Applying Vertical Fragmentation Techniques in Logical Design of Multidimensional Databases[1]

Matteo Golfarelli, Dario Maio, and Stefano Rizzi

DEIS - University of Bologna,
Viale Risorgimento 2, 40136 Bologna, Italy
{mgolfarelli, dmaio, srizzi}@deis.unibo.it

Abstract. In the context of multidimensional databases implemented on relational DBMSs through star schemes, the most effective technique to enhance performances consists of materializing redundant aggregates called views. In this paper we investigate the problem of vertical fragmentation of views aimed at minimizing the workload response time. Each view includes several measures which not necessarily are always requested together; thus, the system performance may be increased by partitioning the views into smaller tables. On the other hand, drill-across queries involve measures taken from two or more views; in this case the access costs may be decreased by unifying these views into larger tables. After formalizing the fragmentation problem as a 0-1 integer linear programming problem, we define a cost function and outline a branch-and-bound algorithm to minimize it. Finally, we demonstrate the usefulness of our approach by presenting a set of experimental results based on the TPC-D benchmark.

1 Introduction

Recently, multidimensional databases have gathered wide research and market interest as the core of decision support applications such as data warehouses [1][11]. A *multidimensional database* (MD) can be seen as a collection of multidimensional "cubes" centered on facts of interest (for instance, the sales in a chain store); within a cube, each cell contains information useful for the decision process, i.e., a set of numerical *measures*, while each axis represents a possible *dimension* for analysis.

An MD implemented on a relational DBMS is usually organized according to the so-called *star scheme* [13], in which each cube is represented by one *fact table* storing the measures and one denormalized *dimension table* for each dimension of analysis. The primary key of each dimension table (usually a *surrogate key*, i.e., internally generated) is imported into the fact table; the primary key of the fact table is defined by the set of these foreign keys. Each dimension table contains a set of *attributes* defining a hierarchy of aggregation levels for the corresponding dimension.

The basic mechanism to extract useful information from elemental data in MDs is aggregation. In order to improve the system performance for a given workload, an MD typically stores, besides the elemental values of measures, also values summarized according to some *aggregation patterns*, i.e., sets of attributes taken from dimension

[1] This research was partially supported by MURST - Interdata Project.

tables to define the coarseness of aggregation. Even data summarized according to each pattern are organized into a star scheme, whose fact table is called a *view* and imports the attributes included in the pattern; measure values are obtained by applying an aggregation operator to the data in another fact table with a finer pattern. In the following we will use the term *view* to denote either the fact tables containing elemental values (*primary views*) or those containing aggregated values (*secondary views*).

Since pre-computing all the possible secondary views is unfeasible, several techniques have been proposed to select the subset of views to materialize in order to optimize the response to the workload (e.g, [3][12][19]). In this paper we investigate how the response can be further enhanced by fragmenting views vertically. By vertical fragmentation we mean the creation of fragments of views, each including measures taken from one or more views with the same pattern as well as the key associated to that pattern. Fragmentation may achieve two goals together: partitioning the measures of a view into two or more tables, and unifying two or more views into a single table. While partitioning may be useful whenever only a subset of the attributes is typically required by each query, unification may be convenient when the workload is significantly affected by drill-across queries, i.e., queries formulated by joining two or more views deriving from different cubes.

It is remarkable that partitioning entails no significant storage overhead. In fact, on the one hand, surrogate keys require a few bytes to be stored. On the other, though on primary views the number of dimensions may exceed the number of measures, this is less likely on secondary views for two reasons. Firstly, it may be necessary to include in them also derived measures and support measures for non distributive aggregation operators [9][10]. Secondly, within a coarse aggregation pattern, one or more dimensions may be completely aggregated (in this case, the corresponding foreign key is dropped from the key of the view).

As compared to operational databases, in MDs the benefits of fragmentation are further enhanced by the multiple query execution plans due to the presence of redundant secondary views. These benefits are particularly relevant if the MD is implemented on a parallel architecture; if disk arrays are adopted and fragmentation is coupled with an allocation algorithm, the queries requiring multiple fragments allocated on different disks can be effectively parallelized [15][16].

The problem of determining the optimal partitioning given a workload has been widely investigated within the context of centralized as well as distributed database systems, considering non-redundant allocation of fragments (for instance, see [6][14][16]); unfortunately, the results reported in the literature cannot be applied here since the redundancy introduced by materializing views binds the partitioning problem to that of deciding on which view(s) each query should be executed. To the best of our knowledge, the problem of vertical fragmentation in MDs has been dealt with only in [15], where no algorithm for determining the optimal fragmentation is proposed. In [7], views are partitioned vertically in order to build *dataindices* to enhance performance in parallel implementations of MDs.

