Data Mining-based Materialized View and Index Selection in Data Warehouses

Kamel Aouiche and Jérôme Darmont
University of Lyon (ERIC Lyon 2)
5 avenue Pierre Mendès-France
69676 Bron Cedex
France
E-mail: kaouiche@eric.univ-lyon2.fr, jerome.darmont@univ-lyon2.fr

Abstract

Materialized views and indexes are physical structures for accelerating data access that are casually used in data warehouses. However, these data structures generate some maintenance overhead. They also share the same storage space. Most existing studies about materialized view and index selection consider these structures separately. In this paper, we adopt the opposite stance and couple materialized view and index selection to take view-index interactions into account and achieve efficient storage space sharing. Candidate materialized views and indexes are selected through a data mining process. We also exploit cost models that evaluate the respective benefit of indexing and view materialization, and help select a relevant configuration of indexes and materialized views among the candidates. Experimental results show that our strategy performs better than an independent selection of materialized views and indexes.

Keywords: Data warehouses, Performance optimization, Materialized views, Indexes, Data mining, Cost models.

1 Introduction

Large-scale usage of databases in general and data warehouses in particular requires an administrator whose principal role is data management, both at the logical level (schema definition) and physical level (files and disk storage), as well as performance optimization. With the wide development of Database Management Systems (DBMSs), minimizing the administration function has become crucial (Chaudhuri & Narasayya, 1997). One important administration task is the selection of suitable physical structures to improve system performance by minimizing data access time (Finkelstein, Schkolnick, & Tiberio, 1988).

Among techniques adopted in data warehouse relational implementations for improving query performance, view materialization and indexing are presumably the most effective (Rizzi

& Saltarelli, 2003). Materialized views are physical structures that improve data access time by precomputing intermediary results. Therefore, end-user queries can be efficiently processed through data stored in views and do not need to access the original data. Indexes are also physical structures that allow direct data access. They avoid sequential scans and thereby reduce query response time. Nevertheless, exploiting either materialized views or indexes requires additional storage space and entails maintenance overhead when refreshing the data warehouse. The issue is thus to select an appropriate configuration (set) of materialized views and indexes that minimizes query response time and the selected data structures' maintenance cost, given a limited storage space.

The literature regarding materialized view and index selection in relational databases and data warehouses is quite abundant. However, we have identified two key issues requiring enhancements. First, the actual selection of suitable candidate materialized views and indexes is rarely addressed in existing approaches. Most of them indeed present scaling problems at this level. Second, none of these approaches dynamically takes into account the interactions that may exist between materialized views, between indexes, and between indexes and materialized views (including the approaches that simultaneously select both materialized views and indexes).

In this paper, we present a novel strategy for optimizing data warehouse performance that aims at addressing both these issues. We have indeed designed a generic approach whose objective is to automatically propose solutions to data warehouse administrators for optimizing data access time. The principle of this approach is to apply data mining techniques on a workload (set of queries) that is representative of data warehouse usage in order to deduce a quasi-optimal configuration of materialized views and/or indexes. Data mining actually helps reduce the selection problem's complexity and improves scalability. Then, cost models help select among the selected materialized views and indexes the most efficient in terms of performance gain/overhead ratio. We have applied our approach on three related problems: isolated materialized view selection, isolated index selection and joint materialized view and index selection. In the last case, we included index-view interactions in our cost models.

The remainder of this paper is organized as follows. Section 2 presents and discusses the state of the art regarding materialized view and index selection. Section 3 motivates and presents the principle of our performance optimization approach. Section 4 further details how we apply this approach to isolated materialized view selection, isolated index selection and joint materialized view and index selection, respectively. We particularly focus on joint materialized view and index selection, which is our latest development. Section 5 presents the experimental results we achieved to illustrate our approach's relevance. Finally, we conclude this paper and provide research perspectives in Section 6.

2 Related work

In this section, we first formalize the materialized view and index selection problem, and then detail and discuss the state of the art regarding materialized view selection, index selection and joint index and materialized view selection, respectively.

2.1 Materialized view and index selection: formal problem definition

The materialized view and index selection problem consists in building a set of materialized views and indexes that optimizes the execution cost of a given workload. This optimization may be realized under constraints, typically the storage space available for storing these physical data structures.

Let V_C and I_C be two sets of materialized views and indexes, respectively, that are termed candidate and are susceptible to reduce the execution cost of a given query set Q (generally supposed representative of system workload). Let $O_C = V_C \cup I_C$. Let S be the storage space allotted by the data warehouse administrator to build objects (materialized views or indexes) from set O_C. The joint materialized view and index selection problem consists of building an object configuration $O \subseteq O_C$ that minimizes the execution cost of Q, under storage space constraint. This NP-hard problem (Comer, 1978; Gupta, 1999) may be formalized as follows:

- $cost(Q, O) = min(cost(Q, \vartheta)) \ \forall \ \vartheta \subseteq O_C$;
- $\sum_{o \in O} size(o) \leq S$, where $size(o)$ is the disk space occupied by object o.

2.2 Materialized view selection

The materialized view selection problem has received significant attention in the literature. Related researches differ in several points:

1. the way the set of candidate views V_C is determined;
2. the framework used to capture relationships between candidate views;
3. the use of mathematical cost models vs. calls to the system's query optimizer;
4. view selection in the relational or multidimensional context;
5. multiple or simple query optimization;
6. theoretical or technical solutions.

Classical papers in materialized view selection introduce a lattice framework that models and captures dependency (ancestor or descendant) among aggregate views in a multidimensional context (Harinarayan, Rajaraman, & Ullman, 1996; Baralis, Paraboschi, & Teniente, 1997; Kotidis & Roussopoulos, 1999; Uchiyama, Runapongsa, & Teorey, 1999). This lattice is greedily browsed with the help of cost models to select the best views to materialize. This problem has first been addressed in one data cube and then extended to multiple cubes (Shukla, Deshpande, & Naughton, 2000). Another theoretical framework, the AND-OR view graph, may also be used to capture the relationships between views (Chan, Li, & Feng, 1999; Nadeau & Teorey, 2002; Valluri, Vadapalli, & Karlapalem, 2002; Gupta & Mumick, 2005). Unfortunately, the majority of these solutions are theoretical and are not truly scalable.

A wavelet framework for adaptively representing multidimensional data cubes has also been proposed (Smith, Li, & Jhingran, 2004). 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 data cube queries. In the same spirit, Sismanis, Deligiannakis, Roussopoulos, and Kotidis (2002) proposed the Dwarf structure, which compresses data cubes. 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.

Other approaches detect common sub-expressions within workload queries in the relational context (Goldstein & Åke Larson, 2001; Baril & Bellahsene, 2003; Rizzi & Saltarelli, 2003). The view selection problem then consists of finding common subexpressions corresponding to intermediary results that are suitable to materialize. However, scanning through numerous intermediary results 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 a workload to enumerate relevant candidate views (Agrawal, Chaudhuri, & Narasayya, 2000). By exploiting the system's 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.

2.3 Index selection

The index selection problem has been studied for many years in databases (Finkelstein et al., 1988; Frank, Omiecinski, & Navathe, 1992; Agrawal et al., 2000; Valentin, Zuliani, Zilio, Lohman, & Skelley, 2000; Feldman & Reouven, 2003; Kratica, Ljubić, & Tošić, 2003; Chaudhuri, Datar, & Narasayya, 2004). In the more specific context of data warehouses, existing research studies may be clustered into two families: algorithms that optimize maintenance cost (Labio, Quass, & Adelberg, 1997) and algorithms that optimize query response time (Gupta, Harinarayan, Rajaraman, & Ullman, 1997; Agrawal, Chaudhuri, & Narasayya, 2001; Golfarelli, Rizzi, & Saltarelli, 2002). In both cases, optimization is realized under storage space constraint. In this paper, we focus on the second family of solutions, which is relevant in our context. Studies falling in this category may be further categorized depending on how the set of candidate indexes I_C and the final configuration of indexes I are built.

Selecting a set of candidate indexes may be automatic or manual. Warehouse administrators may indeed appeal to their expertise and manually provide, from a given workload, a set of candidate indexes (Frank et al., 1992; Choenni, Blanken, & Chang, 1993a, 1993b). Such a choice is however subjective. Moreover, the task may be very hard to achieve when the number of queries is very high. In opposition, candidate indexes can also be extracted automatically, through a syntactic analysis of queries (Chaudhuri & Narasayya, 1997; Valentin et al., 2000; Golfarelli et al., 2002). Such an analysis depends on the DBMS, since each DBMS is queried through a specific syntax derived from the SQL standard.

The methods for building a final index configuration from candidate indexes may be categorized into:

1. ascending or descending greedy methods;
2. methods derived from genetic algorithms;
3. methods assimilating the selection problem to the well-known knapsack optimization problem.

Ascending greedy methods start from an empty set of candidate indexes (Kyu-Young, 1987; Frank et al., 1992; Choenni et al., 1993b; Chaudhuri & Narasayya, 1997). They incrementally add in indexes minimizing cost. This process stops when cost ceases decreasing. Contrarily, descending greedy methods consider the whole set of candidate indexes as a starting point. Then, at each iteration, indexes are pruned (Kyu-Young, 1987; Choenni et al., 1993a). If workload cost before pruning is lower (respectively, greater) than workload cost after pruning, the pruned indexes are useless (respectively, useful) for reducing cost. The pruning process stops when cost increases after pruning.

Genetic algorithms are commonly used to resolve optimization problems. They have been adapted to the index selection problem (Kratica et al., 2003). The initial population is a set of input indexes (an index is assimilated to an individual). The objective function to optimize is the workload cost corresponding to an index configuration. The combinatory construction of an index configuration is realized through the crossover, mutation and selection genetic operators. Eventually, the index selection problem has also been formulated in several studies as a knapsack problem (Ip, Saxton, & Raghavan, 1983; Gundem, 1999; Valentin et al., 2000; Feldman & Reouven, 2003) where indexes are objects, index storage costs represent object weights, workload cost is the benefit function, and storage space is knapsack size.

2.4 Joint materialized view and index selection

Few research studies deal with simultaneous index and materialized view selection. Agrawal et al. (2000) have proposed three strategies. The first one, MVFIRST, selects materialized views first, and then indexes, taking the presence of selected views into account. The second alternative, INDFIRST, selects indexes first, and then materialized views. The third alternative, joint enumeration, processes indexes, materialized views and indexes over these views at the same time. According to the authors, this approach is more efficient than MVFIRST and INDFIRST, but no further details are provided.

