HAL
open science

A Declarative Approach to View Selection Modeling

Imene Mami, Zohra Bellahsene, Remi Coletta

To cite this version:

Imene Mami, Zohra Bellahsene, Remi Coletta. A Declarative Approach to View Selection Modeling. Transactions on Large-Scale Data- and Knowledge-Centered Systems, 2013, Part X - Special Issue on Database- and Expert-Systems Applications, LNCS (8220), pp.115-145. 10.1007/978-3-642-41221-9_5 . lirmm-00950884

HAL Id: lirmm-00950884

https://hal-lirmm.ccsd.cnrs.fr/lirmm-00950884v1

Submitted on 23 Feb 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Declarative Approach to View Selection Modeling

Imene Mami, Zohra Bellahsene, and Remi Coletta

University Montpellier 2, LIRMM, France
{mami,bella,coletta}@lirmm.fr

Abstract. View selection is important in many data-intensive systems e.g., commercial database and data warehousing systems. Given a database (or a data warehouse) schema and a query workload, view selection is to choose an appropriate set of views to be materialized that optimizes the total query cost, given a limited amount of resource, e.g., storage space and total view maintenance cost. The view selection problem is known to be a NP-complete problem. In this paper, we propose a declarative approach that involves a constraint programming technique which is known to be efficient for the resolution of NP-complete problems. The originality of our approach is that it provides a clear separation between formulation and resolution of the problem. For this purpose, the view selection problem is modeled as a constraint satisfaction problem in an easy and declarative way. Then, its resolution is performed automatically by the constraint solver. Furthermore, our approach is flexible and extensible, in that it can easily model and handle new constraints and new heuristic search strategies to reduce the solution space. The performance results show that our approach outperforms the genetic algorithm which is known to provide the best trade-off between quality of solutions in terms of cost saving and execution time.

keywords: Database design, modeling and management, query processing and optimization, view selection, materialized views.

1 Introduction

Selecting the best set of views to materialize for a given query workload, under certain resource constraints, is one of the most common problems in commercial database management systems and data warehousing systems. In many applications, and in particular in a data warehouse application, queries need to be answered over massive amounts of data. Materializing and exploiting previous query results (views) can be important for efficient processing of queries by avoiding re-computation of expensive query operations. Consequently, answering queries using materialized views is significant for improving query performance. To support view selection process, different related issues have to be considered. One of the challenging issues is the view maintenance which is the process of updating a materialized view. Indeed, whenever a data source is changed, the materialized views built on it have to be updated (or at least have to be checked

whether some changes have to be propagated or not) in order to compute up-to-date query results. The view maintenance cost constraint is very important in the view selection process and cannot be ignored. Otherwise, the cost of the view maintenance may offset the performance advantages provided by the view materialization. Besides the view maintenance issue, each materialized view requires additional storage space which must be taken into account when deciding which and how many views to materialize. Hence, there is a need for selecting a set of views to materialize by taking into account three important features: query cost, view maintenance cost and storage space. The problem of choosing which views to materialize that minimize the total query cost given a limited amount of resource such as total view maintenance cost and storage space is known as the view selection problem. This is one of the most complex problem solving: it is known to be a NP-complete problem [11]. Moreover, the number of possible view combinations to materialize grows exponentially with the number of queries and with the numbers of columns, join predicates and grouping clauses.

There has been much work on materialized view selection. A naive method is to apply an algorithm which finds the optimal set of materialized views by browsing through all sets of considered views to materialization. However, an exhaustive search cannot be applied due to the complexity of the problem. The most efficient method for deciding which views to be materialized for a given workload is a randomized method which uses the genetic algorithm [5,18,33]. The main difference between the genetic algorithm and previously designed algorithms, i.e., greedy algorithms [7,8,24,28,32] is that it can be applicable on the large search space. It can find a reasonable solution within a relatively short period of time by trading executing time for quality. However, there is no guarantee of performance because the probabilistic behavior of the genetic algorithms does not insure to find the global optimum. Besides, the quality of the solution (i.e., the quality of the obtained set of materialized views in terms of cost saving) depends on the set-up of the algorithm as well as the extremely difficult fine-tuning of the algorithm that must be performed during many test runs.

In this paper, we have proposed a constraint programming based approach. Constraint programming is a general framework which relies on a combination of techniques that deal with reasoning. It has been applied with success to many domains such as scheduling, planning, vehicle routing, configuration, networks and bioinformatics. More recently, constraint programming has been considered as beneficial in data mining setting [25]. Our motivation to use constraint programming in solving the view selection problem is that it is known to be efficient for the resolution of NP-complete problems and a powerful method for modeling and solving combinatorial optimization problems [26]. To solve a given problem by means of constraint programming, the problem must be represented as a constraint satisfaction problem. This part of the problem solving is called modeling. Then, the resolution of the modeled problem is performed automatically by the constraint solver in the solving stage. The originality of our approach is that it provides a clear separation between formulation and resolution of the problem. Indeed, constraint programming is a declarative programming paradigm: instead

of specifying how to solve the problem, the user has only to specify the problem itself.

Our Goals. Based on the application workload, we select a set of views to materialize over a database (or data warehouse) schema, such that the cost of evaluating queries is minimal, subject to space and maintenance cost constraints. Our goal is to provide better solution quality (i.e., the quality of the obtained set of materialized views in terms of cost saving) with respect to the currently most efficient approach (genetic algorithm). The focus of this study is also to enable optimal view selection by using constraint programming techniques.

Our Contributions. We propose a novel and efficient approach to address the view selection problem. Our approach is based on constraint programming techniques and consists in modeling in a declarative way the view selection as a constraint satisfaction problem. We formalize and study the view selection problem under a limited amount resource, e.g., storage space and view maintenance cost. The contributions of this paper are based on the extension of our previous work [23].

- We include further explanations and illustrations through the paper, i.e., how the constraint programming can be applied to decide which views to materialize (see section 3.2).
- We propose a heuristic search strategy to efficiently search the solution space (see section 5.2.2). We prove that the time that a constraint solver incurs for finding near optimal and optimal solutions is significantly reduced (see section 6.2).
- We also show the effectiveness of our heuristic based search strategy which improves in several magnitudes the quality of the solution provided by the previous version [23]. Hence, our approach achieves significant performance gains compared with the genetic algorithm in terms of cost saving (see section 6.3.1 and 6.3.2). While in our previous work [23] we achieved only a slight improvement.
- We design new and various experiments to prove the efficiency of our approach when we simulate diverse query workloads by generating different query and update distribution and query complexity. The results show that, over all the experiments, the performance of our approach is much better than that of the genetic algorithm (see section 6.3.3 and 6.3.4).
- We perform real experiments on MySQL server in order to measure the *real* query runtime (see section 6.4). The results of these experiments have shown that queries using our proposed views are evaluated faster in comparison with those found by the genetic algorithm. These experiments also confirm the robustness of our approach toward simplified cost models. This requirement is very important for database optimization as it is based on cost estimations.

Paper Outline The rest of this paper is organized as follows. After reviewing and classifying prior work in the view selection context, Section 3 contains the background related to understand the view selection problem and discusses the settings for the problem. In Section 4, we present the framework that we have used for representing views to materialize in order to exhibit common subexpressions. Section 5 describes how to model the view selection problem as a

constraint satisfaction problem as well as the heuristic search strategy that we have designed for optimization purpose. Section 6 gives a performance analysis comparing our approach with the genetic algorithm. The paper ends with a summary and future works in Section 7.

2 Related work

In this section, we review the view selection methods based on what kind of algorithms they use to address the view selection problem. The best-known heuristic algorithms proposed in literature to tackle the problem of finding an appropriate set of views to materialize can be classified into three major groups: deterministic algorithms, randomized algorithms and hybrid algorithms. For a deeper review of the existing view selection approaches, we refer the reader to the survey that we have done in our previous work [21].

Deterministic Algorithms Based Methods Much research work on view selection uses deterministic strategies to address the view selection problem. [27] is the first paper that provides a solution for materializing view indexes which can be seen as a special case of the materialized views. The solution is based on A* algorithm. An exhaustive approach is also presented in [16] for finding the best set of views to materialize. Nevertheless, an exhaustive search cannot compute the optimal solution in a reasonable time.

The authors in [9] present and analyze algorithms for view selection in case of OLAP-style queries. They provide a polynomial-time greedy algorithm to select a set of views to materialize that minimizes the query cost subject to a space constraint. However, this approach does not consider the view maintenance cost. The work in [32] is dealing with more general SQL queries which include select, project, join, and aggregation operations. A greedy algorithm has been designed to select a set of materialized views so that the combined query and view maintenance cost is minimized. However, the view maintenance cost has been overrated since the maintenance cost for a materialized view is the cost used for constructing this view. Besides, the view selection is done without any resource constraint.

A theoretical framework for the view selection problem in data warehousing setting has been developed in [7]. Their work provides a near-optimal polynomial time greedy algorithm for the cases of AND view graph, where each query (or view) has a unique evaluation, and OR view graph, in which any view can be computed from any one of its related views. For the most general case of AND-OR view graph which allows a single query to be answered and updated from multiple paths, they have designed a near-optimal exponential time greedy algorithm. This approach was extended in [8] to study the view selection under a maintenance cost constraint.

The view selection has been studied in [19,30,31] under the condition that the input queries can be answered using exclusively the materialized views. An exhaustive algorithm has been designed in [31] to select a set of materialized views while minimizing the combination of the query and view maintenance cost. This work was extended in [19] by developing greedy algorithms that expand only

a small fraction of the states produced by the exhaustive algorithm. The view selection problem in [30] is addressed under a space constraint. However, their view selection algorithm is still in exponential time.

The system designed in [3] runs a greedy enumeration algorithm to pick a set of views and indexes to materialize by taking into account the space constraint. Nevertheless, this approach does not take into account the view maintenance cost.

