Vertical Fragmentation of Views in Relational Data Warehouses

Matteo Golfarelli, Dario Maio, Stefano Rizzi

DEIS - Università di Bologna
Viale Risorgimento, 2 - 40136 Bologna

Abstract. Within the framework of the data warehouse design methodology we are developing, in this paper we investigate the problem of vertical fragmentation of relational views aimed at minimizing the global query response time. Each view includes several measures which, within the workload, are seldom requested together; thus, the system performance may be increased by partitioning the views to be materialized 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. Within the data warehouse context, the presence of redundant views makes the fragmentation problem more complex than in traditional relational databases since it requires to decide on which views each query should be executed. After formalizing the fragmentation problem as a 0-1 integer linear programming problem, we define a cost function and propose a branch-and-bound algorithm to minimize it. Finally, we demonstrate the usefulness of our approach by presenting a sample set of experimental results.

1 Introduction

Though designing a data warehouse (DW) requires techniques completely different from those adopted for operational systems, no significant effort has been made so far to develop a complete and consistent ad hoc design methodology. In [7] we have outlined a general methodological framework for DW design, based on the conceptual model we developed, called *Dimensional Fact Model*. In [6] we proposed a semi-automated approach to conceptual modelling starting from the pre-existing (conceptual or logical) schemes describing the operational information system; currently, we are working on logical design, which entails producing a logical scheme for the DW starting from the conceptual scheme (which we call dimensional scheme) and from the expected workload. The whole process, including physical design, is sketched in Figure 1.

The issues raised by the topic of DW logical design are many and interesting; among them, the one most commonly considered in the literature is view materialization, which plays a relevant role in determining the system overall performance for a given workload [8]. Assuming that the target logical model is the well-known star scheme [10], in this paper we investigate how the response to the workload can be further enhanced by fragmenting vertically the fact tables which implement the views to be materialized. By vertical fragmentation we mean the

partitioning of the attributes of a table into two or more tables by replicating the key, as well as the unification of two or more tables with the same key into a single table including the union of the attributes. 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 on two or more fact tables.

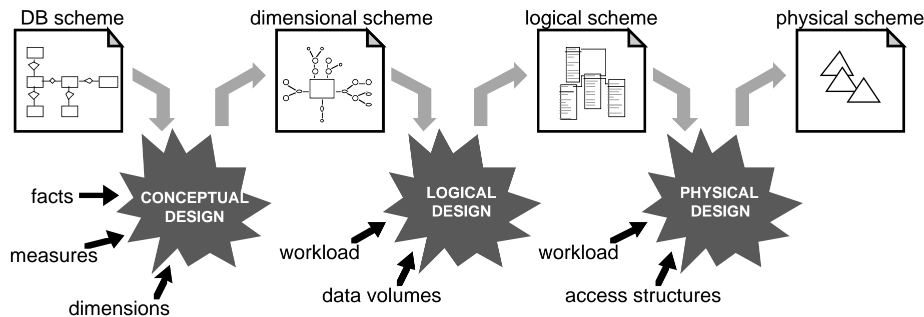

Fig. 1. Design methodology for DWs

As compared to operational databases, in DW environments the benefits of fragmentation are further enhanced by the multiple query execution plans arising from the presence of redundant views. These benefits are particularly relevant if the DW is implemented on a parallel architecture; in particular, if disk arrays are adopted and fragmentation is coupled with an allocation algorithm [13], the queries requiring multiple fragments allocated on different disks can be effectively parallelized.

The problem of determining the optimal partitioning given a workload has been widely investigated within the context of centralized [5] [11] as well as distributed database systems [13]; unfortunately, the results reported in the literature cannot be applied to the DW case 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 DWs has been dealt with only in [12], where it is formalized with reference to multidimensional databases; no algorithm for determining the optimal fragmentation is proposed.

2 Background

In this section we introduce the necessary background in our conceptual model. In particular, after discussing the main features of a fact scheme, we define aggregation patterns as a way to characterize DW queries. Finally, we show how drill-across queries, formulated on two or more fact schemes, can be formulated within the workload.

2.1 The Dimensional Fact Model

The Dimensional Fact Model (DFM) is a graphical formalism for conceptual modeling of DW requirements [6]. The representation of reality built using the DFM is called *dimensional scheme*, and consists of a set of *fact schemes* whose basic elements are facts, dimensions and hierarchies. In this section we briefly recall the features of the DFM useful within this paper, with reference to the *LINEITEM* example shown in Figure 2, which models one of the star schemes included within the TPC-D [15].