In Section 2 we outline the necessary background for the paper. In Section 3 the vertical fragmentation problem is formalized, a cost function is proposed and a branch-and-bound approach is proposed. Section 4 presents some results based on the TPC-D benchmark.

2 Background

2.1 Cubes and Patterns

In this paper we will often need to refer to multidimensional objects from a conceptual point of view, apart from their implementation on the logical level. For this reason we introduce *cubes*.

A multidimensional cube f is characterized by a set of dimensions $Patt(f)$, a set of measures $Meas(f)$ and a set of attributes $Attr(f) \supset Patt(f)$. The attributes in $Attr(f)$ are related into a directed acyclic graph by a set of functional dependencies $a_i \rightarrow a_j$. From now on, by writing $a_i \rightarrow a_j$ we will denote both the case in which a_i directly determines a_j and that in which a_i transitively determines a_j. It is required that

$$\forall a_j \in Attr(f) - Patt(f) \quad \exists a_i \in Patt(f);\, a_i \rightarrow a_j .$$

The cube we will use as a working example, *LineItem*, is inspired by a star scheme in the TPC-D [18] and describes the composition of the orders issued to a company; it is characterized by:

Patt(*LineItem*) = {Part, Supplier, Order, ShipDate},
Meas(*LineItem*) = {Price, Qty, ExtPrice, Discount, DiscPrice, SumCharge, Tax}

and by the attributes and functional dependencies shown in Fig. 1, where circles represent attributes (in gray the dimensions).

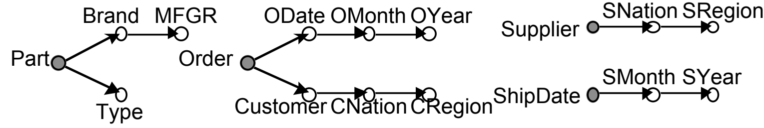

Fig. 1. Functional dependencies in the *LineItem* cube

On relational DBMSs, cubes are usually implemented adopting the star scheme. The star scheme for *LineItem* is:

PART (PartId, Part, Brand, MFGR, Type)
SUPPLIER (SupplierId, Supplier, SNation, SRegion)
ORDER (OrderId, Order, ODate, OMonth, OYear, Customer, CNation, CRegion)
SHIPDATE (ShipDateId, ShipDate, SMonth, SYear)
LINEITEM (PartId, SupplierId, OrderId, ShipDateId, Price, Qty, ExtPrice, Discount, DiscPrice, SumCharge, Tax)

where LINEITEM is the primary view; the other tables are dimension tables. For the sake of simplicity, we will not consider the possibility of normalizing dimension tables to obtain snowflake schemes.

Definition 1. Given a cube f, an *aggregation pattern* (or simply *pattern*) on f is a set $P \subset Attr(f)$ such that no functional dependency exists between each pair of attributes in P: $\forall a_i \in P \left(\nexists a_j \in P;\, a_i \rightarrow a_j \right)$.

With reference to the *LineItem* cube, examples of patterns are *Patt(f)*, {Part, OMonth, SNation}, {Brand, Type}, {}.

Definition 2. Let P_i and P_j be two patterns; we say that P_i is *coarser* than P_j ($P_i \geq P_j$) if $\forall a_h \in P_i \quad \left(a_h \in P_j\right) \vee \left(\exists a_k \in P_j; a_k \rightarrow a_h\right)$.

For instance, {Brand, CRegion} ≥ {Brand, Customer, Supplier}.

2.2 The Workload

In principle, the workload for a MD is dynamic and unpredictable. A possible approach to cope with this fact, adopted in some commercial tools, consists of monitoring the actual workload while the MD is operating. Otherwise, the designer may try to determine a core workload a priori: in fact, on the one hand, the user typically knows in advance which kind of data analysis (s)he will carry out more often for decisional or statistical purposes; on the other, a substantial amount of queries are aimed at extracting summary data to fill standard reports.

As to update queries, we believe they should not be included in the workload. In fact, MDs are typically updated only periodically, in an off-line fashion, and during this process the database is unavailable for querying. Thus, the update process does not directly affect the MD performance, and it is sufficient to ensure that it is properly bounded in time.

Definition 3. The workload is a set of pairs (q_i, η_i), where q_i denotes a query and η_i its expected frequency.

Within the scope of this paper, a query q can be characterized by (1) its pattern, *Patt*(q); (2) the set of measures it requires, *Meas*(q); (3) the selectivity, *sel*(q), defined as the ratio between the number of tuples returned by q and the cardinality of the view at *Patt*(q). For instance, on *LineItem*, the query asking for the total quantity of each medium polished part ordered from each American supplier is characterized by *Patt*(q) = {Supplier, Part}, *Meas*(q) = {Qty} and *sel*(q) = 0.01 (assuming that 5 supplier regions and 20 part types are present, that attribute values are uniformly distributed and that selection predicates are independent, it is *sel*(q) = $\frac{1}{5} \cdot \frac{1}{20}$).