Bellatreche, Karlapalem, and Schneider (2000) studied storage space distribution among materialized views and indexes. First, a set of materialized views and indexes is designed as an initial solution. Then, the approach iteratively reconsiders the solution to further reduce execution cost, by redistributing storage space between indexes and materialized views. Two agents are in perpetual competition: the index spy (respectively, view spy) steals some space allotted to materialized views (respectively, indexes), and *vice versa*. The recovered space is used to create new indexes (respectively, materialized views) and prune views (respectively, indexes), according to predefined replacement policies.

Another approach *a priori* determines a trade-off between storage space allotted to indexes and materialized views, depending on query definition (Rizzi & Saltarelli, 2003). According to the authors, the key factors to leverage query optimization is aggregation level, defined by the attribute list of `Group by` clauses in SQL queries, and the selectivity of attributes present in `Where` and `Having` clauses. View materialization indeed provides a great benefit for queries involving coarse granularity aggregations (few attributes in the `Group by` clause) because they produce few groups among a large number of tuples. On the other hand, indexes provide their best benefit with queries containing high selectivity attributes. Thus, queries with fine aggregations and high selectivity stimulate indexing, while queries with coarse aggregations and weak selectivity encourage view materialization.

Finally, Bruno and Chaudhuri (2006) have recently worked on refining the physical design of relational databases. Their objective was to automatically improve an expert's physical design, to take into account primordial constraints it might violate. Hence, they proposed a transformation architecture base on two fusion and reduction primitives that helps process indexes and materialized views in a unified way.

2.5 Discussion

Existing studies related to index and materialized view selection are numerous and diverse in the field of databases, and quite developed in the field of data warehouses as well. However, we have identified two main points that could be improved in these approaches.

2.5.1 Candidate object selection

Selecting candidate objects (materialized views and indexes) is rarely the focus of existing approaches, most of which do not scale up well at this level. Many index selection strategies indeed rest on human expertise (the warehouse administrator's) to propose an initial candidate index configuration. Given the size and complexity of most data warehouses, an automatic approach is mandatory to apply these methods on a real-life scale. The most recent studies actually take this option, by building the initial index configuration from system workload.

With respect to materialized views, various data structures have been proposed (lattices, graphs, wavelets...) to model inter-view relationships. None of them scale up very well. For instance, browsing a candidate view lattice is very costly when the input data cube is very large. Similarly, building view graphs is as complex as the input workload is large. Hence, it is necessary to carefully evaluate a strategy's complexity before adopting it, and to optimize any data structure used.

2.5.2 Inter-object interaction management

To the best of our knowledge, none of the approaches we have presented in this section dynamically takes into account the interactions that may exist between indexes, between materialized views, and between index and views, including joint selection methods. Existing studies, especially those assimilating the selection problem to the knapsack problem or exploiting genetic algorithms, indeed compute the cost or benefit of an object (index or materialized view) once only, before injecting it in their algorithm. Since no details are provided by Agrawal et al. (2000) about joint enumeration index and view selection, we cannot know exactly how it takes interactions between materialized views and indexes into account. We suspect costs/benefits are also computed once before the selection phase. However, the relevance of selecting a given object may vary from one iteration to the other if another, previously selected object interacts with the first one. It is thus primordial to recompute costs or benefits dynamically before object selection.

The nearest solution is the one by Bellatreche et al. (2000). However, its object replacement policies in the disk spaces allotted to indexes and materialized views do not truly reflect index-view interactions. They indeed only consider joint usage frequency in queries, and not the benefit an object brings with respect to other objects.

3 Data mining-based warehouse performance optimization approach

In this section, we first motivate our performance optimization approach. Then, we present its general principle, detail how candidate objects are selected and how a final object (materialized view and index) configuration is generated.

3.1 Motivation

In this paper, our objective is to address the issues identified in Section 2.5. First, to ensure that candidate object (materialized view or index) selection scales up, it is necessary to devise an automatic approach. Generally, this is achieved by syntactically analyzing the system's query workload, which helps identify query attributes that might support indexes or materialized views. These attributes are then systematically combined to propose multi-attribute indexes or exhaustive view graphs. However, this strategy later leads, in the selection phase, to consider irrelevant objets, i.e., objects that do belong to the workload, but are not interesting in the scope of indexing or view materialization.

To *a priori* eliminate these irrelevant objects, we propose to exploit data mining techniques to directly extract from the workload a configuration of pertinent candidate objects. Our idea is to discover co-occurencies and similarities between workload objects. For indexing, we base our approach on the intuition that the importance of an attribute to index is strongly correlated with its appearance frequency in the workload. For view materialization, devising similar classes of queries also helps build views that are likely to answer all the queries from a given class.

From the smallest possible set of relevant candidate objects, we must then exploit an optimization algorithm (typically a greedy, knapsack or genetic algorithm) to build a quasi-optimal object configuration. However, to take index-view interactions into account, these algorithms must be modified. In a given iteration, an object's cost indeed depends on previously selected objects. Thus, it must be recomputed at each step. For simplicity reasons, we implemented this approach with a greedy algorithm.

3.2 General principle of our approach

Our automatic warehouse performance optimization approach (Figure 1) is not only based on information extracted from the warehouse's data (statistics such as attribute selectivity, for instance) or workload, but also on knowledge. This knowledge includes classical warehouse metadata (we notably exploit the database schema), as well as administration expertise, formalized in cost models (benefit induced by an index and maintenance cost, for instance) or rules. Our approach proceeds in two main steps, which are both piloted by knowledge.

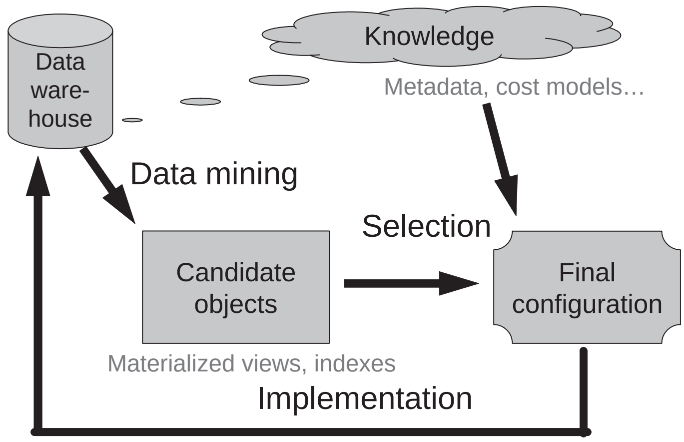

Figure 1: Principle of our automatic performance optimization approach

The first step is building a candidate object configuration O_C. It consists in syntactically analyzing the input workload, which helps identify attributes that might be useful for view materialization or indexing. Applying rules issued from administration-related knowledge can already reduce the size of this attribute set. For instance, a low selectivity attribute such as gender, which has only two values, is not a good candidate index. This set of attributes is then structured as an attribute-value table that can be processed by a data mining algorithm. The output of such an algorithm is directly the candidate object configuration.

Since disk space is constrained, it would be impossible to exploit all the candidate objects from O_C. Thus, the second step in our process is greedily selecting a final object configuration O from O_C. This algorithm exploits cost models we have developed to express, e.g., the benefit brought by a materialized view or an index, as well as their storage and maintenance costs (Section 4). Eventually, the last step in our approach consists of implementing the final object configuration in the data warehouse.

Note that we have designed this approach in a modular fashion, so that it is as generic as possible. Completing the two main steps indeed brought us to perform choices, but other options would be easy to consider. For instance, the data mining technique we selected for building a candidate index configuration is frequent itemset mining, but another study explored clustering instead (Zaman, Surabattula, & Gruenwald, 2004). Besides, we have also used clustering for materialized view selection. Similarly, other optimization algorithms could be substituted to the greedy strategy we adopted to build the final object configuration. Our cost models might also be easily replaced with others if necessary, or by calls to a query optimizer, if it is accessible on the host DBMS.

3.3 Candidate object selection

System workload is typically accessible from the host DBMS' transaction log. A given workload is supposed representative if it has been measured during a time period the warehouse administrator judges sufficient to anticipate upcoming transactions.

Since we are more particularly interested in decision-support query performance and not warehouse maintenance, we only consider interrogation query workloads in this paper. These queries are typically composed of join operations between the fact table and dimensions, restriction predicates and aggregation and grouping operations. More formally, an analytic query q may be expressed as follows in relational algebra: $q = \Pi_{G,M}(\sigma_R(F \bowtie D_1 \bowtie D_2 \bowtie ... \bowtie D_d))$, where G is the set of attributes from dimensions D_i that are present in q's grouping clause, M is a set of aggregate measures from fact table F and R a conjunction of predicates over dimension attributes.

Attributes that may support materialized views or indexes belong to sets G and R (Chaudhuri & Narasayya, 1997; Golfarelli et al., 2002; Valentin et al., 2000; Feldman & Reouven, 2003). We reference them in a so-called "query-attribute" binary matrix whose rows represent workload queries q_i and whose columns are representative attributes a_j. The general term m_{ij} of this matrix is equal to one if attribute a_j is present in query q_i, and to zero otherwise. A simple example of query-attribute matrix based on the workload excerpt from Figure 2 is featured in Table 1. Note that attributes a_2 and a_6 are not present in this matrix. They are indeed used in aggregation operations, and are thus not good candidates for indexing. Hence, they are not considered representative.

q_1	SELECT $F.a_1$, SUM($F.a_2$) FROM F, D_1 WHERE $F.a_1$ = $D_1.a_3$ AND $D_1.a_4$ < 2000 GROUP BY $F.a_1$
q_2	SELECT $F.a_1$, $F.a_5$, AVG($F.a_6$) FROM F, D_1, D_2 WHERE $F.a_1$ = $D_1.a_3$ AND $F.a_5$ = $D_2.a_7$ AND $D_2.a_8$ = 'ABC' GROUP BY $F.a_1$, $F.a_5$
q_3	SELECT $F.a_1$, $F.a_9$, SUM($F.a_2$) FROM F, D_1, D_3 WHERE $F.a_1$ = $D_1.a_3$ AND $F.a_9$ = $D_3.a_{10}$ GROUP BY $F.a_1$, $F.a_9$

Figure 2: Workload excerpt

	a_1	a_3	a_4	a_5	a_7	a_8	a_9	a_{10}
q_1	1	1	1	0	0	0	0	0
q_2	1	1	0	1	1	1	0	0
q_3	1	1	0	0	0	0	1	1

Table 1: Sample query-attribute matrix

This data structure directly corresponds to attribute-value statistical tables that are exploited by data mining algorithms. Here, attributes are the workload queries, and values are all the warehouse's table attributes that are used in queries. Applying a data mining technique onto the query-attribute matrix helps obtain a set of candidate objects (materialized views and indexes) O_c.