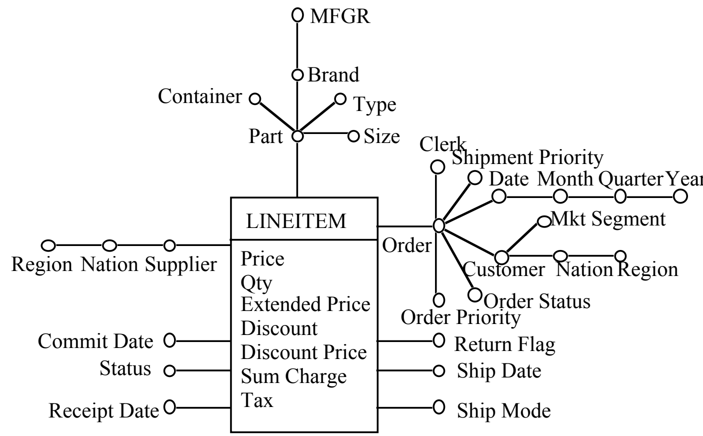

Fig. 2. The *LINEITEM* fact scheme

A fact scheme is structured as a tree whose root is a fact. A *fact* is a focus of interest for the enterprise; it is represented by a box which reports the fact name and, typically, one or more numeric and continuously valued *measures* which "quantify" the fact from different points of view. We will denote with *Meas*(*f*) the set of measures of fact scheme *f*. In the *LINEITEM* scheme, *Price* is an example of measure.

Attributes are represented by circles and may assume a discrete set of values. Each attribute directly attached to the fact is a *dimension*; dimensions determine the granularity adopted for representing facts. We will denote with *Attr*(*f*) and *Patt*(*f*)⊆*Attr*(*f*), respectively, the set of attributes and the set of dimensions (*dimension pattern*) of fact scheme *f*. The dimension pattern of the *LINEITEM* scheme is {*Supplier*, *Part*, *Order*, *Return Flag*, *Ship Mode*, *Status*, *Ship Date*, *Commit Date*, *Receipt Date*}.

Subtrees rooted in dimensions are *hierarchies*, and determine how fact occurrences may be aggregated and selected significantly for the decision-making process. The dimension in which a hierarchy is rooted defines its finest aggregation granularity; the attributes associated to the vertices along each path of the hierarchy starting from the dimension define progressively coarser granularities. The arc connecting two attributes represents a -to-one relationship (functional dependency) between them; thus, every directed path within one hierarchy necessarily represents a functional dependency between the starting and the ending attributes.

2.2 Fact instances and aggregation patterns

Given a fact scheme *f*, each *n*-ple of values taken from the domains of the *n* dimensions of *f* defines an elemental cell where one unit of information for the DW can be represented. We call *primary fact instances* the units of information present within the DW, each characterized by exactly one value for each measure.

Since analysing data at the maximum level of detail is often overwhelming, primary fact instances are typically aggregated into secondary fact instances at different levels of abstraction, each corresponding to an aggregation pattern.

Definition 1. Given a fact scheme *f* with *n* dimensions, a *v*-dimensional *aggregation pattern* is a set $P = \{a_1,...a_v\}$ of attributes of *f* such that $P \neq Patt(f)$ and no directed path of arcs (i.e., no functional dependency) exists between each pair of attributes in *P*.

Examples of aggregation patterns in the *LINEITEM* scheme are {*Part.Type*, *Supplier.Region*, *Order*, *Return Flag*, *Ship Mode*, *Status*, *Ship Date*, *Commit Date*, *Receipt Date*}, {*Part.Brand*, *Supplier*, *Order.Year*}.

Given aggregation pattern $P = \{a_1,...a_v\}$, each *v*-ple of values taken from the domains of the *v* attributes of *P* defines a macro-cell which, from the logical point of view, aggregates all the primary fact instances corresponding to *n*-ples of values which functionally determine the same *v*-ple along the attribute hierarchies. These macro-cells are called *secondary fact instances* for aggregation pattern *P*; each is characterized by exactly one value for each measure, calculated by applying a grouping function (typically the sum) to the values that measure assumes within the primary fact instances aggregated.

In the following, we will use the term *pattern* to denote either the dimension pattern or an aggregation pattern.

2.3 The workload

DWs are primarily directed towards answering quickly all the possible queries on the enterprise data. Since keeping into account all the possible queries is computationally too complex, a reduced set of significant and frequent queries will be considered to be representative of the actual workload.

In [6] we have introduced a simple language for defining, with reference to a dimensional scheme, the queries forming the expected workload for the DW; this language is meant to be used for logical design, hence, it focuses on which data must be retrieved and at which level they must be consolidated. In particular, we represent the typical DW query by the set of fact instances, at any aggregation level, whose measure values are to be retrieved; one or more Boolean predicates on the attributes of the fact scheme may be formulated in order to restrict this set. Within the scope of this paper, it will be sufficient to characterize query *q* with (1) its pattern, *Patt*(*q*); (2) the set of measures required, *Meas*(*q*); (3) the selectivity, *sel*(*q*), defined as the ratio between the number of fact instances returned by *q* and the total number of fact instances, *ns*(*Patt*(*q*)).