Part of the queries the user formulates may require comparing measures taken from distinct, though related, cubes; in the OLAP terminology, these are called drill-across queries.

Definition 4. Let $f_1,\ldots f_m$ be m cubes such that $\exists \overline{P}; \left(\overline{P} \geq Patt\left(f_i\right) \quad \forall i = 1,\ldots m\right)$; a *drill-across query* on $f_1,\ldots f_m$ is a query q characterized by *Patt*(q) = $\overline{P}$ and $Meas\left(q\right) \subseteq \bigcup_{i=1,\ldots m} Meas\left(f_i\right)$, $Meas(q) \cap Meas(f_i) \neq \emptyset$ for i=1,...m. We call the *projection* of q on cube f_i the query q_i characterized by $Patt(q_i) = \overline{P}$ and $Meas(q_i) = Meas(q) \cap Meas(f_i)$.

Consider for instance the cube *Shipment* characterized by:

Patt(*Shipment*) = {Part, ShipTo, ShipFrom, ShipMode, ShipDate},
Meas(*Shipment*) = {QtyShipped, ShippingCost}

and by the attributes and functional dependencies shown in Fig. 2. Since pattern P = {Part, Customer, ShipDate} and all the other patterns coarser than P are common to *LineItem* and *Shipment*, a possible drill-across query is the one asking for the total cost paid by the customers of each region to receive each part, characterized by *Patt*(q) = {CRegion, Part}, *Meas*(q) = {DiscountPrice, ShippingCost} and *sel*(q) = 1.

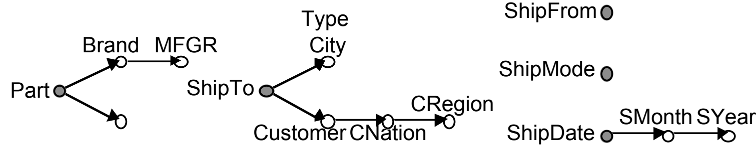

Fig. 2. Functional dependencies in the *Shipment* cube

2.3 Views

Given a cube f, each pattern on f determines a candidate view for materialization. Several algorithms have been proposed to determine the optimal set of views to be materialized, often by significantly reducing the search space [3][12]. Discussing these algorithms is outside the scope of this paper; we will assume that one of them is applied to determine, for each cube, an optimal set of views. To the best of our knowledge, no workload-based materialization algorithms in the literature takes drill-across queries into account; on the other hand, since these queries play a relevant role within our workload, it is necessary to involve them in the optimization process. Thus, when applying the materialization algorithm, every drill-across query is substituted by its projections on the cubes involved.

Let V be the global set of the (primary and secondary) views to be materialized for the MD considered, as determined by the materialization algorithm. Given view $v \in V$, we will denote with *Patt*(v) and *Meas*(v), respectively, the pattern on which v is defined (determined by its primary key) and the set of measures it contains. The primary view for cube f is characterized by *Patt*(v) = *Patt*(f) and *Meas*(v) = *Meas*(f); the secondary views for f by *Patt*(v) ≥ *Patt*(f) and *Meas*(v) = *Meas*(f).

Given query q and view v, q can be answered on v if *Patt*(q) ≥ *Patt*(v) and *Meas*(q) ⊆ *Meas*(v). In particular:

- If q involves only measures from a single cube f, it can always be answered on the primary view for f or, more conveniently, on any secondary view v for f provided that *Patt*(q) ≥ *Patt*(v).
- If q is a drill-across query on cubes $f_1,...f_m$, by definition it is *Patt*(q) ≥ *Patt*(f_i) for each i; thus, q can be solved by first solving all the projections of q on the m cubes, then performing a join between the results (the join attributes are those in *Patt*(q)).

3 Vertical Fragmentation of Views

Each view includes measures which describe the same cube but, within the workload, may be often requested separately. Thus, the system overall performance may be increased by *partitioning* the views determined from the materialization algorithm into smaller tables, each including only the measures which typically appear together within the queries. On the other hand, drill-across queries can be solved by joining views defined on different cubes. The access costs for these queries may be decreased by *unifying* two or more views on the same pattern into larger tables where all the measures required are stored together.

With the term *fragmentation* we denote both partitioning and unification of (primary or secondary) views. The approach we propose is aimed at determining an optimal fragmentation of the views in the set V.