3.4 Final object configuration construction

Our final materialized view and index configuration construction algorithm (Figure 3) is based on an ascending greedy search within the input candidate object set O_C. It starts from an empty final object configuration O, and then adds in it object o from O_C that maximizes objective function f_O, at each iteration. For each object $o \in O_C$, the value of $f_O(o)$ depends on objects already selected in O. Thus, it must be recomputed at each iteration, which helps take view-index interactions into account. The algorithm ends when objective function f_O cannot be improved any more, when there are no more candidate objects in O_C, or when storage space S allocated by the warehouse administrator to materialized views and index is full.

```
O = ∅
s_O = 0
Repeat
    o_max = ∅
    f_max = 0
    For each o ∈ O_C do
        If f_O(o) > f_max then
            o_max = o
            f_max = f_O(o)
        End if
    End for
    If f_O(o_max) > 0 then
        O = O ∪ {o_max}
        s_O = s_O + size(o_max)
        O_C = O_C \ {o_max}
    End if
Until f_O(o_max) ≤ 0 or O_C = ∅ or s_O ≥ S
```

Figure 3: Final object configuration construction algorithm

For a given workload Q and an object configuration O, the objective function f_O may generally be expressed as follows: $f_O(o) = \alpha_o \; benefit_O(o) - \beta_o \; maintenance(o)$. Generally, $benefit_O(o) = \frac{cost(Q,O)-cost(Q,O\cup\{o\})}{size(o)}$. However, taking view-index interactions into account complicates this function's computation (Section 4.3.3).

Cost models help compute the *cost* and *maintenance* functions. Since they depend on the considered data structure (materialized view or index), they are detailed in Section 4's corresponding subsections. Coefficient α_o helps ponder benefit. It is generally equal to one, but may also help favor index that avoid join operations (Section 4.2). Finally, coefficient $\beta_o = |Q|p(o)$ is an estimator for the number of updates of object o. The update probability of object o, $p(o)$, is equal to $\frac{1}{|O|}\frac{\%refreshment}{\%interrogation}$, where $\frac{\%refreshment}{\%interrogation}$ represents the proportion of warehouse updates with respect to interrogations.

4 Applications

This section presents three instances of our automatic data warehouse performance optimization approach: automatic materialized view selection, automatic index selection, and automatic, joint materialized view and index selection. We particularly detail this last, newest application. Moreover, we particularly insist, for each application, on its specificities in terms of candidate object selection (e.g., the data mining technique we exploited) and cost models used in building the final object configuration.

4.1 Clustering-based materialized view selection

In this application, we propose to select materialized views by clustering queries from workload Q. Several syntactically similar queries have indeed a high probability of being resolved by one single materialized view. Then, we must build classes of similar queries from Q. Since the number of classes is *a priori* unknown, we have selected an unsupervised clustering method.

Our approach's principle is similar to SQL workload compression (Chaudhuri, Gupta, & Narasayya, 2002), a technique proposed in the relational database context to optimize, for instance, index selection or approximate answer to aggregation queries. We adapted this idea to the context of relational (with an SQL decision-support workload) and XML (with an XQuery decision-support workload) data warehouses (Aouiche, Jouve, & Darmont, 2006; Mahboubi, Aouiche, & Darmont, 2006).

The main improvement brought by our approach lies at the candidate view selection level. Most anterior methods indeed build a lattice or graph of all syntactically correct views for a given workload. However, in practice, such data structures are complex to build and browse. Using a clustering algorithm helps drastically reduce the number of candidate materialized views by proposing only a couple of views per class (only one in the best case — Section 4.1.1) instead of one view per workload query. This dimensionality reduction helps improve the whole process' efficiency and offers true scaling up capability.

4.1.1 Candidate materialized view selection

Query similarity and dissimilarity. To perform clustering and check out whether query classes are homogeneous, we must define query similarity and dissimilarity measures. Let M be a query-attribute matrix of general term m_{ij}, defined on query set $Q = \{q_i, i = 1..n\}$ and attribute set $A = \{a_j, j = 1..l\}$. We define the elementary similarity and dissimilarity between two queries q_i and $q_{i'}$, regarding attribute a_j, as follows.

$$\partial_{sim/a_j}(q_i, q_{i'}) = \begin{cases} 1 \text{ if } m_{ij} = m_{i'j} = 1 \\ 0 \text{ otherwise} \end{cases} \qquad \partial_{dissim/a_j}(q_i, q_{i'}) = \begin{cases} 1 \text{ if } m_{ij} \neq m_{i'j} \\ 0 \text{ otherwise} \end{cases}$$

Note that these definitions are not symmetric. The absence of a given attribute in two queries does indeed not constitute an element of similarity, unlike its presence. We now extend these definitions onto attribute set A to obtain global similarity and dissimilarity between queries q_i and $q_{i'}$.

$$sim(q_i, q_{i'}) = \sum_{j=1}^{l} \partial_{sim/a_j}(q_i, q_{i'}) \qquad dissim(q_i, q_{i'}) = \sum_{j=1}^{l} \partial_{dissim/a_j}(q_i, q_{i'})$$

Query clustering. The objective of clustering is to build a natural partition of queries that reflects their internal structure. Objects in the same class must be strongly similar, while objects from different classes must be strongly dissimilar. Let $P = \{C_k, k = 1..p\}$ be a partition of p classes (query sets). We define interclass similarity between two distinct classes C_a and C_b from P, as well as intraclass dissimilarity within class C_a from P, as follows.

$$Sim(C_a, C_b) = \sum_{q_i \in C_a, q_{i'} \in C_b,} sim(q_i, q_{i'}) \qquad Dissim(C_a) = \sum_{q_i \in C_a, q_{i'} \in C_a, i<i'} dissim(q_i, q_{i'})$$

Eventually, we define on P a measure of clustering quality $Q(P)$ that helps capture the partition's natural aspect. $Q(P)$ indeed possesses low values for partitions that have a strong intraclass homogeneity and a strong interclass disparity. $Q(P)$ must be minimized.

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

Any algorithm would be suitable to actually perform clustering. However, we selected the Kerouac algorithm (Jouve & Nicoloyannis, 2003a) that bears interesting features in our context. Kerouac indeed natively features a clustering quality measure we can easily replace by $Q(P)$. Furthermore, Kerouac can also integrate constraints in the clustering process, which is useful to exploit knowledge (warehouse administration-related knowledge, in our case) in the clustering process. This feature also allows to satisfy a precondition in the materialized view fusion process (see next paragraph): queries from one given class must share the same joining conditions. Kerouac also determines the number of clusters automatically. Finally, Kerouac's computational complexity is relatively low (log linear regarding the number of queries and linear regarding the number of attributes) ; it can deal with a high number of objects (queries) ; and it can also deal with distributed data (Jouve & Nicoloyannis, 2003b).

Kerouac exploits the notion of partition neighborhood. Partition P_g belongs to partition P_h's neighborhood if P_g may be obtained from P_h by segmenting a class from P_h, P_g may be obtained from P_h by merging two classes of P_h, or $P_g = P_h$. Actual clustering is a greedy heuristic similar to induction graph building. Partition classes are iteratively segmented and/or merged. At each step i, a new partition P_{i+1} is selected from P_i's neighborhood such as it minimizes quality measure $Q(P_{i+1})$. This process is initialized with a raw query partition P_0 and ends up when $P_{i+1} = P_i$. It outputs a partition $P_{\sim nat}$ that is the closest to natural partition P_{nat}, according to Kerouac's authors.

Candidate view fusion. The output of clustering is a set of similar query classes. Our objective is to associate to each class the smallest possible number of materialized views that cover all the class' queries. To achieve this goal, we consider each query as a potential view and run a fusion process to decrease their number. The algorithm we use follows the principle introduced by by Agrawal et al. (2000). However, in our context, it is much more efficient since it is applied onto a limited number of views in each class instead of the whole set of candidate views derived from the workload. The output of fusion applied on classes obtained in the previous step is the set of candidate materialized views.

The fusion v_{ij} of two candidate materialized views v_i and v_j must satisfy two conditions: all queries resolved by v_i and v_j must be resolved by v_{ij}, and the scan cost of (v_i, v_j) must not be significantly greater than that of v_{ij}. The actual fusion algorithm proceeds in four steps:

1. aggregation operations from v_i and v_j are reproduced in v_{ij};
2. all projection and `group by` attributes from v_i and v_j become both projection and `group by` attributes in v_{ij};
3. all attributes in selection predicates that are present in v_i but not in v_j, and *vice versa*, are added into v_{ij}'s `group by` clause;
4. selection predicates that are common to v_i and v_j are reproduced in v_{ij}'s selection predicates.

Finally, fusion is actually performed if $cost(v_{ij}) \geq (cost(v_i)+cost(v_j)) \times x$, with x's value being empirically set up by the administrator (small values of x favor materialized view fusion).

4.1.2 Cost models

In most of the (relational) data warehouse cost models from the literature, the cost of a query q is supposed proportional to the size (in tuples) of the materialized view exploited by q (Golfarelli & Rizzi, 1998). The same assumption is made for view maintenance cost. Hence, we reuse a model that estimates the size of a given materialized view. It has been proposed by Golfarelli and Rizzi (1998) and exploits Yao (1977)'s formula to estimate the number of tuples $|V|$ of a view V composed of k attributes $a_1, a_2, ..., a_k$ and based on fact table F and d dimensions $D_1, D_2, ..., D_d$: $|V| = max\_size(V) \times \left(1 - \prod_{i=1}^{|F|} \frac{max\_size(F) \times (1 - \frac{1}{max\_size(V)}) - i + 1}{max\_size(F) - i + 1}\right)$, where $max\_size(V) = \prod_{i=1}^{k} |a_i|$ and $max\_size(F) = \prod_{i=1}^{d} |D_i|$.