In the DFM, different facts are represented in different fact schemes. However, part of the queries the user formulates on the DW may require comparing measures taken from distinct, though related, schemes; in the OLAP terminology, these are called

drill-across queries. In [6] we defined the rules for combining two *compatible* (i.e., sharing at least one attribute) fact schemes f' and f'' into a new scheme $f' \otimes f''$, which we call their *overlap* and includes the union of the measures of f' and f'' (see Figure 3). A drill-across query can then be expressed on the overlap of two or more fact schemes; for instance, the query asking for the total cost paid by the customers of each region to receive each part, characterized by *Patt*(q) = {*Customer.Region*, *Part*} and *Meas*(q) = {*Discount Price*, *Shipping Cost*}, is formulated on *LINEITEM* ⊗ *SHIPMENT*.

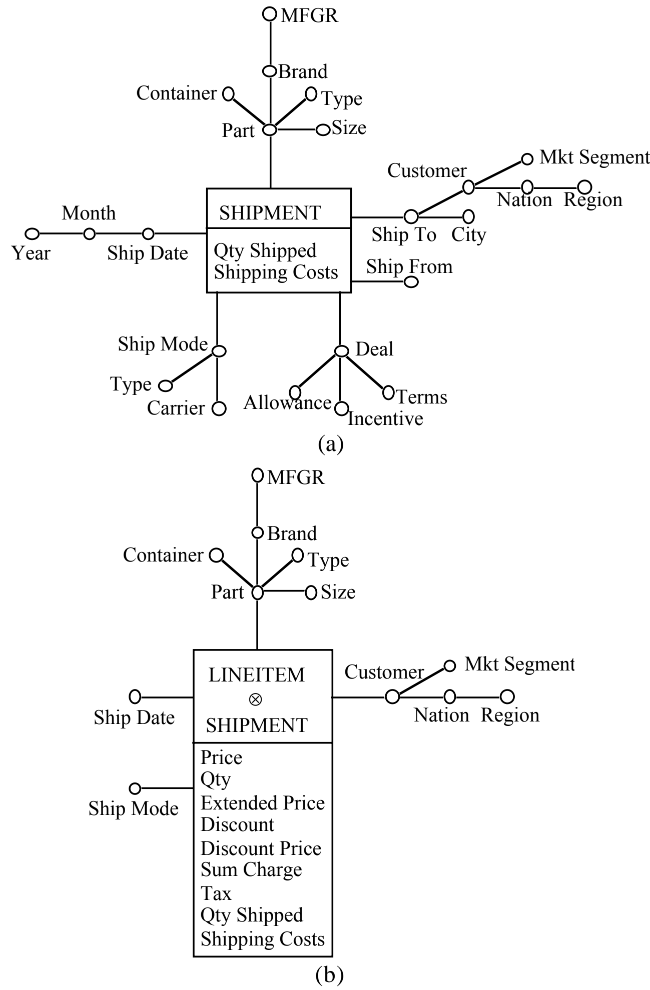

Fig. 3. The *SHIPMENT* fact scheme (a) and the *LINEITEM* ⊗ *SHIPMENT* overlapped fact scheme (b)

Definition 2. The workload on a dimensional scheme is a set of pairs (q_i, η_i), where q_i denotes a query and η_i its expected frequency.

3 Logical design from dimensional schemes

Logical design receives in input a dimensional scheme, a workload and a set of additional information (update frequencies, total disk space available, maximum query response time, time-space trade-off point, etc.) to produce a DW scheme which should minimize the total query response time by respecting the disk space constraint.

At this time, it is necessary to choose the target logical model, relational or multidimensional. In this paper we consider only the relational case which represents, at the moment, the most frequent choice. A dimensional scheme can be mapped on the relational model by adopting the well-known star scheme [10] [3], in which fact instances and hierarchies are stored, respectively, within *fact tables* and denormalized *dimension tables*. A fact table has logical scheme $\mathtt{FT}(\underline{\mathtt{k}_1},\ldots\underline{\mathtt{k}_n},\mathtt{m}_1,\ldots\mathtt{m}_z)$, where each $\mathtt{k}_i$ is a foreign key imported from a dimension table and each $\mathtt{m}_j$ is a measure; in general, fact table v is characterized by the pattern it is defined on, $Patt(v) = \{a_1,...a_n\}$ (where a_i is the attribute corresponding to $\mathtt{k}_i$), and by the set of measures it contains, $Meas(v) = \{m_1,...m_z\}$.

Within a star scheme, the primary fact instances for fact scheme f are stored within a fact table v characterized by $Patt(v) = Patt(f)$ and by $Meas(v) = Meas(f)$. A technique commonly used in order to reduce the response time for frequent queries is to pre-compute and consolidate the secondary fact instances on one or more aggregation patterns. In the literature, fact tables reporting data consolidated from other fact tables are generally called views; in the following we will use the term *view* to denote indifferently the fact tables containing primary fact instances (*primary views*) and those containing secondary fact instances (*secondary views*).