It is remarkable that the effectiveness of fragmentation for MDs may be higher than for operational non-redundant databases; in fact, while in the latter case it is known a priori on which table(s) each query will be executed, in MDs the presence of redundant views makes multiple solutions possible. In the following we consider an example on *LineItem*. Let $V = \{v_1, v_2\}$, where

$Meas(v_1) = Meas(LineItem)$; $Patt(v_1)$ = {SNation, Brand}
$Meas(v_2) = Meas(LineItem)$; $Patt(v_2)$ = {SNation, Part, ODate}

Let the workload include two queries q_1 and q_2 defined as follows:

$Meas(q_1)$ = {Price, Qty, Discount, ExtPrice, DiscPrice}; $Patt(q_1)$ = {SNation, Brand}
$Meas(q_2)$ = {Tax, DiscPrice, SumCharge}; $Patt(q_2)$ = {SNation, Brand}

It is convenient to execute both q_1 and q_2 on v_1 since its cardinality is lower than that of v_2 ($Patt(v_1) > Patt(v_2)$). Now, consider a fragmentation including four fragments:

$Meas(v'_1)$ = {Price, Qty, Discount, ExtPrice, DiscPrice}; $Patt(v'_1) = Patt(v_1)$
$Meas(v''_1)$ = {Tax, SumCharge}; $Patt(v''_1) = Patt(v_1)$
$Meas(v'_2)$ = {DiscPrice}; $Patt(v'_2) = Patt(v_2)$
$Meas(v''_2)$ = {Price, Qty, Discount, ExtPrice, Tax, SumCharge}; $Patt(v''_2) = Patt(v_2)$

This solution is optimal for q_1, which will be executed on v'_1. As to q_2, while Tax and SumCharge are retrieved from v''_1, it might be more convenient to retrieve DiscPrice from v'_2 rather than from v'_1, depending on the trade-off between reading less measures and accessing less tuples. In general, another factor to be considered in the trade-off is the number of attributes forming the fact table key: for coarser patterns, the length of the key is shorter and the size of the tuples read is smaller. The possibility of answering a query by jointly accessing fragments of different patterns, impacts on the optimization of the query execution tree by enabling additional push-downs and pull-ups of group-by operators.

3.1 Problem Statement

In principle, the fragmentation algorithm should be applied to the whole set of views, V. On the other hand, it may be convenient to unify two measures belonging to two different cubes f' and f'' only if at least two views with the same pattern have been materialized on f' and f'' and the workload includes at least one drill-across query on f'

and *f''* which can be answered on these two views; in this case, we say that *f'* and *f''* are *strictly related*. Two cubes are *related* if (1) they are strictly related or (2) a third cube related to both exists. The transitive notion of relatedness induces a partitioning onto the set of cubes belonging to the MD, which in turn partitions the set of queries and the set of views according to the cube(s) they are defined on; in order to decrease complexity, fragmentation is meant to be applied separately to each set of queries on the corresponding set of related cubes.

Let FS be a set of related cubes and QS be the set of queries on the cubes in FS. Let $VS \subseteq V$ be the set of views materialized on the cubes in FS and PS be the set of patterns characterizing the views in VS.

Definition 5. Given cube $f \in FS$, we partition[2] $Meas(f)$ into the largest subsets of measures which appear all together in at least one query of QS and do not appear separately in any other query in QS:

$$\exists q_i \in QS; M \subseteq Meas(q_i) \wedge \forall q_j \in QS \left(Meas(q_j) \cap M \neq \varnothing \Rightarrow M \subseteq Meas(q_j)\right) .$$

We call each subset a *minterm* of *f*, and denote with $MS(f)$ the set of all minterms of *f*.

For instance, on the *LineItem* cube, given $QS = \{q_1, q_2\}$ where $Meas(q_1)$ = {Price, Qty, ExtPrice, Discount} and $Meas(q_2)$ = {Price, Qty, DiscPrice, SumCharge}, it is $MS(LineItem)$ = {{Price, Qty}, {ExtPrice, Discount}, {DiscPrice, SumCharge}}.

Definition 6. Given the set of related cubes FS, we define a *term* as a set of measures which either is a minterm of a cube in FS or is the union of the minterms, even from different cubes in FS, within a set $\overline{MS}$ such that $\forall M_k \in \overline{MS} \quad \exists q_i \in QS, \exists M_j \in \overline{MS}, j \neq k; M_k \subset Meas(q_i) \wedge M_j \subset Meas(q_i)$. We denote with TS the set of terms for FS.

For example above, it is $TS = MS(LineItem) \cup \{Meas(q_1), Meas(q_2)\}$.

Given FS, a solution to the fragmentation problem is encoded by a *fragmentation array*, i.e., a binary array $\boldsymbol{C}$ with three dimensions corresponding to, respectively, the queries $q_i \in QS$, the patterns $P_j \in PS$ and the terms $T_k \in TS$. The set of *fragments* defined by $\boldsymbol{C}$ is

$$VS' = \left\{ v_{jk} ; \exists\, j,k; \sum_{q_i \in QS} C_{ijk} \geq 1 \right\}$$

where fragment v_{jk} is characterized by $Meas(v_{jk}) = T_k$ and $Patt(v_{jk}) = P_j$.