The authors in [28] demonstrate that using multi-query optimization techniques in conjunction with a greedy heuristic provides significant benefit. The greedy heuristic is used to iteratively pick from the AND-OR view graph the set of views to materialize that minimizes the query cost. This study was extended in [24] to consider how to optimize the view maintenance cost. However, the view selection has been studied without any resource constraint.

In order to improve the query performance as well as save the storage space, the study in [29] aims at materializing only a part of the relations instead of considering all tuples in the relations. An efficient algorithm has been designed which uses clustering techniques to select the set of views to be materialized.

The above methods take a deterministic approach either by exhaustive search or by some heuristics such as greedy. However, greedy search is subjected to the known caveats, i.e., sub-optimal solutions may be retained instead of the globally optimal one since initial solutions influence the solution greatly. As a result, other algorithms have been developed to improve the solutions of the view selection problem, namely: randomized algorithms and hybrid algorithms which we describe in next sections.

Randomized Algorithms Based Methods Typical randomized algorithms are genetic or use simulated annealing. Genetic algorithms generate solutions using techniques inspired by the natural evolution process such as selection, mutation, and crossover. The search strategy for these algorithms is very similar to biological evolution. Genetic algorithms start with a random initial population and generate new populations by random crossover and mutation. The fittest individual found is the solution. The algorithms terminate as soon as there is no further improvement over a period.

A genetic algorithm has been used in [33] to solve the view selection problem. The materialized views have been selected according to their reduction in the combined query and view maintenance cost. However, because of the random characteristic of the genetic algorithm, some solutions can be infeasible. For example, in the maintenance cost constrained model, when a view is selected, the benefit will not only depend on the view itself but also on other views that are selected. One solution to this problem is to add a penalty value as part of the fitness function to ensure that infeasible solutions will be discarded. For instance, a penalty function has been applied in [18] which reduces the fitness each time the maintenance cost constraint is not satisfied. This approach minimizes the query cost given varying upper bounds on the view maintenance cost, assuming unlimited amount of storage space. In order to let the genetic algorithm converge faster, they represent the initial population as a favorable configuration based

on external knowledge about the problem and its solution rather than a random sampling, i.e., the views with a high query frequency are most likely selected for materialization.

The approach proposed in [10] use simulated annealing algorithms to address the view selection problem. These algorithms are motivated by an analogy to annealing in solids. Simulated Annealing algorithms start with an initial configuration, generate new configurations by random walk along the different solutions of the solution space according to a cooling schedule and terminate as soon as no applicable ones exist or lose all the energy in the system. The view selection problem is solved in [10] under the case where either the space constraint or the maintenance cost constraint is considered. Further, randomized search has been applied to solve two more issues. First, they considered the case where both space and maintenance constraints exist. Next they applied a randomized search in the context of dynamic view selection.

In contrast with simulated annealing algorithms, genetic algorithms use a multi-directional search which allows to efficiently search the space and find better solution quality. For more details about this observation we refer the reader to [18]. Randomized algorithms can be applied to complex problems dealing with large or even unlimited search spaces. Recent works [13,14] have shown that randomized search heuristic techniques, in comparison to greedy techniques, are able to select comparatively better quality views for higher dimensional data sets. However, they may have a tendency to converge toward local optima due to their random characteristics. Besides, their successes often depend on the set-up of the algorithm as well as the extremely difficult fine-tuning of algorithm that must be performed during many test runs.

Hybrid Algorithms Based Methods Hybrid algorithms combine the strategies of deterministic and randomized algorithms in their search in order to provide better performance in terms of solution quality. Solutions obtained by deterministic algorithms are used as initial configuration for simulated annealing algorithms or as initial population for genetic algorithms.

A hybrid approach has been applied in [34] which combines heuristic algorithms i.e., greedy algorithms and genetic algorithms to solve three related problems. The first one is to optimize queries. The second one is to choose the best global processing plan from multiple processing plans for each query. The third problem is to select materialized views from a given global processing plan. Their experimental results confirmed that hybrid algorithms provide better performance than either genetic algorithms or heuristic algorithms i.e., greedy algorithms used alone in terms of solution quality. However, their algorithms are more time consuming and may be impractical due to their excessive computation time.

3 Background

3.1 View Selection Problem and Cost Model

View Selection Problem The problem of view selection that we consider in this paper is to select a set of views to be materialized in order to speed up a given set

of queries constrained by a storage space capacity and maintenance costs to keep the materialized views in synchronization with the underlying base relations.

More precisely, the view selection problem can be defined as follows: *Given a query workload* $Q = \{q_1, q_2, ..., q_q\}$ *and their query frequency* $fQ = \{f_{q_1}, f_{q_2}, ..., f_{q_q}\}$ *over a given database (or data warehouse) schema* $R = \{r_1, r_2, ..., r_r\}$*, a set of updates* $U = \{u_1, u_2, ..., u_u\}$ *on base relations and their update frequency* $fU = \{fu_{r_1}, fu_{r_2}, ..., fu_{r_r}\}$ *and a limited amount of resource, e.g., storage space* Sp_{max} *and view maintenance cost limit* U_{max}*, the problem is to find a set of views to materialize* $MV = \{v_1, v_2, ..., v_v\}$ *such as the cost of evaluating the query workload is minimal.*

Cost Model The cost model assigns an estimated cost e.g., query cost or maintenance cost to any view (or query) in the search space. In our approach, we use a cost model similar to [6,20]. Hence, the query and view maintenance costs are estimated with respect to CPU and IO costs. In this paper we consider selection-projection-join (SPJ) queries that may involve aggregation and a group by clause as well. The formulas used for cost operations estimation are given below with the following assumptions:

- Formulas to estimate the cost of executing every relational operation take into account its implementation, e.g., we consider sequential scans and nested loop joins.
- The CPU cost is estimated as the time needed to process each tuple of the relation e.g., checking selection conditions.
- The IO cost estimation is the time necessary for fetching each tuple of the relation.
- The costs are estimated according to the size of the involved relations and in terms of time.

Estimated cost of relational operations.

- Estimated cost of unary operations
  - $cost(op) = (IO * card * length) + (CPU * card * lengthP)$ where op is a selection operation
  - $cost(op) = (IO * card * log(card) * length) + (CPU * card * log(card) * lengthP)$ where op is a projection operation
  - $cost(op) = (IO * card * length) + (CPU * card * lengthA)$ where op is an aggregation operation
- Estimated cost of binary operations
  - $cost(op) = (IO * lcard * rcard * (llength + rlength)) + (CPU * lcard * rcard * lengthP)$ where op is a join operation

Where *card* is the number of tuples of the operand, *length* is the length (in bytes) of a tuple, *lengthP* is the length of columns checked by predicates, *lengthA* is the length of the tuples being aggregated, *lcard* and *rcard* are respectively the number of tuples of the left and right operands (the same for *llength* and *rlength*).

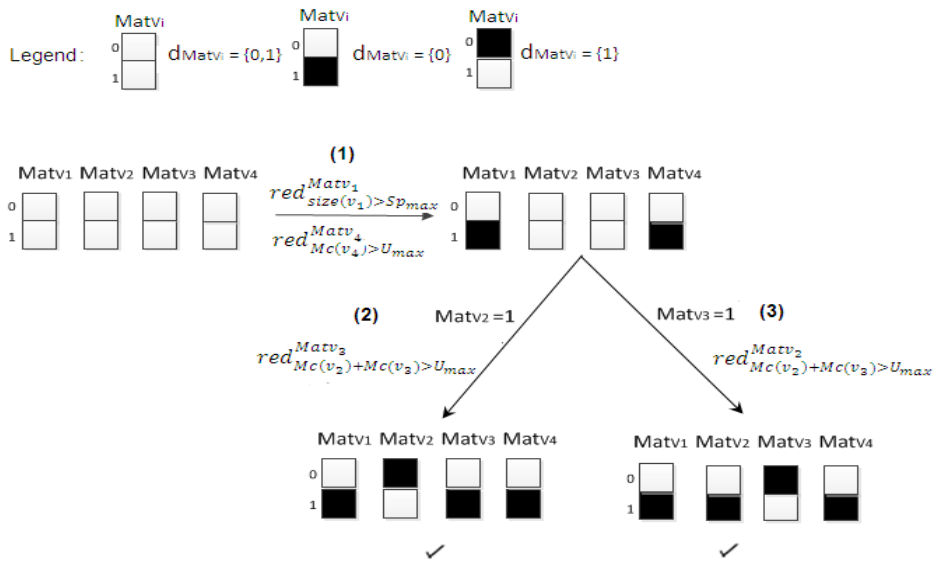


Fig. 1: Search tree using constraint propagation

3.2 Constraint Programming Notions

Constraint programming has been successfully applied in numerous combinatorial search problems [26] such as scheduling and timetabling. Constraint programming allows to solve combinatorial problems modeled as a Constraint Satisfaction Problem (CSP). Indeed, the principle idea of constraint programming is to solve problems by stating constraints which must be satisfied by the solution.

Formally, a CSP is defined by a triplet (VAR;DOM;CST):

- Variables. $VAR = \{var_1, var_2, ..., var_n\}$ is the set of variables of the problem.
- Domains. $DOM = \{d_{var_1}, d_{var_2}, ..., d_{var_n}\}$ is the set of possible values that can be assigned to each variable var_i.
- Constraints. $CST = \{c_1, c_2, ..., c_n\}$ is the set of constraints that describes the relationship between subsets of variables. Formally, a constraint C_{ijk} between the variables var_i, var_j, var_k is any subset of the possible combinations of values of var_i, var_j, var_k, i.e., $C_{ijk} \subset d_{var_i} \times d_{var_j} \times d_{var_k}$. The subset specifies the combinations of values that the constraint allows.

