Clustering-Based Materialized View Selection in Data Warehouses

Kamel Aouiche, Pierre-Emmanuel Jouve, and Jérôme Darmont

ERIC Laboratory – University of Lyon 2
5, av. Pierre Mendès-France
F-69676 BRON Cedex – FRANCE
{kaouiche, pjouve, jdarmont}@eric.univ-lyon2.fr

Abstract. materialized view selection is a non-trivial task. Hence, its complexity must be reduced. A judicious choice of views must be cost-driven and influenced by the workload experienced by the system. In this paper, we propose a framework for materialized view selection that exploits a data mining technique (clustering), in order to determine clusters of similar queries. We also propose a view merging algorithm that builds a set of candidate views, as well as a greedy process for selecting a set of views to materialize. This selection is based on cost models that evaluate the cost of accessing data using views and the cost of storing these views. To validate our strategy, we executed a workload of decision-support queries on a test data warehouse, with and without using our strategy. Our experimental results demonstrate its efficiency, even when storage space is limited.

1 Introduction

Among the techniques adopted in relational implementations of data warehouses to improve query performance, view materialization and indexing are presumably the most effective ones [14]. Materialized views are physical structures that improve data access time by precomputing intermediary results. Then, user queries can be efficiently processed by using data stored within views and do not need to access the original data. Nevertheless, the use of materialized views requires additional storage space and entails maintenance overhead when refreshing the data warehouse.

One of the most important issues in data warehouse physical design is to select an appropriate set of materialized views, called a configuration of views, which minimizes total query response time and the cost of maintaining the selected views, given a limited storage space. To achieve this goal, views that are closely related to the workload queries must be materialized.

The view selection problem has received significant attention in the literature. Researches about it differ in several points: (1) the way of determining candidate views; (2) the frameworks used to capture relationships between candidate views; (3) the use of mathematical cost models *vs.* calls to the query optimizer; (4) view

selection in the relational or multidimensional context; (5) multiple or simple query optimization; and (6) theoretical or technical solutions.

The classical papers in materialized view selection introduce a lattice framework that models and captures dependency (ancestor or descendent) among aggregate views in a multidimensional context [2, 9, 12, 19]. This lattice is greedily browsed with the help of cost models to select the best views to materialize. This problem has been firstly addressed in one data cube and then extended to multiple cubes [15]. Another theoretical framework called the AND-OR view graph may also be used to capture the relationships between views [5, 8, 13, 20]. The majority of these solutions are theoretical and are not truly scalable. In opposition to these studies, we exploit a query clustering involving similarity and dissimilarity measures defined on the workload queries, in order to capture the relationships existing between the candidate views derived from this workload. This approach is scalable thanks to the low complexity of our clustering (log linear regarding the number of queries and linear regarding the number of attributes).

A wavelet framework for adaptively representing multidimensional data cubes has also been proposed [17]. This method decomposes data cubes into an indexed hierarchy of wavelet view elements that correspond to partial and residual aggregations of data cubes. An algorithm greedily selects a non-expensive set of wavelet view elements that minimizes the average processing cost of the queries defined on the data cubes. In the same spirit, Kotidis *et al.* proposed the Dwarf structure, which compresses data cubes [16]. Dwarf identifies prefix and suffix redundancies within cube cells and factors them out by coalescing their storage. Suppressing redundancy improves the maintenance and interrogation costs of data cubes. These approaches are very interesting, but they are mainly focused on computing efficient data cubes by changing their physical design. In opposition, we aim at optimizing performance in relational warehouses without modifying their design.

Other approaches detect common sub-expressions within workload queries in the relational context [3, 6, 14]. The problem of view selection consists in finding common subexpressions corresponding to intermediary results that are suitable to materialize. However, browsing is very costly and these methods are not truly scalable with respect to the number of queries.

Finally, the most recent approaches are workload-driven. They syntactically analyze the workload to enumerate relevant candidate views [1]. By calling the query optimizer, they greedily build a configuration of the most pertinent views. A workload is indeed a good starting point to predict future queries because these queries are probably within or syntactically close to a previous query workload. In addition, extracting candidate views from the workload ensures that future materialized views will probably be used when processing queries.

Our approach is also workload-driven. Its originality lies in exploiting knowledge about how views can be used to resolve a set of queries to cluster these queries together. For this purpose, we define the notion of query similarity and dissimilarity in order to capture closely related queries. These queries are

grouped in the same cluster and are used to build a set of candidate views. Furthermore, these candidate views are merged to resolve multiple queries. This merging process can be seen as iteratively building a lattice of views. The merging process time can be expensive when the number of candidate views is high. However, we apply merging over candidate views present in each cluster instead of the whole set of candidate views as in [1]. This reduces the complexity of the merging process, since the number of candidate views per cluster is significantly lower.

The remainder of this paper is organized as follows. We first present in Section 2 our materialized view selection strategy. Then, we show in Section 3 how we build a candidate view configuration through our merging process. Next, we detail in Section 4 the cost models used for building the final configuration of views to materialize. To validate our approach, we also present some experiments in Section 7. We finally conclude and provide research perspectives in Section 8.