When ratio $\frac{max\_size(F)}{max\_size(V)}$ is high enough, Cardenas (1975)'s formula helps obtain a good approximation: $|V| = max\_size(V) \times \left(1 - \left(1 - \frac{1}{max\_size(V)}\right)^{|F|}\right)$.

V's size in bytes is then $size(V) = |V| \times \sum_{i=1}^{n} size(d_i)$, where $size(d_i)$ is the size in bytes of dimension d_i from V (which can be directly obtained from the warehouse metadata) and n the number of dimensions in V. Yao and Cardenas' formulas assume data are uniformly distributed and tend to overestimate view size. However, they are easy to implement and fast to compute. Other, more precise methods exploit data sampling and statistical laws (Shukla, Deshpande, Naughton, & Ramasamy, 1996; Chaudhuri & Motwani, 1999; Nadeau & Teorey, 2001), but they are much harder to implement.

Eventually, this cost model is very easy to adapt to the XML context by establishing equivalences between relations and XML documents on one hand, and tuples and XML elements on the other hand. The only true difference lies in $size(d_i)$'s computation, but it is also obtained from warehouse metadata in the XML context.

4.2 Frequent itemset mining-based index selection

In this application, we work on optimizing the execution of join operations in a decision-support query workload. We propose an index selection method based on the extraction from the workload of frequent attributes that may support indexes.

We have first worked on classical, B-tree-like indexes (Aouiche, Darmont, & Gruenwald, 2003). We focus in this paper on bitmap index selection (Aouiche, Darmont, Boussaïd, & Bentayeb, 2005). These data structures (O'Neil & Graefe, 1995) are particularly adapted to the data warehouse context. They indeed render logical and counting operations efficient (they operate directly on bitmaps stored in the main memory), and help precompute join operations at index creation time. Moreover, bitmap storage space is small, especially when the indexed attributes' cardinality is low, which is usually the case in a warehouse's dimensions.

Our approach's originality mainly lies in the use of frequent itemset mining for selecting the most pertinent candidate indexes. However, it also has another advantage. The few approaches that help select multi-attribute indexes exploit an iterative process to build them: mono-attribute indexes in the first iteration, 2-attribute indexes in the second, and so on (Chaudhuri & Narasayya, 1997). In our approach, frequent itemsets, which are attribute sets of variable size, help directly propose multi-attribute candidate indexes. Furthermore, these candidate indexes are *a priori* pertinent, while combinations generated from smaller candidate indexes are not necessarily all pertinent. Thus, our approach avoids pruning them by providing a smaller set of pertinent candidates.

Eventually, most existing index selection techniques (Section 2.3) only exploit B-tree indexes. Though this type of index is widely used in DBMSs, it is not the best adapted to index voluminous data and low cardinality attributes. In the data warehouse context, bitmap join indexes have indeed been demonstrated to be more efficient (Sarawagi, 1997; Wu, 1999).

4.2.1 Candidate index selection

When building the extraction context (query-attribute matrix) that is exploited by a data mining algorithm to select candidate indexes, we use knowledge relative to database administration and performance optimization, much like Feldman and Reouven (2003). Such an attribute preselection helps reduce the mining algorithm search space and, mechanically, improves its response time.

Knowledge is formalized under the form of "if-then" rules, e.g., "if a predicate is like $attribute \neq value$, then *attribute* must not be selected". Such a predicate would indeed not exploit an index defined on *attribute*, all its values being scanned but *value*.

We base the final selection of candidate indexes on the intuition that the importance of an attribute to index is strongly correlated to its appearance frequency in the workload. Frequent itemset mining (Agrawal & Srikant, 1994) appears as a natural solution to extract these attributes. Many frequent itemset mining algorithms are available in the literature. We selected Close (Pasquier, Bastide, Taouil, & Lakhal, 1999), which presents several advantages in our context.

First, Close helps process voluminous workloads. It indeed exploits Galois closure operators, which reduce the number of accesses to the extraction context when searching for frequent itemsets. Close is also efficient when the extraction context is dense, which is our case, since query sets often form logical suites. Eventually, frequent closed itemsets[1] extracted by this algorithm are fewer than all frequent itemsets (which can nonetheless be generated from the frequent closed itemsets). This helps reduce computing time and avoid multiplying useless candidate indexes.

4.2.2 Cost models

Data access cost through a bitmap join index. Data access is performed in two steps: scanning the index' bitmaps, and then reading the tuples. If access to the bitmaps is direct and data are uniformly distributed, which is a reasonable assumption according to Choenni et al. (1993a), index traversal cost is $d\frac{|A||F|}{8S_p}$. d is the number of predicates applied on indexed attribute A. F is the fact table. S_p is the size of a disk page. $\frac{|A||F|}{8}$ represents the size of the bitmap index (Wu & Buchmann, 1998).

The number of tuples read by a query using d bitmaps is $d\frac{|F|}{|A|}$ if data are uniformly distributed. The number of corresponding input/output is then equal to $p_F(1-e^{-\frac{d|F|}{p_F|A|}})$ (O'Neil & Quass, 1997), where p_F is the number of disk pages that are necessary to store F. Finally, $cost = d\frac{|A||F|}{8S_p} + p_F(1-e^{-\frac{d|F|}{p_F|A|}})$.

If bitmap access is performed through a B-tree, as is the case in the Oracle DBMS, for

[1] An itemset I is closed with respect to the Galois connection (f, g) iff $g \circ f(I) = I$ (Pasquier et al., 1999).

instance, B-tree descent cost must be taken into account: $log_m|A| - 1$, where m is the B-tree order. Leaf nodes traversal cost is then $\frac{|A|}{m-1}$ at worst. However, bitmap index traversal cost is reduced to $d\frac{|F|}{8S_p}$. Then, $cost = log_m|A| - 1 + d\frac{|F|}{8S_p} + p_F(1 - e^{-\frac{d|F|}{p_F|A|}})$.

Bitmap join index maintenance cost. Let a bitmap join index be defined on attribute A from dimension D. When inserting a tuple into fact table F, D must be traversed to find the tuple that must be joined to the one inserted in F: p_D pages are read. Then, the index' bitmaps must be updated. At worst, they are all traversed and $\frac{|A||F|}{8S_p}$ pages are read. Hence, $maintenance_F = p_D + \frac{|A||F|}{8S_p}$.

When inserting a tuple into dimension D, update may be without domain expansion, then a bit corresponding to the inserted tuple must be added to each bitmap; or with domain expansion, then a new bitmap must be created. Then, $maintenance_D = p_F + (1 + \xi)\frac{|A||F|}{8S_p}$, where $\xi = 1$ when expanding the domain and $\xi = 0$ otherwise.

4.3 Joint materialized view and index selection

In this eventual application, we seek to select a configuration of materialized views and indexes that are mutually beneficial, in order to further optimize the response time of decision-support queries. More precisely, we aim at truly taking view-index interactions into account and at optimizing storage space sharing between materialized views and indexes. Existing approaches indeed consider indexes and materialized views as distinct objects, whose benefit and maintenance cost are invariant and independent from already-selected objects. Moreover, few consider indexing materialized views.

Bellatreche et al. (2000)'s approach, which is closest to ours, starts from an initial solution composed of indexes and materialized views separately selected under storage space constraint. Taking this constraint into account *a priori* might eliminate solutions that are susceptible to become pertinent in the next iterations of the selection process. Hence, we only introduce the storage space constraint *a posteriori*, within the selection algorithm. Furthermore, object replacement policies in storage spaces respectively allotted to indexes and materialized views exploit these objects' usage frequency, and not the benefit brought by their simultaneous usage.

4.3.1 Candidate object selection

First, let us detail and specialize the automatic performance optimization strategy presented in Section 3.2 for joint materialized view and index selection. Here, we exploit the modular structure of our approach: our input is a set of candidate objects (materialized views and indexes) obtained with any existing selection algorithm, such as the ones we propose. Then, we exploit specific data structures and cost models to recommend a pertinent configuration of materialized views and indexes through the following steps (Figure 4):

1. extract a representative query set Q from system workload;
2. select a set of candidate materialized views V_C using the approach described in Section 4.1.1, with Q as input;
3. select a set of candidate indexes I_C using the approach described in Section 4.2.1, with $Q \cup V_C$ as input;
4. simultaneously select materialized views and indexes from $O_C = V_C \cup I_C$ with the algorithm from Figure 3;
5. build the final configuration of materialized views and indexes $O \subseteq O_C$ under storage space constraint S.

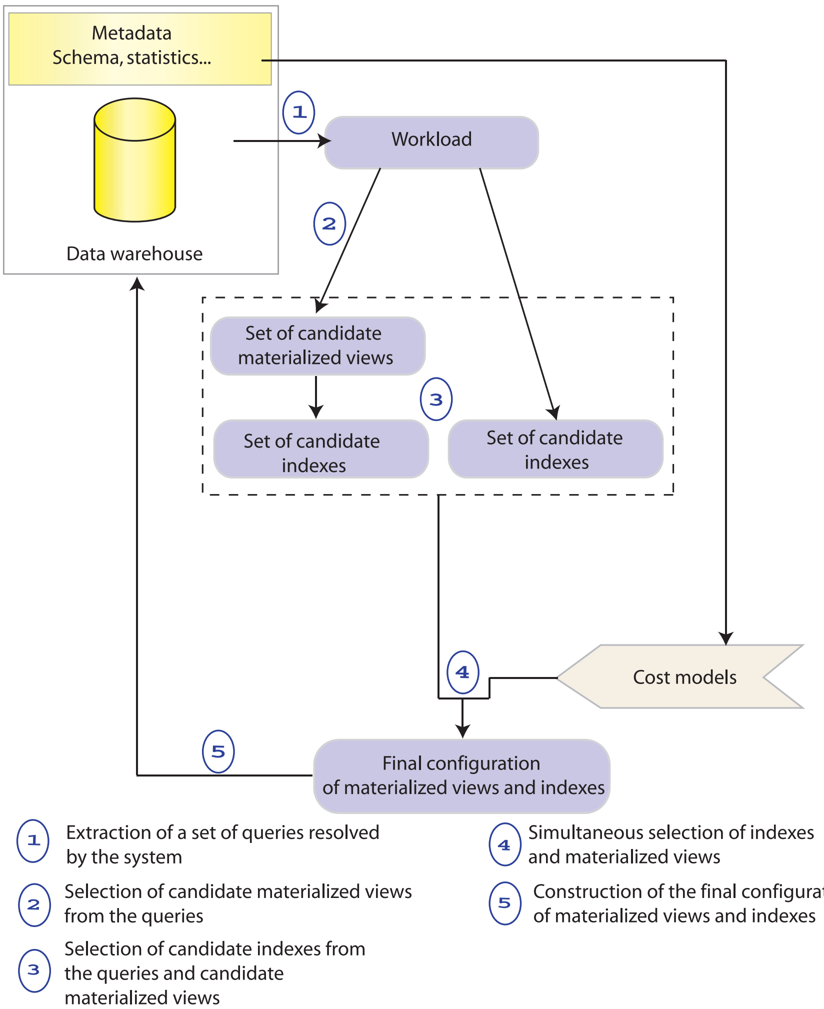

Figure 4: Joint materialized view and index selection approach

4.3.2 Specific data structures

After building the set of candidate materialized views, indexes and indexes on views O_C, we aim at combining them to recommend a pertinent configuration of materialized views and indexes O. To consider the relationships between these objects in this process, we need to materialize them. For this purpose, we use three binary matrices: the "query-view" matrix, the "query-index" matrix and the "view-index" matrix that we detail in the following paragraphs.