A feasible solution to a CSP is an assignment of a value from its domain to every variable, so that the constraints on these variables are satisfied. For optimization purpose some cost expression on these variables takes a maximal or minimal value.

Most algorithms for solving CSPs usually use constraint propagation to reduce the size of the search space to be explored [15]. When a value of a variable is fixed, constraint propagation is applied to restrict the domains of other variables whose values are not currently fixed. This means that when a value is assigned to

the current variable, any value in the domain of a future variable which conflicts with this assignment is removed from the domain.

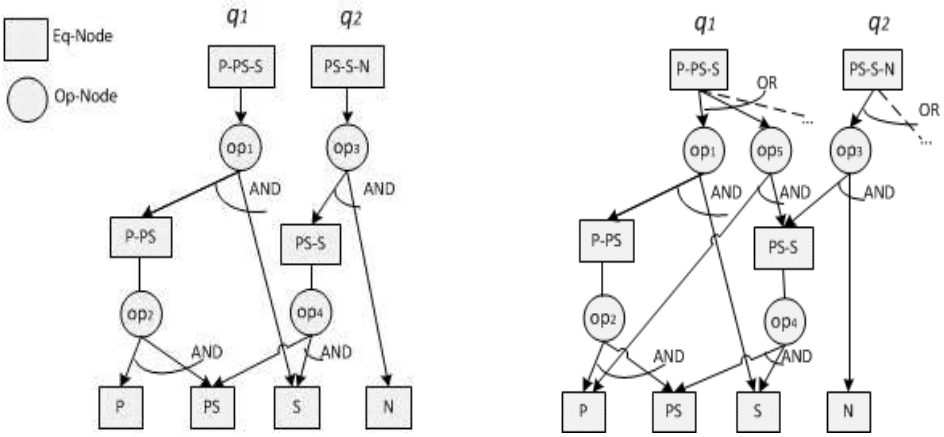
Let us now illustrate this in the context of view selection problem. Figure 1 shows the domain reduction of four variables Mat_{v_1}, Mat_{v_2}, Mat_{v_3} and Mat_{v_4} where Mat_{v_i} denotes for each view v_i if it has been materialized or has not been materialized. It is a binary variable, $d_{Mat_{v_i}}$ ={0,1} (0: v_i has not been materialized, 1: v_i has been materialized). The problem is to select a set of views to materialize subject to a space and maintenance cost constraints. The space constraint ensures that the total space occupied by the materialized views is less than Sp_{max}. Let as assume that Sp_{max}=3MB, $size(v_1)$=4MB, $size(v_2)$=2MB, $size(v_3)$=1MB and $size(v_4)$=1MB; where $size(v_i)$ is the size of the view v_i. While, the maintenance cost constraint guarantees that the time to update the set of materialized views is less than U_{max}. Note that $U_{max} = 3sec$, $Mc(v_1)$=1sec, $Mc(v_2)$=2sec, $Mc(v_3)$=2sec and $Mc(v_4)$=5sec; where $Mc(v_i)$ denotes the cost of maintaining the view v_i.

At the beginning, the initial variable domains, $d_{Mat_{v_1}}$=$d_{Mat_{v_2}}$=$d_{Mat_{v_3}}$=$d_{Mat_{v_4}}$ ={0,1}, are represented by four columns of white squares. Considering the space and maintenance cost constraints, it appears that Mat_{v_1} and Mat_{v_4} cannot take the value 1 because otherwise the total space and maintenance cost of the materialized views will be respectively greater than Sp_{max} and U_{max}. In the stage (1), $red^{Mat_{v_1}}_{size(v_1)>Sp_{max}}$ and $red^{Mat_{v_4}}_{Mc(v_4)>U_{max}}$ filters respectively the inconsistent value 1 from $d_{Mat_{v_1}}$ and $d_{Mat_{v_4}}$. The deleted values are marked with a black square. After this stage some variable domains are not reduced to singletons, the constraint solver takes one of these variables and tries to assign to it each of the possible values in turn. For example, if the solver selects the view v_2 to be materialized ($Mat_{v_2} = 1$, see stage (2)), $red^{Mat_{v_3}}_{Mc(v_2)+Mc(v_3)>U_{max}}$ eliminates the value 1 from $d_{Mat_{v_3}}$. Otherwise, if the view v_3 is selected to be materialized ($Mat_{v_3} = 1$, see stage (3)), $red^{Mat_{v_2}}_{Mc(v_2)+Mc(v_3)>U_{max}}$ withdraws the value 1 from $d_{Mat_{v_2}}$. This enumeration stage leads in our example to two solutions. These solutions are of various quality or cost.

In addition to providing a rich constraint language to model a problem as a CSP and techniques such as constraint propagation to reduce the search space by excluding solutions where the constraints become inconsistent, constraint programming offers facilities to control the search behavior. This means that search strategies can be defined to decide in which order to explore the created child nodes in an enumeration tree which can significantly reduce the execution time. Furthermore, constraint programming provides ways to limit the tree search regarding different criteria. For instance performing the search until reaching a feasible solution in which all constraints are satisfied, or until reaching a search time limit or until reaching the optimal solution.

4 Framework for detecting common views

In our approach, the task of a view selection module is to recognize possibilities of shared views and then to apply a strategy that use constraint programming



(a) AND view graph (b) AND-OR view graph

Fig. 2: DAG representation of two queries q_1 and q_2

techniques for deciding which views to materialize. The first task involves setting up the search space by identifying common sub-expressions between the different queries of workload. This feature can be exploited for sharing computation, updates and storage space. The most commonly used frameworks in the context of representing SQL queries in order to exhibit common sub-expressions are the AND view graph and the AND-OR view graph. In what follows, we start by giving a formal definition of these representations.

Definition 4.1 (AND View Graph) *An AND view graph is formed from the union of individual AND-DAG representations of each query. An AND-DAG representation for a query or a view v is a directed acyclic graph having the base relations as leaf nodes and the node v as a root node and consists of a set of operation nodes (Op-Nodes) and equivalence nodes (Eq-Nodes). The Op-nodes have only Eq-nodes as children and Eq-nodes have only Op-nodes as children. Each Op-Node corresponds to an algebraic expression (Select-Project-Join) with possible aggregate function. It represents the expression defined by the operand and its inputs. An Eq-Node represents an expression that is defined by the child operation node and its inputs. Each Eq-Node represents a view that could be selected for materialization. In an AND-DAG representations, each Op-node op_i has associated with it an AND arc which is indicated by drawing a semicircle, through the edges (op_i,v_{c_1}),(op_i,v_{c_2}),...,(op_i,v_{c_i}). This dependence means that all the views v_{c_1}, v_{c_2},...,v_{c_i} that are the child nodes of op_i are needed to compute the view v_p which is the parent node of op_i.*

Definition 4.2 (AND-OR View Graph) *A graph is called an AND-OR view graph if for each query or a view v, there is an AND-OR-DAG representation. All the possible AND-DAG representations for v, described in the previous definition,*

become the AND-OR DAG which consists of all possible execution plans for v. If a parent view v_p has outgoing edges to children operation nodes op_1,op_1,...,op_i, then v_p can be computed from any one of its children. This dependence is indicated by drawing a semicircle, called an OR arc. The AND-OR view graph can be constructed by merging the AND-OR DAG for each query where the common sub-expressions are represented once.

The DAG representation of the queries q_1: P $\bowtie$ PS $\bowtie$ S and q_2: PS $\bowtie$ S $\bowtie$ N, are shown in figure 2. The subscripts P, PS, S and N denote respectively the base relations of TPC-H benchmark: Part, PartSupp, Supplier and Nation. In the AND view graph (see figure 2a), there is only one way to answer or update a view (or query). Indeed, the views P-PS-S and PS-S-N, corresponding respectively to the result of the query q_1 and q_2, can be computed or updated on only one way (it consider optimal query plans):

q_1**:((P $\bowtie$ PS) $\bowtie$ S)**
q_2**:((PS $\bowtie$ S) $\bowtie$ N)**

However, all possible ways for evaluating the queries have been considered in the AND-OR view graph (see figure 2b). For simplicity, we represent only two execution plans for the view P-PS-S which is the query result of q_1 and one execution plan for the view PS-S-N that is the query result of q_2:

q_1**:{((P $\bowtie$ PS) $\bowtie$ S), (P $\bowtie$ (PS $\bowtie$ S))}** // **two execution plans**
q_2**:((PS $\bowtie$ S) $\bowtie$ N)** // **one execution plan**

The remaining execution plans are just indicated in figure 2b by dashed lines.

In this paper, we use the AND-OR view graph to compactly represent alternative query plans and exhibit common subexpression. For more details about constructing the AND-OR view graph for the queries of workload, we may refer the reader to [28].

Our motivation to use the AND-OR representation rather than the AND representation since the latter makes local optimal choices, and may miss global optimal plans. The choice of materialized views must be done in conjunction with choosing execution plans for queries. For instance, a plan that seems quite inefficient could become the best plan if some intermediate result of the plan is chosen to be materialized and maintained as the following example demonstrates it.

Example. Let us consider the views P-PS-S and PS-S-N which are respectively computed by using the plan ((P $\bowtie$ PS) $\bowtie$ S) and the plan ((PS $\bowtie$ S) $\bowtie$ N), as it is shown in figure 2a. These execution plans represent the optimal plans for q_1 and q_2. However, if we choose the alternative plan (P $\bowtie$ (PS $\bowtie$ S)) to compute the view P-PS-S, the view PS-S becomes a common subexpression (see figure 2b). It can be computed once and used for both queries q_1 and q_2. This alternative with sharing of the view PS-S may be the global optimal choice.

In the context of view maintenance, common sub-expressions can be exploited to find an efficient plan for maintenance of a set of views. Indeed, the view P-PS-

S may also be used for sharing updates and hence reducing the view maintenance cost.