2 Strategy for materialized view selection

The architecture of our materialized view selection strategy is depicted in Figure 1. We assume that we have a workload composed of representative queries for which we want to select a configuration of materialized views in order to reduce their execution time. The first step is to build, from the workload, a context for clustering. This context is modelled as a matrix having as many lines as the extracted queries and as many columns as the extracted attributes from the whole set of queries. We define similarity and dissimilarity measures that help clustering together relatively similar queries. We apply a merging process on each query cluster to build a configuration of candidate views. Then, the final view configuration is created with a greedy algorithm. This step exploits cost models that evaluate the cost of accessing data using views and the cost of their storage.

2.1 Query workload analysis

The workloads we consider are sets of GPSJ (Generalized Projection-Selection-Join) queries. A GPSJ query q is composed of joins, selection predicates and aggregations. As such, it may be expressed in relational algebra over a star schema as follows: $q = \pi_{G,M}\sigma_S(F \bowtie D_1 \bowtie D_2 \bowtie \ldots \bowtie D_d)$, where S is a conjunction of simple range predicates on dimension table attributes, G is a set of attributes from dimension tables D_i (grouping set), and M is a set of aggregated measures each defined by applying aggregation operator to a measure in fact table F. For example, query q_1 in Figure 2 may be expressed as follows: $q_1 = \pi_{sales.time_id,sum(quantity_sold)}\sigma_{fiscal_day=2}(sales \bowtie times)$.

The first step consists in extracting from the workload the attributes that are representative of each query. We mean by representative attributes those that are present in `Where` (join and selection predicate attributes) and `Group by` clauses. We also save for each query their aggregation operators and joined

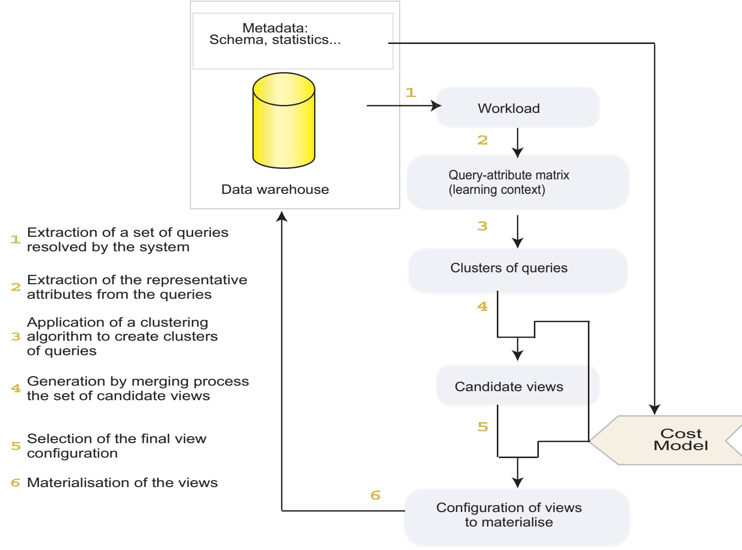

Fig. 1. Strategy of materialized view selection

tables. A query q_i is then seen as a line in a matrix composed of cells that correspond to the representative attributes. The general term q_{ij} of this matrix is set to 1 if the extracted attribute is present in the query and to 0 otherwise. This matrix represents our clustering context. Moreover, we store in an appendix matrix the existing associations between the join attributes and queries, in the same manner. We illustrate this step by an example: from the workload shown in Figure 2, we build the clustering context depicted in Figure 3.

2.2 Building the candidate view set

In practice, it is hard to search all the views that are syntactically relevant (candidate views) from the workload queries, because the search space is very large [1]. To reduce the size of this space, we propose to cluster the queries. Indeed, we group in a same cluster all the queries that are closely similar. Closely similar queries are queries having a close binary representation in the query-attribute matrix. Two closely similar queries can be resolved by using only one materialized view. Used within a clustering process, the similarity and dissimilarity measures defined in the next section ensures that queries within a same cluster are strongly related to each others whereas queries from different clusters are significantly distant to each others.

Similarity measure. Let QA be a query-attribute matrix that has a set of queries $Q = \{q_i, i = 1..n\}$ as rows and a set of attributes $A = \{a_j, j = 1..p\}$ as columns. The value q_{ij} is equal to 1 if attribute a_j is extracted from query

```
(q1) select sales.time_id, sum(quantity_sold) from sales, times
     where sales.time_id = times.time_id and times.day_year = 2
     group by sales.time_id;
(q2) select sales.prod_id, sum(amount_sold) from sales, products, promotions
     where sales.prod_id = products.prod_id and sales.promo_id = promotions.promo_id and
     promotions.promo_category = 'newspaper'
     group by sales.prod_id;
(q3) select sales.cust_id, sum(amount_sold) from sales, customers, products, times
     where sales.cust_id = customers.cust_id and sales.prod_id = products.prod_id and
     sales.time_id = times.time_id and times.fiscal_day = 3 and customers.cust_marital_status
     ='single' and products.prod_category ='Women'
     group by sales.cust_id;
...
```

Fig. 2. Example of workload

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}
q_1	1	1	1	0	0	0	0	0	0	0	0	0
q_2	0	0	0	1	0	1	1	1	1	0	0	0
q_3	0	0	0	1	0	1	1	1	1	1	1	1
..												
..												

a_1 times.time_id a_2 times.fiscal_day
a_3 sales.time_id a_4 products.prod_id
a_5 products.prod_category a_6 sales.promo_id
a_7 promotions.promo_id a_8 sales.prod_id
a_9 promotions.promo_category a_{10} sales.cust_id
a_{11} customers.cust_marital_status a_{12} customers.cust_id

Fig. 3. Example of clustering context

q_i. Otherwise, q_{ij} is equal to 0. We describe query q_i by a vector of p values $q_i = [q_{i1}, ..., q_{ip}]$. These p values describe respectively the presence ($q_{ij} = 1$) or absence ($q_{ij} = 0$) of attribute a_j. This description model helps comparing two queries. Then, for example, we can consider queries q_1 and q_2 as closely similar if vectors $[q_{11}, ..., q_{1p}]$ and $[q_{21}, ..., q_{2p}]$ have the majority of their cells equal. This introduces the notion similarity and dissimilarity between queries.