A fragmentation array not only denotes a fragmentation of the views in VS; at the same time, it specifies on which fragment(s) each query is assumed to be executed. In fact, a 1 in cell C_{ijk} denotes that, when answering query q_i, the measures in $Meas(q_i) \cap T_k$ will be obtained from v_{jk}.

The fragmentation encoded by $\boldsymbol{C}$ is *feasible* if the following constraints are satisfied:

[2] We assume that each measure appears in at least one query of the workload.

(1) For each query, every measure required must be obtained from exactly one fragment (non ambiguous query execution):

$$\forall q_i \in QS, \forall m \in Meas(q_i) \left(\sum_{P_j \in PS;\ Patt(q_i) \geq P_j} \sum_{T_k \in TS;\ m \in T_k} C_{ijk} = 1 \right).$$

(2) For each pattern, each measure must belong to at most one fragment (non redundant fragmentation):

$$\forall P_j \in PS, \forall T_k, T_h \in TS; \left(k \neq h \wedge T_k \cap T_h \neq \varnothing\right) \left(\left(\sum_{q_i \in QS} C_{ijk} = 0 \right) \vee \left(\sum_{q_i \in QS} C_{ijh} = 0 \right) \right).$$

(3) Each fragment in VS' must be a fragmentation of one or more views in VS (consistency with the views materialized):

$$\forall P_j \in PS, \forall T_k \in TS; \sum_{q_i \in QS} C_{ijk} \geq 1 \quad \left(\forall m \in T_k\ \exists v \in VS;\ Patt(v) = P_j \wedge m \in Meas(v)\right).$$

Though constraint (3) states that every fragment must derive from a view in VS, we are not guaranteed that every measure in every view is included in a fragment. In fact, fragments will be produced only for the minterms, among those deriving from each view, actually used to answer at least one query. Given a view, we will call *lost minterms* those not included in any term generating a fragment. The fragmentation of primary views must necessarily be lossless, thus, every lost primary minterm must be reconsidered *a posteriori*, either by creating a separate fragment or by unifying it with one of the fragments determined. On the other hand, as to lost minterms from secondary views, it is not obvious whether generating the fragments is convenient or not; in fact, the space saved could probably be more profitably employed to store indices or additional views.

In the following we consider a small example on FS = {*LineItem*, *Shipment*}. Let $QS = \{q_1, q_2, q_3, q_4, q_5\}$; q_1, q_2, q_3 are defined on *LineItem*, q_4 on *Shipment* and q_5 is a drill-across query:

$Meas(q_1)$ = {Price, Qty, Discount}; $Patt(q_1)$ = {ShipDate}
$Meas(q_2)$ = {ExtPrice, DiscPrice}; $Patt(q_2)$ = {Part, Customer}
$Meas(q_3)$ = {SumCharge, Tax}; $Patt(q_3)$ = {Part, CNation}
$Meas(q_4)$ = {QtyShipped, ShippingCost}; $Patt(q_4)$ = {CNation, MFGR, ShipDate}
$Meas(q_5)$ = {ExtPrice, DiscPrice, ShippingCost}; $Patt(q_5)$ = {Brand, CNation}

We assume that, besides the primary views v_1 and v_2, two secondary views v_3 and v_4 have been materialized on *LineItem*, one secondary view v_5 on *Shipment*:

$Patt(v_3)$ = {Part, Customer}; $Patt(v_4) = Patt(v_5)$ = {Part, CNation}

(for each view, the measures are those of the corresponding cube). Fig. 3 shows the fragmentation array which represents a feasible solution to this fragmentation problem, which features five fragments:

$Meas(v'_1)$ = {Price, Qty, Discount}; $Patt(v'_1) = Patt(LineItem)$
$Meas(v'_2)$ = {QtyShipped, ShippingCost}; $Patt(v'_2) = Patt(Shipment)$
$Meas(v'_3)$ = {ExtPrice, DiscPrice}; $Patt(v'_3) = Patt(v_3)$
$Meas(v'_4)$ = {SumCharge, Tax}; $Patt(v'_4) = Patt(v_4)$
$Meas(v'_5)$ = {ExtPrice, DiscPrice, ShippingCost}; $Patt(v'_5) = Patt(v_4) = Patt(v_5)$

the first four of which are obtained by partitioning, the last one by coupling partitioning and unification. The array also denotes that, for instance, query q_1 is executed on v'_1.