Note that in the AND-OR view graph, each equivalence node, which represents a candidate view v_i to materialization, has the following parameters associated to it: Query cost Qc (the evaluation cost of the cheapest embedded expression AND-DAG for v_i), maintenance cost Mc (the cost required for updating v_i when the related base relations are changed), reading cost Rc, query frequency fq (if the equivalence node is a root node) and the update frequency fu (the frequency of updating v_i in response to change to the underlying data). To each operation node op_i, that represents a relational operator, a cost is associated with it which is the cost incurred during the computation of the parent node of op_i from the children nodes of op_i.

The view selection problem for AND-OR view graphs can be formulated as follows: *Given an AND-OR view graph G, a maximum storage space Sp_{max} (available space), a total view maintenance limit U_{max}(available maintenance time), the problem is to select a set of views to be materialized MV, a subset of the equivalence nodes of G, that minimizes the total query cost, under the constraint that the total space occupied by MV is less or equal than Sp_{max} and the total maintenance time of MV (i.e., view maintenance cost) is less or equal than U_{max}.*

5 Our view selection approach

Let us now introduce the constraint satisfaction model that we have proposed for the view selection problem. We then present the search strategy that we have defined within the constraint solver for optimization purpose.

5.1 Modeling View Selection Problem as a Constraint Satisfaction Problem (CSP)

This section describes how to model the view selection problem as a CSP. Then, its resolution is supported automatically by the constraint solver. In the following, we define all the symbols as well as the variables that we have used in our constraint satisfaction model.

- G. The AND-OR view graph described in the previous section.
- $Q(G)$. The views which correspond to the query results (the root nodes in the AND-OR view graph G).
- $V(G)$. The set of views in G which are candidate to materialization.
- $U(v_i)$. The set of updates on v_i in response to changes of the associated base relations.
- $\delta(v_i, u)$: The differential result of view v_i with respect to update u.
- $f_q(v_i)$. The access frequency or importance of the associated view (or query) v_i.
- $f_u(v_i)$. The frequency of propagating the changes of each associated base relation to the view v_i.

- Sp_{max}. The maximum storage space that can be used to view materialization.
- U_{max}. The time that can be allotted to keep up to date the materialized views.
- $size(v_i)$. The size of the view v_i in terms of number of bytes.

CSP variables and their domains

- Mat_{v_i}. The materialization variable which denotes for each view v_i (equivalence node in the AND-OR view graph G), if it is materialized or not materialized. It is a binary variable, $d_{Mat_{v_i}}$ ={0,1} (0: v_i is not materialized, 1: v_i is materialized).
- $Qc(v_i)$. The query cost corresponding to the view v_i. The domain is a finite subset of $\mathbb{N}^*$ such as $d_{Qc(vi)} \subset \mathbb{N}^*$.
- $Mc(v_i)$. The maintenance cost corresponding to a view v_i, where $d_{Qc(vi)} \subset \mathbb{N}^*$.

The view selection problem can be formulated by the following constraint satisfaction model. It consists in specifying in a declarative way the CSP variables, their domains, and the constraints that are over them.

$$minimize \sum_{v_i \in Q(G)} \Big(f_q(v_i) * Qc(v_i) \Big) \tag{1}$$

$$subject\ to \sum_{v_i \in V(G)} \Big(Mat_{v_i} * size(v_i) \Big) \leq Sp_{max} \tag{2}$$

$$\sum_{v_i \in V(G)} \Big(Mat_{v_i} * f_u(v_i) * Mc(v_i) \Big) \leq U_{max} \tag{3}$$

In our approach, the main objective is the minimization of the total query cost. It is computed by summing over the cost of processing each input query rewritten over the materialized views. Constraints (2) and (3) state that the views are selected to be materialized under a limited amount of resources. Constraint (2) ensures that the total space occupied by the materialized views is less than or equal to the maximum storage space capacity. Constraint (3) guarantees that the total maintenance cost of the set of materialized views is less than or equal to the total view maintenance cost limit.

The query and maintenance costs corresponding to a view are implemented by using a depth-first traversal of the AND-OR view graph. We have been inspired by the formulas described in [24,28] to compute these two costs. Note that the query and maintenance costs corresponding to a base relation are equal to zero. They may be formulated as follows.

Query cost

$$Qc(v_i) \quad = \begin{cases} CCost(v_i) \;\; if \;\; Mat_{v_i} = 0 \\ Rc(v_i) \; otherwise \end{cases} \tag{4}$$

$$\begin{aligned} &where \\ &CCost(v_i) = \min_{op_j \in child(v_i)} \left(cost(op_j) + \sum_{v_k \in child(op_j)} Qc(v_k) \right) \end{aligned} \tag{5}$$

Constraint (4) states that the query cost corresponding to each given view in the AND-OR view graph is the minimum cost paths from the view to its related base relations or views. The reading cost is considered if the view has been materialized. Constraint (5) ensures that the minimum cost path is selected for computing a given view. Each minimum cost path includes all the cost of executing the operation nodes on the path and the query cost corresponding to the related bases relations or views.

View maintenance Cost

$$Mc(v_i) \quad = \begin{cases} 0 \;\; if \;\; Mat_{v_i} = 0 \\ \sum_{u \in U(v_i)} Mcost(v_i, u) \; otherwise \end{cases} \tag{6}$$

$$\begin{aligned} &where \\ &Mcost(v_i, u) = \min_{op_j \in child(v_i)} \Bigg(cost(op_j, u) \\ &\qquad + \sum_{v_k \in child(op_j)} UCost(v_k, u) \Bigg) \end{aligned} \tag{7}$$

$$UCost(v_k, u) = \begin{cases} Mcost(v_k, u) \;\; if \;\; Mat_{v_k} = 0 \\ \delta(v_k, u) \; otherwise \end{cases} \tag{8}$$

Constraint (6) guarantees that there is no maintenance cost if the view has not been materialized. Otherwise, the view maintenance cost is computed by summing the number of changes in the base relations from which the view is updated. We assume incremental maintenance to estimate the view maintenance cost. Therefore, the maintenance cost is the differential results of materialized views given the differential (updates) of the bases relations. Constraints (7) and (8) insure that the best plan with the minimum cost will be selected to maintain a view. The view maintenance cost is computed similarly to the query cost, but the cost of each minimum path is composed of all the cost of executing the operation nodes with respect to the updates on the path and the maintenance cost corresponding to the related base relations or views.

5.2 Search strategy

A key ingredient of any constraint satisfaction approach is an efficient search strategy. As mentioned in Section 3.2, the search is organized as an enumeration tree, where each node corresponds to a subspace of the search. The tree is progressively constructed by applying series of branching strategies that define the way to branch from a tree search node. In the constraint solver, branching has been applied to decision variables. In our constraint satisfaction model, the materialization variable Mat_{v_i} is the decision variable since the aim of the view selection problem is to decide which views to materialize. The most common branching strategies in the constraint solver are based on the assignment of a selected variable to one or several selected values (one assignment in each branch). Variable selector defines the way to choose a non instantiated variable on which the next decision will be made. Once the variable has been chosen, the solver has to compute its value.

5.2.1 The default search strategy The default search strategy is applied to the decision variables of the solver when no search strategy is specified. The default strategy selects the decision variables to be instantiated by using the following branching strategies.

Variable selection heuristic: DomOverWDeg. The strategy selects the variable Mat_{v_i} with the smallest ratio r:

$$r = \frac{dom}{w * deg}$$

where dom is the current domain size, deg is the current number of non instantiated constraints involving the variable, and w the sum of the counters of the failures caused by each constraint from the beginning of the search. To each variable $Mat(v_i)$ are associated, at any time the dom, deg and w values.

Value selection heuristic: MinVal. The variable Mat_{v_i} which has been chosen (by applying the variable selection heuristic) is then assigned, in the first branch, to its smallest value:

$$val = \min(d_{Mat_{v_i}})$$

In the next branch, the value val is removed from the variable domain $d_{Mat_{v_i}}$.

5.2.2 Our own search strategy As mentioned in Section 3.2, constraint programming offers facilities to control the search behavior. Defining our own search strategy is very important since a well-suited search strategy can reduce the number of expanded nodes and hence the time that the solver takes to find solutions to the view selection problem. In the following we describe the variable and value selection heuristics that we have defined in the search strategy.

Variable and value selection heuristics. Our aim is to minimize the query cost with a constraint on update time (maintenance cost constraint) and storage space (space constraint). Low query cost can be obtained by materializing all the queries of the workload (materializing the root level in the AND-OR view graph). In this case the view maintenance cost will be high. Low view maintenance cost can be achieved by leaving all the views virtual and in this case the query cost will be high (replicating the base relations which are in the leaf level of the AND-OR view graph). For this matter, our strategy consists in finding an intermediary level for each query tree in the AND-OR view graph that optimizes the query cost without violating the maintenance cost and space constraints. Therefore, our strategy is based on the notion of level in the AND-OR view graph [4]. For this purpose, each view (equivalence node) is associated to a level, which is defined as follows:

$$level(baserelation) = 0$$
$$level(view) = \max_{v_c \in child(view)} level(v_c) + 1$$

As presented in the code below, we explain how to compute for each query the relative query cost reduction associated to the different levels in the query tree.

```
levels = ∅ //set of levels with their cost saving
for each q in Q(G) do
  levelCS = ∅//Map : key = level; val = cost saving
  // each view in the query tree is associated to a level
  for each l in AllLevels(q) do
    space = 0
    maint = 0
    for each v in AllViews(l)  do
      space = space + size(v)
      maint = maint + Mc(v)
    end for
    if space ≤ Sp_Max and maint ≤ U_Max then
      LevelCostSaving(q, l)
      //LevelCostSaving is defined as the relative
      //query cost reduction when the views associated
      // to level l are materialized
    else
      LevelCostSaving(q, l) = −1
    end if
    levelCS.put(l, LevelCostSaving)
  end for
  levels = levels ∪ {levelCS}
end for
```

In order to guide the search to the optimal solution, the variable selector has to start by instantiating the materialization variables of the recommended views. These views are those associated to the levels that minimize the query cost subject to space and maintenance cost constraints. For this purpose, we sort the query levels according to their *LevelCostSaving* in descending order (as it is presented below). We iterate over the sorted set starting with the levels which have the highest query cost reduction. We then store each view associated to these levels in the variable MV.

```
//sort the levels according to their LevelCostSaving in
//descending order
LSort = SortLevels(levels)
for each l_s in LSort  do
  for each v_s in l_s  do
    MV = MV ∪ {Mat_{v_s}}
  end for
end for
```

Finally, the variable selector will choose the materialization variables to be instantiated in the order they appear in MV. Once the variable has been chosen, the value selector will assign the materialization variable to its highest value: $max(d_{Mat_{v_i}})$. Note that these variable and value heuristics do not inhibit the solver to compute solutions in which it will start by materializing another set of views. By defining these heuristics in the search strategy, we expect the solver to converge faster to the optimal solution and avoid browsing a large number of inferior solutions.