Similarity and dissimilarity between queries. We define the notion of similarity and dissimilarity between queries by two functions $\delta_{sim_j}(q_k, q_l)$ and $\delta_{dissim_j}(q_k, q_l)$ that measure the similarity between two queries q_k and q_l with respect to attribute a_j.

$$\delta_{sim_j}(q_k, q_l) = \begin{cases} 1 \text{ if } q_{kj} = q_{lj} = 1 \\ 0 \text{ otherwise} \end{cases}$$

This first function defines the notion of similarity between q_k and q_l following attribute a_j: two queries q_k and q_l are considered similar regarding attribute a_j if and only if $q_{kj} = q_{lj} = 1$, i.e., attribute a_j is extracted from both queries.

$$\delta_{dissim_j}(q_k, q_l) = \begin{cases} 0 \text{ if } q_{kj} = q_{lj} \\ 1 \text{ if } q_{kj} \neq q_{lj} \end{cases}$$

This second function defines the notion of dissimilarity between queries q_k and q_l according to attribute a_j: two queries q_k and q_l are considered dissimilar according to attribute a_j if only and if $q_{kj} \neq q_{lj}$, i.e., if one and only one of the queries does not contain a_j. Note that there is not a complete symmetry between the notion of similarity and dissimilarity: non similar queries according to an attribute are not necessarily dissimilar according to this attribute. For example, let

q_1 and q_2 be queries such that $q_{1j} = 0$ and $q_{2j} = 0$, respectively. Then we have $\delta_{sim_j}(q_1, q_2) = 0$ (q_1 and q_2 are not considered similar) and $\delta_{dissim_j}(q_1, q_2) = 0$ (q_1 and q_2 are not considered dissimilar). This absence of full symmetry underlines the fact that the absence of the same attribute in two queries does not give an element of similarity or dissimilarity between these queries.

These measures can be extended to an attribute set $A = \{a_1, \ldots, a_p\}$ such that we get the degree of global similarity and dissimilarity between two queries: $sim(q_k, q_l) = \sum_{j=1}^{p} \delta_{sim_j}(q_k, q_l)$ and $dissim(q_k, q_l) = \sum_{j=1}^{p} \delta_{dissim_j}(q_k, q_l)$, where $0 \leq sim(q_k,\ q_l) \leq p$ and $0 \leq dissim(q_k, q_l) \leq p$. Hence, the closer $sim(q_a, q_b)$ (resp. $dissim(q_a,\ q_b)$) is to p the more q_a and q_b can be considered globally similar (resp. dissimilar).

Similarity and dissimilarity between query sets. As we do for two queries, we introduce two functions that take into account the degree of similarity and dissimilarity between two query sets. A set of queries (subset of Q) is denoted C_a. In order to translate the level of similarity (resp. dissimilarity) between query sets, we use function $Sim(C_a, C_b)$ (resp. $Dissim(C_i, C_j)$) that determines the number of similarities (resp. dissimilarities) between two different sets of queries C_a and C_b ($C_a \neq C_b$):

$$Sim(C_a, C_b) = \sum_{q_k \in C_a, q_l \in C_b} sim(q_k, q_l)$$

$$Dissim(C_a, C_b) = \sum_{q_k \in C_a, q_l \in C_b} dissim(q_k, q_l)$$

where $0 \leq Sim(C_a, C_b) \leq card(C_a) \times card(C_b) \times p$ and $0 \leq Dissim(C_a, C_b) \leq card(C_a) \times card(C_b) \times p$. Hence, the closer $Sim(C_a, C_b)$ (resp. $Dissim(C_a, C_b)$) is to $card(C_a) \times card(C_b) \times p$ the more C_a and C_b can be considered similar (resp. dissimilar).

Similarity and dissimilarity within a query set. The notion of similarity (resp. dissimilarity) within a query set corresponds to the number of similarities (resp. dissimilarities) between queries of a same set C_a. It consists of an extension of the similarity and dissimilarity functions defined between query sets: $Sim(C_a) = \sum_{q_k \in C_a, q_l \in C_a, k<l} sim(q_k, q_l)$ and $Dissim(C_a) = \sum_{q_k \in C_a, q_l \in C_a, k<l} dissim(q_k, q_l)$, where $0 \leq Sim(C_a) \leq \frac{card(C_a) \times (card(C_a)-1) \times p}{2}$ and $0 \leq Dissim(C_a) \leq \frac{card(C_a) \times (card(C_a)-1) \times p}{2}$. Hence, the close $Sim(C_a)$ (resp. $Dissim(C_a)$) is to $\frac{card(C_a) \times (card(C_a)-1) \times p}{2}$ the more C_a contains queries that are globally similar (resp. dissimilar).