Definition 3. Let P_i and P_j be two distinct aggregation patterns on fact scheme f; we say that P_i is *projectable* on P_j ($P_j \geq P_j$) iff, for each attribute $a_h \in P_i$, there exists one attribute $a_k \in P_j$ such that a_h functionally depends on a_k. Obviously, since each attribute functionally depends on itself, each pattern is projectable on itself.

For example, with reference to the *LINEITEM* fact scheme, the pattern P_i = {*Part.Brand*,*Order.Region*} is projectable on P_j = {*Part.Brand*, *Order.Customer*, *Supplier*}.

Given a query q and a view v, q can be answered on v iff $Patt(q) \geq Patt(v)$ and $Meas(q) \subseteq Meas(v)$.

3.1 View materialization

Before facing the problem of vertical fragmentation of views, it is necessary to decide *which* secondary views should be materialized. Several algorithms have been proposed to determine the optimal set of views, often by reducing significantly the search space [2] [9]. Discussing these algorithms is outside the scope of this paper; we will assume that one of them is applied to determine, for each fact scheme, an optimal set of views.

To the best of our knowledge, no materialization algorithms in the literature takes drill-across queries into account. On the other hand, since drill-across queries play a relevant role within our workload, it is necessary to involve them in the optimization process by transforming them into queries on single fact schemes. Let q be a drill-

across query on the overlapped fact scheme $f = f_1 \otimes ... \otimes f_z$; from the point of view of view materialization, q is equivalent to z queries $q_1,...q_z$ characterized by $Patt(q_i) = Patt(q)$ and $Meas(q_i) = Meas(q) \cap Meas(f_i)$.

Let V be the set of (primary and secondary) views to be materialized determined in output from the materialization algorithm. Each view $v \in V$, associated to fact scheme f, is characterized by its pattern $Patt(v) \geq Patt(f)$ and by $Meas(v) = Meas(f)$.

4 Vertical fragmentation of views

Vertical fragmentation is an important issue to be addressed in order to minimize the global query response time; it is aimed at optimizing both the queries requiring a subset of measures and drill-across queries.

Each view includes several measures which describe the same fact but, within the workload, are seldom requested together. 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 are formulated on overlapped fact schemes; as such, they involve measures taken from two or more views. The access costs for these queries may be decreased by *unifying* these views into larger tables where all the measures required are stored together.

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

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

$Meas(v_1) = Meas(LINEITEM)$; $Patt(v_1) = \{Supplier.Nation, Part.Brand\}$
$Meas(v_2) = Meas(LINEITEM)$; $Patt(v_2) = \{Supplier.Nation, Part, Order.Date\}$

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

$Meas(q_1) = \{Price, Qty, Discount, ExtPrice, DiscPrice\}$; $Patt(q_1) = \{Supplier.Nation, Part.Brand\}$
$Meas(q_2) = \{Tax, DiscPrice, SumCharge\}$; $Patt(q_2) = \{Supplier.Nation, Part.Brand\}$

It is convenient to execute both q_1 and q_2 on v_1 since its cardinality is lower than that of v_2 ($Patt(v_1) > Patt(v_2)$). Let the optimal fragmentation include four fragmented views:

$Meas(v'_1) = \{Price, Qty, Discount, ExtPrice, DiscPrice\}$; $Patt(v'_1) = Patt(v_1)$
$Meas(v'_2) = \{Tax, SumCharge\}$; $Patt(v'_2) = Patt(v_1)$
$Meas(v'_3) = \{DiscPrice\}$; $Patt(v'_3) = Patt(v_2)$
$Meas(v'_4) = \{Price, Qty, Discount, ExtPrice, Tax, SumCharge\}$; $Patt(v'_4) = Patt(v_2)$

This solution is optimal for q_1, which will be executed on v'_1, i.e., on pattern $\{Supplier.Nation, Part.Brand\}$. As to q_2, it could be now convenient to retrieve measure *DiscPrice* from view v'_3 on pattern $\{Supplier.Nation, Part, Order.Date\}$,

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: the coarser the aggregation pattern, the lower the length of the key, the smaller the tuples to be read.

4.1 Problem statement

In principle, the fragmentation algorithm should be applied to the whole set V. On the other hand, it may be convenient to unify two measures belonging to two different fact schemes 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, defined on $f' \otimes f''$, which could be answered on these two views; in this case, we say that f' and f'' are *related*. The notion of relatedness induces a partitioning onto the set of fact schemes belonging to the dimensional scheme, which in turn partitions the set of queries and the set of views according to the fact scheme(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 fact schemes.

Let FS be a set of related fact schemes and QS be the set of queries on the schemes in FS. Let $VS \subseteq V$ be the set of views materialized on the fact schemes in FS and PS be the set of patterns associated to the views in VS.