TS \ PS	q_1: $Patt(LineItem)$	q_1: $Patt(Shipment)$	q_1: $Patt(v_3)$	q_1: $Patt(v_4)=Patt(v_5)$	q_2: $Patt(LineItem)$	q_2: $Patt(Shipment)$	q_2: $Patt(v_3)$	q_2: $Patt(v_4)=Patt(v_5)$	q_3: $Patt(LineItem)$	q_3: $Patt(Shipment)$	q_3: $Patt(v_3)$	q_3: $Patt(v_4)=Patt(v_5)$	q_4: $Patt(LineItem)$	q_4: $Patt(Shipment)$	q_4: $Patt(v_3)$	q_4: $Patt(v_4)=Patt(v_5)$	q_5: $Patt(LineItem)$	q_5: $Patt(Shipment)$	q_5: $Patt(v_3)$	q_5: $Patt(v_4)=Patt(v_5)$
{Discount,Qty,Price}	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{DiscPrice,ExtPrice}	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
{SumCharge,Tax}	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
{ShippingCost}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{QtyShipped}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{ShippingCost,QtyShipped}	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
{DiscPrice,ExtPrice,ShippingCost}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Fig. 3. Fragmentation array representing a feasible solution

3.2 The Cost Function

Among all the feasible solutions to the fragmentation problem, we are interested in the one which minimizes the cost for executing the workload. We believe it is convenient to keep logical design separate from the physical level in order to both provide a more general solution and reduce complexity; thus, the cost function we propose intentionally abstracts from any assumptions on the access paths, being based on the number of disk pages in which the tuples of interest for a given query are stored. In particular, the cost of query q_i within fragmentation $\boldsymbol{C}$ is defined as:

$$cost\left(q_i,\boldsymbol{C}\right)=\sum_{P_j\in PS,T_k\in TS}\Phi\left(sel\left(q_i\right)\cdot ns\left(P_j\right),\left\lceil\frac{ns\left(P_j\right)}{\beta_{jk}}\right\rceil\right)\cdot C_{ijk}$$

where:

- $ns(P_j)$ is the cardinality for the view on pattern P_j (estimated for instance as shown in [8]).
- $sel(q_i)$ is the selectivity of q_i; thus, $sel(q_i)\cdot ns(P_j)$ is the number of tuples of the view on pattern P_j which must be accessed in order to answer q_i.
- β_{jk} is the number of tuples per disk page for fragment v_{jk} characterized by P_j and T_k:

$$\beta_{jk}=\frac{size\left(page\right)}{size\left(tuple\right)}=\frac{size\left(page\right)}{\sum_{a\in P_j}size\left(a\right)+\sum_{m\in T_k}size\left(m\right)}\ .$$

Thus, $\left\lceil \frac{ns(P_j)}{\beta_{jk}} \right\rceil$ is the number of pages in which v_{jk} is contained.

- $\Phi\left(sel(q_i) \cdot ns(P_j), \left\lceil \frac{ns(P_j)}{\beta_{jk}} \right\rceil \right)$ is the expected number of pages in which the tuples necessary for q_i are stored, estimated with the Cardenas formula Φ [5].

Thus, $cost(q_i,\boldsymbol{C})$ expresses the total number of disk pages which must be accessed in order to solve q_i. Though the actual number of pages read when executing the query may be higher depending on the access path followed, we believe that this function represents a good trade-off between generality and accuracy.

It should be noted that, whenever two views are unified, the resulting fragment may be used to answer not only drill-across queries, but also queries on the single cubes; thus, it must contain the union of their tuples. Let *f'* and *f"* be two cubes, let *P* be a pattern common to *f'* and *f"*, and let *ns'*(*P*) and *ns"*(*P*) be the cardinalities of views *v'* on *f'* and *v"* on *f"*, respectively. The cardinality of the view unifying *v'* and *v"* can be estimated as:

$$ns'''(P) = ns'(P) + ns''(P) - \frac{ns'(P) \times ns''(P)}{cs(P)}$$

where *cs*(*P*) is the product of the domain cardinalities for the attributes in *P*.

3.3 A Branch-and-Bound Approach

The problem of vertical fragmentation (*VFP*) can be formulated as follows: *Find, for the binary decision array* ***C****, the value which minimizes function*

$$tcost(QS,\boldsymbol{C}) = \sum_{q_i \in QS} \eta_i \cdot cost(q_i,\boldsymbol{C})$$

subject to constraints (1), (2), (3) expressed in Section 3.1. *VFP* is a 0-1 integer linear programming problem like *set covering with additional constraints*, and is known to be NP-hard [17]. Thus, a branch-and-bound approach can be adopted to solve it optimally.