To better illustrate how these data structures are designed, let us consider the workload sample from Figure 5. Candidate materialized views and indexes obtained from this workload by applying our strategy are featured in Figures 6 and 7, respectively.

q_1
```
select sales.time_id, sum(amount_sold)
from sales, times
where sales.time_id = times.time_id
and times.time_fiscal_year = 2000
group by sales.time_id
```

q_2
```
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 = 'news paper'
group by sales.prod_id
```

q_3
```
select customers.cust_gender, sum(amount_sold)
from sales, customers, products,
where sales.cust_id = customers.cust_id
and sales.prod_id = products.prod_id
and customers.cust_marital_status ='single'
and products.prod_category = 'women'
group by customers.cust_gender
```

q_4
```
select products.prod_name, sum(amount_sold)
from sales, products, promotions
where sales.prod_id = products.prod_id
and sales.promo_id = promotions.prom_id
and promotions.promo_category = 'TV'
group by products.prod_name
```

q_5
```
select promotions.promo_name,
sum(amount_sold)
from sales, promotions
where sales.promo_id = promotions.promo_id
and promotions.promo_begin_date='30/01/2000'
and promotions.promo_end_date='30/03/2000'
group by promotions.promo_name
```

q_6
```
select customers.cust_marital_status,
sum(quantity_sold)
from sales, customers, products
where sales.cust_id = customers.cust_id
and sales.prod_id = products.prod_id
and customers.cust_gender = 'woman'
and products.prod_name = 'shampooing'
group by customers.cust_first_name
```

q_7
```
select products.prod_name, sum(amount_sold)
from sales, products, promotions
where sales.prod_id = products. prod_id
and sales.promo_id =promotions.promo_id
and products.prod_category='tee shirt'
and promotions.promo_end_date='30/04/2000'
group by products.prod_name
```

q_8
```
select channels.channel_desc, sum(quantity_sold)
from sales, channels
where sales.channel_id = channels.channel_id
and channels.channel_class = 'Internet'
group by channels.channel_desc
```

Figure 5: Sample workload

Query-view matrix. The query-view matrix (QV) captures existing relationships between workload queries and the materialized views extracted from these queries, i.e., views that are exploited by at least one workload query. This matrix may be viewed as the result of rewriting queries with respect to candidate materialized views. The query-view matrix' rows and columns are workload queries and candidate materialized views, respectively. Its general term QV_{qv} is equal to one if a given query q exploits the corresponding view v, and to zero otherwise. Table 2 presents the query-view matrix corresponding to the example from Figures 5 and 6.

	v_1	v_2	v_3	v_4	v_5	v_6	v_7
q_1	1	0	0	0	0	0	0
q_2	0	0	0	1	0	0	0
q_3	0	0	1	0	0	0	0
q_4	0	0	0	1	0	0	0
q_5	0	0	0	0	0	0	1
q_6	0	0	1	0	0	0	0
q_7	0	0	0	0	0	0	1
q_8	0	1	0	0	0	1	0

Table 2: Sample query-view matrix

```
v1  create materialized view v1 as
    select sales.time_id, times.time_fiscal_year,
    sum(amount_sold)
    from sales, times
    where sales.time_id = times.time_id
    group by sales.time_id, times.times_fiscal_year

v2  create materialized view v2 as
    select sales.prod_id, sales.cust_id, channels.channel_desc,
    channels.channel_class, sum(quantity_sold)
    from sales, channels, products, customers
    where sales.prod_id = products.prod_id
    and sales.channel_id = channels.channel_id
    and sales.cust_id = customers.cust_id
    group by sales.prod_id, sales.cust_id, channels.channel_desc,
    channels.channel_class

v3  create materialized view v3 as
    select customers.cust_first_name, products.prod_name,
    products.prod_category, customers.cust_gender,
    customers.cust_marital_status, sum(sales.quantity_sold)
    from sales, customers, products
    where sales.cust_id = customers.cust_id
    and sales.prod_id = products.prod_id
    group by customers.cust_first_name, products.prod_name,
    products.prod_category, customers.cust_gender,
    customers.cust_marital_status

v4  create materialized view v4 as
    select products.prod_name, products.prod_category,
    promotions.promo_category, sum(amount_sold)
    from sales, products, promotions
    where sales.prod_id = products.prod_id
    and sales.promo_id = promotions.promo_id
    group by products.prod_name, products.prod_category,
    promotions.promo_category

v5  create materialized view v5 as
    select sales.prod_id, products.prod_category,
    promotions.promo_category, sum(amount_sold)
    from sales, products, promotions
    where sales.prod_id = = products.prod_id
    and sales.promo_id = promotions.promo_id

v6  create materialized view v6 as
    select channels.channel_class, products.prod_name, channels.channel_desc,
    products.prod_category, sum(sales.quantity_sold), sum(sales.amount_sold)
    from sales, channels, products
    where sales.prod_id = products.prod_id
    and sales.channel_id = channels.channel_id
    group by channels.channel_class, products.prod_name,
    products.prod_category, channels.channel_desc

v7  create materialized view v7 as
    select sales.prod_id, products.prod_category,
    channels.channel_desc, promotions.promo_name,
    promotions.promo_begin_date, promotions.promo_end_date,
    products.prod_name, sum(sales.quantity_sold), sum(sales.amount_sold)
    from sales, products, promotions
    where sales.prod_id = products.prod_id
    and sales.promo_id = promotions.promo_id
    and sales.channel_id = channels.channel_id
    group by sales.prod_id, products.prod_category, channels.channel_desc,
    promotions.promo_name, promotions.promo_begin_date,
    promotions.promo_end_date, products.prod_name
```

Figure 6: Candidate materialized views

Query-index matrix. The query-index matrix (QI) stores the indexes built on base tables. This matrix may be viewed as the result of rewriting queries with respect to candidate indexes. The query-index matrix' rows and columns are workload queries and candidate indexes, respectively. Its general term QI_{qi} is equal to one if a given query q exploits the corresponding index i, and to zero otherwise. Table 3 presents the query-index matrix corresponding to the example from Figures 5 and 7.

View-index matrix. The view-index matrix (VI) identifies candidate indexes that are recommended for candidate materialized views returned by our view selection algorithm. The query-index matrix' rows and columns are candidate views and candidate indexes on these views, respectively. Its general term VI_{vi} is equal to one if a given materialized view v is susceptible to exploit index i (i.e., if the attributes indexed by i are also present in v), and to zero otherwise. Table 4 presents the view-index matrix corresponding to the example from

Indexes	Indexed attributes
i_1	promotions.promo_category
i_2	channels.channel_desc
i_3	channels.channel_class
i_4	customers.cust_marital_status
i_5	customers.cust_gender
i_6	times.time_begin_date
i_7	times.time_end_date
i_8	times.fiscal_year
i_9	products.prod_name
i_{10}	products.prod_category
i_{11}	promotions.promo_name
i_{12}	customers.cust_first_name

Figure 7: Candidate indexes

	i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8	i_9	i_{10}	i_{11}	i_{12}
q_1	0	0	0	0	0	0	0	1	0	0	0	0
q_2	1	0	0	0	0	0	0	0	0	0	0	0
q_3	0	0	0	1	1	0	0	0	0	1	0	0
q_4	1	0	0	0	0	0	0	0	1	0	0	0
q_5	0	0	0	0	1	1	0	0	0	0	1	0
q_6	0	0	0	0	1	0	0	0	1	0	0	1
q_7	0	0	0	0	0	0	1	0	1	1	0	0
q_8	0	1	1	0	0	0	0	0	0	0	0	0

Table 3: Sample query-index matrix

Figures 6 and 7.

	i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8	i_9	i_{10}	i_{11}	i_{12}
v_1	0	0	0	0	0	0	0	1	0	0	0	0
v_2	0	1	0	0	0	0	0	0	0	0	0	0
v_3	0	0	0	1	1	0	0	0	1	1	0	1
v_4	1	0	0	0	0	0	0	0	1	1	0	0
v_5	1	0	0	0	0	0	0	0	0	1	0	0
v_6	0	1	1	0	0	0	0	0	1	1	0	0
v_7	0	1	0	0	0	1	1	0	1	1	1	0

Table 4: Sample view-index matrix

4.3.3 Cost models

We have already presented in Sections 4.1.2 and 4.2.2 cost models relative to materialized views and bitmap join indexes, respectively. Since indexes defined on materialized views are generally B-trees or derivatives, we first recall here the cost models that relate to these indexes. Then, we discuss the benefit of view materialization vs. indexing.

Data access cost through a B-tree index. Data access through an index is subdivided into to steps: index traversal to find key values corresponding to the query ($C_{traversal}$ cost), and then searching for these identifiers in the database (C_{search} cost). Let q be a query, ζ a set of indexes, SNA_q the set of attributes that are present in query q's restriction clause (the `Where` clause in SQL), BF_a the bloc factor of the index built on attribute a (the average number of $(key, identifier)$ couples per disk page), SF_a the selectivity factor of attribute a, and finally v the accessed materialized view. Then: $C_{traversal} = \sum_{a \in (\zeta \cap SNA_q)} \lceil log_{BF_a}|v| \rceil + \left\lceil \frac{SF_a|v|}{BF_a} \right\rceil - 1$.