Query clustering. Clustering consists in determining a kind of natural grouping of objects (here, queries) that shows the internal structure of data. These groups must be such as they are composed of objects with high similarity and objects from different clusters present a high dissimilarity.

Let us consider clustering P_h of a query set, a quality measure of this clustering can be built as follows:

$$Q(P_h) = \sum_{\substack{a=1..z,\\ b=1..z, a<b}} Sim(C_a, C_b) + \sum_{a=1}^{z} Dissim(C_a)$$

$$0 \leq Q(P_h) \leq \sum_{i=1..z, j=1..z, i<j} card(C_i) \times card(C_j) \times p + \sum_{i=1}^{z} \frac{card(C_i) \times (card(C_i)-1) \times p}{2}$$

This measure permits to capture the natural aspect of a clustering. Hence, $Q(P_h)$ measures simultaneously similarities between queries within the same cluster and dissimilarities between queries within different clusters. Thus, $Q(P_h)$ evaluates simultaneously the homogeneity of clusters as well as the heterogeneity between clusters. Therefore, the clustering presenting a high intra-cluster homogeneity and a high inter-cluster disparity has a weak value of $Q(P_h)$ and thereby appears as the most natural.

Jouve and Nicoloyannis proposed such a solution in the Kerouac clustering algorithm and its associated clustering quality measure [10]. We have chosen this algorithm because it has several interesting properties: (1) its computational complexity is relatively low (log linear regarding the number of queries and linear regarding the number of attributes) ; (2) it can deal with a high number of objects (queries) ; (3) it can deal with distributed data [11].

3 View merging process

If we materialize all the different views derived from the query clusters obtained in the previous step, we can obtain a high number of materialized views, especially if the number of queries within the workload is high. A view configuration obtained this way would not be very relevant if the storage space allotted by the data warehouse administrator was limited. Instead of materializing each view, it is better to only materialize views that can be used to resolve multiple queries. To solve this problem, we must enumerate the space of views that can be merged, determine how to guide the merging process, and finally build the set of merged views. View merging is relevant if the queries are strongly similar. As we cluster together closely similar queries, it is logical to apply the merging process on the set of queries present in each cluster. This significantly reduces the number of possible combinations when merging views. We detail in the following sections how we merge two views and then generalize this process to many views.

Merging of view couples. The merging of two views must ensure these conditions: (1) all queries resolved by each view must also be resolved by the merged view, and (2) the cost of using the couple of views must not be significatively greater than the cost obtained when using the merged view. Let v_1 and v_2 be a couple of views of the same cluster and $s_{11}, \ldots, s_{1m}$ the selection predicates that are in v_1 and not in v_2. In a dual way, let $s_{21}, \ldots, s_{1n}$ be the selection conditions

present in v_2 and not in v_1. Merged view v_{12} is obtained by applying Algorithm 1.

Algorithm 1 *Merge_View_Pair*(v_1, v_2)

1: put v_1 and v_2 aggregation operations in v_{12} operation aggregations
2: put the union of projection and group by attributes v_1 and v_2 in projection and group by clause of v_{12}
3: put all attributes $s_{11}, \ldots, s_{1m}$ and $s_{21}, \ldots, s_{1n}$ in the group by clause of v_{12}
4: put the selection predicates shared between v_1 and v_2 in the selection predicate clause of v_{12}

Algorithm 2 *Mergin_View_Generation*

1: $M = V_1$
2: **for** ($k = 2$; $V_{k-1} \neq \emptyset$; $k++$) **do**
3: $\quad C_k = \texttt{View_Gen}(V_{k-1})$
4: $\quad M \leftarrow M \cup C_k$
5: $\quad$ **for all** (view $v \in M$) **do**
6: $\quad\quad$ Remove the parents of v from M
7: $\quad$ **end for**
8: **end for**
9: **return** M

The merging of two views v_1 and v_2 is effective if $cost(v_{12}) \geq ((cost(v_1) + cost(v_2)) * x)$. Cost computation is detailed in Section 4. The value of x is fixed empirically by the administrator. If it is small (resp. high), we privilege (resp. disadvantage) view merging.

Property 1 *The view obtained by merging views v_1 and v_2 is the smallest view that resolves the query resolved by both v_1 and v_2.*

Proof. To show that the view obtained by merging views v_1 and v_2 is the smallest view, we have to show that there is no view v'_{12} such as the data within v'_{12} are also included within v_{12}. We denote respectively views v_1, v_2 and v_{12} $\pi_{G_1,M_1}\sigma_{S_1}(F \bowtie \ldots)$, $\pi_{G_2,M_2}\sigma_{S_2}(F \bowtie \ldots)$ and $\pi_{G_{12},M_{12}}\sigma_{S_{12}}(F \bowtie \ldots)$, respectively, where:
– G_1, G_2 are respectively the attribute set of the group by clause of views v_1 and v_2;
– S_1, S_2 are respectively the attribute set of the selection predicates of v_1 and v_2;
– $G_{12} = G_1 \cup G_2 \cup (S_1 \cup S_2 - S_1 \cap S_2)$ is the attribute set of the group by clause of merged view v_{12};
– $S_{12} = S_1 \cap S_2$ is the set of attribute selection predicates within merged view v_{12}.