> **Definition 4**. Given fact scheme $f \in FS$, we partition *Meas*(f) into the largest subsets of measures which appear together in at least one query of QS and do not appear separately in any other query in QS. 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* scheme, 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 5**. Given the set of related fact schemes FS, we define a *term* as a set of measures which (1) is a minterm of one of the fact schemes in FS; or (2) is the union of two or more minterms, even belonging to different fact schemes in FS, required together by at least a query. We denote with TS the set of terms for FS.

In the example above, if FS = {*LINEITEM*}, it is TS = {{*Price*, *Qty*}, {*ExtPrice*, *Discount*}, {*DiscPrice*, *SumCharge*}, {*Price*, *Qty*, *ExtPrice*, *Discount*}, {*Price*, *Qty*, *DiscPrice*, *SumCharge*}}.

Given FS and VS, a solution to the fragmentation problem is encoded by a *fragmentation cube*, 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 fragmented views defined by $\boldsymbol{C}$ is

$$VS' = \left\{ v_{jk} \mid \exists j,k \mid \sum_{q_i \in QS} C_{ijk} \geq 1 \right\} \qquad \textbf{(1)}$$

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

A fragmentation cube not only denotes a fragmentation of the views in VS; at the same time, it specifies on which view(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* with reference to QS and VS iff the following constraints are satisfied:

1. for each query, every measure required must be obtained from exactly one view (non ambiguous query execution);
2. for each pattern, each measure must belong to exactly one view (non redundant fragmentation);
3. each view in VS' must be a fragmentation of one or more views in VS (consistency with view materialization).

It should be noted that, if some measures of a fact scheme are used by no query in the workload, they do not generate any minterm, thus, they are not involved in the fragmentation algorithm. These measures will be reconsidered *a posteriori*, after an optimal fragmentation has been determined, by either creating new fragments including them only or by adding them to one of the fragments determined. Of course, if the workload has been properly defined and materialization has been executed correctly, the amount of unused measures should be negligible.

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 on *LINEITEM* ⊗ *SHIPMENT*:

$Meas(q_1)$ = {*Price*, *Qty*, *Discount*}; $Patt(q_1)$ = {*ReturnFlag*, *Status*, *ShipDate*}
$Meas(q_2)$ = {*ExtPrice*, *DiscPrice*}; $Patt(q_2)$ = {*Part*, *Customer*}
$Meas(q_3)$ = {*SumCharge*, *Tax*}; $Patt(q_3)$ = {*Part*, *Customer.Nation*}
$Meas(q_4)$ = {*QtyShipped*, *ShippCost*}; $Patt(q_4)$ = {*Customer.Nation*, *Part.MFGR*, *ShipDate*}
$Meas(q_5)$ = {*ExtPrice*, *DiscPrice*, *ShippCost*}; $Patt(q_5)$ = {*Part.Brand*, *Customer.Nation*}

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*}　　$Patt(v_4)$ = {*Part*, *Customer.Nation*}
$Patt(v_5)$ = {*Part*, *Customer.Nation*}

(for each view, the measures are those of the corresponding fact scheme). Figure 4 shows the fragmentation cube representing a feasible solution to this fragmentation problem, which features five fragmented views:

$Meas(v'_1)$ = {*Price*, *Qty*, *Discount*}; $Patt(v'_1) = Patt(LINEITEM)$
$Meas(v'_2)$ = {*ExtPrice*, *DiscPrice*}; $Patt(v'_2)$ = {*Part*, *Customer*}
$Meas(v'_3)$ = {*SumCharge*, *Tax*}; $Patt(v'_3)$ = {*Part*, *Customer.Nation*}
$Meas(v'_4)$ = {*QtyShipped*, *ShippCost*}; $Patt(v'_4) = Patt(SHIPMENT)$
$Meas(v'_5)$ = {*ExtPrice*, *DiscPrice*, *ShippCost*}; $Patt(v'_5)$ = {*Part.Brand*, *Customer.Nation*}

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

TS \ PS	Patt(LINEITEM)	Patt(SHIPMENT)	Patt(v_3)	Patt(v_4) = Patt(v_5)	Patt(LINEITEM)	Patt(SHIPMENT)	Patt(v_3)	Patt(v_4) = Patt(v_5)	Patt(LINEITEM)	Patt(SHIPMENT)	Patt(v_3)	Patt(v_4) = Patt(v_5)	Patt(LINEITEM)	Patt(SHIPMENT)	Patt(v_3)	Patt(v_4) = Patt(v_5)	Patt(LINEITEM)	Patt(SHIPMENT)	Patt(v_3)	Patt(v_4) = Patt(v_5)
{Disc,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
{ShippCost}	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
{ShippCost,QtyShipped}	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
{DiscPrice,ExtPrice,ShippCost}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	q_1				q_2				q_3				q_4				q_5			

Fig. 4. Fragmentation cube representing a feasible solution

4.2 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. The cost could be defined in several ways, depending on the assumptions made on the DBMS features and, consequently, on the access paths which can be followed to solve the queries. However, we believe that 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 in this paper intentionally abstracts from any assumptions on the access paths. Of course, other cost functions specifically tuned for the different DBMSs could be adopted as well.