6 Performance Evaluation

In this section, we evaluate the performance of our approach through experimentations over the database schema of the TPC-H benchmark [2]. Our approach takes as input a set of selection-projection-join (SPJ) queries that may involve aggregation and group by clause as well. For each query, we consider all possible execution plans which represent its execution strategies. Then, all the queries are merged into the same graph (see Section 4) in order to detect the overlapping and capture the dependencies among them. Our approach produces as output the set of materialized views. The performance of our approach was evaluated by measuring the gain in solution quality obtained by the materialized views.

The rest of this section is organized as follows. In Section 6.1, we describe our experimental setup, and the randomized method used for comparison. In Section 6.2, we study the impact of variable and value selection heuristics on the search space explored by our approach. In Section 6.3, we first report experimental results when the view selection is decided under resource constraints and we present the results on performance by increasing the number of queries. Then,

we evaluate the effect of the frequency of queries and updates as well as the query complexity on performance. In Section 6.4, we study the benefit of using materialized views to improve query performance. Finally, we summarize the performance results in Section 6.5.

6.1 Experimental Setup

We have implemented our approach and compared it with a randomized method i.e., genetic algorithm . The latter was chosen for comparison since it has been argued that the genetic algorithm provides a good balance between the computing costs that an algorithm incurs for finding the materialized views and the gain to be realized in query processing by materializing these views (see Section 2). All the algorithms are implemented in Java and all the experiments were carried out on an Intel Core 2 Duo P8600 CPU @ 2.40 GHz machine running with 3GB of RAM and Windows XP Professional SP3.

In order to solve the view selection problem as a constraint satisfaction problem, we have used the latest powerful version of CHOCO [1] (knowing that the constraint solvers are structured around annual competitions [17]). For the genetic algorithm, we have implemented the one presented in [5] by incorporating space and maintenance cost constraints into the algorithm and without taking into account the data placement. In order to let the genetic algorithm converge quickly, we generated an initial population which represents a favorable view configuration rather than a random sampling. Favorable view configuration such as the views which minimize the query cost without violating space and maintenance cost constraints are most likely selected for materialization.

To evaluate the performance of view selection methods, we measure the following metric.

1. **Solution Quality**. The performance of view selection methods was evaluated by measuring the solution quality which results from evaluating the quality of the obtained set of materialized views in terms of cost saving. In the experimental results, the solution quality denoted by Q_s is computed as follows:

$$Q_s = \frac{WM - \sum_{v_i \in Q(G)} \big(f_q(v_i) * Qc(v_i)\big)}{WM - ALLM} \tag{9}$$

   Where WM is the total query cost obtained using the "WithoutMat" approach which does not materialize views and always recomputes queries, $AllM$ is the "AllMat" approach which materializes the result of each query of the workload. The "WithoutMat" and "AllMat" approaches are used as a benchmark for our normalized results. As defined in Section 5, $Qc(v_i)$ is the query cost corresponding to the view v_i and $f_q(v_i)$ is the frequency of the view v_i.

2. **Space constraint**. In the case where the view selection problem is decided under a space constraint, the total space occupied by the materialized views

has to be less than or equal to the maximum storage space Sp_{max} . Similar to [10] Sp_{max} is computed as a function of the size of the associated query workload.

$$Sp_{max} = \alpha * Sp_{AllM} \tag{10}$$

where Sp_{AllM} is the size of the whole workload and α is a constant. In our experiments, we assume the case where the view selection is studied under restrictive constraints and hence we set α to 10%. We also examine the case where the constraints are not very tight and at that case α was set to 30%.

3. **Maintenance Cost Constraint**. In the maintenance cost constrained model, the total maintenance cost of the set of materialized views has to be less than or equal to the total view maintenance cost limit U_{max}. As in previous work [10], U_{max} is calculated as a function of the total maintenance cost when all the queries are materialized.

$$U_{max} = \beta * Mc_{AllM} \tag{11}$$

where Mc_{AllM} is the total maintenance cost when the result of each query of the workload is materialized and β is a constant. The value of β was set similar to α (see above).

4. **Runtime**. The Runtime which we consider here is the time that MySQL server takes to compute query results using materialized view. This metric has been used in Section 6.4 to measure the running time of the query workload given a set of materialized views. Thus, the runtime is a good metric to study the benefit that materialized views found by our approach bring to query evaluation. It is also a good indicator for comparing the performance of our approach to those of the genetic algorithm.

6.2 Impact of variable and value selection heuristics

Here, we study the impact of variable and value selection heuristics that we have presented in Section 5.2.2, on the search space explored by our approach. To evaluate this, we attempted to compare the solution quality found by the constraint solver in the case where (i) the default search strategy is used and (ii) the variable and value selection heuristics that we have defined in Section 5.2.2 are implemented in the search strategy. As mentioned in Section 3.2, the constraint solver (CHOCO Solver) can find a set of feasible solutions in which all the constraints are satisfied before reaching the optimal solution. In this case, we use *timeout* condition to evaluate the quality of the different solutions found by the solver. A workload of 20 queries suffices to illustrate this. α and β, which define respectively the storage space and the view maintenance cost limits, was set to 30%. The results are shown in figure 3. The solver is left to run until reaching the optimal solution. *default search* denotes the default search strategy

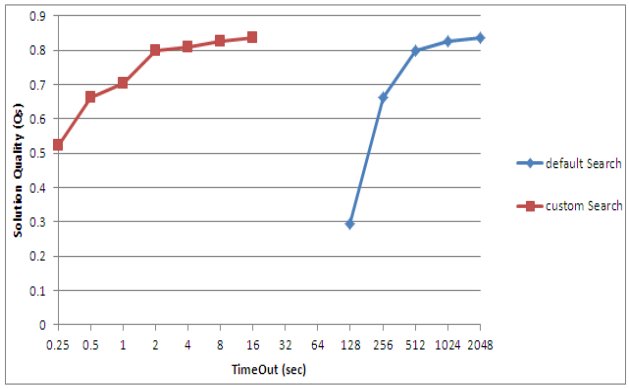
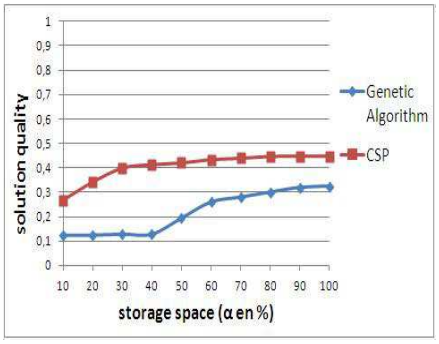
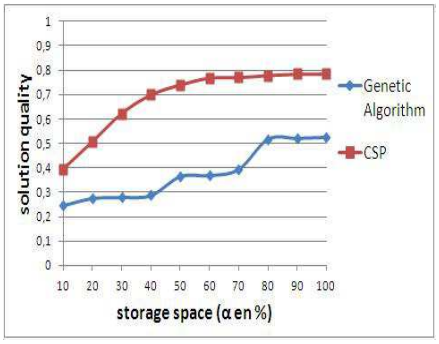


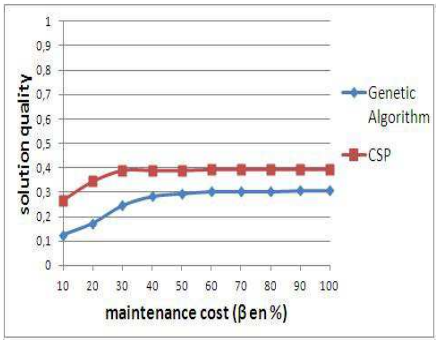
Fig. 3: Impact of heuristics on the search



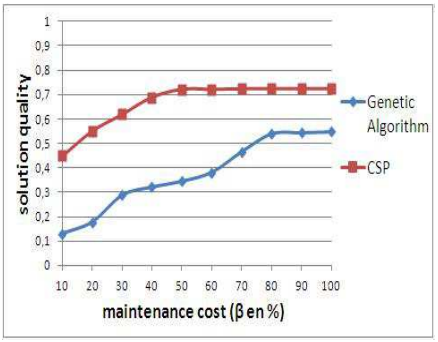
(a) β=10%



(b) β=30%



(c) α=10%



(d) α=30%

Fig. 4: Solution quality while varying the space or the maintenance cost constraint

while *custom search* requires the variable and value selection heuristics that we have defined in the search strategy. We can observe from figure 3 that the time that a solver incurs in the presence of *custom search* for finding near optimal and optimal solutions is significantly reduced. This is because the variable and value selection heuristics that we have defined in the search strategy reduce significantly the search space explored by the CHOCO solver. Consequently, our approach can provide high solution quality in a short time. In the following experiments, we use the *custom search* in the constraint satisfaction model.