The number of identifiers to search for is then $N = |v| \prod_{a \in (\zeta \cap SNA_q)} SF_a$. According, to Cardenas (1975)'s formula, the number of disk pages to access is: $C_{search} = S_p \left(1 - (1 - \frac{1}{S_p})^N\right)$, where S_p represents disk page size.

Finally, data access cost through a B-tree index is $cost = C_{traversal} + C_{search}$.

B-tree index maintenance cost. Classically, this cost is expressed as follows: $maintenance = \sum_{op \in \{ins, del, upd\}} \sum f_{op} \sum_{a \in A_{op}} C_{op}(a)$; where f_{ins}, f_{del} and f_{upd} are insert, delete and update frequencies, respectively; and $C_{ins}(a)$, $C_{del}(a)$ and $C_{upd}(a)$ are maintenance costs related to an insert, delete or update operation on attribute a, respectively. A_{op} is the set of considered attributes. $A_{ins} = A_{del} = \zeta$, where ζ is the set of indexes to maintain. $A_{upd} = \zeta \cap SNA_{upd}$, where SNA_{upd} is the set of attributes to update. Finally, maintenance costs are the following (Whang, 1985): $C_{ins}(a) = C_{del}(a) = \lceil log_{BF_a}|v| \rceil$ and $C_{upd}(a) = \lceil log_{BF_a}|v| \rceil + \left\lceil \frac{|v|SF_a}{2BF_a} \right\rceil - 1$.

View materialization and indexing benefit. In the general case (Section 3.4), the benefit brought by selecting an object o is defined as the difference between the execution cost of query workload Q before and after inserting o into the final object configuration O. Taking view-index relationships into account implies redefining the benefit function. Let $i \in O_C$ and $v \in O_C$ be a candidate index and a candidate materialized view, respectively. Adding i or v into O may lead to the benefit cases enumerated in Tables 5 and 6, respectively, depending on interactions between i and v.

	$VI_{vi} = 1$	$VI_{vi} = 0$
$v \in O$	min(materialization, indexing v)	indexing
$v \notin O$	—	indexing

Table 5: Benefit brought by index i

	$VI_{vi} = 1$	$VI_{vi} = 0$
$i \in O$	—	materialization
$i \notin O$	min(indexing v, materialization)	materialization

Table 6: Benefit brought by materialized view v

Indexing and view materialization benefits for Q, brought by adding index i or view v into O, respectively, may hence be expressed as follows.

$$benefit_O(i) = \begin{cases} \frac{cost(Q,O)-cost(Q,O\cup\{i\})}{size(i)} \text{ if } VI_{vi} = 0 \ \forall \ v \in V \ (V \subseteq O) \\ \frac{cost(Q,O)-cost(Q,O\cup\{i\}\cup V')}{size(i)+\sum v' \in V' size(v')} \text{ if } V' = \{v \in V, VI_{vi} = 1\} \neq \emptyset \\ 0 \text{ otherwise} \end{cases}$$

$$benefit_O(v) = \begin{cases} \frac{cost(Q,O)-cost(Q,O\cup\{v\})}{size(v)} \text{ if } VI_{vi} = 0 \ \forall \ i \in I \ (I \subseteq O) \\ \frac{cost(Q,O)-cost(Q,O\cup\{v\}\cup I')}{size(v)+\sum i' \in I' size(i')} \text{ if } I' = \{i \in I, VI_{vi} = 1\} \neq \emptyset \\ 0 \text{ otherwise} \end{cases}$$

5 Experiments

In order to experimentally validate our generic approach for optimizing data warehouse performance, we have run several series of tests. We summarize the main results in the following sections. Regarding isolated materialized view or index selection, the interested reader can refer to (Aouiche et al., 2003, 2005, 2006; Mahboubi et al., 2006) for more complete results.

5.1 Experimental conditions

All our tests have been run on a 1 GB data warehouse implemented within Oracle 9*i*, on a Pentium 2.4 GHz PC with 512 MB RAM and a 120 GB IDE disk. Our test data warehouse is actually derived from Oracle's sample data warehouse. Its star schema is composed of one fact table: `Sales`; and five dimensions: `Customers`, `Products`, `Times`, `Promotions` and `Channels`. Table 7 features the characteristics of our test data warehouse. The workload we executed on this data warehouse is composed of 61 decision-support queries involving aggregation operations and multiple joins between the fact table and dimension tables. Due to space constraints, we do not reproduce here the full data warehouse schema nor the detail of each workload query, but they are both available on-line[2].

Table	Number of tuples	Size (MB)
Sales	16,260,336	372.17
Customers	50,000	6.67
Products	10,000	2.28
Times	1,461	0.20
Promotions	501	0.04
Channels	5	0.0001

Table 7: Test data warehouse's characteristics

Note that our experiments are based on an ad-hoc benchmark because, at the time we performed them, there was no standard benchmark for data warehouses. TPC-H (TPC, 2005) does indeed not feature a true multidimensional schema and thus does not qualify, and TPC-DS' (TPC, 2007) draft specifications had not been issued yet.

[2] `http://eric.univ-lyon2.fr/~kaouiche/adbis.pdf`

5.2 Materialized view selection results

We plotted in Figure 8 the variation of workload execution time with respect to the storage space allotted for materialized views. This figure shows that the views we select significantly 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 S is large and thereby better improve execution time. Let S_V be the disk space that is necessary to store all the candidate materialized views. The average gain in performance is indeed 68.9% when $S = 35.4\% \times S_V$. It is equal to 94.9% when $S = 100\% \times S_V$ (when the storage space constraint is relaxed).

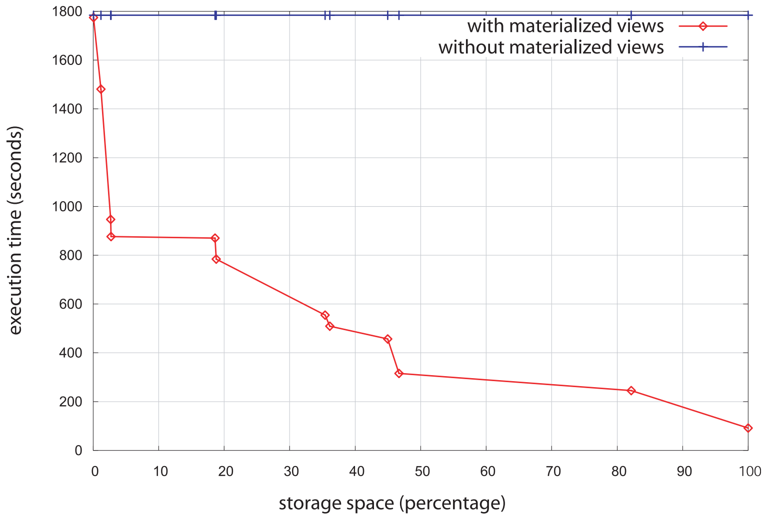

Figure 8: Views materialization experiment results

Moreover, we have demonstrated the relevance of the materialized views that are selected with our approach by computing query cover rate, i.e., the proportion of queries resolved by using views. When the storage space constraint is hard ($S = 0.05\% \times S_V$), average cover rate is already 23%. It reaches 100% when the storage space constraint is relaxed.

5.3 Index selection results

In these experiments, we have fixed the minimal support parameterized in the Close frequent itemset mining algorithm to 1%. We empirically selected this value, basing on our experimental results, as a good trade-off in number of frequent itemsets (and thus, of candidate indexes): enough to allow good quality index selection, but not too many to preserve scalability. It is completely dependent on the workload, though, and would require further investigation in another test environment. In these experiments, we can vary storage space S within a wide interval. We have measured query execution time according to the percentage of storage space allotted for indexes. This percentage is computed from the space S_I occupied by all indexes.

Figure 9 shows that execution time decreases when storage space occupation increases. This is predictable because we create more indexes and thus better improve execution time. We also observe that the maximal time gain is about 30% and it is reached for space occupation $S = 59.64\% \times S_I$.

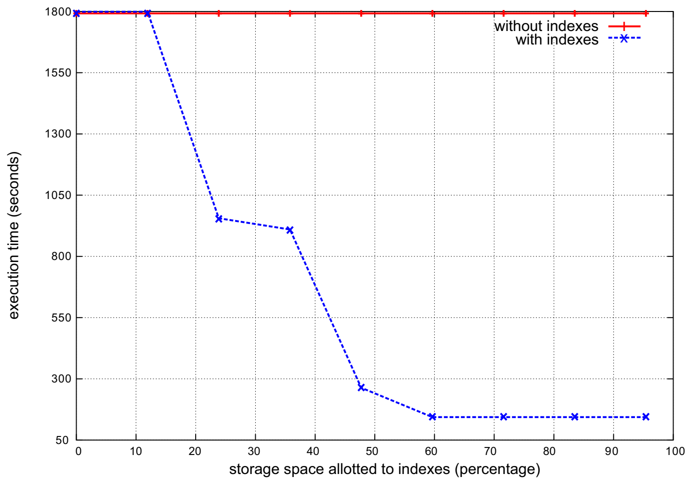

Figure 9: Indexing experiment results

Finally, these experiments also showed that our index selection strategy helped select a portion of candidate indexes that allows to achieve roughly the same performances than the whole set of candidate indexes. This guarantees substantial gains in storage space (40% on an average) and decreases index maintenance cost.

5.4 Joint index and materialized view selection results

Eventually, we have compared the efficiency of isolated materialized view selection, isolated index selection and joint materialized view and index selection. We have measured query execution time in the following cases: without materialized views nor indexes (reference plot), with materialized views only, with indexes only and with both materialized views and indexes (simultaneously selected). Figure 10 represents the variation of response time with respect to the storage space S allotted to materialized views and indexes, for isolated materialized view selection (*with materialized views* plot), isolated index selection (*with indexes* plot), joint materialized view and index selection (*with materialized views and indexes* plot), and *without materialized views nor indexes* (for reference), respectively. S is expressed in percentage of total space S_{VI} occupied by all indexes and materialized views, achieved when we apply our strategy without any storage space constraint. Note that we used a logarithmic scale on the X axis to better visualize the results.