Note that sets G_{12} and S_{12} are obtained by applying lines 1 and 2 of Algorithm 1. Let us now assume that the data in view v'_{12}, denoted $\pi_{G'_{12},M'_{12}}\sigma_{S'_{12}}(F \bowtie \ldots)$ are all in v_{12}. This means that both of the following conditions hold: (1) $G_{12} \subset G'_{12}$, (2) $S_{12} \supset S'_{12}$.

From the first condition, there is at least one attribute x such that $x \in G'_{12}$ and $x \notin G_{12}$. As we have $x \notin G_{12}$, then $x \notin G_1$, $x \notin G_2$ and $x \notin S_1 \cup (S_2 - S_1 \cap S_2)$ because $G_{12} = G_1 \cup G_2 \cup (S_1 \cup S_2 - S_1 \cap S_2)$. As $x \notin G_1$ and $x \notin G_2$, then x is not in any clause of v_2. This means that $x \notin G'_{12}$, which contradicts condition $x \in G'_{12}$.

From the second condition, there is at least one attribute y such that $y \in S_{12}$ and $y \notin S'_{12}$. As we have $y \in S_{12}$, then $y \in S_1$ and $y \in S_2$ because $S_{12} = S_1 \cup S_2$. As $y \in S_1$ and $y \in S_2$, then y must be in all the predicates of the views obtained by merging v_1 and v_2. This means that $y \in S'_{12}$, which contradicts condition $y \notin S'_{12}$.

Merged view generation algorithm. The algorithm of view generation by merging is similar to algorithms searching for frequent itemsets. A frequent itemset lattice looks like a lattice of views within a given cluster. The lattice nodes represent the space of views obtained by merging.

Algorithm 3 *Function View_Gen(V_{k-1})*

```
1: C_k = ∅
2: for all (view v ∈ V_{k-1}) do
3:   for all (view u ∈ V_{k-1}) do
4:     if (v[1] = u[1] ∧ ... ∧ v[k-2] = u[k-2] ∧ v[k-1] < u[k-1]) then
5:       c = Merge_View_Pair (v,u)
6:       if (cost(c) ≥ ((cost(v)+cost(u))*x)) then
7:         C_k = C_k ∪ {c}
8:       end if
9:     end if
10:  end for
11: end for
12: return C_k
```

Algorithm 4 *View_Configuration_Construction*

```
1: S ← ∅
2: repeat
3:   v_max ← ∅
4:   F_max ← 0
5:   for all v_j ∈ V − S do
6:     if F_/S(v_j) > F_max then
7:       F_max ← F_/S(v_j)
8:       v_max ← v_j
9:     end if
10:  end for
11:  if F_/S(v_max) > 0 then
12:    S ← S ∪ {v_max}
13:  end if
14: until (F_/S(v_max) ≤ 0 or V − S = ∅)
```

The algorithm of view generation by merging (Algorithm 3) uses an iterative approach by level to generate a new view. It explores the view lattice in breadth first. The input of the algorithm is V_1, a set of candidate views extracted from a given cluster. This algorithm outputs a set of candidate views obtained by merging. In the kth iteration, view set V_{k-1} obtained by merging the $k-1^{th}$ level's views from the lattice (computed in the last step) is used to generate the set C_k of k-candidate views. This set is added to set M (line 4). The parents of each view obtained by merging are then removed from set M (lines 5 to 7).

The function for view generation by merging `View_Gen(`V_{k-1}`)`, called on line 3, takes as argument V_{k-1} and returns C_k. Two views v and u within V_{k-1} form a k-view c if and only if they have $(k-2)$ views in common. This is expressed using a lexicographic order in the condition of line 3. We denote by $v[1] \dots v[k-2]v[k-1]$ the merged views in the kth iteration that are used to derive v. Function `Merge_View_Pair`(v,u) (Algorithm 1) called on line 5 of `View_Gen` generates a new view c. The condition of line 6 ensures, after generating a k-view from two $k-1$-views, that the candidate view does not have a cost greater than the cost of its parents.

4 Cost models

The number of candidate views is generally as high as the input workload is large. Thus, it is not feasible to materialize all the proposed views because of storage space constraints. To circumvent these limitations, we use cost models allowing to conserve only the most pertinent views. In most data warehouse cost models [7], the cost of a query q is assumed to be proportional to the number of tuples in the view on which q is executed. In the following section, we detail the cost model that estimates the size of a given view.

Let $ms(F)$ be the maximum size of fact table F, $|F|$ be the number of tuples in F, D_i_ID be a primary key of dimension D_i, $|D_i_ID|$ be the cardinality of the attribute(s) that form the primary key, and N be the number of dimension tables. Then, $ms(F) = \prod_{i=1}^{N} |D_i_ID|$.

Let $ms(V)$ be the maximum size of a given view v that has attributes $a_1, a_2, \ldots, a_k$ in its group by clause, where k is the number of attributes in v and $|a_i|$ is the cardinality of attribute a_i. Then, $ms(v) = \prod_{i=1}^{k} |a_i|$.

Golfarelli *et al.* [7] proposed to estimate the number of tuples in a given view v by using Yao's formula [21] as follows:
$|v| = ms(v) \times \left[1 - \prod_{i=1}^{|F|} \frac{ms(F) \times d - i + 1}{ms(F) - i + 1}\right]$, where $d = 1 - \frac{1}{ms(v)}$. If $\frac{ms(F)}{ms(v)}$ is sufficiently large, then Cardenas' formula [4] approximation gives:
$|v| = ms(v) \times \left(1 - \left(1 - \frac{1}{ms(v)}\right)^{|F|}\right)$, where $d = 1 - \frac{1}{ms(v)}$.