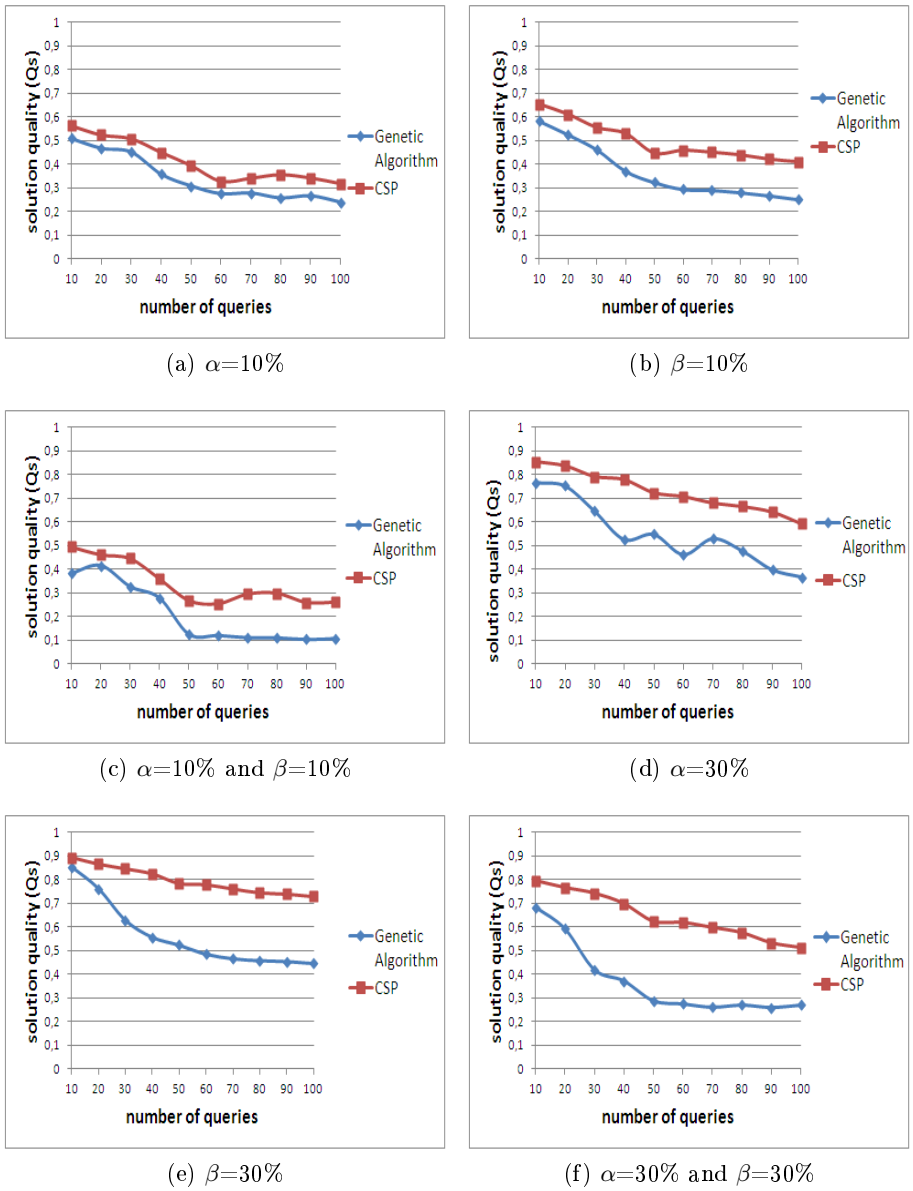
6.3 Solution quality: Our approach versus Genetic algorithm

In this section, we examined the effectiveness of our approach by measuring the gain in solution quality obtained by using our approach versus the genetic algorithm. First, we compare the performance of our approach to those of the genetic algorithm for various values of storage space and maintenance cost limits and then we present the impacts on performance by increasing the number of queries. We also evaluate the solution quality found by view selection methods with respect to different query and update distributions. Finally, we evaluate our approach and the genetic algorithm according to query complexity. In order to allow a fair comparison with the genetic algorithm and since our approach is able to provide a solution at any time, the CHOCO solver was left to run until the convergence of the genetic algorithm in the following experiments. More precisely, the *timeout* condition was set to the time required by the genetic algorithm to solve the view selection problem. Since the heuristic based search strategy allows the solver to find a high solution quality very fast (as described in the previous section) and the genetic algorithm requires an amount of time to converge, we expect to achieve significant performance gains in comparison with the genetic algorithm in terms of cost saving.

6.3.1 Resource constraints In this experiment, we first examine the impact of space and maintenance cost constraints on solution quality. For this evaluation, we consider a workload of 50 queries. Recall that for each query, we consider all possible execution plans which represent its execution strategies. The query and update frequencies are at scale 1. The values of α and β which define respectively the storage space capacity and the view maintenance cost limit are varied from 10% to 100%. All the results are shown in figure 4.

Figure 4a and Figure 4b investigate respectively the influence of space constraint on solution quality for each value of α where β was set to 10% and 30%, while figure 4c and figure 4d examine respectively the impact of maintenance cost constraint on solution quality for each value of β where α was set to 10% and 30%. We note from these experiments that the quality of the solutions produced by our approach and genetic algorithm improves when α (see figure 4a and figure 4b) or β (see figure 4c and figure 4d) increases. However, there is no improvement in the solution quality from certain values of α or β because the maintenance cost constraint or the space constraint becomes the significant factor.

We also observe from figure 4 that our approach provides better solution quality in the case where the view selection is decided under a maintenance cost constraint (i.e., $Q_s \approx 0.8$ when α=100% and β=30% in figure 4b while $Q_s \approx 0.7$ when β=100% and α=30% in figure 4d). The reason is the maintenance cost of a



(a) α=10%

(b) β=10%

(c) α=10% and β=10%

(d) α=30%

(e) β=30%

(f) α=30% and β=30%

Fig. 5: Solution quality on large workloads under different resource constraints

view may decrease with selection of other views for materialization. Hence, there is time to update more views. This non monotonic nature of view maintenance cost is formally defined in [8].

Finally, we conclude from these experiments that our approach outperforms the genetic algorithm for different values of α and β in terms of cost saving. Indeed, we can see that our approach generates solutions with cost saving up to 2 times more than the genetic algorithm.

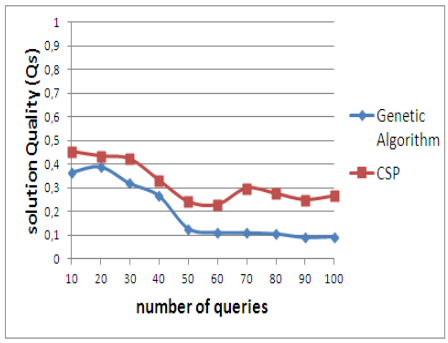
6.3.2 Large query workload Let us now evaluate the performance of our approach and the one of genetic algorithm on larger query workload. For this purpose, we generated workloads of 10, 20, 30, 40, 50, 60, 70, 80, 90 and 100 queries. The solution quality of our approach and the genetic algorithm is evaluated when the view selection is decided under the case where (i) only the space constraint is considered (see figure 5a and figure 5d); (ii) the limiting factor is the view maintenance cost(see figure 5b and figure 5e); and (iii) both maintenance cost and space constraints exists (see figure 5c and figure 5f). On each of these cases, we consider the case where the resource constraints become very tight (α and/or $\beta = 10\%$) as well as the case where we relax them (α and/or $\beta = 30\%$).

For this collection of experiments, we make the following observations. Our approach provides in all the cases better performances in terms of the solution quality while varying the number of queries. For example for a workload of 100 queries where α and β was set to 30% (see figure 5f), our approach provides a cost saving of 24% more than the genetic algorithm ($Qs_{CSP} = 0.512$ while $Qs_{GeneticAlgorithm} = 0.27$). Another remark based on figure 5 is that in our

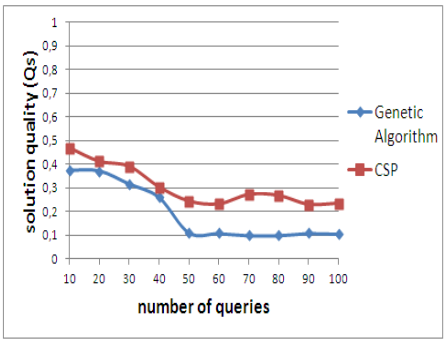
q_{random}	The values of the query frequencies have been assigned randomly to each query of the workload.
$q_{uniform}$	All the queries of the workload have the same query frequency
$q_{gaussian}$	Queries from certain levels have higher probability to be queried. The frequency distribution is normal with $\mu = 1/2$ and $\sigma = 1$.
u_{random}	The values of the update frequencies have been assigned using a random distribution.
$u_{uniform}$	All the views in the AND-OR view graph have the same update frequency
$u_{gaussian}$	The views which are at the lower level of the AND-OR view graph have higher probability to be updated than those which are on the upper level (guassian distribution with $\mu = 1/2$ and $\sigma = 1$).