Figure 10 shows that jointly selecting materialized views and indexes allows better performance than selecting indexes or views separately when storage space is large. However, when

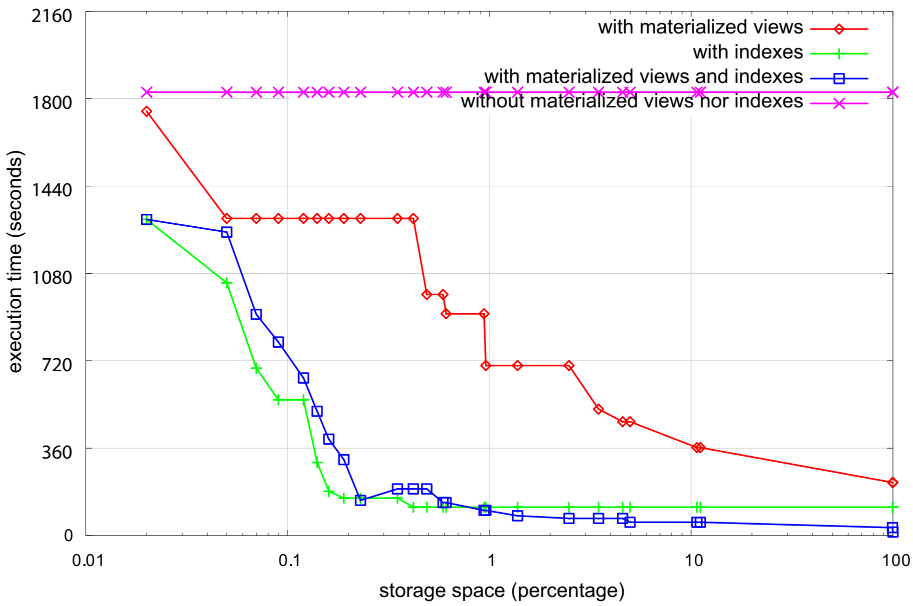

Figure 10: Joint view materialization and indexing experiment results

it is small, isolated index selection is more competitive than the other solutions. This may be explained by the fact that index size is generally significantly smaller than materialized view size. Then, we can store many more indexes than materialized views in a small space and achieve a better performance. In conclusion, indexes should thus be privileged when storage space is strongly constrained.

5.5 Discussion

In this section, we experimentally demonstrated the efficiency of our approach against a "no index – no view" policy. Though it would be much more interesting to compare it to similar works from the literature, especially the one by Agrawal et al. (2000), which is the most closely related to ours, we could not do so.

One problem is that Agrawal et al. (2000) recommend B-tree-like indexes for OLTP (On-Line Transaction Processing) databases. Data warehouse multidimensional models, which are aimed at OLAP (On-Line Analytical Processing), are fundamentally different. They require specific indexes such as the bitmap join indexes we focus on (Section 4.2). In consequence, the underlying cost models used by Agrawal et al. (2000) and us are different, making the comparison partly irrelevant.

A second problem relates to implementing the solution of Agrawal et al. (2000) to achieve this comparison anyway. Although the published algorithms clearly illustrate the approach's principle, we encountered many problems with implementation details and found it quite difficult to faithfully reproduce. Some of these problems came from the solution being so closely tied to Microsoft SQL Server (while we worked on Oracle, but system is actually unimportant since our solution is DBMS-independent). Finally, we failed to contact the authors to get hints at how to solve all these problems.

6 Conclusion and perspectives

We have presented in this paper an approach for automatic data warehouse performance optimization. Our main contribution in this field mainly relates to exploiting knowledge about the data warehouse and its usage. Knowledge may either be formalized expertise, or automatically extracted with the help of data mining techniques. This approach allowed us to reduce the dimensionality of the materialized view and index selection problem, by proposing a reduced and pertinent candidate object configuration. We also have explicitly taken view-index interactions into account, to propose a final object configuration that is as close as possible to the optimum.

We have designed our approach to be generic. Data mining techniques and cost models we exploit are indeed not related to any system in particular. They may be applied on any host DBMS. Our materialized view and index strategies are also modular: each step (candidate object selection, cost computation...) exploits interchangeable tools. The data mining techniques and cost models we used could easily be replaced by others. Moreover, we could also extend our approach to other performance optimization techniques, such as buffering, physical clustering or partitioning (Agrawal, Chaudhuri, Kollár, Marathe, Narasayya, & Syamala, 2004; Zilio, Rao, Lightstone, Lohman, Storm, Garcia-Arellano, & Fadden, 2004; Bellatreche, Boukhalfa, & Mohania, 2005).

Though we have systematically tried to demonstrate the efficiency of our proposals by experimenting on real-life systems such as Oracle, up to now, we have not been able to compare our proposals to existing approaches *in situ*. Those that are developed by DBMS vendors are not always published for obvious industrial property issues. They are also tied to a given system, and necessitate its acquisition. Furthermore, they are implemented as "black boxes" that are often hard to trigger and tinker with. On the other hand, research proposals from the literature are not always available as source or executable code and, when it is the case, they operate in one given environment and must often be reimplemented. Finally, both research and industrial solutions are hard to implement from the published sources that, due to space constraints, generally focus on their principle and not on implementation details. However, we shall have to get over these true difficulties to complete our solutions' experimental validation. In particular, we aim at demonstrating our solution's scalability with respect to existing approaches.

Finally, the main possible evolution for our work resides in improving our solutions' automaticity. We indeed perform static performance optimization. If the input query workload significantly evolves, we must rerun the whole process to preserve performance. Dynamic materialized view selection approaches that have been proposed to optimize refreshing times (Kotidis & Roussopoulos, 1999; Shah, Ramachandran, & Raghavan, 2006) are more efficient than static approaches. We must work in this direction for optimizing query response time.

Our main lead is to exploit our approach's modularity by replacing the data mining techniques we used by incremental frequent itemset mining (Leung, Khan, & Hoque, 2005) or clustering (Jain, Murty, & Flynn, 1999) techniques. Studies related to session detection that are based on entropy computation (Yao, Huang, & An, 2005) could also be exploited to detect when to rerun the (incremental) selection of materialized views and indexes.

Acknowledgments

The authors would like to thank the anonymous reviewers for their detailed and constructive feedback, as well as the JIIS editors, who greatly helped improve this paper.

References

Agrawal, R., & Srikant, R. (1994). Fast algorithms for mining association rules. In *20th International Conference on Very Large Data Bases, (VLDB 1994)*, pp. 487–499.

Agrawal, S., Chaudhuri, S., Kollár, L., Marathe, A., Narasayya, V., & Syamala, M. (2004). Database Tuning Advisor for Microsoft SQL Server 2005. In *30th International Conference on Very Large Data Bases (VLDB 2004), Toronto, Canada*, pp. 1110–1121.

Agrawal, S., Chaudhuri, S., & Narasayya, V. (2001). Materialized view and index selection tool for Microsoft SQL Server 2000. In *ACM SIGMOD International Conference on Management of Data (SIGMOD 2001), Santa Barbara, USA*, p. 608.

Agrawal, S., Chaudhuri, S., & Narasayya, V. R. (2000). Automated selection of materialized views and indexes in SQL databases. In *26th International Conference on Very Large Data Bases (VLDB 2000), Cairo, Egypt*, pp. 496–505.

Aouiche, K., Darmont, J., Boussaïd, O., & Bentayeb, F. (2005). Automatic selection of bitmap join indexes in data warehouses. In *7th International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2005), Copenhagen, Denmark*, Vol. 3589 of *LNCS*, pp. 64–73 Heidelberg, Germany. Springer Verlag.

Aouiche, K., Darmont, J., & Gruenwald, L. (2003). Frequent itemsets mining for database auto-administration. In *7th International Database Engineering and Application Symposium (IDEAS 2003), Hong Kong, China*, pp. 98–103.

Aouiche, K., Jouve, P., & Darmont, J. (2006). Clustering-Based Materialized View Selection in Data Warehouses. In *10th East-European Conference on Advances in Databases and Information Systems (ADBIS 2006), Thessaloniki, Greece*, Vol. 4152 of *LNCS*, pp. 81–95.

Baralis, E., Paraboschi, S., & Teniente, E. (1997). Materialized views selection in a multidimensional database. In *23rd International Conference on Very Large Data Bases (VLDB 1997), Athens, Greece*, pp. 156–165.

Baril, X., & Bellahsene, Z. (2003). Selection of materialized views: a cost-based approach. In *15th International Conference on Advanced Information Systems Engineering (CAiSE 2003), Klagenfurt, Austria*, pp. 665–680.

Bellatreche, L., Boukhalfa, K., & Mohania, M. (2005). An evolutionary approach to schema partitioning selection in a data warehouse environmen. In *7th International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2005), Copenhagen, Denmark*, Vol. 3589 of *LNCS*, pp. 115–125 Heidelberg, Germany. Springer Verlag.

Bellatreche, L., Karlapalem, K., & Schneider, M. (2000). On efficient storage space distribution among materialized views and indices in data warehousing environments. In *9th International Conference on Information and Knowledge Management (CIKM 2000), McLean, USA*, pp. 397–404.

Bruno, N., & Chaudhuri, S. (2006). Physical Design Refinement: The "Merge-Reduce" Approach. In *10th International Conference on Extending Database Technology (EDBT 2006), Munich, Germany*, Vol. 3896 of *LNCS*, pp. 386–404.

Cardenas, A. F. (1975). Analysis and performance of inverted data base structures. *Communication of the ACM*, *18*(5), 253–263.

Chan, G. K. Y., Li, Q., & Feng, L. (1999). 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*, pp. 42–47.

Chaudhuri, S., Datar, M., & Narasayya, V. (2004). Index selection for databases: A hardness study and a principled heuristic solution. *IEEE Transactions on Knowledge and Data Engineering*, *16*(11), 1313–1323.

Chaudhuri, S., Gupta, A., & Narasayya, V. (2002). Compressing SQL workloads. In *2002 ACM SIGMOD International Conference on Management of Data (SIGMOD 2002), Madison, Wisconsin*, pp. 488–499.

Chaudhuri, S., & Narasayya, V. (1997). An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *23rd international Conference on Very Large Data Bases (VLDB 1994), Santiago de Chile, Chile*, pp. 146–155.