Cardenas' and Yao's formulaes are based on the assumption that data is uniformly distributed. Any skew in the data tends to reduce the number of tuples in the aggregate view. Hence, the uniform assumption tends to overestimate the size of the views and give a crude estimation. However, they have the advantage to be simple to implement and fast to compute. In addition, because of the modularity of our approach, it is easy to replace the cost model module by another more accurate one.

From the number of tuples in v, we estimate its size, in bytes, as follows: $size(v) = |v| \times \sum_{i=1}^{c} size(c_i)$, where $size(c_i)$ denotes the size, in bytes, of column c_i of v, and c is the number of columns in v.

5 Objective functions

In this section, we describe three objective functions to evaluate the variation of query execution cost, in number of tuples to read, induced by adding a new view. The query execution cost is assimilated to the number of tuples in the fact table when no view is used or to the number of tuples in view(s) otherwise. The workload execution cost is obtained by adding all execution costs for each query within this workload.

The first objective function advantages the views providing more profit while executing queries, the second one advantages the views providing more benefit and occupying the smallest storage space, and the third one combines the first two in order to select at first all the views providing more profit and then keep only those occupying the smallest storage space when this resource becomes critical. The first function is useful when storage space is not limited, the second one is useful when storage space is small and the third one is interesting when storage space is larger.

5.1 Profit objective function

Let $V = \{v_1, ..., v_m\}$ be a candidate view set, $Q = \{q_1, ..., q_n\}$ a query set (a workload) and S a final view set to build. The profit objective function, noted P, is defined as follows:
$P_{/S}(v_j) = \left(C_{/S}(Q) - C_{/S \cup \{v_j\}}(Q) - \beta\, C_{maintenance}(\{v_j\})\right)$, where $v_j \notin S$.
– $C_{/S}(Q)$ denotes the query execution cost when all views in S are used. If this set is empty, $C_{/\emptyset}(Q) = |Q| \times |F|$ because all the queries are resolved by accessing

fact table $|F|$. When a view v_j is added to S, $C_{/S\cup\{v_j\}}(Q) = \sum_{k=0}^{|Q|} C(q_k, \{v_j\})$ denotes the query execution cost for the views that are in $S \cup \{v_j\}$. If query q_k exploits v_j, the cost $C(q_k, \{v_j\})$ is then equal to C_{v_j} (number of tuples in v_j). Otherwise, $C(q_k, \{v_j\})$ is equal to the minimum value between $|F|$ and values of $C(q_k, \{v\})$ (executing cost of q_k exploiting $v \in S$ with $v \neq v_j$).
– Coefficient $\beta = |Q|\, p(v_j)$ estimates the number of updates for view v_j. The update probability $p(v_j)$ is equal to $\frac{1}{number\ of\ views} \frac{\%update}{\%query}$, where $\frac{\%update}{\%query}$ represents the proportion of updating *vs.* querying the data warehouse.
– $C_{maintenance}(\{v_j\})$ represents the maintenance cost for view v_j.

5.2 Profit/space ratio objective function

If view selection is achieved under a space constraint, the profit/space objective function $R_{/S}(v_j) = \frac{P_{/S}(v_j)}{size(v_j)}$ is used. This function computes the profit provided by v_j in regard to the storage space $size(v_j)$ that it occupies.

5.3 Hybrid objective function

The constraint on the storage space may be relaxed if this space is relatively large. The hybrid objective function H does not penalize space–"greedy" views if the ratio $\frac{remaining_space}{storage_space}$ is lower or equal than a given threshold α $(0 < \alpha \leq 1)$, where *remaining_space* and *storage_space* are respectively the remaining space after adding v_j and the allotted space needed for storing all the views. This function is computed by combining the two functions P and R as follows:

$$H_{/S}(v_j) = \begin{cases} P_{/S}(v_j) \text{ if } \frac{remaining_space}{storage_space} > \alpha, \\ R_{/S}(v_j) \text{ otherwise.} \end{cases}$$

6 View selection algorithm

The view selection algorithm (Algorithm 4) is based on a greedy search within the candidate view set V. Objective function F must be one of the functions P or R described previously. If R is used, we add to the algorithm's input the space storage M allotted for views.

In the first algorithm iteration, the values of the objective function are computed for each view within V. The view v_{max} that maximizes F, if it exists ($F_{/S}(v_{max}) > 0$), is then added to S. If R is used, the whole space storage M is decreased by the amount of space occupied by v_{max}.

The function values of F are then recomputed for each remaining view in $V - S$ since they depend on the selected views present in S. This helps taking into account the interactions that probably exist between the views.

We repeat these iterations until there is no improvement ($F_{/S}(v) \leq 0$) or until all views have been selected ($V - S = \emptyset$). If function R is used, the algorithm also stops when storage space is full.