The cost function we adopt is based on the number of disk pages in which the fact instances of interest for a given query are stored. In particular, the cost of query q_i within fragmentation $\boldsymbol{C}$ is defined as:

$$cost(q_i, \boldsymbol{C}) = \sum_{P_j \in PS, T_k \in TS} \Phi\left(sel(q_i) \cdot ns(P_j), \left\lceil \frac{ns(P_j)}{\beta_{jk}} \right\rceil \right) \boldsymbol{C}_{ijk} \qquad (2)$$

This formula can be explained as follows:

- $sel(q_i) \cdot ns(P_j)$ is the number of fact instances on pattern P_j involved in q_i.
- β_{jk} is the number of tuples per disk page for view v_{jk} characterized by P_j and T_k: 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 fact instances involved in q_i are stored.

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.

The total cost for the workload QS turns out to be:

$$tcost(QS, \boldsymbol{C}) = \sum_{q_i \in QS} \eta_i \cdot cost(q_i, \boldsymbol{C}) \quad (3)$$

4.3 A branch-and-bound approach

The problem of vertical fragmentation (VFP) can be formulated as follows: Find, for the binary decision array $\boldsymbol{C}$, the value which minimizes function $tcost(QS, \boldsymbol{C})$, subject to constraints (1), (2), (3) expressed in Section 4.1. VFP is a 0-1 integer linear programming problem of kind *set covering with additional constraints*, and is known to be NP-hard [14]. In this section we propose a branch-and-bound approach to solve it optimally.

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

1. 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" cube $\boldsymbol{C}(VFP_\alpha)$, 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 fragmented view on pattern P_j to be used for retrieving some measures T_k to solve a query q_i.
2. A *subproblem selection rule* for choosing the next (most promising) subproblem to be processed. The rule adopted will be explained later.
3. 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 fragmented views defined on the same pattern.
4. A *lower bounding* procedure for calculating the cost of the relaxation. VFR_α consists of one set covering problem for each query q_i, which can be easily solved by adopting one of the algorithms in the literature [4]. Since in solving VFR_α the number of eligible views is higher than that for VFP_α, the cost of VFR_α will necessarily be lower or equal to that of VFP_α.

The branch-and-bound algorithm has the following structure (see Figure 5):

```
// ub: upper bound to the solution cost
// lb: lower bound to the solution cost
// next: next subproblem
// curr: current subproblem
ub ← +∞;
next ← VFP;
while next is not null do
{ curr ← most promising subproblem in SUB(next);
  if C(curr) is a feasible solution to VFP then
```

```
  { if tcost(QS,C(curr)) < ub then
        ub ← tcost(QS,C(curr));
    next ← most promising problem;      // backtracking
  }
  else
  { C' ← solution to the relaxation of curr;
    lb ← tcost(QS,C');
    if lb < ub then
        next ← curr;
    else
        next ← most promising problem;   // backtracking
  }
}
```

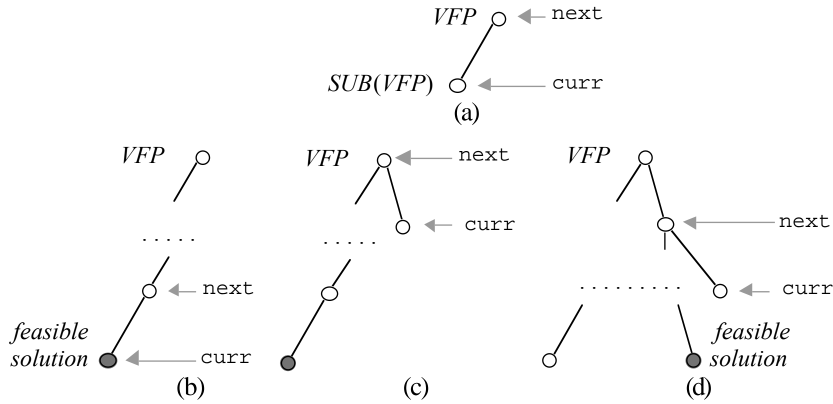

Fig. 5. Four steps in the branch-and-bound algorithm. The circles represent the subproblems generated; the grey ones correspond to the best solution found so far

