD International Conference, Montreal, Canada, June 1996, pp. 469–480.

[6] L.S. Colby, A. Kawaguchi, D.F. Lieuwen, I.S. Mumick, K.A. Ross, Supporting multiple view maintenance policies, in: Proceedings of the ACM SIGMOD International Conference, Tucson, USA, May 1997, pp. 405–416.
[7] S. Chakravarthy, J. Muthuraj, R. Varadarajan, S.B. Navathe, An objective function for vertically partitioning relations in distributed databases and its analysis, Distributed and Parallel Databases 2 (1) (1993) 183–207.
[8] S. Ceri, S. Navathe, G. Wiederhold, Distributed design of logical database schemas, IEEE Transactions on Software Engeneering 9 (4) (1983).
[9] C.I. Ezeife, S.R. Baksh, A partition-selection scheme for warehouse aggregate views, in: Proceeding of the Ninth International Conference on Computing and Information, Canada, July 1998.
[10] C.I. Ezeife, K. Barker, Distributed object based design: vertical fragmentation of classes, Journal of Distributed and Parallel Database Systems, Kluwer academic Publishers, vol. 6, 4, 1998, pp. 327–360.
[11] 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, IEEE publication, August 1997, pp. 151–160.
[12] C.I. Ezeife, Accommodating dimension hierarchies in a data warehouse view/indexes selection scheme, in: W.G. Wojtkowski, W. Wojtkowski, S. Wrycza, J. Zupancic (Eds.), Systems Development Methods for the Next Century, Plenum Press, New York, 1997, pp. 195–211.
[13] J. Gray, A. Bosworth, A. Layman, H. Pirahesh, Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals, in: Proceedings of the 12th International Conference on Data Engineering, 1996, pp. 152–159.
[14] H. Gupta, V. Harinarayan, A. Rajaraman, J. Ullman, Index Selection for OLAP, in: International Conference on Data Engineering, Burmingham, UK, 1997.
[15] V. Harinarayan, A. Rajaraman, J. Ullman, Implementing data cubes efficiently, in: ACM SIGMOD International Conference on Management of Data, June 1996, pp. 205–216.
[16] W.H. Inmon, Building the Data Warehouse, second ed, Wiley, New York, 1996.
[17] R. Kimball, The Data Warehousing Tool kit: Practical Techniques for Building Dimensional Data Warehouses, Wiley, New York, 1996.
[18] Y. Kotidis, N. Roussopoulos, DynaMat: A dynamic view management system for data warehouses, in: ACM SIGMOD International Conference on Management of Data, Philadelphia, USA, June 1999, pp. 371–382.
[19] M.E. Meredith, A. Khader, Divide and aggregate: designing large warehouses, Database Programming and Design 9 (6) (1996).
[20] M.T. Ozsu, P. Valduriez, Principles of Distributed Database Systems, second ed, Prentice-Hall, Englewood Cliffs, NJ, 1999.
[21] R. Ramakrishnan, Database Management Systems, McGraw-Hill, New York, 1998.
[22] S. Sarawagi, R. Agrawal, A. Gupta, On computing the data cube, in: Technical Report RJ 10026, IBM Almaden Research center, San Jose, California, 1996.
[23] S. Tideman, R. Chu, Building efficient data warehouses: Understanding the issues of data summarization and partitioning, in: Proceedings of the 21st Annual SAS Users Group International Conference, SUGI 21, vol. 1, 1996, pp. 520–527.
[24] J. Widom, Research problems in data warehousing, in: Proceedings of the 4th International Conference on Information and Knowledge Management (CIKM), November 1995.
[25] M. Xu, C.I. Ezeife, Maintaining horizontally partitioned warehouse views, in: Proceedings of the 2nd International Conference on Data Warehousing and Knowledge Discovery (DaWaK), DEXA, September 2000, published in the Lecture Notes in Computer Science by Springer, Berlin.
[26] M. Xu, Maintaining horizontally partitioned warehouse views, Masters Thesis, Computer Science department, University of Windsor, December 1999.

Christie I. Ezeife received her M.Sc. in Computer Science from Simon Fraser University, Canada in 1988 and a Ph.D. in Computer Science from the University of Manitoba, Canada in 1995. She has held academic positions in a number of universities and is now an associate professor of Computer Science at the University of Windsor, Canada. Her research interests include distributed object-oriented database systems, data warehousing and mining. She has authored several technical publications including two comprehensive journal articles in the international journal of distributed and parallel databases by Kluwer academic publishers.