Table 1: Distribution of query and update frequencies

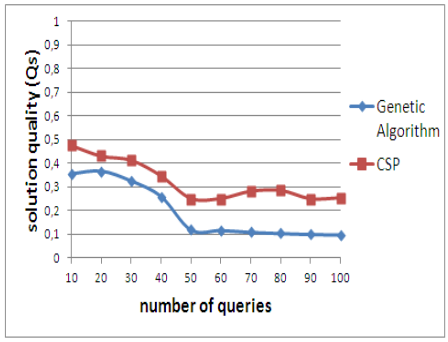
approach the gain in solution quality tends to be relatively more significant when we have more resource constraints. For instance, the gain in solution quality obtained by our approach is up to 10% (in figure 5a) and 16% (in figure 5b) more than the genetic algorithm. While this gain is up to 18% in figure 5c. This



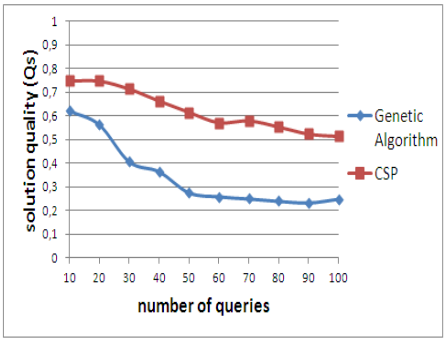
(a) q_{random}, α=10% and β=10%



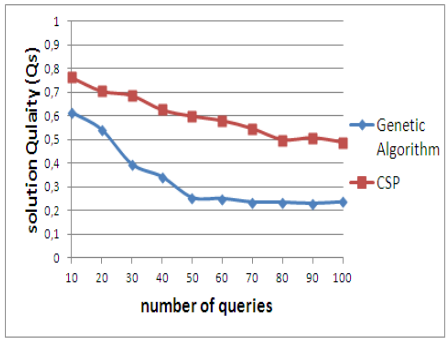
(b) $q_{uniform}$, α=10% and β=10%



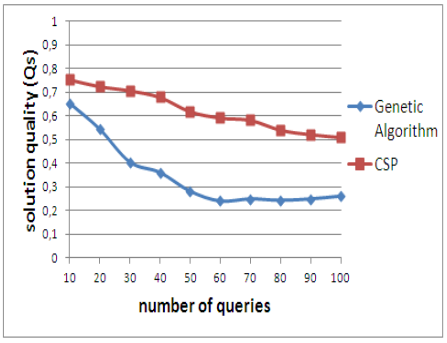
(c) $q_{gaussian}$, α=10% and β=10%



(d) q_{random}, α=30% and β=30%



(e) $q_{uniform}$, α=30% and β=30%



(f) $q_{gaussian}$, α=30% and β=30%

Fig. 6: Solution quality for different query distributions

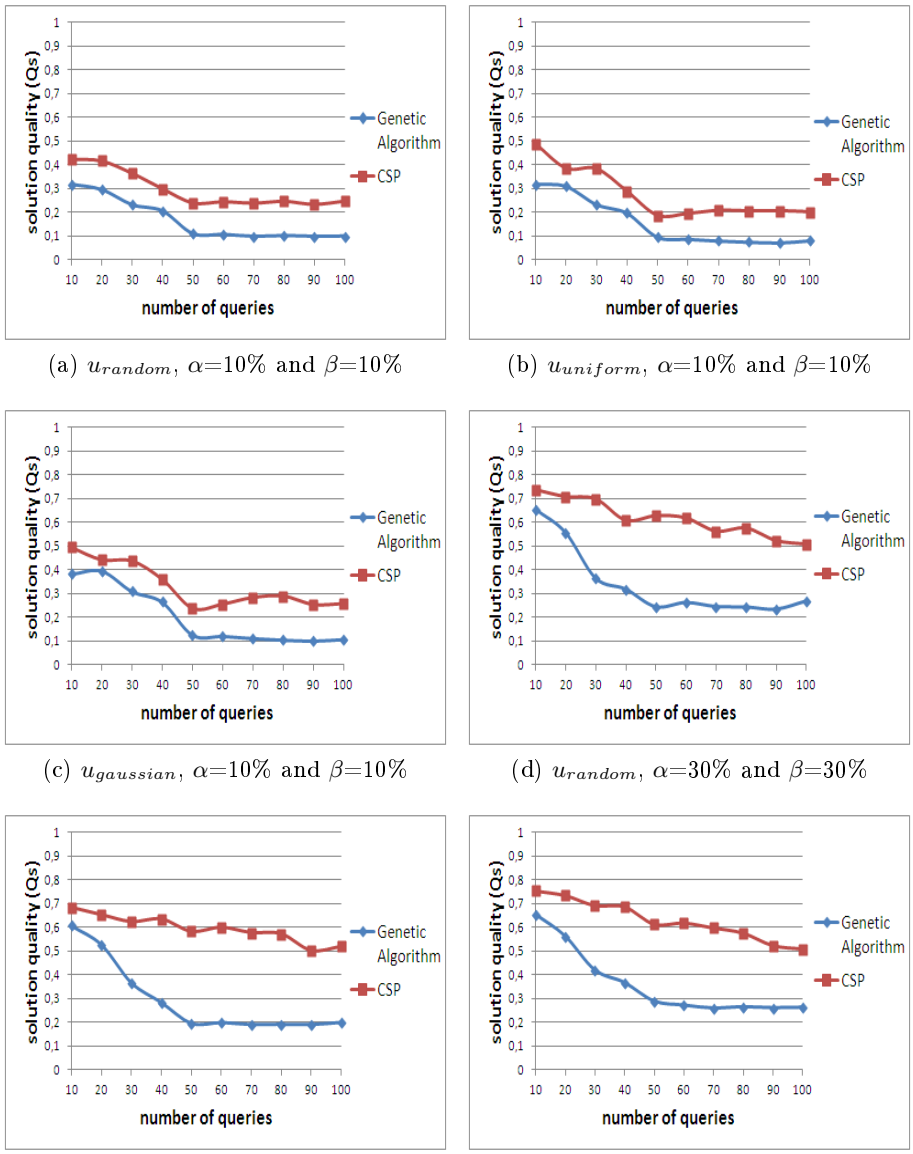
is because the idea of constraint programming is to solve problems by stating constraints and the search space is reduced when there are more constraints. This result is similar to the case where we relax the constraints (see figures 5d, 5e and 5f).

6.3.3 Query and update distributions We now study the behavior of view selection methods while varying the query and update frequencies. For this purpose, we generated different query and update distribution to simulate various workloads (see table 1). The random distribution assigns random values to query or update frequencies. While, the uniform distribution simulates cases where all views (or queries) have equal probability to be queried and updated. The last distribution which is the gaussian distribution favors views (or queries) from lower levels in the AND-OR view graph that have higher probability to be queried or updated. For example, queries of the TPC-H benchmark which contain less relational operators have higher probability to be queried.

Figures 6 illustrates the quality of the solutions produced by the two methods for different query distributions (q_{random}, $q_{uniform}$, $q_{gaussian}$). In the first combination, α and β were set to 10% (see figures 6a, 6b and 6c). While, for the other combination, α and β were set to 30% (see figures 6d, 6e and 6f). The update frequencies are at scale 1. We have made the same experiments for different update distributions in which the query frequencies was at scale 1 (see figure 7).

We can see that the quality of the solutions found by our approach is always better than those of the genetic algorithm for different query and update distributions. For example, in figure 6 and in the worst case which arises at the random workload (q_{random};α=10%;β=10%), our approach provides solutions with a cost saving of 4% more than the genetic algorithm. While, in the best case which arises at the gaussian workload ($q_{gaussian}$;α=30%;β=30%), the cost saving is 35% more than the genetic algorithm.

6.3.4 Query complexity We study the effect of query complexity on view selection performance. More specifically, we evolved the number of join operators $N_{Join_{Op}}$ for each query of the workload since the complexity of binary operators is more important than the one of unary operators. This results to three different workloads: (i) c_query_01 ($N_{Join_{Op}} < 2$); (ii) c_query_02, ($2 \leq N_{Join_{Op}} < 4$); and (iii) c_query_03, ($N_{Join_{Op}} \geq 4$). We run experiments with a workload of 50 queries and we measure the gain in solution quality according to the set of the obtained materialized views. The frequencies for access and update are at scale 1. Figure 8 shows the cost saving found by our approach and the genetic algorithm for both cases: (i) α and β was set to 10% (see figure 8a) and (ii) α and β was set to 30% (see figure 8b). We can see that our approach produce the best results. Indeed, our approach provides a cost saving up to 27.2% when α and β was set to 10% and 63.3% when α and β was set to 30%. While the genetic algorithm achieve a cost saving of only 12.9% when α and β was set to 10% and 29.3% when α and β was set to 30%. We also observe, in the graphic



(a) u_{random}, α=10% and β=10%

(b) $u_{uniform}$, α=10% and β=10%

(c) $u_{gaussian}$, α=10% and β=10%

(d) u_{random}, α=30% and β=30%

(e) $u_{uniform}$, α=30% and β=30%

(f) $u_{gaussian}$, α=30% and β=30%

Fig. 7: Solution quality for different update distributions

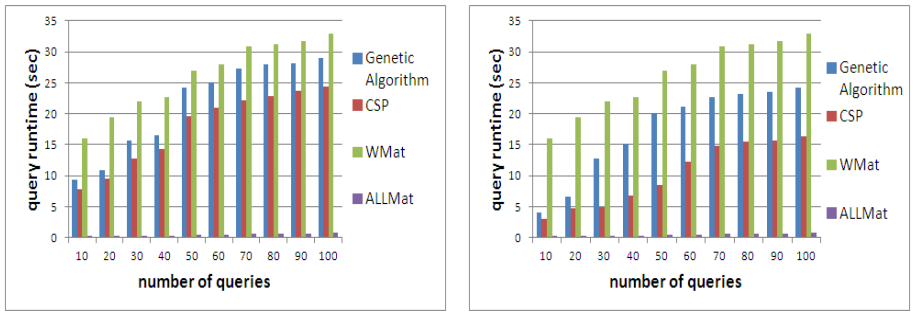
depicted in figure 8, that the quality of the solutions produced by our approach slightly decrease with an increasing complexity of the query workload. Hence, we confirm that the performance of our approach is not significantly influenced by an increasing of query complexity.