Besides the cube $\boldsymbol{C}(VFP_\alpha)$, expressing a partial solution, each subproblem VFP_α is also associated to another cube $\boldsymbol{D}(VFP_\alpha)$ which represents the "pool" of possible choices (cube elements which could be set to 1) for generating the subproblems in $SUB(VFP_\alpha)$. Let $VFP_{\alpha+1} \in SUB(VFP_\alpha)$ denote the subproblem generated by setting $\boldsymbol{C}_{\bar{i}\bar{j}\bar{k}}$ to 1; $\boldsymbol{D}(VFP_{\alpha+1})$ is obtained from $\boldsymbol{D}(VFP_\alpha)$ by applying the following algorithm:

```
D(VFP_{α+1}) ← D(VFP_α);
for each k such that T_k ∩ T_{k̄} ≠ ∅ do
{ for each j do D(VFP_{α+1})_{īj k} ← 0;
  if k ≠ k̄ then
     for each u do D(VFP_{α+1})_{i j̄ k} ← 0;
}
```

which drops from the set of possible choices those made unfeasible (due to constraints (1) and (2)) by setting $C_{\bar{i}\bar{j}\bar{k}}$ to 1. The cube $\boldsymbol{D}(VFP)$ associated to the global problem VFP is initialized as follows:

$$D(VFP)_{ijk} = \begin{cases} 1 \text{ if } \left(\exists v \in VS \mid Patt(v) = P_j \wedge Meas(v) \cap T_k \neq \varnothing\right) \\ \qquad \wedge \left(Meas(q_i) \cap T_k \neq \varnothing\right) \wedge \left(Patt(q_i) \geq P_j\right) \\ 0 \text{ otherwise} \end{cases} \quad \textbf{(4)}$$

which encodes all the possible views which may be derived by fragmenting views in VS and all the queries which can be answered on each of them. The starting cube $\boldsymbol{D}(VFP)$ for the example presented in Section 4.1 is shown in Figure 6.

TS \ PS	q1: Patt(LINEITEM)	q1: Patt(SHIPMENT)	q1: Patt(v_3)	q1: Patt(v_4) = Patt(v_5)	q2: Patt(LINEITEM)	q2: Patt(SHIPMENT)	q2: Patt(v_3)	q2: Patt(v_4) = Patt(v_5)	q3: Patt(LINEITEM)	q3: Patt(SHIPMENT)	q3: Patt(v_3)	q3: Patt(v_4) = Patt(v_5)	q4: Patt(LINEITEM)	q4: Patt(SHIPMENT)	q4: Patt(v_3)	q4: Patt(v_4) = Patt(v_5)	q5: Patt(LINEITEM)	q5: Patt(SHIPMENT)	q5: Patt(v_3)	q5: Patt(v_4) = Patt(v_5)
{Disc,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	1	0	1	0	0	0	0	0	0	0	0	0	1	0	1	1
{SumCharge,Tax}	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0
{ShippCost}	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1
{QtyShipped}	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
{ShippCost,QtyShipped}	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
{DiscPrice,ExtPrice,ShippCost}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Fig. 6. Starting cube $\boldsymbol{D}(VFP)$

The algorithm uses two subproblem selection rules for choosing, respectively, the current subproblem from $SUB(next)$ and the next subproblem to be processed. The first rule, in order to select from $\boldsymbol{D}(next)$ the element $C_{\bar{i}\bar{j}\bar{k}}$ to be set to 1 in $\boldsymbol{C}(next)$ to generate $\boldsymbol{C}(curr)$, operates as follows:

```
among the i,j,k such that D(next)_ijk = 1
{ find P_j̄ such that ns(P_j̄) is minimum;
  find ī,k̄ such that Meas(q_ī)∩T_k̄ has max. cardinality;
}
```

The second rule selects, from the set of all the problems generated so far (except those such that all their subproblems have already been generated and processed), the one for which the cost of the relaxation is minimum.

5 Experimental tests

The tests we have carried out are based on the well-known TPC-D benchmark [15], which features two related fact schemes (*LINEITEM* and *PARTSUPPLIER*). The

number of primary fact instances generated for the two schemes is 6.000.000 and 800.000, respectively; the total amount of data is about 1 Gbyte.

We have tested our approach with five workloads, the first including only the 17 TPC-D queries (all with the same frequency), the others being progressively extended with more queries. For each workload, the views to be fragmented have been selected by means of the heuristic approach to view materialization proposed in [2], by considering a global space constraint of 2 Gbytes (1 Gbyte for primary views + 1 Gbyte for secondary views).

Table I reports the results obtained. The percentage saving is evaluated by comparing the workload cost on the set of views selected by the materialization algorithm and the one on the set of fragmented views; the time for finding the optimal fragmentation has been measured on a Pentium II - 300 MHz processor.