7 Experiments

In order to validate our approach for materialized view selection, we have run tests on a 1 GB data warehouse implemented within Oracle 9i, on a Pentium 2.4 GHz PC with a 512 MB main memory and a 120 GB IDE disk. This data warehouse is composed of the fact table `Sales` and five dimensions `Customers`, `Products`, `Times`, `Promotions` and `Channels`. We executed on our data warehouse a workload composed of sixty-one decision-support queries involving aggregation operations and several joins between the fact table and dimension tables. Due to space constraints, the data warehouse schema and the detail of each workload query are available at `http://eric.univ-lyon2.fr/~kaouiche/adbis.pdf`. Our experiments are based on an ad-hoc benchmark because, as far as we know, there is no standard benchmark for data warehouses. TPC-R [18] has no multidimensional schema and does not qualify, for instance.

We first applied our selection strategy with the profit function. This function gives us the maximal number of materialized views (twelve views) because it does not specify any storage space constraint. This point gives us the upper boundary of the storage space occupation. Then, we applied the profit/space ratio and hybrid functions under a storage space constraint. We have measured query execution time with respect to the percentage of storage space allotted for materialized views. This percentage is computed from the upper boundary computed when applying the profit function. This helps varying storage space within a wider interval.

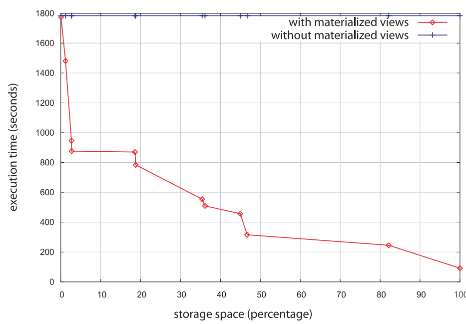

Fig. 4. Profit/space ratio function

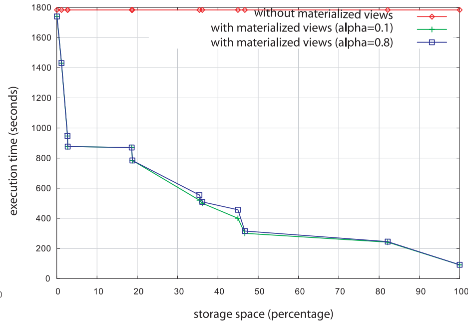

Fig. 5. Hybrid function

Ratio profit/space function experiment. We plotted in Figure 4 the variation of workload execution time with respect to the storage space allotted for materialized views. This figure shows that the selected views improve query execution time. Moreover, execution time decreases when storage space occupation increases. This is predictable because we create more materialized views when storage space is large and thereby better improve execution time. We also observe

that the maximal gain is equal to 94.86%. It is reached for a space occupation of 100% (no constraint on storage space). This case is also reached when using the profit function, because it corresponds to the upper boundary.

Hybrid function experiment. We repeated the previous experiment with the hybrid objective function. We varied the value of parameter α between 0.1 and 1 by 0.1 steps. The obtained results with $\alpha \in [0.1, 0.7]$ and $\alpha \in [0.8, 1]$ are respectively equal to those obtained with $\alpha = 0.1$ and $\alpha = 0.8$. Thus, we plotted in Figure 5 only the results obtained with $\alpha = 0.1$ and $\alpha = 0.7$. This figure shows that for percentage values of space storage under 18.6%, the hybrid function with $\alpha = 0.1$ and $\alpha = 0.8$ behaves as the ratio function. When the storage space becomes critical, the hybrid function behaves as the ratio profit/space function. On the other hand, for the percentage values of storage space greater than 18.6%, the results obtained with $\alpha = 0.8$ are slightly better than those obtained with $\alpha = 0.1$. This is explained by the fact that for the high values of α, the hybrid function chooses the views providing the most profit and thereby improving the best the execution time. The maximal gain in execution time observed for the values 0.1 and 0.8 of α is equal to 96%.

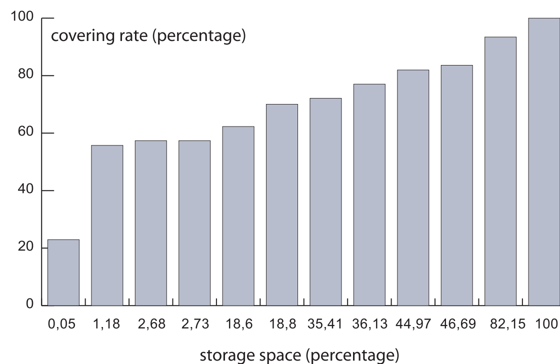

Fig. 6. Query covering rate by the selected materialized views

Selected view pertinence experiment. In order to show if our strategy provides pertinent views for a given workload, we measured the covering rate of the workload query results by the selected views. We mean by covering rate the ratio between the number of queries resolved from the materialized views and the total number of queries within the workload. Thus, the highest the rate value, the most pertinent the selected views. In this experiment, the percentage of storage space is also computed from the upper boundary. We plotted in Figure 6 the covering rate according to storage space occupation. This figure shows that the covering rate increases with storage space. When storage space gets larger,

we materialize more views and thereby we recover more query results from these views. When materializing all the views (100% storage space occupation), all the data corresponding to query results are recovered from the materialized views. This shows that, without storage space constraint, the selected views are pertinent. For example, for 0.05% storage space occupation, 22.95% of the query results are recovered from the selected views. This shows that, even for a limited storage space, our strategy helps building views that cover a maximum number of queries. This experiment shows that materialized view selection based on workload syntactical analysis is efficient to guarantee the exploitation of the selected views by the workload queries.