Chaudhuri, S., & Motwani, R. (1999). On sampling and relational operators. *IEEE Data Engineering Bulletin*, *22*(4), 41–46.

Choenni, S., Blanken, H. M., & Chang, T. (1993a). Index selection in relational databases. In *5th International Conference on Computing and Information (ICCI 1993), Ontario, Canada*, pp. 491–496.

Choenni, S., Blanken, H. M., & Chang, T. (1993b). On the selection of secondary indices in relational databases. *Data Knowledge Engineering*, *11*(3), 207–238.

Comer, D. (1978). The difficulty of optimum index selection. *ACM Transactions on Database Systems*, *3*(4), 440–445.

Feldman, Y. A., & Reouven, J. (2003). A knowledge–based approach for index selection in relational databases. *Expert System with Applications*, *25*(1), 15–37.

Finkelstein, S. J., Schkolnick, M., & Tiberio, P. (1988). Physical database design for relational databases. *ACM Transactions on Database Systems*, *13*(1), 91–128.

Frank, M. R., Omiecinski, E., & Navathe, S. B. (1992). Adaptive and Automated Index Selection in RDBMS. In *3rd International Conference on Extending Database Technology, (EDBT 1992), Vienna, Austria*, Vol. 580 of *LNCS*, pp. 277–292.

Goldstein, J., & Åke Larson, P. (2001). Optimizing queries using materialized views: a practical, scalable solution. In *ACM SIGMOD international conference on Management of data (SIGMOD 2001), Santa Barbara, USA*, pp. 331–342.

Golfarelli, M., Rizzi, S., & Saltarelli, E. (2002). Index selection for data warehousing. In *4th International Workshop on Design and Management of Data Warehouses (DMDW 2002), Toronto, Canada*, pp. 33–42.

Golfarelli, M., & Rizzi, S. (1998). A methodological framework for data warehouse design. In *1st ACM International Workshop on Data Warehousing and OLAP (DOLAP 1998), New York, USA*, pp. 3–9.

Gundem, T. I. (1999). Near optimal multiple choice index selection for relational databases. *Computers & Mathematics with Applications*, *37*(2), 111–120.

Gupta, H. (1999). *Selection and Maintenance of Views in a Data Warehouse*. Ph.D. thesis, Stanford University.

Gupta, H., Harinarayan, V., Rajaraman, A., & Ullman, J. D. (1997). Index selection for OLAP. In *13th International Conference on Data Engineering (ICDE 1997), Birmingham, UK*, pp. 208–219.

Gupta, H., & Mumick, I. S. (2005). Selection of views to materialize in a data warehouse. *IEEE Transactions on Knowledge and Data Engineering*, *17*(1), 24–43.

Harinarayan, V., Rajaraman, A., & Ullman, J. D. (1996). Implementing data cubes efficiently. In *ACM SIGMOD International Conference on Management of data (SIGMOD 1996), Montreal, Canada*, pp. 205–216.

Ip, M. Y. L., Saxton, L. V., & Raghavan, V. V. (1983). On the selection of an optimal set of indexes. *IEEE Transactions on Software Engineering*, *9*(2), 135–143.

Jain, A. K., Murty, M. N., & Flynn, P. J. (1999). Data clustering: a review. *ACM Computing Surveys*, *31*(3), 264–323.

Jouve, P., & Nicoloyannis, N. (2003a). KEROUAC: an algorithm for clustering categorical data sets with practical advantages. In *International Workshop on Data Mining for Actionable Knowledge (DMAK/PAKDD 2003), Seoul, Korea*.

Jouve, P., & Nicoloyannis, N. (2003b). A new method for combining partitions, applications for distributed clustering. In *International Workshop on Paralell and Distributed Machine Learning and Data Mining (ECML/PKDD 2003), Cavtat-Dubrovnik, Croatia*, pp. 35–46.

Kotidis, Y., & Roussopoulos, N. (1999). Dynamat: A dynamic view management system for data warehouses. In *ACM SIGMOD International Conference on Management of Data (SIGMOD 1999), Philadelphia, USA*, pp. 371–382.

Kratica, J., Ljubić, I., & Tošić, D. (2003). A genetic algorithm for the index selection problem. In *Applications of Evolutionary Computing, EvoWorkshops 2003: EvoBIO, EvoCOP, EvoIASP, EvoMUSART, EvoROB, EvoSTIM*, Vol. 2611 of *LNCS*, pp. 281–291.

Kyu-Young, W. (1987). *Index Selection in Relational Databases*, pp. 497–500. Foundation of Data Organization. Plenum Publishing.

Labio, W., Quass, D., & Adelberg, B. (1997). Physical database design for data warehouses. In *13th International Conference on Data Engineering (ICDE 1997), Birmingham, UK*, pp. 277–288.

Leung, C., Khan, Q., & Hoque, T. (2005). CanTree: A Tree Structure for Efficient Incremental Mining of Frequent Patterns. In *5th IEEE International Conference on Data Mining (ICDM 2005), Houston, USA*, pp. 274–281.

Mahboubi, H., Aouiche, K., & Darmont, J. (2006). Materialized View Selection by Query Clustering in XML Data Warehouses. In *4th International Multiconference on Computer Science and Information Technology (CSIT 2006), Amman, Jordan*, Vol. 2, pp. 68–77.

Nadeau, T. P., & Teorey, T. J. (2001). A pareto model for OLAP view size estimation. In *4th Conference of the Centre for Advanced Studies on Collaborative Research (CASCON 2001), Toronto, Canada*, p. 13.

Nadeau, T. P., & Teorey, T. J. (2002). Achieving scalability in OLAP materialized view selection. In *5th ACM International Workshop on Data Warehousing and OLAP (DOLAP 2002), McLean, USA*, pp. 28–34.

O'Neil, P., & Graefe, G. (1995). Multi–table joins through bitmapped join indices. *SIGMOD Record*, *24*(3), 8–11.

O'Neil, P., & Quass, D. (1997). Improved query performance with variant indexes. In *ACM SIGMOD International Conference on Management of Data (SIGMOD 1997), Tucson, USA*, pp. 38–49.

Pasquier, N., Bastide, Y., Taouil, R., & Lakhal, L. (1999). Discovering frequent closed itemsets for association rules. In *7th International Conference on Database Theory (ICDT 1999), Jerusalem, Israel*, Vol. 1540 of *LNCS*, pp. 398–416.

Rizzi, S., & Saltarelli, E. (2003). View materialization vs. indexing: Balancing space constraints in data warehouse design. In *15th International Conference on Advanced Information Systems Engineering (CAiSE 2003), Klagenfurt, Austria*, pp. 502–519.

Sarawagi, S. (1997). Indexing OLAP data. *Data Engineering Bulletin*, *20*(1), 36–43.

Shah, B., Ramachandran, K., & Raghavan, V. (2006). A Hybrid Approach for Data Warehouse View Selection. *International Journal of Data Warehousing and Mining*, *2*(2), 1–37.

Shukla, A., Deshpande, P., & Naughton, J. F. (2000). Materialized view selection for multicube data models. In *7th International Conference on Extending DataBase Technology (EDBT 2000), Konstanz, Germany*, pp. 269–284.

Shukla, A., Deshpande, P. M., Naughton, J. F., & Ramasamy, K. (1996). Storage estimation for multidimensional aggregates in the presence of hierarchies. In *22nd International Conference on Very Large Data Bases (VLDB 1996), Bombay, India*, pp. 522–531.

Sismanis, Y., Deligiannakis, A., Roussopoulos, N., & Kotidis, Y. (2002). Dwarf: shrinking the petacube. In *ACM SIGMOD International Conference on Management of Data (SIGMOD 2002), Madison, USA*, pp. 464–475.

Smith, J. R., Li, C.-S., & Jhingran, A. (2004). A wavelet framework for adapting data cube views for OLAP. *IEEE Transactions on Knowledge and Data Engineering*, *16*(5), 552–565.

TPC (2005). *TPC Benchmark H Standard Specification revision 2.3.0*. Transaction Processing Performance Council.

TPC (2007). *TPC Benchmark DS Standard Specification, Draft Version 52*. Transaction Processing Performance Council.

Uchiyama, H., Runapongsa, K., & Teorey, T. J. (1999). A progressive view materialization algorithm. In *2nd ACM International Workshop on Data warehousing and OLAP (DOLAP 1999), Kansas City, USA*, pp. 36–41.

Valentin, G., Zuliani, M., Zilio, D., Lohman, G., & Skelley, A. (2000). DB2 advisor: An optimizer smart enough to recommend its own indexes. In *16th International Conference on Data Engineering, (ICDE 2000), California, USA*, pp. 101–110.

Valluri, S. R., Vadapalli, S., & Karlapalem, K. (2002). View relevance driven materialized view selection in data warehousing environment. In *13th Australasian Database Technologies (ADC 2002), Melbourne, Australia*, pp. 187–196.

Whang, K. (1985). Index selection in relational databases. In *International Conference on Foundations of Data Organization (FODO 1985), Kyoto, Japan*, pp. 487–500.

Wu, M. (1999). Query optimization for selections using bitmaps. In *ACM SIGMOD International Conference on Management of Data (SIGMOD 1999), Philadelphia, USA*, pp. 227–238.

Wu, M., & Buchmann, A. (1998). Encoded bitmap indexing for data warehouses. In *14th International Conference on Data Engineering (ICDE 1998), Orlando, USA*, pp. 220–230.

Yao, Q., Huang, J., & An, A. (2005). Machine learning approach to identify database sessions using unlabeled data. In *7th International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2005), Copenhagen, Denmark*, Vol. 3589 of *LNCS*, pp. 254–255.

Yao, S. B. (1977). Approximating block accesses in database organizations. *Communications of the ACM*, *20*(4), 260–261.

Zaman, M., Surabattula, J., & Gruenwald, L. (2004). An Auto-Indexing Technique for Databases Based on Clustering. In *15th International Workshop on Database and Expert Systems Applications (DEXA Workshops 2004), Zaragoza, Spain*, pp. 776–780.

Zilio, D., Rao, J., Lightstone, S., Lohman, G., Storm, A., Garcia-Arellano, C., & Fadden, S. (2004). DB2 Design Advisor: Integrated Automatic Physical Database Design. In *30th International Conference on Very Large Data Bases (VLDB 2004), Toronto, Canada*, pp. 1087–1097.