Table 1. Results of experimental tests

n. queries in the workload	*n. secondary views*	*n. subproblems generated*	*computing time*	*percentage saving*
17	8	2775	about 1 min	8.2%
25	12	4439	about 2 mins	22.0%
30	13	348925	about 30 mins	20.4%
35	14	51099	about 12 mins	9.0%
40	16	403420	about 75 mins	11.8%

In the following we describe in more detail the last experiment. The total number of fragments obtained is 29, 3 of which are primary; in fact, the primary view on *LINEITEM* is splitted in two fragments with measures {*Discount Price*, *Sum Charge*, *Extended Price*, *Qty*} and {*Discount*, *Price*, *Tax*}, respectively. We consider in particular two representative cases:

- Queries q_{11} and q_{12}, on *PARTSUPPLIER*, are characterized by $Patt(q_{11})$ = {*Part.Container*, *Date.Week*}, $Meas(q_{11})$ = {*Available Qty*, *Supply Cost*} and $Patt(q_{12})$ = {*Part*, *Date.Week*}, $Meas(q_{12})$ = {*Available Qty*}; the frequencies are $\eta_{11} = 10$ and $\eta_{12} = 26$. In the absence of fragmentation, both queries would be executed on the secondary view v characterized by $Patt(v)$ = {*Part*, *Date.Week*}, $Meas(v)$ = *Meas*(*PARTSUPPLIER*) = {*Available Qty*, *Supply Cost*}, yielding a total cost 10×16490+26×323232=8568932. Within the optimal fragmentation, v is partitioned into two fragments v' and v'' characterized by $Meas(v')$ = {*Available Qty*}, $Meas(v'')$ = {*Supply Cost*}. In this case, q_{11} requires both v' and v'' to be read and its cost raises to 32240; on the other hand, q_{12} reads smaller tuples and its cost is decreased to 243386. The total cost is thus decreased to 10×32240+26×243386=6650436.
- Query q_{19}, on *LINEITEM* ⊗ *PARTSUPPLIER*, is characterized by $Patt(q_{19})$ = {*Part.Brand*, *Supplier*}, $Meas(q_{19})$ = {*Discount*, *Available Qty*, *Supply Cost*}. In the absence of fragmentation, this query would require two views to be read, yielding a cost of 98700. Within the optimal fragmentation, these two views are partitioned by eliminating the unused measures and unified; as a result, q_{19} is executed on a fragment v''' characterized by $Meas(v''') = Meas(q_{19})$, yielding a cost of 44860. It should be noted that, if the two views had been partitioned but not unified, the cost would have been 82940.

6 Conclusion

In this paper we have proposed an approach to vertical fragmentation of views in data warehouses. The experimental results presented confirm the utility of the approach in terms of reduction of the cost for executing the expected workload.

Our future work on the topic of DW logical design will address the problem of horizontal fragmentation of views, aimed at enhancing the performance for the queries which operate on subsets of fact instances.

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. Baralis, E., Paraboschi, S., Teniente, E.: Materialized view selection in multidimensional database. In: Proc. 23rd Int. Conf. on Very Large Data Bases, Athens, Greece (1997) 156-165
3. Barquin, R., Edelstein, S.: Planning and Designing the Data Warehouse. Prentice Hall (1996)
4. Beasley, J.E.: An Algorithm for Set Covering Problems. European Journal of Operational Research 31 (1987) 85-93
5. Chu, W.W., Ieong, I.T.: A transaction-based approach to vertical partitioning for relational database system. IEEE Trans. on Software Engineering 19(8) (1993) 804-812
6. Golfarelli, M., Maio, D., Rizzi, S.: The Dimensional Fact Model: a Conceptual Model for Data Warehouses. International Journal of Cooperative Information Systems, 7(2&3) (1998) 215-247
7. Golfarelli, M., Rizzi, S.: Designing the data warehouse: key steps and crucial issues. Journal of Computer Science and Information Management 2(3) (1999)
8. Gupta, A., Mumick, I.S.: Maintenance of materialized views: problems, techniques, and applications. Bulletin of the Technical Committee on Data Engineering 18(2) (1995) 3-18
9. Harinarayan, V., Rajaraman, A., Ullman, J.: Implementing Data Cubes Efficiently. In: Proc. of ACM Sigmod Conf., Montreal, Canada (1996)
10. Kimball, R.: The data warehouse toolkit. John Wiley & Sons (1996)
11. Lin, X., Orlowska, M., Zhang, Y.: A graph-based cluster approach for vertical partitioning in database design. Data & Knowledge Engineering 11 (1993) 151-169
12. Munneke, D., Wahlstrom, K., Mohania, M.: Fragmentation of multidimensional databases. In: Proc. 10th Australasian Database Conf., Auckland, New Zealand (1999) 153-164
13. Özsu, M.T., Valduriez, P.: Principles of distributed database systems. Prentice-Hall Int. Editors (1991)
14. Papadimitriou, C.H., Steiglitz, K.: Combinatorial optimization. Prentice Hall, Englewood Cliffs (1982)
15. Raab, F. (ed.): TPC Benchmark(tm) D (Decision Support). Proposed Revision 1.0. Transaction Processing Performance Council, San Jose, CA 95112 (1995)