The ingredients of a branch-and-bound procedure for a discrete optimization problem such as *VFP* are [1]:

(i) A *branching rule* for breaking up the problem into subproblems. Let VFP_α be the problem of choosing, given a partial solution to *VFP* represented by an "incomplete" array[3] $\boldsymbol{C}(VFP_\alpha)$, the remaining elements C_{ijk} to be set to 1 in the complete solution. We denote with $SUB(VFP_\alpha)$ the set of subproblems in which VFP_α is broken up; each is defined by choosing one element C_{ijk} to be set to 1 in the partial solution, which means adding to the current solution a

[3] Non feasible since one or more queries cannot be answered (constraint (1) is not satisfied).

fragment on pattern P_j to be used for retrieving some measures T_k to solve query q_i.

(ii) A *subproblem selection rule* for choosing the next (most promising) subproblem to be processed. The element C_{ijk} chosen is the one for which $ns(P_j)$ is minimum and $Meas(q_i) \cap T_k$ has maximum cardinality.

(iii) A *relaxation* of VFP_α, i.e. an easier problem VFR_α whose solution bounds that of VFP_α. We relax VFP_α by removing constraint (2): in VFR_α, some measures may be replicated in two or more fragments defined on the same pattern.

(iv) A *lower bounding* procedure to calculate the cost of the relaxation. VFR_α consists of one set covering problem for each q_i, which can be solved by adopting one of the algorithms in the literature [4]. Since in solving VFR_α the number of eligible fragments is higher than that for VFP_α, the cost of VFR_α will be lower or equal to that of VFP_α.

4 Experimental Tests

In this paper we have proposed an approach to vertical fragmentation of views in multidimensional databases. The experimental results we present in this section confirm the utility of the approach in terms of reduction of the cost for executing the expected workload. The tests we have carried out are based on the well-known TPC-D benchmark [18], which features two cubes *LineItem* and *PartSupplier* with cardinalities 6.000.000 and 800.000, respectively; the total amount of data is about 1 Gbyte.

We have tested our approach on the Informix DBMS with a workload based on the 17 TPC-D queries (all with the same frequency). The views to be fragmented have been selected by means of the heuristic approach to materialization proposed in [3], by considering a global space constraint of 2 GB (1 GB for primary views + 1 GB for secondary views); as a result, 11 secondary views were created besides the 2 primary views. The fragmentation algorithm determined 14 fragments (3 from the primary views) and 9 lost minterms. Indices on all the attributes belonging to keys in both fact and dimension tables were created.

Fig. 4 shows, for each query, the ratio between the number of disk pages read without and with fragmentation; above each column, the number of disk pages read without fragmentation. Overall, fragmentation decreases the workload cost from 265904 to 59986 pages (more than 4 times).

Fig. 5 shows how fragmentation affects the total storage space; above each column, the storage space without fragmentation. Overall, the unfragmented views require 368840 pages; while materializing only the fragments (no lost minterms) decreases the space required to 306042 pages (-17.0%), materializing also lost minterms increases the space required to 442097 pages (+19.8%).

It should be noted that the next view to be materialized beyond the 2 GB constraint would take 126460 disk pages, and decrease the workload cost by 1%; fragmentation is more convenient since it takes only 73257 extra pages and decreases the cost by 77%. In fact, while materializing one more view typically benefits few queries, several queries may take advantage from using the same disk space for fragmentation. Furthermore, while fragmentation does not require extra space for dimension tables, each new view may require adding new tuples in dimension tables to be referenced by the aggregated tuples in the view.

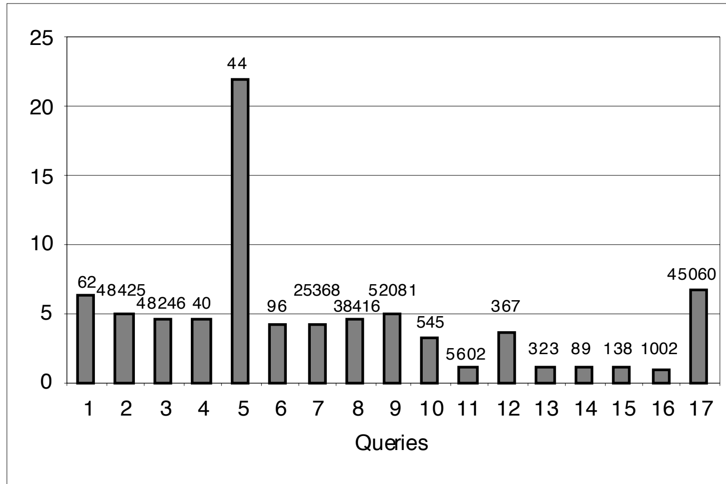

Fig. 4. Ratio between the number of disk pages read without and with fragmentation