8 Conclusion

In this paper, we presented an automatic strategy for materialized view selection in data warehouses. This strategy exploits the results of clustering applied on a given workload to build a set of syntactically relevant candidate views. Our experimental results show that our strategy guarantees a substantial gain in performance. It also shows that the idea of using data mining techniques for data warehouse auto-administration is a promising approach.

This work opens several future research axes. First, we are still currently experimenting in order to better evaluate system overhead in terms of materialized view building and maintenance. The maintenance cost is currently derived from the query frequencies (Section 4). We are envisaging a more accurate cost model to estimate update costs. We also plan to compare our approach to other materialized view selection methods. Furthermore, it could be interesting to design methods that select both indexes and materialized views, since these data structures are often used together. More precisely, we are currently developing methods to efficiently share the available storage space between indexes and views. Finally, our strategy is applied on a workload that is extracted from the system during a given period of time. We are thus performing static optimization. It would be interesting to make our strategy dynamic and incremental, as proposed in [12]. Studies dealing with dynamic or incremental clustering may be exploited. Entropy-based session detection could also be beneficial to determine the best moment to run view reselection.

References

1. S. Agrawal, S. Chaudhuri, and V. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *26th International Conference on Very Large Data Bases (VLDB 2000), Cairo, Egypt*, pages 496–505, 2000.
2. E. Baralis, S. Paraboschi, and E. Teniente. Materialized views selection in a multidimensional database. In *23rd International Conference on Very Large Data Bases (VLDB 1997), Athens, Greece*, pages 156–165, 1997.
3. X. Baril and Z. Bellahsene. Selection of materialized views: a cost-based approach. In *15th International Conference (CAiSE 2003), Klagenfurt, Austria*, pages 665–680, 2003.

4. A. F. Cardenas. Analysis and performance of inverted data base structures. *Communication in ACM*, 18(5):253–263, 1975.
5. G. K. Y. Chan, Q. Li, and L. Feng. Design and selection of materialized views in a data warehousing environment: a case study. In *2nd ACM international workshop on Data warehousing and OLAP (DOLAP 1999), Kansas City, USA*, pages 42–47, 1999.
6. J. Goldstein and P. Larson. Optimizing queries using materialized views: a practical, scalable solution. In *ACM SIGMOD international conference on Management of data (SIGMOD 2001), Santa Barbara, USA*, pages 331–342, 2001.
7. M. Golfarelli and S. Rizzi. A methodological framework for data warehouse design. In *1st ACM international workshop on Data warehousing and OLAP (DOLAP 1998), New York, USA*, pages 3–9, 1998.
8. H. Gupta and I. S. Mumick. Selection of views to materialize in a data warehouse. *IEEE Transactions on Knowledge and Data Engineering*, 17(1):24–43, 2005.
9. V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *ACM SIGMOD International Conference on Management of data (SIGMOD 1996), Montreal, Canada*, pages 205–216, 1996.
10. P. Jouve and N. Nicoloyannis. KEROUAC: an algorithm for clustering categorical data sets with practical advantages. In *International Workshop on Data Mining for Actionable Knowledge (DMAK'2003, in conjunction with PAKDD03)*, 2003.
11. P. Jouve and N. Nicoloyannis. A new method for combining partitions, applications for distributed clustering. In *International Workshop on Paralell and Distributed Machine Learning and Data Mining (ECML/PKDD03)*, pages 35–46, 2003.
12. Y. Kotidis and N. Roussopoulos. DynaMat: A dynamic view management system for data warehouses. In *ACM SIGMOD International Conference on Management of Data, (SIGMOD 1999), Philadelphia, USA*, pages 371–382, 1999.
13. T. P. Nadeau and T. J. Teorey. Achieving scalability in OLAP materialized view selection. In *5th ACM International Workshop on Data Warehousing and OLAP (DOLAP 2002), McLean, USA*.
14. S. Rizzi and E. Saltarelli. View materialization vs. indexing: Balancing space constraints in data warehouse design. In *15th International Conference (CAiSE 2003), Klagenfurt, Austria*, pages 502–519, 2003.
15. A. Shukla, P. Deshpande, and J. F. Naughton. Materialized view selection for multi-cube data models. In *7th International Conference on Extending DataBase Technology (EDBT 2000), Konstanz, Germany*, pages 269–284, 2000.
16. Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Dwarf: shrinking the petacube. In *ACM SIGMOD International Conference on Management of Data (SIGMOD 2002), Madison, USA*, pages 464–475, 2002.
17. J. R. Smith, C.-S. Li, and A. Jhingran. A wavelet framework for adapting data cube views for OLAP. *IEEE Transactions on Knowledge and Data Engineering*, 16(5):552–565, 2004.
18. Transaction Processing Council. *TPC Benchmark R Standard Specification*, 1999.
19. H. Uchiyama, K. Runapongsa, and T. J. Teorey. A progressive view materialization algorithm. In *2nd ACM International Workshop on Data warehousing and OLAP (DOLAP 1999), Kansas City, USA*, pages 36–41, 1999.
20. S. R. Valluri, S. Vadapalli, and K. Karlapalem. View relevance driven materialized view selection in data warehousing environment. In *30th Australasian conference on Database technologies, Melbourne, Australia*, pages 187–196, 2002.
21. S. B. Yao. Approximating block accesses in database organizations. *Communication in ACM*, 20(4):260–261, 1977.