(a) α=10% and β=10% (b) α=30% and β=30%

Fig. 8: Query complexity on view selection performance

6.4 Query performance using materialized views

In this section, we study the benefit of using materialized views to improve query performance. For a workload involving 10, 20, 30, 40, 50, 60, 70, 80, 90 and 100 queries, we materialized the views proposed by our approach and the genetic algorithm. Then, we run the query workload using these views. We also consider the two basic strategies that we have defined above: the "WithoutMat" and the "AllMat" approaches. Recall that the "WithoutMat" approach does not materialize views and always recomputes queries. While the "AllMat" approach materializes the result of each query without any resource constraint. The frequencies for access and update are at scale 1. In order to measure the query runtime, the experiments were performed on MySQL server through JDBC interface. The query runtime is expressed in seconds (sec).



(a) α=10% and β=10% (b) α=30% and β=30%

Fig. 9: Query runtime using materialized views

The results are shown in figure 9. The view selection has been decided under space and maintenance cost constraints: (i) α and β was set to 10% in figure 9a and (ii) α and β was set to 30% in figure 9b. The results indicate that the benefit

of using materialized views is significant. Indeed, queries using our proposed views or those of the genetic algorithm are evaluated faster in comparison with the "WithoutMat" approach. We can also see that our approach provides the better quality of the obtained set of materialized views. For instance as can be seen in figure 9b, when comparing the runtimes of the workload of 100 queries, our approach requires $\approx 16 seconds$ while genetic algorithm takes $\approx 24 seconds$. According to the equation (9) in section 6.1,

$$Qs_{CSP} = 0.518 = \frac{32.86(WM) - 16.297(CSP)}{32.86(WM) - 0.892(ALLM)} \quad (12)$$

$$Qs_{GeneticAlgorithm} = 0.273 = \frac{32.86(WM) - 24.126(GeneticAlgorithm)}{32.86(WM) - 0.892(ALLM)} \quad (13)$$

This result confirms our expectation in section 6.3.2 that for a workload of 100 queries where α and β was set to 30%, our approach provides a cost saving of 24% more than the genetic algorithm ($Qs_{CSP} = 0.512$ while $Qs_{GeneticAlgorithm} = 0.27$). Another important remark is that our approach is able to provide materialized views that produce higher cost savings even if the underlying cost model is simplified (see section 3). Thus, our approach is robust toward simplified cost models which is an important requirement for a practical solution to the view selection problem.

6.5 Concluding remarks

Our experiments show that our approach outperforms the genetic algorithm in many cases. We achieve impressive cost saving factors when (i) we study the view selection under resource constraints, (ii) we increase the number of queries and (iii) we simulate various query workloads. We also show the efficiency of our approach when we run the query workloads on MySQL server i.e., queries using our proposed views are evaluated faster in comparison with those found by the genetic algorithm. The experiment results confirm our expectation that our own search strategy allows our approach to achieve significant performance gains in comparison with the genetic algorithm.

7 Conclusion

The most efficient algorithm proposed so far for deciding which views to materialize is the genetic algorithm that provides the best trade-off between quality of solutions and execution time. However, there is no guarantee of performance because the probabilistic behavior of the genetic algorithms does not insure to find the global optimum. Besides, the quality of the solution depends on the set-up of the algorithm as well as the extremely difficult fine-tuning of algorithm that must be performed during many test runs.

In this paper, we proposed a declarative approach which simply modeled the view selection problem as a CSP without the need of being interested in the way the problem is solved. Indeed, its resolution was supported automatically by the constraint solver. We also designed a heuristic search strategy within the constraint solver to reduce the solution space and hence the execution time. The experiment results confirm our expectation that our own search strategy allows

our approach to achieve significant performance gains in comparison with the genetic algorithm.

More recently, the view selection has been investigated in data placement in a distributed setting [5]. For this purpose, we have extended our constraint satisfaction model to deal with the distributed setting. The formulation of the view selection problem in a distributed context can be found in our recent work [22]. As a future work, we are planning to solve the view selection problem in a large scale distributed environments such as peer to peer or cloud computing environments.

In our proposals, all queries are assumed to be known and given in advance and there is a frequency of occurrence associated with each query. One line of recent research [12] has explored the problem of identifying subject area specific queries from which frequent queries are selected. As a result, they obtain significant performance improvements when processing queries. However, the proposed approach is based on a given workload and chooses accordingly the set of views to materialize. In order to respond to the changes in the query workload over time, views need to be selected continuously. Consequently, the dynamic view selection issue will be a part of our planned future work while studying the view selection in large scale distributed environments.

References

1. Choco, open-source software for constraint satisfaction problems. http://www.emn.fr/z-info/choco-solver.
2. The TPC benchmark H (TPC-H). http://www.tpc.org/tpch/spec/tpch2.14.3.pdf.
3. Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, pages 496–505, Cairo, Egypt, 2000.
4. Xavier Baril and Zohra Bellahsene. Selection of materialized views: A cost-based approach. In *CAiSE*, pages 665–680, Klagenfurt, Austria, 2003.
5. Leonardo Weiss F. Chaves, Erik Buchmann, Fabian Hueske, and Klemens Böhm. Towards materialized view selection for distributed databases. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 1088–1099, New York, NY, USA, 2009. ACM.
6. Weimin Du, Ravi Krishnamurthy, and Ming chien Shan. Query optimization in heterogeneous dbms. In *In Proc. of VLDB, pp 277–91*, ancouver, British Columbia, Canada, 1992.
7. Himanshu Gupta. Selection of views to materialize in a data warehouse. In *ICDT*, pages 98–112, Delphi, Greece, 1997.
8. Himanshu Gupta and Inderpal Singh Mumick. Selection of views to materialize under a maintenance cost constraint. In *ICDT*, pages 453–470, Jerusalem, Israel, 1999.
9. Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In *SIGMOD Conference*, pages 205–216, Montreal, Canada, 1996.
10. Panos Kalnis, Nikos Mamoulis, and Dimitris Papadias. View selection using randomized search. *Data Knowl. Eng.*, 42(1):89–111, 2002.
11. Howard J. Karloff and Milena Mihail. On the complexity of the view-selection problem. In *PODS*, pages 167–173, Philadephia, Pennsylvania, USA, 1999.

12. T. V. Vijay Kumar, Gaurav Dubey, and Archana Singh. Frequent queries selection for view materialization. In *ACITY (2)*, pages 521–530, 2012.
13. T. V. Vijay Kumar and Santosh Kumar. Materialized view selection using genetic algorithm. In *IC3*, pages 225–237, 2012.
14. T. V. Vijay Kumar and Santosh Kumar. Materialized view selection using iterative improvement. In *ACITY (3)*, pages 205–213, 2012.
15. Vipin Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, 1992.
16. Wilburt Labio, Dallan Quass, and Brad Adelberg. Physical database design for data warehouses. In *Proceedings of the Thirteenth International Conference on Data Engineering*, ICDE '97, pages 277–288, Washington, DC, USA, 1997. IEEE Computer Society.
17. Christophe Lecoutre, Olivier Roussel, and Marc R. C. van Dongen. Promoting robust black-box solvers through competitions. *Constraints*, 15(3):317–326, 2010.
18. Minsoo Lee and Joachim Hammer. Speeding up materialized view selection in data warehouses using a randomized algorithm. *Int. J. Cooperative Inf. Syst.*, 10(3):327–353, 2001.
19. Spyros Ligoudistianos, Dimitri Theodoratos, and Timos K. Sellis. Experimental evaluation of data warehouse configuration algorithms. In *DEXA Workshop*, pages 218–223, Vienna, Austria, 1998.
20. Lothar F. Mackert and Guy M. Lohman. R* optimizer validation and performance evaluation for local queries. In *Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, SIGMOD '86, pages 84–95, New York, NY, USA, 1986. ACM.
21. Imene Mami and Zohra Bellahsene. A survey of view selection methods. *SIGMOD Record*, 41(1):20–29, 2012.
22. Imene Mami, Zohra Bellahsene, and Remi Coletta. View selection under multiple resource constraints in a distributed context. In *DEXA (2)*, pages 281–296, Vienna, Austria, 2012.
23. Imene Mami, Remi Coletta, and Zhora Bellahsene. Modeling view selection as a constraint satisfaction problem. In *DEXA (2)*, pages 396–410, Toulouse, France, 2011.
24. Hoshi Mistry, Prasan Roy, S. Sudarshan, and Krithi Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *SIGMOD Conference*, pages 307–318, Santa Barbara, California, USA, 2001.
25. Luc De Raedt, Tias Guns, and Siegfried Nijssen. Constraint programming for itemset mining. In *KDD*, pages 204–212, Las Vegas, USA, 2008.
26. Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.
27. Nick Roussopoulos. The logical access path schema of a database. *IEEE Trans. Software Eng.*, 8(6):563–573, 1982.
28. Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD Conference*, pages 249–260, Dallas, Texas, USA, 2000.
29. Jong-Soo Sohn, Jin-Hyuk Yang, and In-Jeong Chung. Improved view selection algorithm in data warehouse. In *ICITCS*, pages 921–928, 2012.
30. Dimitri Theodoratos, Spyros Ligoudistianos, and Timos K. Sellis. View selection for designing the global data warehouse. *Data Knowl. Eng.*, 39(3):219–240, 2001.
31. Dimitri Theodoratos and Timos K. Sellis. Data warehouse configuration. In *VLDB*, pages 126–135, Athens, Greece, 1997.

32. Jian Yang, Kamalakar Karlapalem, and Qing Li. Algorithms for materialized view design in data warehousing environment. In *VLDB*, pages 136–145, Athens, Greece, 1997.
33. Chuan Zhang and Jian Yang. Genetic algorithm for materialized view selection in data warehouse environments. In *DaWaK*, pages 116–125, Florence, Italy, 1999.
34. Chuan Zhang, Xin Yao, and Jian Yang. An evolutionary approach to materialized views selection in a data warehouse environment. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 31(3):282–294, 2001.