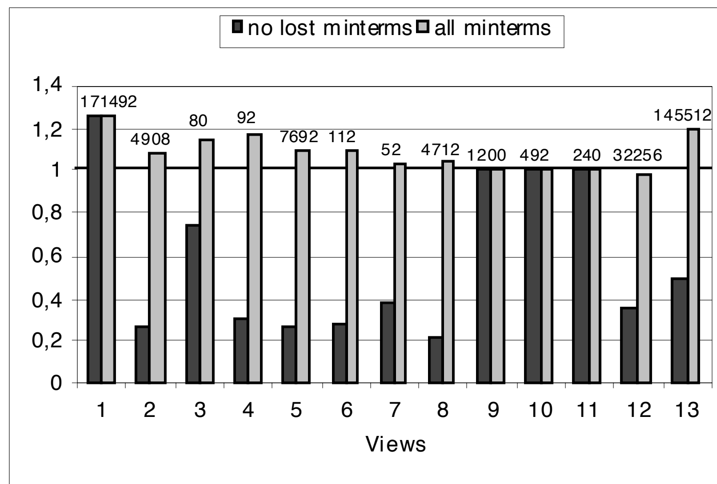

Fig. 5. Ratio between the storage space with and without fragmentation

In order to evaluate the algorithm complexity, we have defined four more workloads, each progressively extending the TPC-D; the results are shown in Table I. The computing time does not depend strictly on the workload size, of course it is also determined by the relationships between the queries.

Table I. Results of the complexity tests

n. queries in the workload	*n. subproblems generated*	*computing time*
17	2775	about 1 min
25	4439	about 2 mins
30	348925	about 30 mins
35	51099	about 12 mins
40	403420	about 75 mins

References

1. Balas, E., Toth, P.: Branch and bound methods. In: Lawler, E. et al. (eds.): The traveling salesman problem. John Wiley & Sons (1985) 361-397
2. Agrawal, R., Gupta, A., Sarawagi, S.: Modeling multidimensional databases. IBM Research Report (1995)
3. Baralis, E., Paraboschi, S., Teniente, E.: Materialized view selection in multidimensional database. Proc. 23rd Int. Conf. on Very Large Data Bases, Athens, Greece (1997) 156-165
4. Beasley, J.E.: An Algorithm for Set Covering Problems. European Journal of Operational Research 31 (1987) 85-93
5. Cardenas, A.F.: Analysis and performance of inverted database structures. Comm. ACM 18(5) (1975) 253-263
6. Chu, W.W., Ieong, I.T.: A transaction-based approach to vertical partitioning for relational database system. IEEE Trans. on Software Eng. 19(8) (1993) 804-812
7. Datta, A., Moon, B., Thomas, H.: A case for parallelism in data warehousing and OLAP. Proc. IEEE First Int. Workshop on Data Warehouse Design and OLAP Technology (1998)
8. Golfarelli, M., Rizzi, S.: Designing the data warehouse: key steps and crucial issues. Journal of Computer Science and Information Management 2(3) (1999)
9. Golfarelli, M., Rizzi, S.: View Materialization for Nested GPSJ Queries. To appear on Proc. DMDW'2000, Stockholm (2000)
10. Gray, J., Bosworth, A., Lyman, A., Pirahesh, H.: Data-Cube: a relational aggregation operator generalizing group-by, cross-tab and sub-totals. Technical Report MSR-TR-95-22, Microsoft Research (1995)
11. Gyssens, M., Lakshmanan, L.V.S.: A foundation for multi-dimensional databases. Proc. 23rd Very Large Database Conf., Athens, Greece (1997) 106-115
12. Harinarayan, V., Rajaraman, A., Ullman, J.: Implementing Data Cubes Efficiently. Proc. of ACM Sigmod Conf., Montreal, Canada (1996)
13. Kimball, R.: The data warehouse toolkit. John Wiley & Sons (1996)
14. Lin, X., Orlowska, M., Zhang, Y.: A graph-based cluster approach for vertical partitioning in database design. Data & Knowledge Engineering 11 (1993) 151-169
15. Munneke, D., Wahlstrom, K., Mohania, M.: Fragmentation of multidimensional databases. Proc. 10th Australasian Database Conf., Auckland (1999) 153-164
16. Özsu, M.T., Valduriez, P.: Principles of distributed database systems. Prentice-Hall Int. Editors (1991)
17. Papadimitriou, C.H., Steiglitz, K.: Combinatorial optimization. Prentice Hall, Englewood Cliffs (1982)
18. Raab, F. (ed.): TPC Benchmark(tm) D (Decision Support), Proposed Revision 1.0. Transaction Processing Performance Council, San Jose, CA 95112 (1995)
19. Yang, J., Karlaplem, K., Li, Q.: Algorithms for Materialized View Design in Data Warehousing Environments. Proc. 23rd Int. Conf. on Very Large Databases, Athens, Greece (1997) 136-145